Summer 2000

A Flowchart Structure for Modification of a MODSIM Process Model

Murali K. Adatrao
Old Dominion University

A FLOWCHART STRUCTURE FOR MODIFICATION OF A MODSIM PROCESS MODEL

by

Murali K Adatrao
B.Tech. July 1996, Jawaharlal Nehru Technological University, Hyderabad, India

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

MASTER OF SCIENCE

COMPUTER ENGINEERING

OLD DOMINION UNIVERSITY
August 2000

Approved by:

[illegible]
Dr. James F. Leathrum, Jr. (Director)

Dr. Roland R. Mielke (Member)

[illegible]
Dr. John W. Stoughton (Member)

ABSTRACT

A FLOWCHART STRUCTURE FOR
MODIFICATION OF A MODSIM PROCESS MODEL

Murali K Adatrao
Old Dominion University, 1997
Director: Dr. James F. Leathrum, Jr.

There are many software processes and software development models that support the development of a software model prior to implementation. However, more often than not, these practices are not followed resulting in a lack of documentation of the intended functionality of the software. As a result, software often becomes a black box for later developers. Even a simple bug fix can turn into an exhaustive task for the developers, as they must attempt to infer the intended system behavior. New designers cannot make any changes to the software or extend the software behavior, as the underlying model within the software itself is unclear. An understanding of the underlying model helps both the designers and programmers to do their job quickly and effectively.

This thesis proposes a model to describe an existing undocumented software system developed in MODSIM III to facilitate future development efforts. The model provides a way to understand the underlying model of the software system by first documenting the software using flowchart constructs. The model also ensures designers and programmers make respective changes in the flowcharts before implementing the proposed changes. This helps document the recent architecture of the software process in the form of flowcharts. The model is described in this thesis and demonstrated on the port simulation PORTSIM.

This thesis is dedicated to my mother.

ACKNOWLEDGMENTS

I would like to thank my committee members for their patience in guiding my research and editing of this manuscript. The untiring efforts of my major advisor who helped me in all stages of this thesis deserve special recognition.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SECTION 1

INTRODUCTION

1.1 Overview

This thesis proposes a systematic approach that can be followed during the modification and maintenance of an undocumented, complex software system and is applied to simulations developed in MODSIM III [13]. The process is demonstrated on a port simulation, PORTSIM [12]. The objectives of this process are to abstract documentation of the software in the form of process flow in order to understand the underlying model and use it while developing the software. The documentation should provide required information for the designers and the programmers for modeling and debugging. Here debugging represents removing modeling errors but not coding errors. The document should, however, help to isolate code containing coding errors. The model is specific to MODSIM in order to capture the language constructs inherent in MODSIM, but new models can be developed for other languages in a similar manner.

The main problem in working on undocumented software is that the tasks it performs are identifiable, but the process of accomplishing those tasks is not known. In order to make any changes in the software, it is necessary to understand the way in which the software is accomplishing its tasks.

In software development models, the software system is first analyzed and designed from the specifications before going into implementation phase. So by the time the software

This Thesis follows the *IEEE Software Engineering Journal* Model reaches the implementation phase the designers have a good understanding of what is happening in the system. Also, the programmers, if they are not the designers, can learn about the system by referring to design documents. The design documents are also helpful during debugging. By understanding the kind of bug the software is suffering from, one can make a good guess on the part of the code that is causing the problem. Hence there is a great need to document the undocumented software to continue to develop the software effectively.

In most cases, designers design the system in accordance to the programming environment that will be used to develop the system. Knowledge of the programming environment enables one to adopt design strategies relative to the programming environment. The way one documents the software should reflect the environment of the programming language used in developing the software. By doing this, any changes proposed by the designers through the documentation can be easily implemented by the programmers. One should take care to make the software's document represent its underlying model instead of the sequence of executable statements. If possible the system should be divided into different logical groups while documenting. This helps to track a bug during debugging once we understand the type of bug and also to speed up the process of modification by only concentrating on the enclosing logical unit while making changes to the system.

1.2 How the Objectives are Achieved

In the proposed model, the above mentioned requirements are met by first developing a set of pictorial representations of MODSIM III constructs and using them in the flowcharts to represent the program flow within the software. By doing this, the flowcharts themselves provide the software's programming environment to the designers. The programmers can easily convert any modifications in the flowcharts into MODSIM III code as long as the designers design their modification within the boundary of the pictorial set. A logical unit is constructed by referring to the related flowcharts from within a flowchart.

1.3 Section Overview

Section 2 provides a brief introduction to the programming language, MODSIM III, used to develop the software, PORTSIM, and to the software itself. It also provides an overview of well-known software models and discusses whether PORTSIM can follow any of the models. In the conclusion, the importance of flowcharts in documenting PORTSIM is discussed.

Section 3 discusses in-depth the important MODSIM III language constructs and how they can be represented pictorially. It later discusses how the pictorial representation of MODSIM III constructs can be used in developing flow diagrams representing the program flow within PORTSIM and how the resulting flow diagrams can be used in the development process of the PORTSIM.

Section 4 provides examples on how to use the flowcharts for debugging and for modeling the code to make improvements.

Section 5 gives the summary of the benefits and the drawbacks and possible solutions of the proposed model.

SECTION 2

BACKGROUND

This section provides an introduction to MODSIM III programming language, PORTSIM, a simulation tool developed to simulate military cargo flow through commercial ports, and to the software engineering models that are used to develop software systems. Finally, this section discusses how the present models are inefficient to use in the development cycle of PORTSIM.

2.1 MODSIM III

MODSIM III is a modular, object-oriented, strongly typed, block structured, discrete-event simulation language [5]. Every MODSIM III program contains a main module. Programs can be divided into modules with each module supporting some particular functionality. Any module can be compiled separately, making the maintenance of the code easy. Each module may contain a number of objects. An object is an encapsulation of data items which describe the state of the object and methods which describe the interface of the object. MODSIM III also supports inheritance and polymorphism. Every expression and method parameter is checked at compilation time for type consistency. This strongly typed feature helps to identify errors at compilation time rather than at run time. The scope and visibility of variables is restricted to the block in which they are declared. The library modules and the language features provide all the capabilities for developing discrete-event simulation models. MODSIM III demands that simulation models be developed in terms of process. The process is capable of performing multiple concurrent activities. The activities can be modeled to operate independently or in a synchronized fashion. This process model helps to code a related group of activities in one routine.

2.2 PORTSIM

PORTSIM is a database-driven, object-oriented, computer simulation [12]. The software is being developed in MODSIM III. PORTSIM is useful for force deployment through a given commercial port [14]. It is useful in estimating the time of embarkation and disembarkation of a force, given the parameters of the port and the force [11]. The purpose of this software is to study the effects of initial parameter values on the end results. The software can be used to simulate either port of embarkation (POE) or port of disembarkation (POD) operations. In either mode, we can either create a new scenario in which PORTSIM relies heavily on the underlying database to get required data or load an existing scenario for which PORTSIM reads data from the scenario file.

2.3 Software Models

There are many systematic development models that are being used for developing a software model. The first model that was used to develop a software model is the Waterfall model. The various stages involved in the Waterfall model is shown in Fig. 1 [1]. Though the Waterfall model does have disadvantages, it showed that a systematic approach in developing the software does make improvement in the quality of the code and the time taken to develop the code. Other significant models which were developed later are:

1. Spiral model [1], [2].
2. Booch method.
3. Object Modeling Technique (OMT) [10].
4. Object-Oriented Software Engineering (OOSE) [9].
5. Unified Modeling Language (UML) [7], [8].

UML is the result of combining the Booch, OMT and OOSE methods [3].

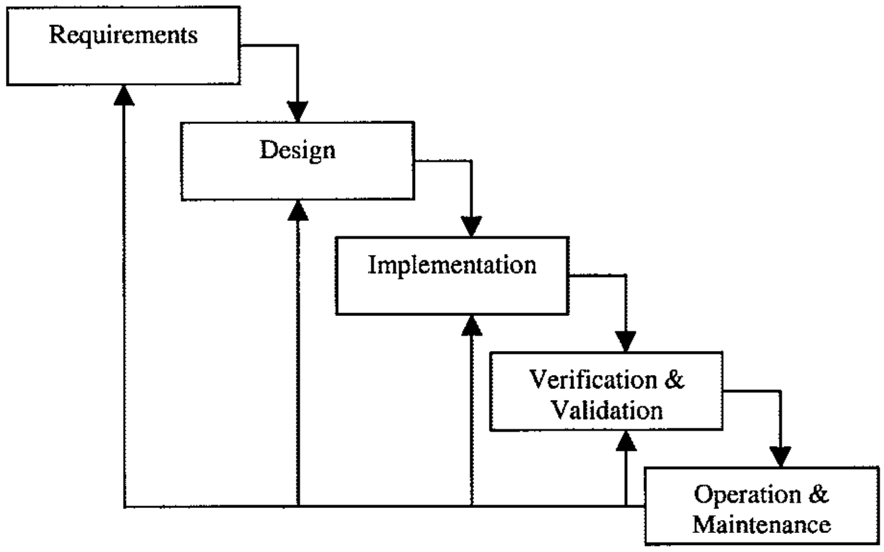


Fig. 1 Waterfall model

2.3.1 Introduction to the Booch Model

This section discusses the Booch model as an example which uses Object-Oriented Analysis and Object-Oriented Design [4]. The Booch model divides the process of software development into two processes, namely:

1. Macro process.
2. Micro process.

Macro process: Booch's Macro development process [4] is shown in Fig. 2, which describes the overall process of software development.

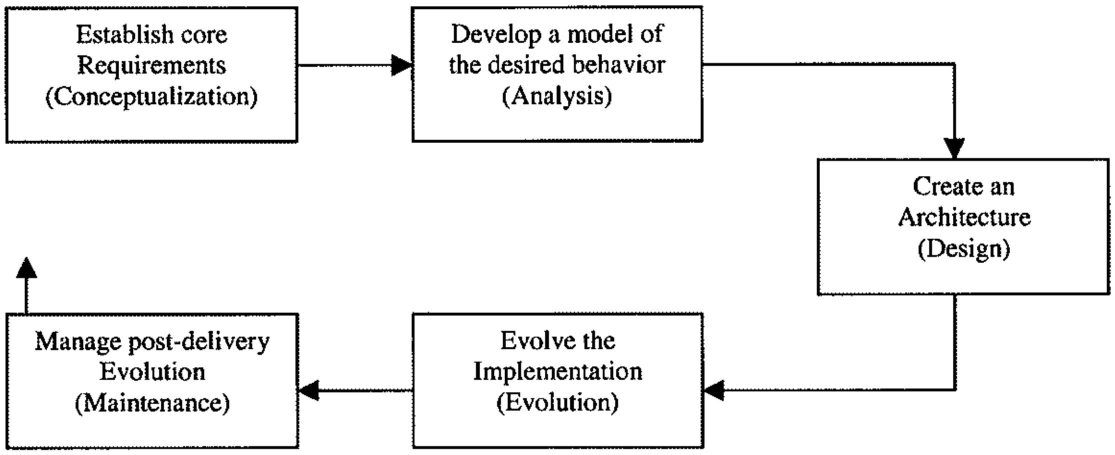

Establish core Requirements (Conceptualization)
Develop a model of the desired behavior (Analysis)
Create an Architecture (Design)
Evolve the Implementation (Evolution)
Manage post-delivery Evolution (Maintenance)

Fig. 2 Booch's Macro development process

Micro process: Booch's Micro development process is shown in Fig. 3 [4], which is used to carry out iteratively the activities in the Macro process.

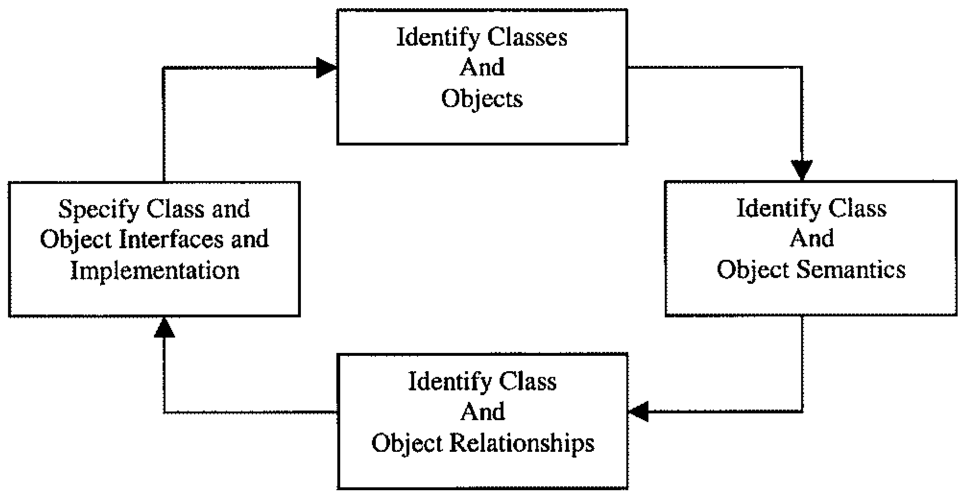

Identify Classes And Objects
Identify Class And Object Semantics
Identify Class And Object Relationships
Specify Class and Object Interfaces and Implementation

Fig. 3 Booch's Micro development process

Actually, Booch's Macro process can be represented as shown in Fig. 4, where each small circle represents a micro process. For each activity in macro process emphasis will be given on different activity in the corresponding micro process.

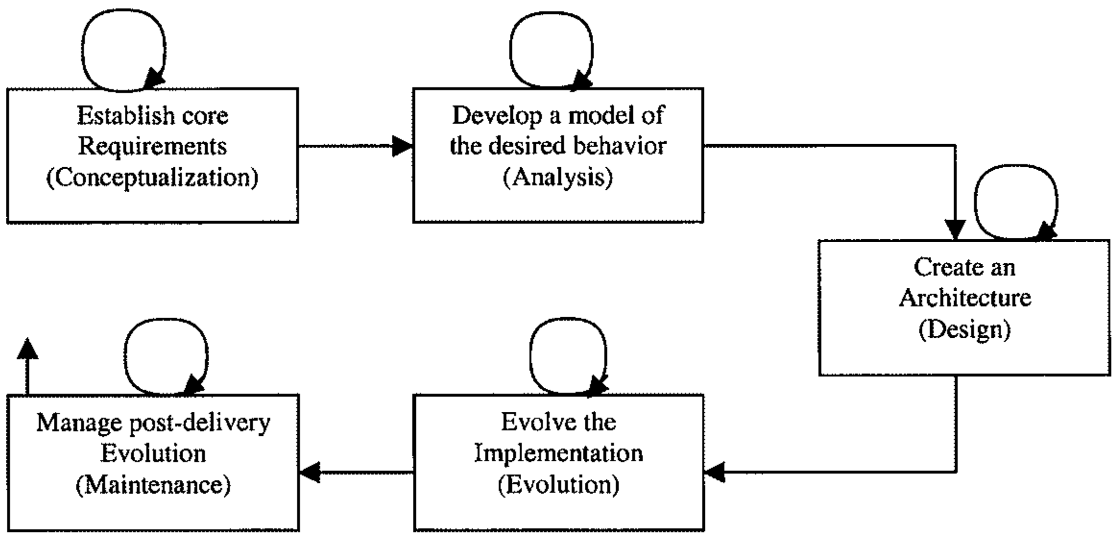

Establish core Requirements (Conceptualization)
Develop a model of the desired behavior (Analysis)
Create an Architecture (Design)
Evolve the Implementation (Evolution)
Manage post-delivery Evolution (Maintenance)

Fig. 4 Alternate representation of Booch's Macro development process

In both the Booch method and the UML, the class diagram is used to specify static relationships between classes in the design phase. Class diagrams do not represent dynamic relationships such as when objects are created or invoke services of other objects.

2.3.2 Software development Models and PORTSIM

From the above discussion, we can see that the present software development models are useful for developing software from software's overall specification to final product. The models actually start from a higher level model of the software and go down to implementation or a lower level model in the later stages. Also, in any given intermediate state of development the models seem to have a proper documentation for the software under development. The absence of any such documentation for PORTSIM and the software structure suggest that PORTSIM did not follow any of the development methods mentioned above. So, to successfully use any of the development methods for PORTSIM, we need to document the software by developing the higher level model of the software from its implementation. As the software is already in an implementation stage, deciding on which development method to use is difficult. Also, the higher level model developed from the software may not be consistent with the higher level model that may be developed if the chosen development method is actually used from the beginning. So it is not wise to think of using any of the above mentioned development methods in the modeling of PORTSIM. The only way left to document PORTSIM is to represent the present architecture in such a way that it will be useful to identify the underlying model, to identify the dynamic relationships between objects and also to make future enhancements easily and successfully.

2.4 Flowcharting

Flowcharts help to represent the underlying model and the dynamic relationships between objects, thus eliminating any need to develop a higher level model to make future enhancements. Since PORTSIM is implemented in MODSIM III, it is necessary to develop flowchart constructs to represent MODSIM III language constructs. By doing this, the resulting flowchart representation of PORTSIM helps both designers and programmers. Designers can now make changes in the flowcharts using the standardized representation and programmers can convert the changes into MODSIM III constructs. That is, flowcharts developed with a one-to-one relation between the flowchart constructs and programming language constructs act like a common interface between designers and programmers.

SECTION 3

PROCESS FOR MODIFICATION OF EXISTING CODE USING FLOW DIAGRAMS

The modification process of existing software can be classified into bug fixes in the software and modified behavior. Though both of these processes produce new problems in the code if not done properly, the latter one is more prone to this effect, as care should be taken to leave the functionality of the rest of the code unchanged. The first thing to do when modifying existing code is to classify the purpose of modification into the modification process for 1) bug fixing or for 2) better performance. This section presents the process of modeling existing MODSIM software for modification.

3.1 MODSIM III Constructs and Flow Diagram Constructs

For both modification processes, we need to understand the program flow within the software code. An appropriate way to represent the program flow is by means of flow diagrams. The complexity of the flow diagrams depends on the complexity of the programming language in which the software is developed. Therefore, it is always a good idea to understand the capabilities of the host language and adopt a notation that reflects its capabilities as closely as possible. This helps to translate code into flow diagrams and flow diagrams into code with consistency. PORTSIM, which will be used to demonstrate the modification process, was developed using MODSIM III. The following section discusses the MODSIM III constructs that need to be represented properly to reflect the functionality of the code.

3.1.1 MODSIM III constructs

MODSIM III supports process-oriented simulation instead of event-oriented simulation [5]. In case of an event-oriented simulation, passage of time is handled by scheduling the next event for the object currently being processed. This type of timing model is well suited to small models, but in the case of large models, where there is a lot of unrelated event routines in-between, following a single object's behavior is more difficult. MODSIM III simplifies larger models by allowing many aspects of an object's behavior in a model to be described in one method that allows for the passage of time at one or more points in its code. To support this concept, MODSIM utilizes several constructs to handle timing issues between concurrent processes, which are not present in all languages. There are also libraries to support capabilities like a Graphical User Interface (GUI) and methods like **NEW** and **DISPOSE** for creating and destroying objects for which it is undesirable to model the internal components in detail. The following section discusses the constructs that MODSIM III contains to build simulation models.

3.1.1.1 MODSIM III constructs from the programmer's perspective

DEFINITIONS:

Activity: What occurs in the model as time elapses. No code is executed

Event: A point in time at which the state of the model changes in some way. Code is executed, but time does not elapse.

In MODSIM III, a programmer has complete control of the simulation process with the help of ASK, TELL and WAITFOR methods [5]. A call to an ASK method is similar to that of a procedure call, i.e., when an ASK method is invoked, the calling code waits until the method is executed completely. Simulation time is not allowed to elapse in an ASK method. So an ASK method symbolizes the occurrence of an **event**. A call to a TELL method is similar to that of an asynchronous call, i.e., when a TELL method is invoked, the calling code does not wait for the completion of the method. Simulation time is allowed to elapse in a TELL method using a WAIT statement, so a Wait statement symbolizes the occurrence of an **activity**. A TELL method can also be scheduled to execute at some time in the future. A call to a WAITFOR method is similar to that of a procedure call with the ability to elapse simulation time, i.e., WAITFOR method has both the required properties of TELL and ASK methods. One important use of TELL and WAITFOR methods is that a programmer can achieve arbitrary synchronization by the use of TRIGGER objects within them. Triggers are used along with WAIT statements. TRIGGERS can be used when some processes need to wait until a specific condition occurs. A TRIGGER object can have any number of processes waiting for it. The processes will continue to wait till the trigger object's TRIGGER method is invoked by some other process.

Another important programming construct that needs to be represented carefully is the selection construct. This construct is important because the program control may follow any of the associated branches, depending on the result of the selection condition at a given time. So if an activity, event or logical process occurs as a branch in the selection construct, the program control may or may not go through that branch depending on the result of the selection condition. Hence the branches could be alternate activities, alternate events or alternate logical processes respectively.

3.1.1.2 MODSIM III constructs from the GUI developer's perspective

SIMGRAPHICS II is the MODSIM III'S graphics package [6]. Since MODSIM III is an Object Oriented language, the graphical interface is also implemented using objects. Many basic objects have already been provided. The most important ones are shown in TABLE 1 below [6].

TABLE 1

Basic objects and their properties

OBJECT NAME	PROPERTIES
WindowObj	Standard system window which acts as a container for all graphical objects.
ImageObj	Basic object used for static icons and backgrounds.
DynImageObj	Basic graphic object used for animation.
DialogBoxObj	Receives various types of input from the user. Controls, such as buttons, check boxes, list boxes etc. can be part of it.
PaletteObj	Receives input from two-state palette buttons.
MenuBarObj	Receives simple menu selections.

An instance of any of these objects can be made visible by calling their respective DRAW methods and can be erased by calling their respective ERASE methods. In SIMGRAPHICS II, a developer can have complete control on the behavior of graphical objects with the help of the BeSelected, AcceptInput and BeClosed methods. The BeSelected method is automatically invoked when a graphical object is clicked on. This method can be overridden to receive asynchronous input. Within a BeSelected method of a graphical object, the LastPicked field can be checked to determine which control was last clicked on. Depending on the control, a programmer can model the behavior of the graphical object. AcceptInput method is used to retrieve synchronous input from a graphical object. AcceptInput returns the contents of the LastPicked field after the selection has been made. However, if the graphical instance is erased or disposed, Acceptinput returns NILOBJ [6]. BeClosed method is automatically invoked whenever a user chooses to close a graphical object. This method can be overridden to change its default behavior associated with a graphical object.

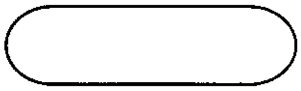
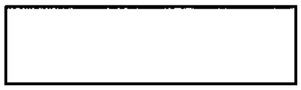
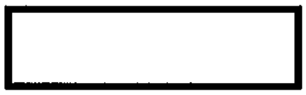
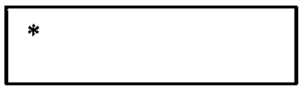
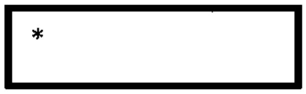
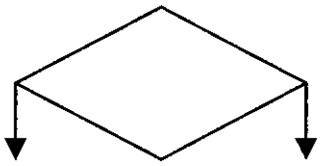
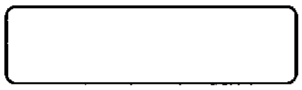
Of all the three methods, the behavior of the Acceptinput method cannot be changed. This very fact makes the representation of data flow difficult if a dialog box uses both Acceptinput and BeSelected methods in its life cycle. We cannot tell whether the selected input is returned to the AcceptInput method first or to the BeSelected method. No matter which of the two methods receive the selected input first, by the time the statement after a call to AcceptInput method is executed we can say that both the BeSelected and AcceptInput methods are returned. So the point after a call to AcceptInput method can be represented as the point of return for both AcceptInput and BeSelected methods. So in any situations where the actual implementation is hidden in the library routines, we should look for the first point at which we are absolutely sure about the cumulative result, i.e., we need to concentrate on final result rather than the actual sequence of data flow. These issues complicate the modeling process.

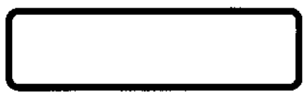
3.1.1.3 MODSIM III constructs from the designer's perspective

From the designer's point of view, a call to any type of method is a way of passing messages from one object to another object or to itself. Objects respond to messages by executing the associated code, which may in turn send messages. In some cases this message sending will be so nested that it is a good idea to subdivide the code associated with a message into logical processes where each logical process represents a set of closely related messages. Closely related messages are messages that are used together to accomplish a single target. In fact, any involved message can also be treated like a logical process, so any system can be represented as sequence of logical processes that are to be completed to get the results. The following section shows the notations for various MODSIM III constructs.

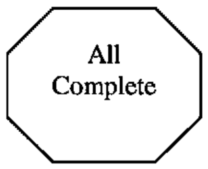
3.1.2 Flow Diagram constructs for MODSIM III constructs

This section shows the flowchart constructs that can be used to represent program flow within the software developed in MODSIM III.

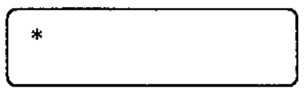
Flowchart Construct	**Purpose**
	Begin/End of logical process.
	Represents an event within a logical process.
	Represents an activity within a logical process.
	Represents a logical process within a logical process.
	Represents a logical process containing an activity.
	Represents a condition within a logical process.
	Represents an alternate event as a result of condition or as a result of selection.



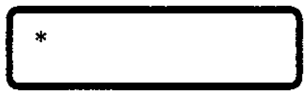
Represents an alternate activity as a result of condition or as a result of selection.



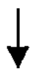
Represents the point of return of multiple methods irrespective of order of completion of the methods.



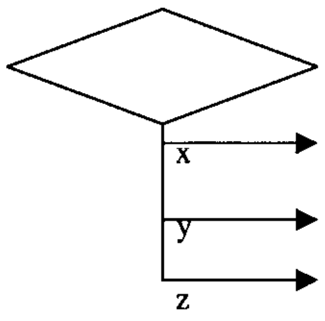
Represents an alternate logical process within a logical process.



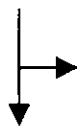
Represents an alternate logical process containing an activity.



Represents the direction of flow of program control.



Represents that the program control will follow any of the directed paths depending on the outcome of the selection statement.



Represents an asynchronous call and hence the program control continues to flow down without waiting for the completion of process started by the branched arrow.

Flowchart Construct	**Purpose**
⊥	Represents a terminator.

3.1.3 Flow Diagram Examples for MODSIM III constructs

This section shows how the above notations can be used to represent different constructs in MODSIM III.

```
....................
ASK watch TO setTime(IST);
....................
```

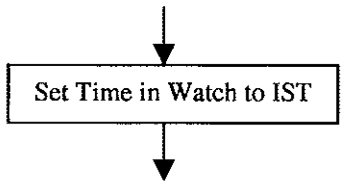


Fig. 5 Example 1: An Event

```
....................
WAIT DURATION 30
END WAIT;
....................
```

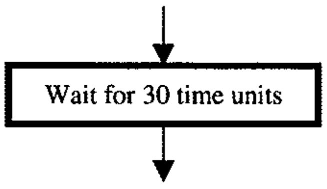


Fig. 6 Example 2: An Activity

```
....................
IF time < 10
  ASK SELF TO setTime(0);
ELSE
  ASK SELF TO setTime(12);
END IF:
....................
```

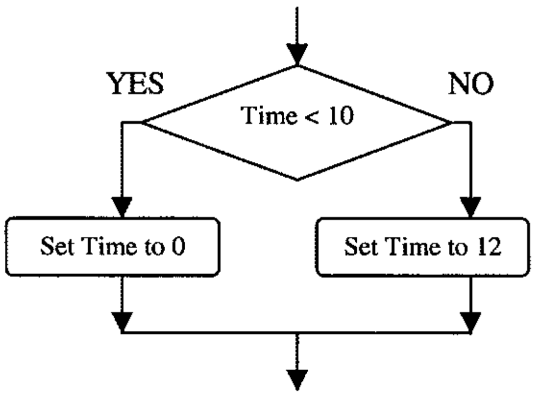


Fig. 7 Example 3: A Condition

```
.....................
TELL Flight20 TO goto(Paris);
.....................
```

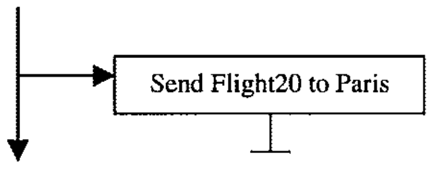


Fig. 8 Example 4: A Asynchronous call

```
.....................
IF month >=1 AND month < 4
 ASK SELF TO setQuarter(1);
ElSIF month >= 4 AND month < 7
 ASK SELF TO setQuarter(2);
ELSIF month >=7 AND month < 10
 ASK SELF TO setQuarter(3);
ELSE
 ASK SELF TO setQuarter(4);
END IF;
.....................
```

OR

```
.....................
CASE month
 WHEN 1..3:
  ASK SELF TO setQuarter(1);
 WHEN 4..6:
  ASK SELF TO setQuarter(2);
 WHEN 7..9:
  ASK SELF TO setQuarter(3);
 OTHERWISE
  ASK SELF TO setQuarter(4);
END CASE;
.....................
```

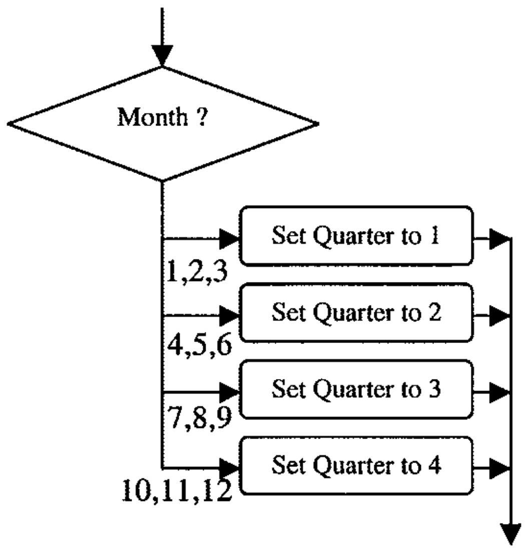


Fig. 9 Example 5: A Selection

```
....................
WHILE (time < 10*60)
 WAIT DURATION 30
 END WAIT;
 time := ASK watch TO getTime();
END WHILE;
```

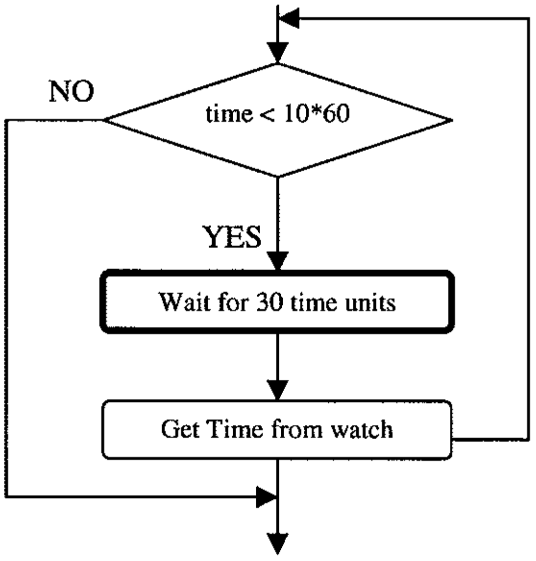


Fig. 10 Example 6: Loop with an activity and an event

```
DEFINITION MODULE x;
.........
SomeDialogBox = OBJECT(DialogBoxObj);
 .................
 OVERRIDE
  ASK METHOD BeSelected;
END OBJECT;
..........
END MODULE.

IMPLEMENTATION MODULE x;
........
OBJECT SomeDialogBox;
 ..............
  REPEAT
   Item := ASK SELF TO AcceptInput();
  UNTIL (ASK item ReferenceName = 'cancel');
 ..................
END OBJECT;
...........
END MODULE.
```

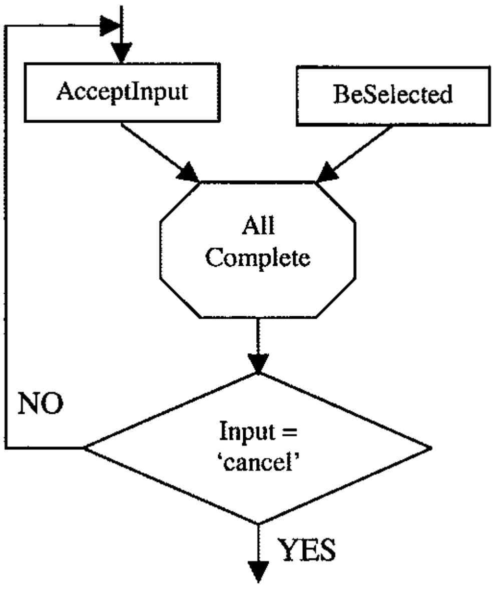


Fig. 11 Example 7: Representation of code where order of data flow is hidden in library routines

3.2 Developing Flow Diagrams

This section describes the method of partitioning the code and representing it as a flow diagram by using the code associated with a method from PORTSIM as an in depth example. Two other examples can be found in Section 4.

3.2.1 Method

The following steps describe the way to develop flow diagrams from MODSIM III code.

1) Define the functionality of the method to be decoded by its object type, name, in-put parameters, out put parameters and by the context of the code from which the method is being called.
2) Go through the code and mark off the method calls;
3) Mark off the activities, events and the asynchronous calls from method calls.
4) Identify sets of method calls where available that are used to perform a single task and name that set of methods depending on the task they are performing.
5) Identify any method calls that need to be further expanded (involved messages).
6) Construct the flow diagrams using the notations developed previously.

3.2.2 PORTSIM Example

This following section shows the result of applying the above mentioned procedure of converting code into flow diagrams to the method called "callForwardFlatcars" in PORTSIM'S code.

Step 1

The characteristics of the method and their values are given in TABLE 2

TABLE 2

Characteristics of "CallForwardFlatcars" method

CHARACTERISTIC OF METHOD	VALUE
Object Type	Spur
Name of Method	CallForwardFlatcars
Input Parameters	Interchange Yard Object & Random Object
Output Parameters	None
Context	The method is being called by an instance of port object while initializing spurs.

Functionality: This method is used for getting flatcars from the interchange yard to the instance of a spur object to which this message is passed.

Steps 2 through 5

Fig. 12 shows the result of applying the steps 2 through 5 on the code shown in Fig. C.1. The following comments are made for Fig. 12 :

1) The statements in boxes represents synchronous method calls out of which statements containing "Wait" represent activities and remaining statements represent events.
2) Asynchronous calls are represented by bold text.
3) Sets of method calls performing a single task are marked along with their names.
4) Statements in boxes that are followed by "expand" represents method calls that can be treated as logical processes.

```
TELL METHOD callForwardFlatcars(IN interchangeYard :interchangeYardObj;IN ranGen :RandomObj);
  BEGIN
    WHILE (SimTime() <= (simBeginTime + timeToSimulate))
     IF (ordering = "Vehicles/Unit Equipment Only")
      In IY Count Flatcars with Vehicles as their First Cargo
       WHILE (numFlatcarsCarryingVehicles = 0)
         Wait for Train Classified Trigger; In IY Count Flatcars with Vehicles as their First Cargo
       END WHILE;
     ELSIF (ordering = "Containers Only")
      In IY Count Flatcars with Container as their First Cargo
      WHILE (numFlatcarsCarryingContainers = 0)
         Wait for Train Classified Trigger; In IY Count Flatcars with Container as their First Cargo
      END WHILE;
     END IF;
     IF (status = "idle")
      Change Status to Busy
      WHILE (stringOfFlatcars.numberIn = 0)  ------>        Start "Flatcars Processing"
       Build List of Flatcars at IY (expand)
       IF (stringOfFlatcars.numberIn = 0)
         Wait for 30 Minutes
       END IF;
      END WHILE;
      TELL IYTrigger TO Trigger;
      Wait for Locomotive; Wait for Locomotive arrival at IY; Wait for Coupling Flatcars at IY
      IF (type = "OPEN STAGING")
         Wait for Flatcars Switch to Spur
      ELSIF (type = "COVERED STAGING")
         Wait for Flatcars Switch to Dock
      ELSE
         Wait for Flatcars Switch to Berth
      END IF;
      Wait to Uncouple Flatcars;Return Locomotive;Wait to Discharge all Flatcar contents(expand)
      Change Status to Idle                   ----> End "Flatcars Processing"
      TELL spurDoneTrigger TO Trigger;
     ELSE
         Wait for Spur Done Trigger
     END IF;
    END WHILE;
END METHOD {callForwardFlatcars};
```

Fig. 12 Result of applying steps 2 through 5 to "callForwardFlatcars" method

Step 6

In this step, Fig. 12 is converted into flow diagrams using the previously adopted notations. Fig. 13 represents the program flow within the “callForwardFlatcars” logical process, with the identified logical processes in it marked properly. Fig. 14 represents the program flow within the “Flatcar Processing” logical process, with the identified logical processes in it marked properly. Every logical process in Fig. 13 and in Fig. 14 is followed by a figure number, which represents the program flow within that particular logical process.

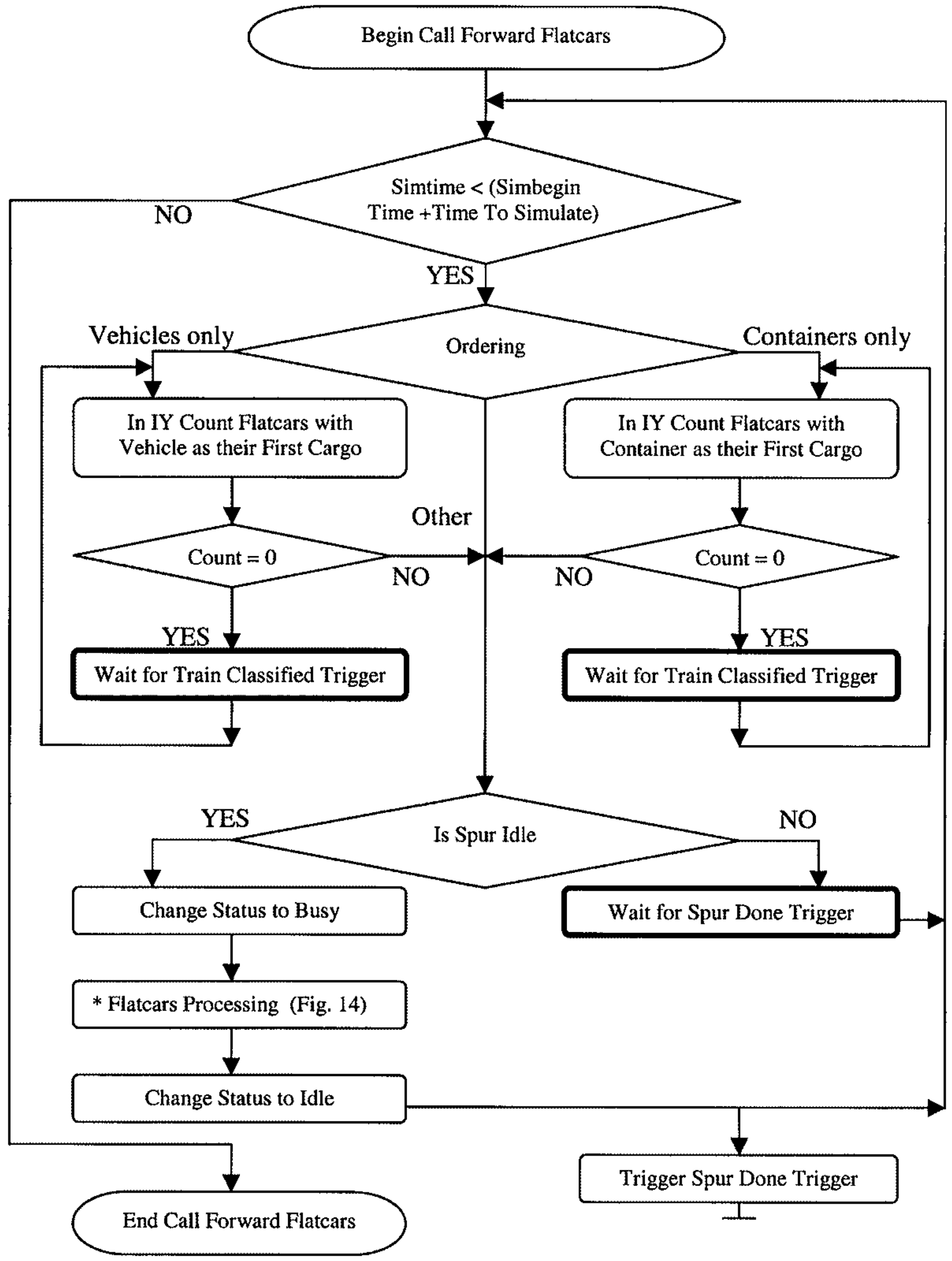

Begin Call Forward Flatcars
Simtime < (Simbegin Time +Time To Simulate)
NO
YES
Vehicles only
Ordering
Containers only
In IY Count Flatcars with Vehicle as their First Cargo
In IY Count Flatcars with Container as their First Cargo
Other
Count = 0
NO
NO
Count = 0
YES
YES
Wait for Train Classified Trigger
Wait for Train Classified Trigger
YES
Is Spur Idle
NO
Change Status to Busy
Wait for Spur Done Trigger
* Flatcars Processing (Fig. 14)
Change Status to Idle
Trigger Spur Done Trigger
End Call Forward Flatcars

Fig. 13 Program flow within "callForwardFlatcars" method

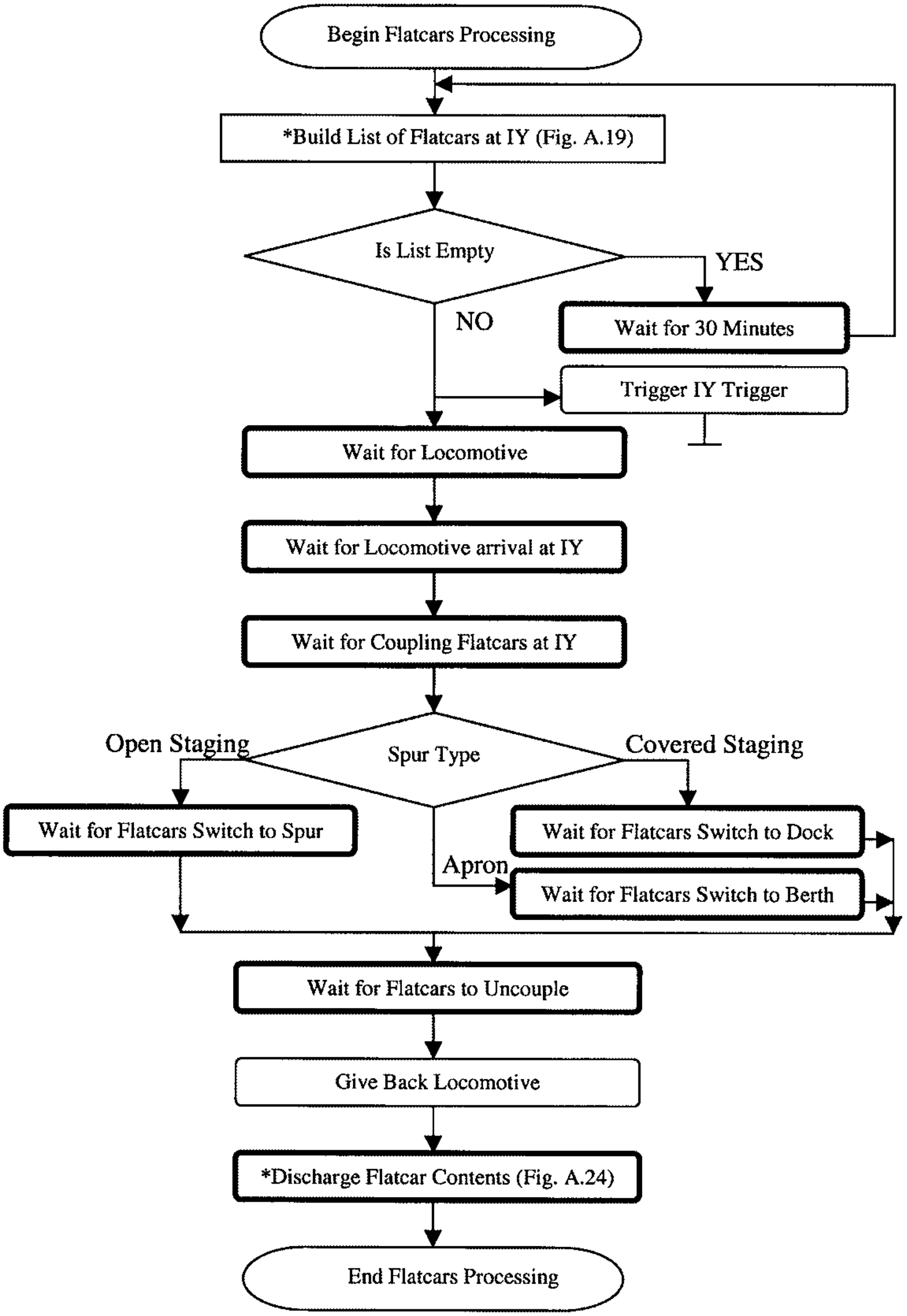

Begin Flatcars Processing
*Build List of Flatcars at IY (Fig. A.19)
Is List Empty
YES
NO
Wait for 30 Minutes
Trigger IY Trigger
Wait for Locomotive
Wait for Locomotive arrival at IY
Wait for Coupling Flatcars at IY
Open Staging
Spur Type
Covered Staging
Wait for Flatcars Switch to Spur
Wait for Flatcars Switch to Dock
Apron
Wait for Flatcars Switch to Berth
Wait for Flatcars to Uncouple
Give Back Locomotive
*Discharge Flatcar Contents (Fig. A.24)
End Flatcars Processing

Fig. 14 Program flow within "Flatcars Processing"

3.3 Modifying Code using Flow Diagrams

This section shows how flow diagrams developed in the above mentioned fashion can be used to modify the code to fix a known bug.

3.3.1 Method for modifying code using Flow Diagrams

The following steps describe the way to use flow diagrams to fix bugs in MODSIM III code.

1) Define and analyze the problem.
2) Identify the set of flow diagrams representing the code that may contain the bug.
3) Go through the flow diagrams keeping in mind the type of problem and identify the problem causing segments within the flow diagrams.
4) Define the condition that needs to be satisfied to prevent the problem.
5) Change the flow diagrams so that the problem is rectified. The change may span more than one flow diagram as we have divided the system into logical processes.
6) Run through the flow diagrams again and make sure that the original functionality is not disturbed because of the changes made in the flow diagrams. Go to step 5 if the original functionality is changed.
7) Now convert the modified flow diagrams into MODSIM III constructs using the developed notations.
8) Execute the code with modifications and check if the problem is solved. If the problem persists, go to step 3.

The following section shows how the above mentioned procedure is applied in solving a problem.

3.3.2 PORTSIM Example: Call Forward Flatcars Problem

Step 1

Problem definition: Only the first available interchange yard is being serviced, and all other available interchange yards are not serviced even though they have flatcars in them and spurs are available for use.

From the problem definition, we may surmise that the first serviced interchange yard is not allowing other interchange yards to get service or is forcing the code into an infinite loop.

Step 2

The flow diagrams involving interchange yards are shown in Fig. A.14 and Figures A.16 through A.21.

Step 3

1) From step 1, we decide that we need to concentrate on loops in the flow diagrams.
2) From Fig. A.14, we can see that each available interchange yard is trying to get served by a spur. But in order to understand the inner loop completely, we need to understand A.16.
3) From Fig. A.16, we can see that each available spur is calling flatcars from the interchange yard. Fig. A.16 points us to Fig. A.17, which expands the "call Forward Flatcars" logical process.

4) From Fig. A.17, we can see that:

   - If the ordering is either "vehicles only" or "containers only," the program control loops until there are flatcars in the interchange yard. However, we cannot say that this flow diagram segment is the one that is causing the problem because we need to solve the problem for interchange yards that have flatcars in them.
   - If the spur is idle, then its status is changed to "busy" before executing the "flatcars processing" logical process and is changed back to "idle" after "flatcars processing" logical process. The interchange yard then sends "spur done" signal.
   - If the spur is not idle, then the interchange yard waits until it receives "spur done" signal.

5) From Fig. A.18, which expands the "flatcars processing" logical process, we can see that:

   - If there are no more flatcars left within the interchange yard, the program control loops until more flatcars are available in the interchange yard. We can say that this segment of the flow diagram is the one that is causing the problem. By the time the program control reaches the loop, the status of the spur is already marked as "busy," making it unavailable to other interchange yards. ***There is no way of getting out of this loop after all the flatcars in the interchange yard are processed.***

Step 4

The condition that needs to be satisfied to solve the problem is that an interchange yard should make the spur, to which it is sending flatcars, available to other waiting interchange yards after all of the flatcars in it are processed.

Step 5

We can remove the problem causing loop from the "Flatcars processing" logical process and add it before the spur's status check in the "call Forward Flatcars" logical process. This makes sure that the spur's status is marked “busy” only when there are flatcars in the interchange yard. Figures 15 and 16 show the modified parts of the flow diagrams shown in Figures 13 and 14.

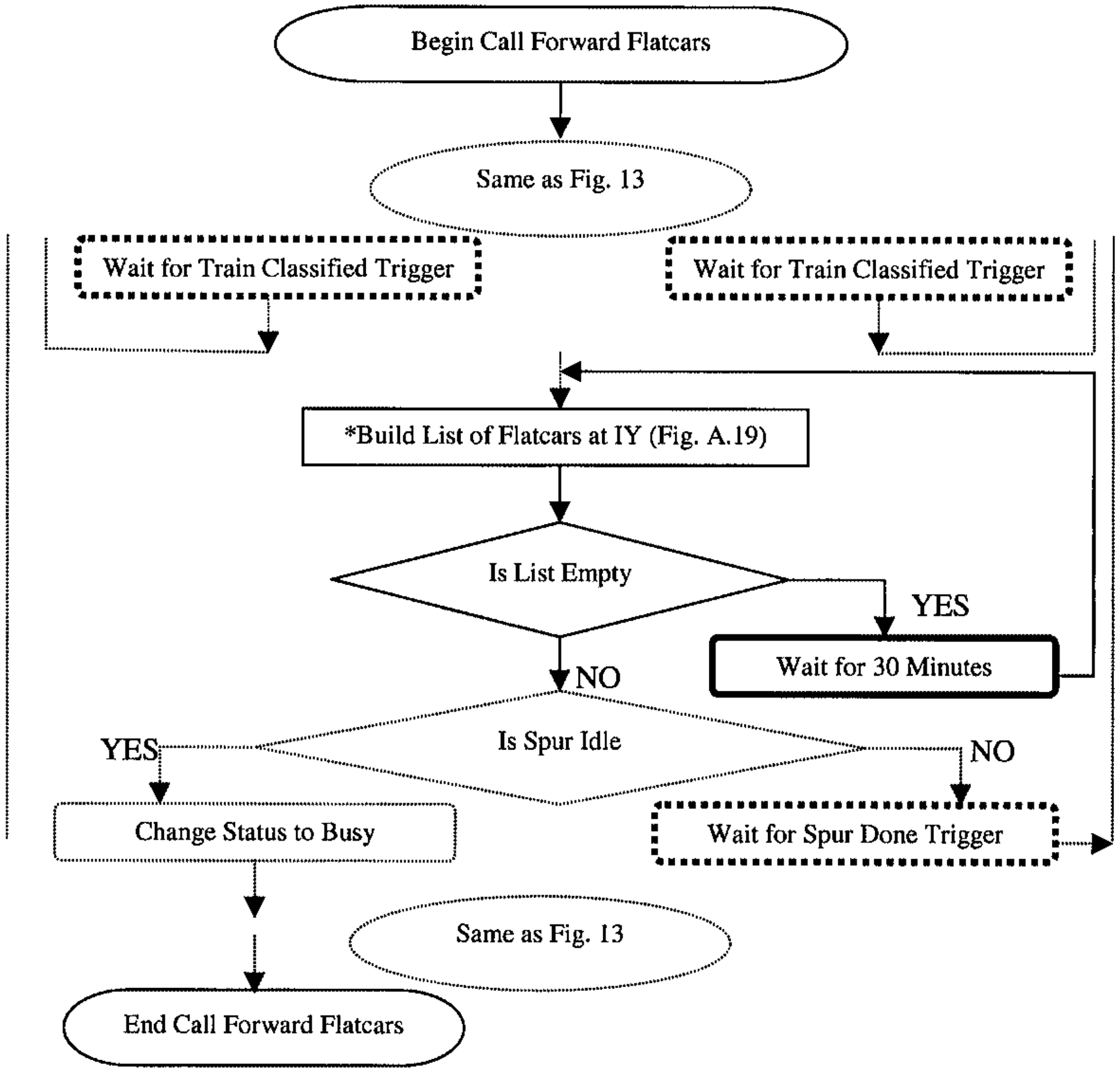

Begin Call Forward Flatcars
Same as Fig. 13
Wait for Train Classified Trigger
Wait for Train Classified Trigger
*Build List of Flatcars at IY (Fig. A.19)
Is List Empty
YES
Wait for 30 Minutes
NO
Is Spur Idle
YES
NO
Change Status to Busy
Wait for Spur Done Trigger
Same as Fig. 13
End Call Forward Flatcars

Fig. 15 Modified representation of Fig. 13

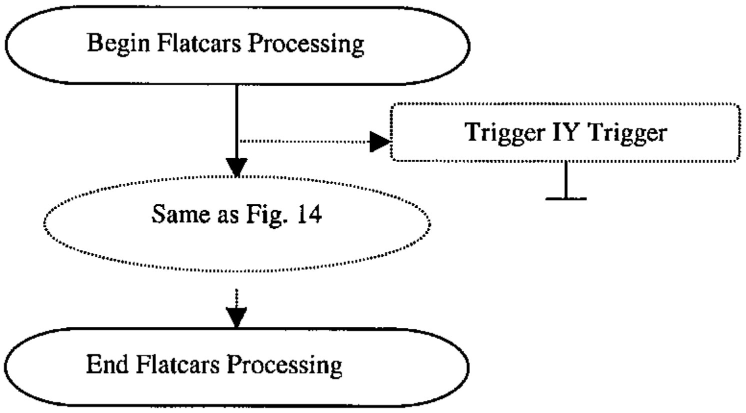


Fig. 16 Modified representation of Fig. 14

Step 6

The model developed above satisfies the condition mentioned in step 4 by not allowing any interchange yard to monopolize the use of the spurs. However, to understand whether the above model leaves the original functionality of the "call Forward Flatcars" logical process unchanged, it is necessary to understand the effect of shifting the position of "Build List of Flatcars at IY" logical process along with the loop. From Fig. A.19, which expands "Build List of Flatcars at IY" logical process, we can see that it is necessary to understand the "Adding Flatcars with Vehicles" logical process. From figure A.20, which expands the "Adding Flatcars with Vehicles" logical process, we can see that

1) While building the list of flatcars, flatcars are removed from the interchange yard, and the capacity of the interchange yard is incremented by their length.

If after building a list of flatcars the spur is found to be busy, the interchange yard will wait for a spur done signal. After receiving the signal it again loops back and builds another list of flatcars, so the flatcars in the previous list were destroyed without being processed.

Step 5 Revisited

We can change the condition for marking the spur's status as "Busy," without shifting the loop from the "Flatcars Processing" logical process to the "call Forward Flatcars" logical process, in such a way that the spur is marked busy only when there are flatcars in an interchange yard. This can be done by marking the status of the spur as "Busy" after the problem causing loop in the "Flatcars processing" logical process instead of in the "call Forward Flatcars" logical process. Now, if the program control loops indefinitely when there are no flatcars left to process in the interchange yard, the status of the spur will remain as "Idle" making it available for other interchange yards. Also, we need to check the status of the spur in the loop before building a list of flatcars from an interchange yard to make sure that the spur is still available after waiting for thirty minutes for more flatcars in the interchange yard. Figures 17 and 18 show the modified parts of the flow diagrams shown in Figures 13 and 14.

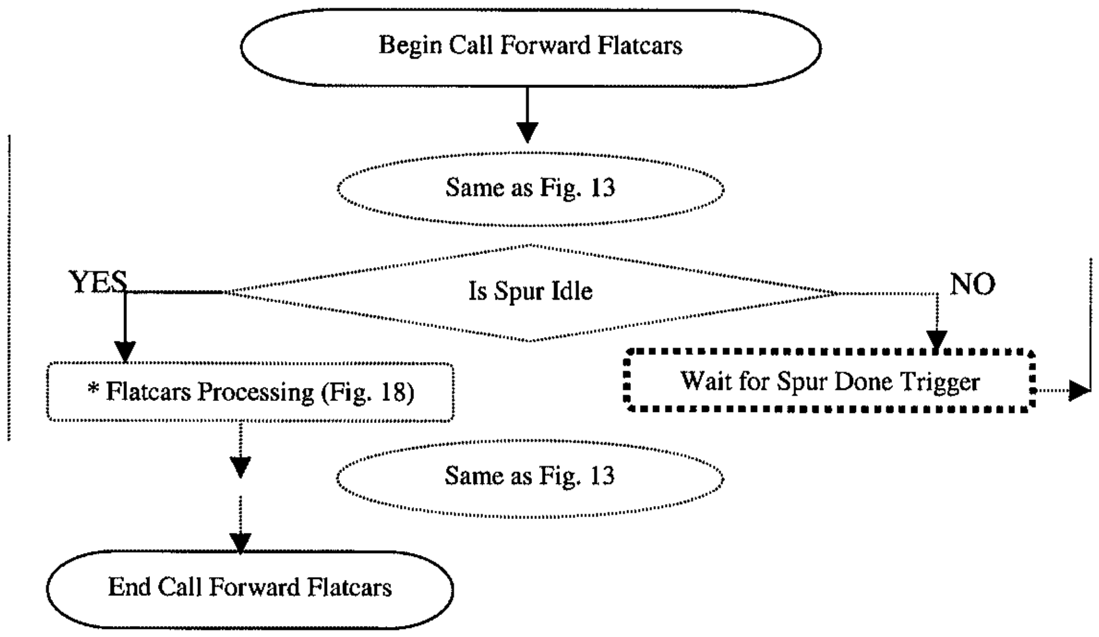

Begin Call Forward Flatcars
Same as Fig. 13
Is Spur Idle
YES
NO
* Flatcars Processing (Fig. 18)
Wait for Spur Done Trigger
Same as Fig. 13
End Call Forward Flatcars

Fig. 17 Revised representation of Fig. 13

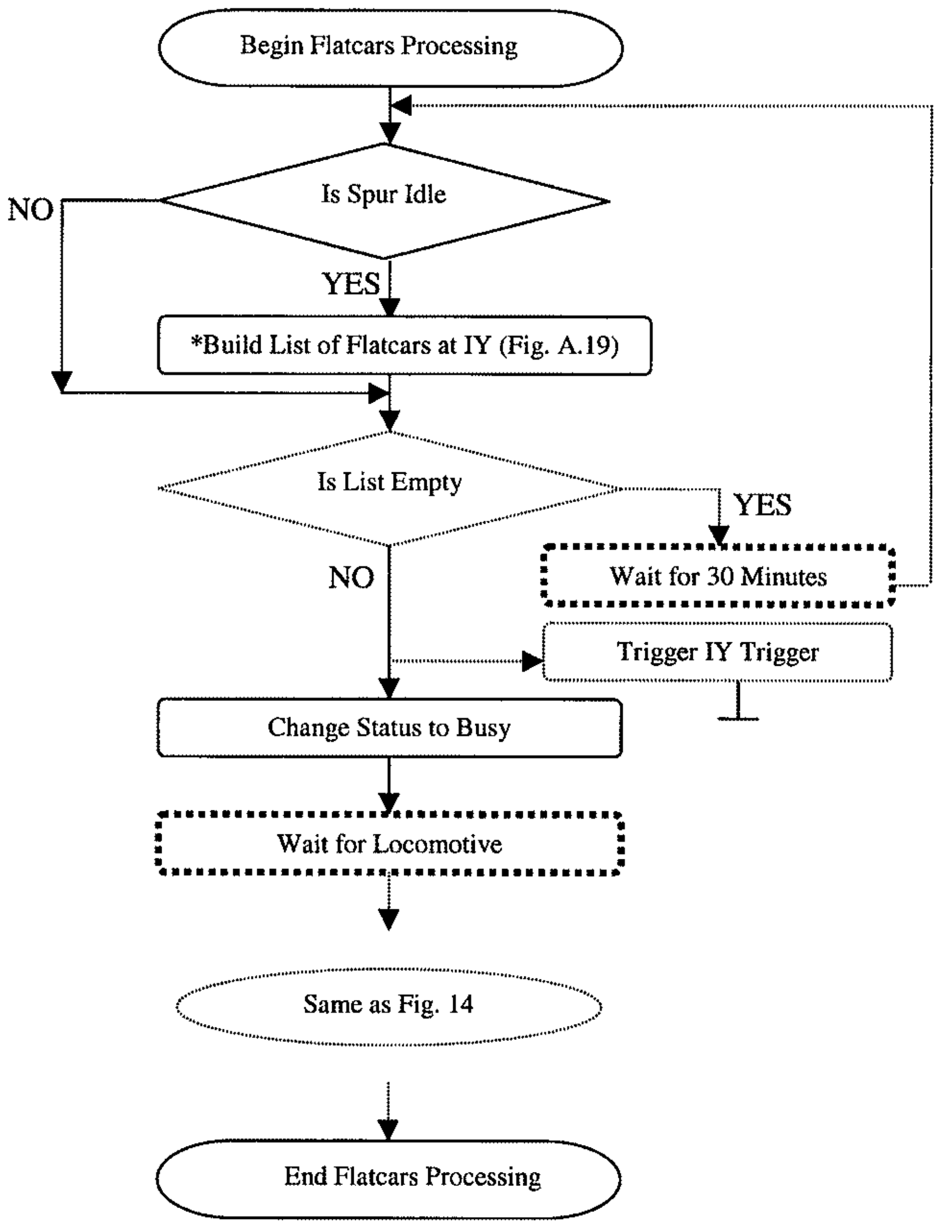


Fig. 18 Revised representation of Fig. 14

Step 6 revisited

The alternate model developed above functions exactly like the original model when there are flatcars in an interchange yard to process, and it also satisfies the condition mentioned in step 4.

Step 7

Now we need to convert the flow diagrams shown in Figures 17 and 18 into MODSIM III constructs using the notations developed. Fig. 19 shows the MODSIM III constructs where the changes are indicated by bold italic text. The modified code is shown in Fig. C.2.

Step 8

The modified code was executed and found that the available spurs are servicing all interchange yards with available flatcars.

```
TELL METHOD callForwardFlatcars(IN interchangeYard :interchangeYardObj;IN ranGen :RandomObj);
   BEGIN
      WHILE (SimTime() <= (simBeginTime + timeToSimulate))
       IF (ordering = "Vehicles/Unit Equipment Only")
         In IY Count Flatcars with Vehicles as their First Cargo
         WHILE (numFlatcarsCarryingVehicles = 0)
            Wait for Train Classified Trigger; In IY Count Flatcars with Vehicles as their First Cargo
         END WHILE;
       ELSIF (ordering = "Containers Only")
         In IY Count Flatcars with Container as their First Cargo
         WHILE (numFlatcarsCarryingContainers = 0)
            Wait for Train Classified Trigger; In IY Count Flatcars with Container as their First Cargo
         END WHILE;
       END IF;
       IF (status = "idle")  ────────▶ Removed the call to spur status change in the IF construct
         WHILE (stringOfFlatcars.numberIn = 0)  ────────▶  Start "Flatcars Processing"
          IF (status = "idle")
            Build List of Flatcars at IY (expand)
          END IF;
          IF (stringOfFlatcars.numberIn = 0)
            Wait for 30 Minutes
          END IF;
         END WHILE;
         TELL IYTrigger TO Trigger;
         Change Status to Busy
         Wait for Locomotive; Wait for Locomotive arrival at IY; Wait for Coupling Flatcars at IY
         IF (type = "OPEN STAGING")
            Wait for Flatcars Switch to Spur
         ELSIF (type = "COVERED STAGING")
            Wait for Flatcars Switch to Dock
         ELSE
            Wait for Flatcars Switch to Berth
         END IF;
         Wait to Uncouple Flatcars;Return Locomotive;Wait to Discharge all Flatcar contents(expand)
         Change Status to Idle                      └──▶ End "Flatcars Processing"
       TELL spurDoneTrigger TO Trigger;
       ELSE
            Wait for Spur Done Trigger
       END IF;
      END WHILE; END METHOD {callForwardFlatcars};
```

Fig. 19 MODSIM III constructs showing the changes made in Figures 13 & 14

SECTION 4

DEMONSTRATION OF USING FLOW DIAGRAMS FOR CODE MODIFICATION IN PORTSIM

This section demonstrates the use of the model developed in the previous section to fix bugs and to modify the code using flowcharts. The "Ship Exiting Problem" as described below demonstrates a second use of the model in fixing bugs similar to the "Call Forward Flatcars" problem detailed in Section 3. The "Dialog Boxes Closing Problem," as described later, demonstrates the use of the model in making improvements in the software.

4.1 Ship Exiting Problem

Step 1

Problem definition: Ships are not leaving a port if they have remaining cargo space and all cargo has been loaded onto a ship.

From the problem definition, we may surmise that this problem is due to either an infinite loop or an inappropriate condition that allows a ship to leave the port.

Step 2

The flow diagrams involving ship life cycle are shown in Figures A.46 through A.78.

Step 3

1) From step 1, we decide that we need to concentrate on loops and conditions in the flow diagrams.

2) From Fig. A.46 we can see that

    - The time the last cargo was loaded onto the ship is set to the present simulation time.
    - A ship in the arrival list is then sent to dock at a berth.

    In order to understand the docking process, we need to understand Fig. A.47.

3) From Fig. A.47 we can see that

    - If none of the berths are available for military use then the ship is added to ship queue.
    - If any of the berths are available, berthing is done depending on the ship type.

    In order to understand the berthing process for various types of ships, we need to understand Figures A.48 through A.51.

4) From Figures A.48 to A.51, we can see that to understand these figures we need to understand Fig. A.52.

5) From Fig. A.52, we can see that if a berth is available and suitable for the ship, then

    - The berth is marked busy.
    - The ship is served or loaded at that berth.

    In order to understand the process of serving a ship, we need to understand Fig. A.53.

6) From Fig. A.53, we can see that

    - If space on the ship is not available, the ship is released from berth.

- If space on the ship is available, then
    - Staging areas with maximum number of vehicles, containers and pallets ready to load are selected.
    - If none of the cargo items are available then the program waits for sixty minutes.
    - If any of the cargo items are available, those cargo items are call forwarded from their respective staging areas.
    - Cargo items are loaded onto the ship.
    - If any of the cargo items are left on berth, those cargo items are sent back to staging areas.
    - If the difference between present time and the time the last cargo item was loaded is less than the maximum wait time for cargo of that ship, the above process is repeated. Otherwise, the ship is released from the berth. This implies that a ship is released from berth if none of the cargo items are available within the maximum allowed wait time for cargo of the ship.

But in order to understand the processes of call forwarding cargo, loading cargo and ship release, we need to understand Figures A.54, A.66 and A.78, respectively.

7) From Fig. A.54, we can see that a call forward list is built depending on the ship type, so to understand Fig. A.54 we need to understand Figures A.55, A.56 and A.63.

8) From Figures A.55, A.56 and A.63 we can see that

   - If the cargo items and the port assets required to load them onto the ship are not available, the program again waits for sixty minutes and checks for the availability of cargo items and the port assets required by them. This loop continues till both the cargo items and the port assets required by them are available.

   We can say that this segment of the flow diagrams is the one that is causing the problem. If there are no more cargo items left to occupy the present staging area, from which the call forward list is being built, there is no way for the program control to get out of this loop. So, the ship cannot be released from the berth once the program control is caught in this loop.

Step 4

The condition that needs to be satisfied to solve the problem is that a ship should be released from a berth when all of the cargo items in a staging area are loaded onto the ship.

Step 5

We can remove the defective loop from the Figures A.55, A.56 and A.63 and convert it into a condition statement as shown in Figures 20, 21 and 22, respectively. This will allow a ship to be released from the berth when all the cargo items in a staging area are loaded onto it by making the "Call Forward Cargo" process behave as a WAIT statement, as shown in Fig. 23.

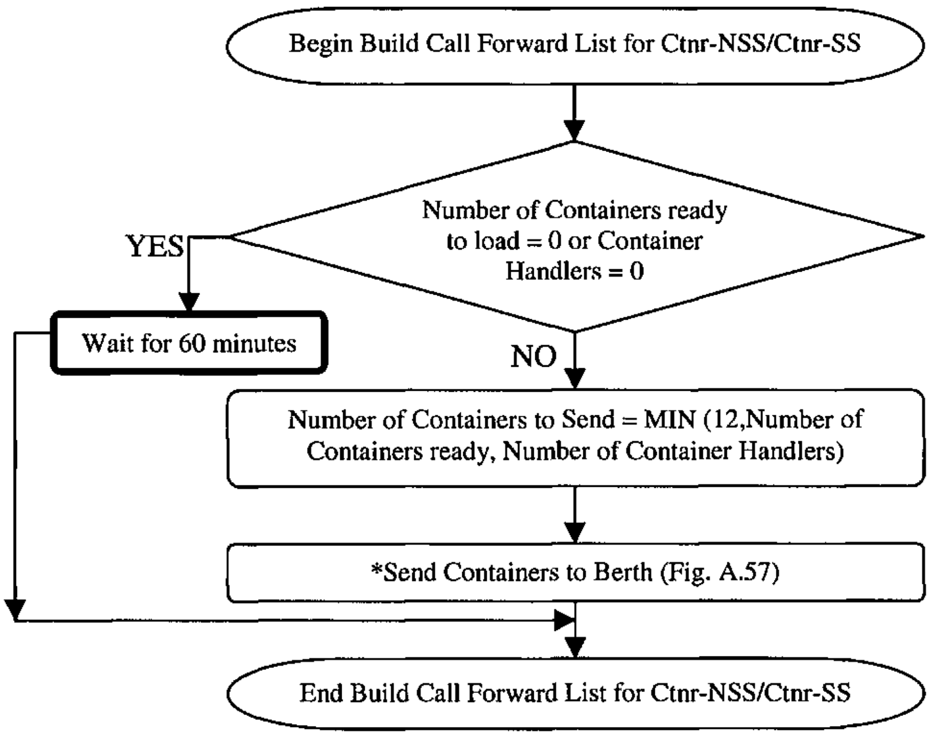

Begin Build Call Forward List for Ctnr-NSS/Ctnr-SS
Number of Containers ready to load = 0 or Container Handlers = 0
YES
Wait for 60 minutes
NO
Number of Containers to Send = MIN (12,Number of Containers ready, Number of Container Handlers)
*Send Containers to Berth (Fig. A.57)
End Build Call Forward List for Ctnr-NSS/Ctnr-SS

Fig. 20 Modified representation of Fig. A.55

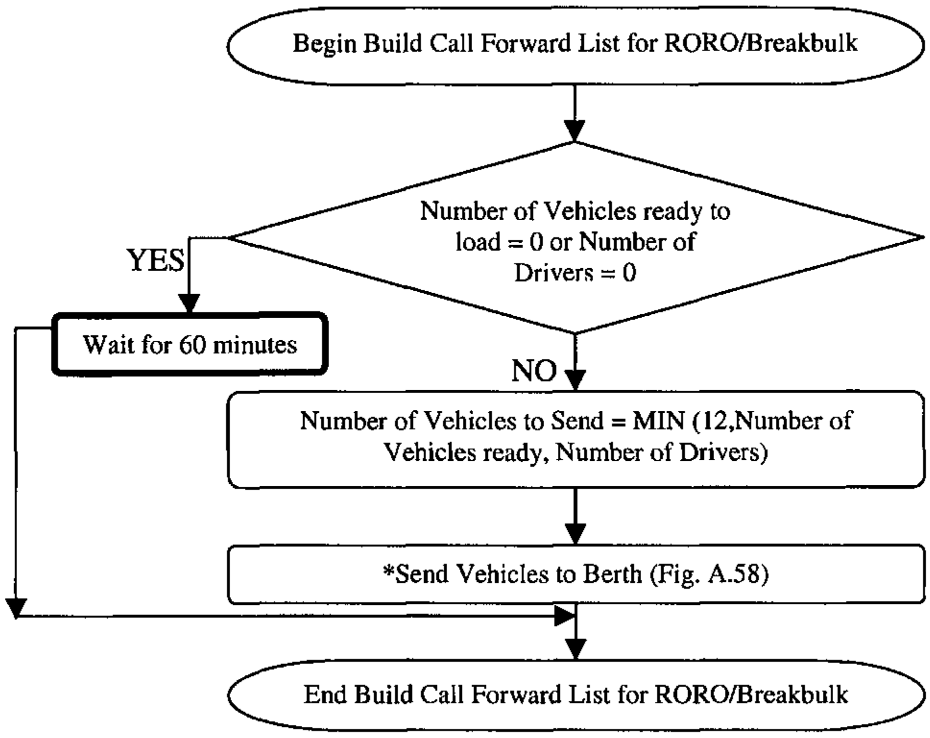

Begin Build Call Forward List for RORO/Breakbulk
Number of Vehicles ready to load = 0 or Number of Drivers = 0
YES
Wait for 60 minutes
NO
Number of Vehicles to Send = MIN (12,Number of Vehicles ready, Number of Drivers)
*Send Vehicles to Berth (Fig. A.58)
End Build Call Forward List for RORO/Breakbulk

Fig. 21 Modified representation of Fig. A.56

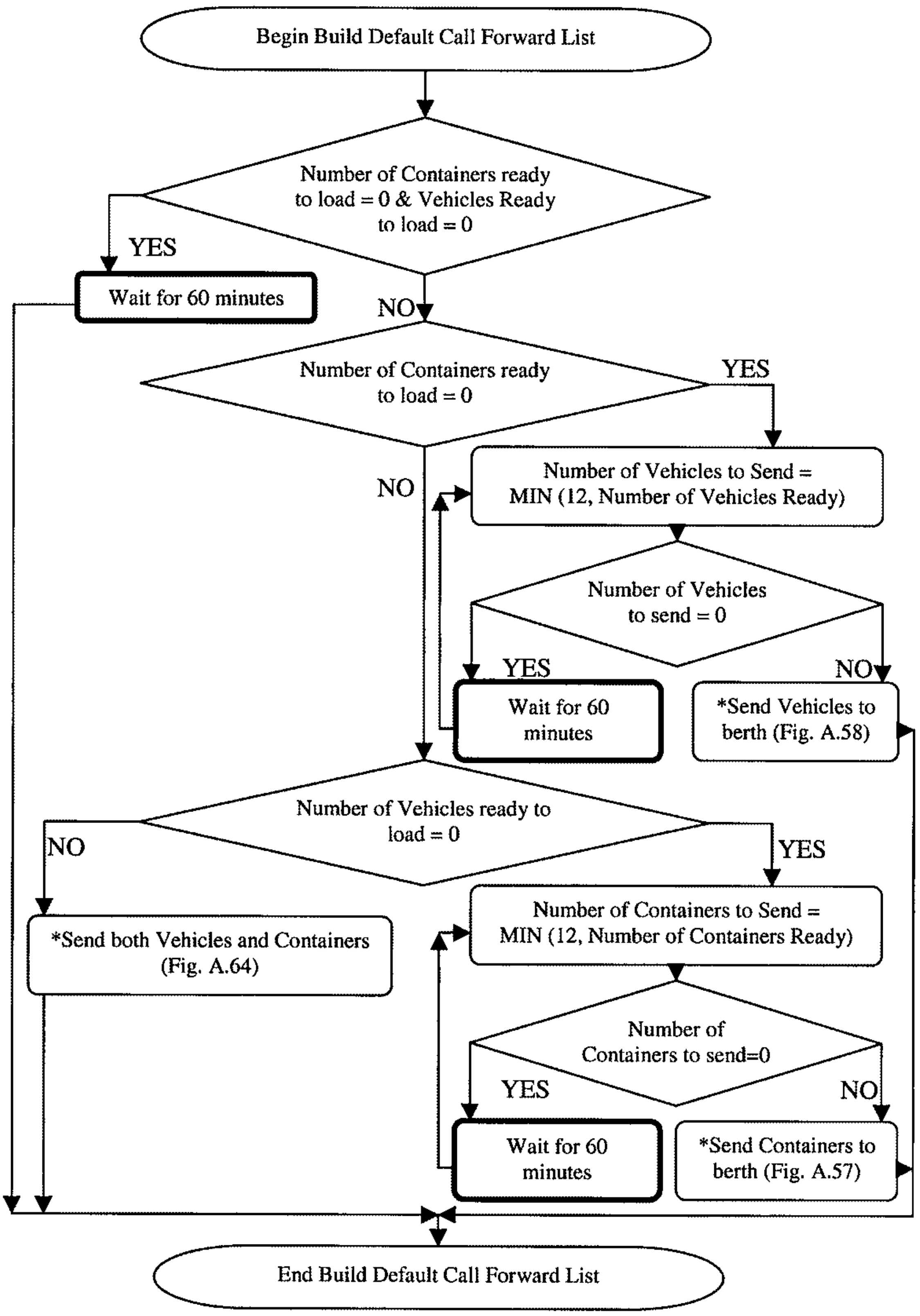

Begin Build Default Call Forward List
Number of Containers ready to load = 0 & Vehicles Ready to load = 0
YES
Wait for 60 minutes
NO
Number of Containers ready to load = 0
YES
NO
Number of Vehicles to Send = MIN (12, Number of Vehicles Ready)
Number of Vehicles to send = 0
YES
NO
Wait for 60 minutes
*Send Vehicles to berth (Fig. A.58)
Number of Vehicles ready to load = 0
NO
YES
*Send both Vehicles and Containers (Fig. A.64)
Number of Containers to Send = MIN (12, Number of Containers Ready)
Number of Containers to send=0
YES
NO
Wait for 60 minutes
*Send Containers to berth (Fig. A.57)
End Build Default Call Forward List

Fig. 22 Modified representation of Fig. A.63

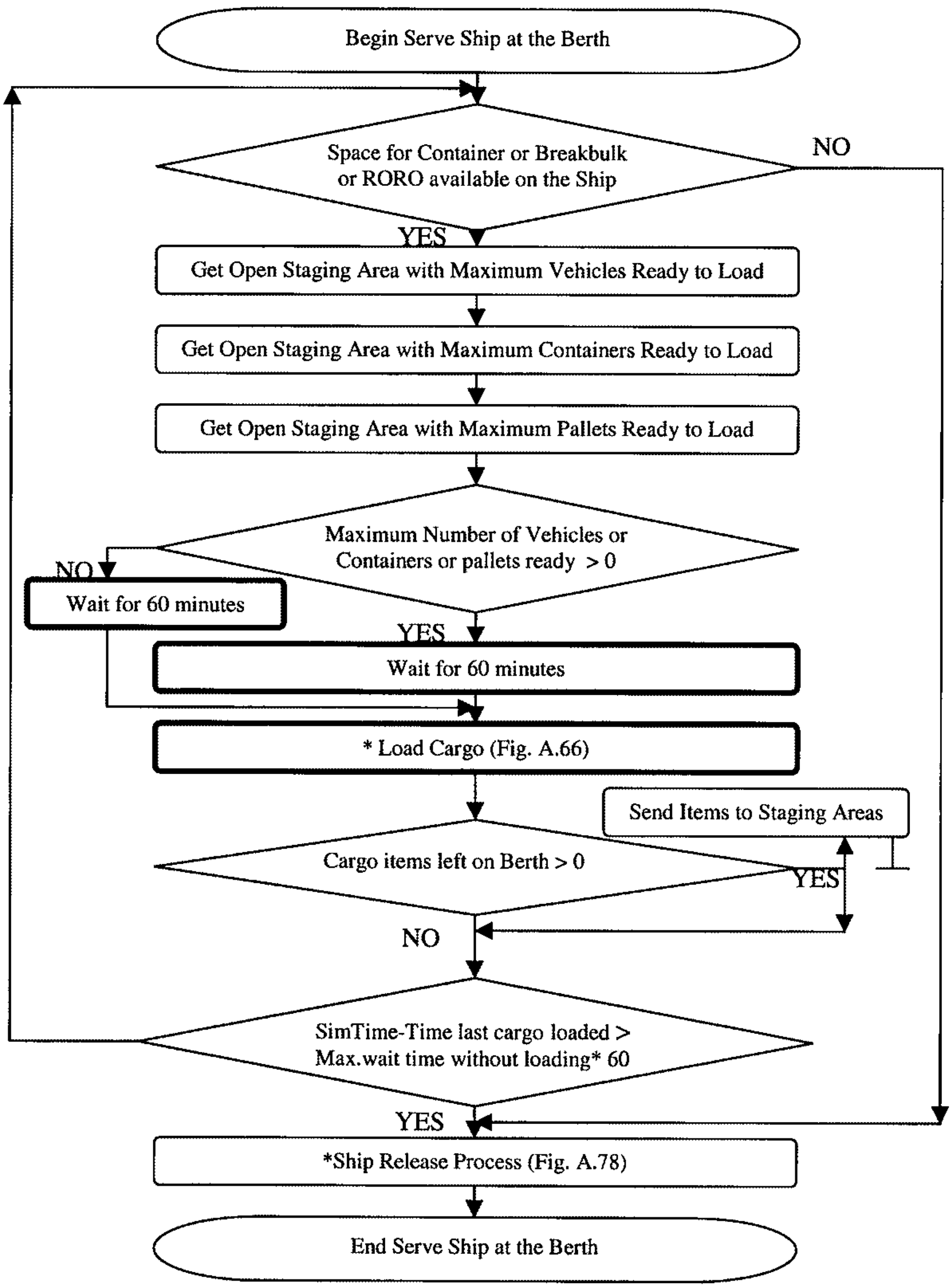

Begin Serve Ship at the Berth
Space for Container or Breakbulk or RORO available on the Ship
NO
YES
Get Open Staging Area with Maximum Vehicles Ready to Load
Get Open Staging Area with Maximum Containers Ready to Load
Get Open Staging Area with Maximum Pallets Ready to Load
Maximum Number of Vehicles or Containers or pallets ready > 0
NO
Wait for 60 minutes
YES
Wait for 60 minutes
* Load Cargo (Fig. A.66)
Send Items to Staging Areas
Cargo items left on Berth > 0
YES
NO
SimTime-Time last cargo loaded > Max.wait time without loading* 60
YES
*Ship Release Process (Fig. A.78)
End Serve Ship at the Berth

Fig. 23 Logical representation of Fig. A.53 when there is no cargo left to load

Step 6

From the understanding of Fig. A.53, we can see that we made the berth release the ship when there are no cargo items left to load. However, to see whether the above model leaves the original functionality of the call forward cargo process unchanged, it is necessary to understand the purpose of using the loop during the preparation of call forward list. From Figures A.55, A.56 and A.63, we can see that the loop is making sure that the program waits for at least sixty minutes for more cargo to arrive at the staging area. The model developed above is not supporting this minimum waiting period functionality.

Step 5 Revisited

We can remove the loop and wait for sixty minutes if there are no cargo items left in the staging area. Once the program control returns from the wait statement, we can check for new cargo items in the staging area again. Figures 24, 25 and 26 show this new model.

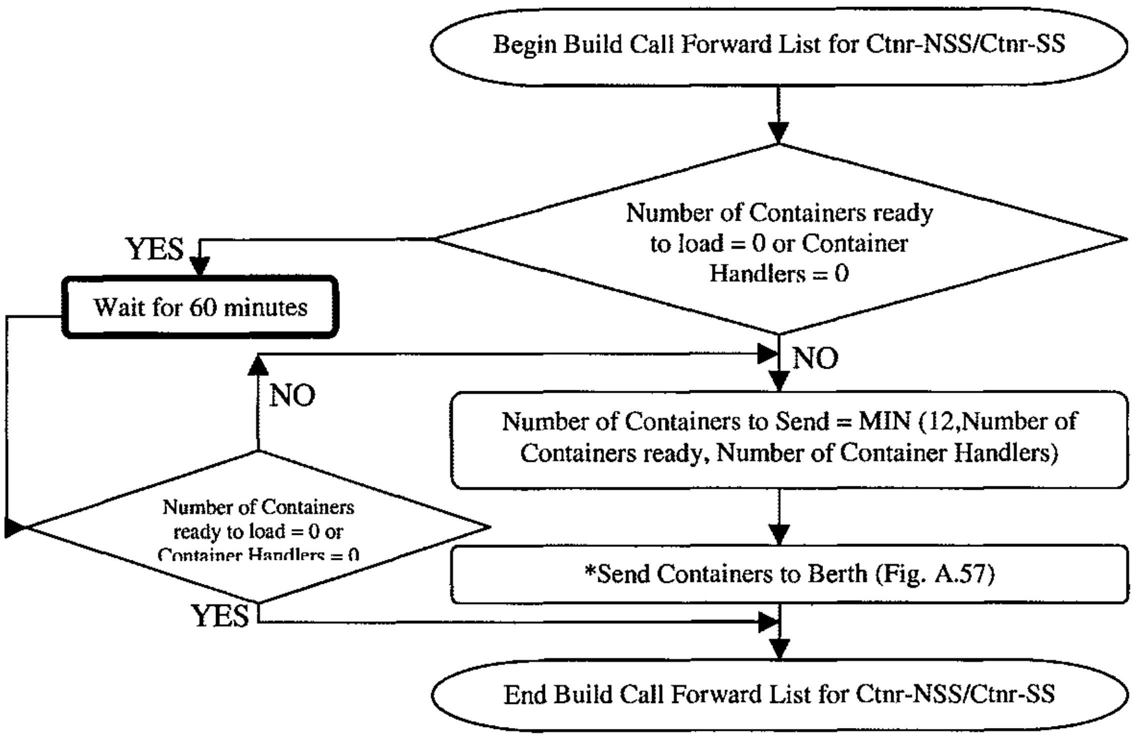

Begin Build Call Forward List for Ctnr-NSS/Ctnr-SS
Number of Containers ready to load = 0 or Container Handlers = 0
YES
Wait for 60 minutes
NO
Number of Containers to Send = MIN (12,Number of Containers ready, Number of Container Handlers)
NO
Number of Containers ready to load = 0 or Container Handlers = 0
*Send Containers to Berth (Fig. A.57)
YES
End Build Call Forward List for Ctnr-NSS/Ctnr-SS

Fig. 24 Modified representation of Fig. 20

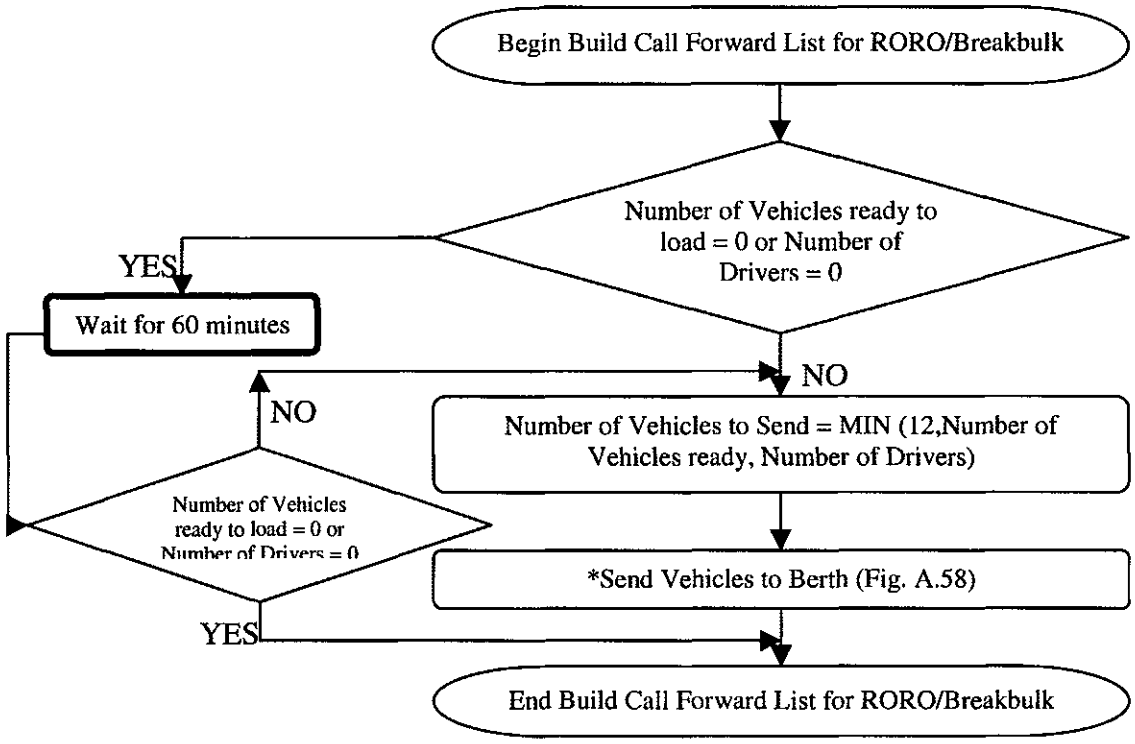

Begin Build Call Forward List for RORO/Breakbulk
Number of Vehicles ready to load = 0 or Number of Drivers = 0
YES
Wait for 60 minutes
NO
Number of Vehicles to Send = MIN (12,Number of Vehicles ready, Number of Drivers)
NO
Number of Vehicles ready to load = 0 or Number of Drivers = 0
*Send Vehicles to Berth (Fig. A.58)
YES
End Build Call Forward List for RORO/Breakbulk

Fig. 25 Modified representation of Fig. 21

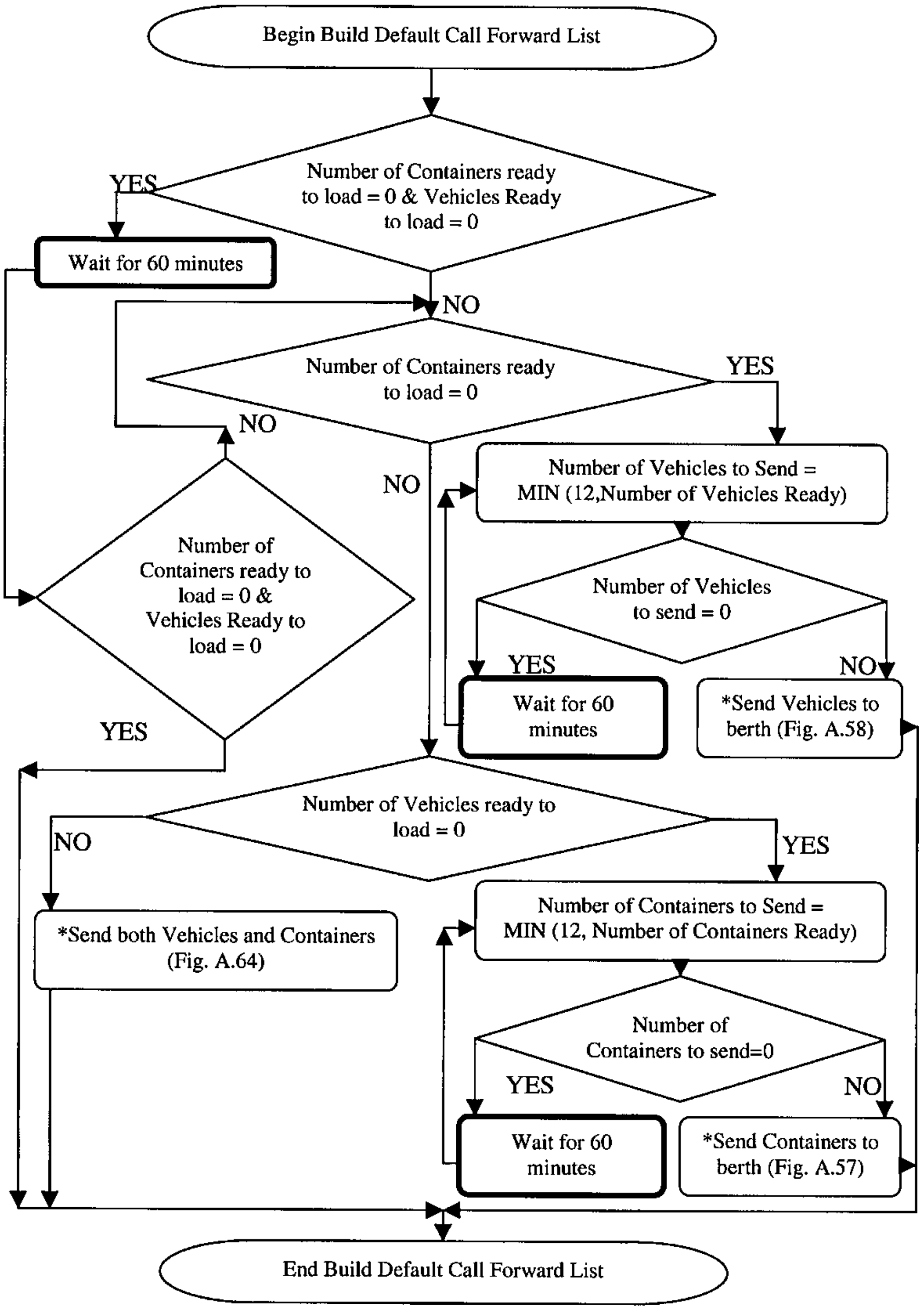

Begin Build Default Call Forward List
Number of Containers ready to load = 0 & Vehicles Ready to load = 0
YES
Wait for 60 minutes
NO
Number of Containers ready to load = 0
YES
NO
Number of Containers ready to load = 0 & Vehicles Ready to load = 0
NO
Number of Vehicles to Send = MIN (12,Number of Vehicles Ready)
Number of Vehicles to send = 0
YES
NO
Wait for 60 minutes
*Send Vehicles to berth (Fig. A.58)
YES
Number of Vehicles ready to load = 0
NO
YES
*Send both Vehicles and Containers (Fig. A.64)
Number of Containers to Send = MIN (12, Number of Containers Ready)
Number of Containers to send=0
YES
NO
Wait for 60 minutes
*Send Containers to berth (Fig. A.57)
End Build Default Call Forward List

Fig. 26 Modified representation for Fig. 22

Step 6 Revisited

The alternate model developed above also makes the "Call Forward Cargo" process behave as a WAIT statement when there are no more cargo items left to arrive at the staging area, and it also satisfies the minimum waiting period for more cargo items to arrive at the staging area.

Step 7

Now we can convert the flow diagrams shown in Figures 24, 25 and 26 into MODSIM III constructs using the notations developed. The MODSIM III code for these figures is shown in Fig. C.3.

Step 8

The modified code has been executed and we found that the ships were leaving the port when there is no more cargo left to load, irrespective of the remaining cargo space on ships.

It should also be noted that this fix allows the code to operate as originally modeled. However, this is not the correct port operation. A significant effort is required to develop a new berthing/call forward model. This need was not uncovered until this modeling effort allowed insight to the implementation model by the domain experts.

4.2 Dialog Boxes Closing Problem

Step 1

Problem definition: Users are not able to close any of the graphical windows that pop up during the course of creating and running a scenario, by clicking on the window's close button. Close buttons are simply the **X** buttons on the right hand top corner of the graphical windows. The main graphical windows behave correctly; only the sub-windows require examination.

From the problem statement, we may surmise that there is a difference in the object types of the windows that are responding to the close buttons and the windows that are not responding to the close buttons. The difference in the object types may be due to the difference in the base object types or due to the difference in the way the objects are derived from base object.

Step 2

The flow diagrams representing the functionality of various graphical window objects are shown in Figures B.1 through B.7.

Step 3

1) From step 1, we decide that we need to concentrate on the difference between the graphical object types and the code associated with them.
2) From Figures B.1 through B.7 we can see that
   - There is a difference in the base object types from which the graphical objects shown in Fig. B.1 and Figures B.2 to B.7 are derived. One is of WindowObj type and another is of DialogBoxObj type.
   - The "cancel" button, which is being used to stop/cancel any process of modifying the parameters of an object that the graphical window is representing, closes the graphical window.
3) From Fig. B.1, which represents the functionality of a graphical object derived from WindowObj, we can see that there is no explicit code written to respond to the close button. Same is the case with Figures B.2 through B.7, which represent the functionality of graphical objects derived from DialogBoxObj.

From Section 3.1.1.2, we know that whenever a user clicks on the close button of a graphical object, the corresponding BeClosed method is called automatically. The default action of the BeClosed method for the WindowObj object is to terminate the application. Also, from problem definition, we infer that the graphical objects that are responding to close buttons are of type WindowObj. Thus, we can infer that the default action of the BeClosed method for the DialogBoxObj object is to do nothing.

Step 4

We need to overwrite the Beclosed method for the objects of type DialogBoxObj in such a way that it actually helps to close the graphical window to which it belongs.

Step 5

The important decision has to be made about the complete functionality of the close button before solving this problem. We should keep in mind that by achieving the condition mentioned in step 4, we are actually changing the behavior of the dialog boxes. Therefore, we need to concentrate on the consistent behavior of the dialog boxes. As closing generally signifies that the graphical object and hence its services are not needed anymore, it is a good idea to make the complete functionality of the close button same as that of "cancel" button of that dialog box.

From Figures B.2 through B.7, we can see that it is impossible to have a unique solution for all types of dialog boxes and yet have their previous functionality unchanged. Thus, the best way to solve this problem is by finding a solution for each type of dialog box.

1. Solution for instances of DialogBoxObj object:

In order to overwrite the default behavior of the BeClosed method of the DialogBoxObj object, an additional object of type wrapDialogBoxObj is derived from DialogBoxObj object with only its BeClosed method overwritten. The overwritten method should contain code to make the AcceptInput return when a user clicks on the close button. This can be achieved by either disposing the instance or erasing the instance within the

BeClosed method. However, from Figures B.2 through B.6 we can see that the instance cannot be disposed as it is referred after AcceptInput returns, so the best way to make the AcceptInput return is by erasing the instance in the Beclosed method. Fig. 27 shows the representation of BeClosed method.

Solution for Fig. B.2:

From Fig. B.2, we can see that the above described modification alone is sufficient to solve the problem for this type of instances of DialogBoxObj object. The solution is achieved by simply changing the type of instances with behavior shown in Fig. B.2 from DialogBoxObj to wrapDialogBoxobj. Fig. 28 shows the modified representation of Fig. B.2.

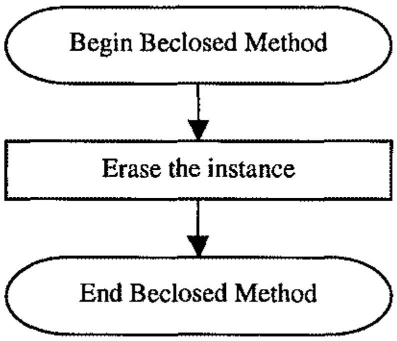


Fig. 27 Representation of BeClosed method using "Erase"

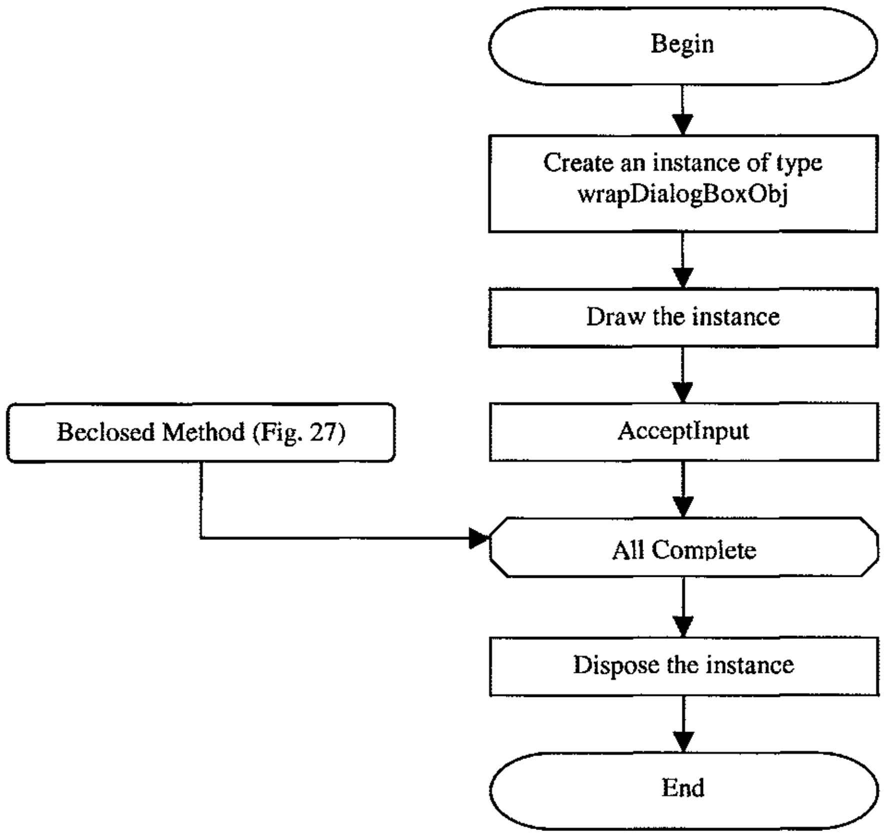

Begin
Create an instance of type wrapDialogBoxObj
Draw the instance
Beclosed Method (Fig. 27)
AcceptInput
All Complete
Dispose the instance
End

Fig. 29 Modified representation of Fig. B.2

Solution for Fig. B.3:

From Fig. B.3, we can see that the loop has to be broken in addition to changing the type of instance from DialogBoxObj to wrapDialogBoxObj to solve the problem for this type of instances. The loop can be broken by checking the returned value from the AcceptInput method. If the returned object is of type NILOBJ, we exit from the loop

[3.1.1.2]. The returned object type has to be checked again outside the loop to make the behavior of the close button the same as that of the cancel button. Fig. 29 shows the modified representation of Fig. B.3.

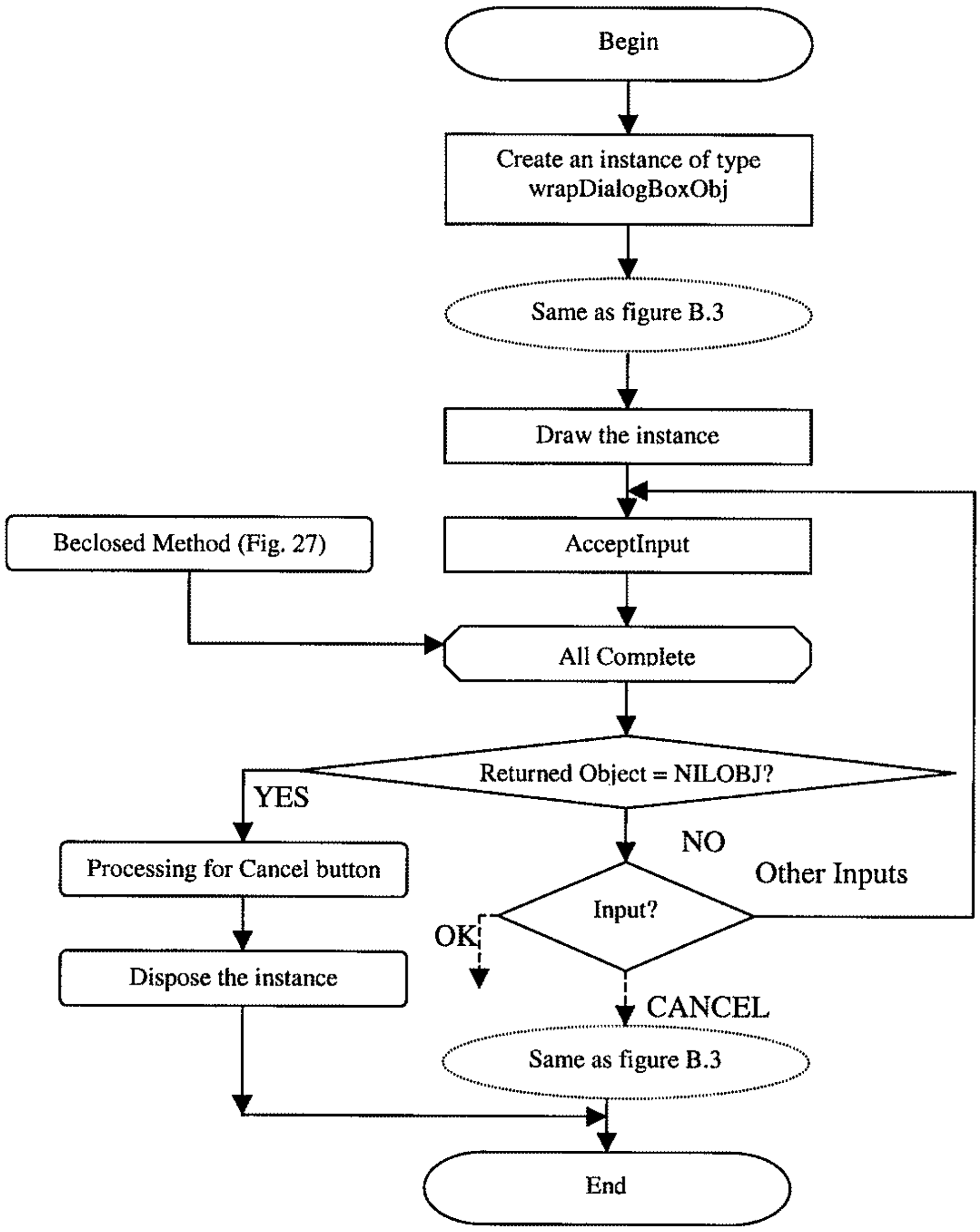


Fig. 29 Modified representation of Fig. B.3

Solution for Fig. B.4:

From Fig. B.4, we can see that for this type of instances we also need to check the object type of the returned object from AcceptInput method immediately after it returns in order to make the behavior of the close button same as that of the cancel button. Fig. 30 shows the modified representation of Fig. B.4.

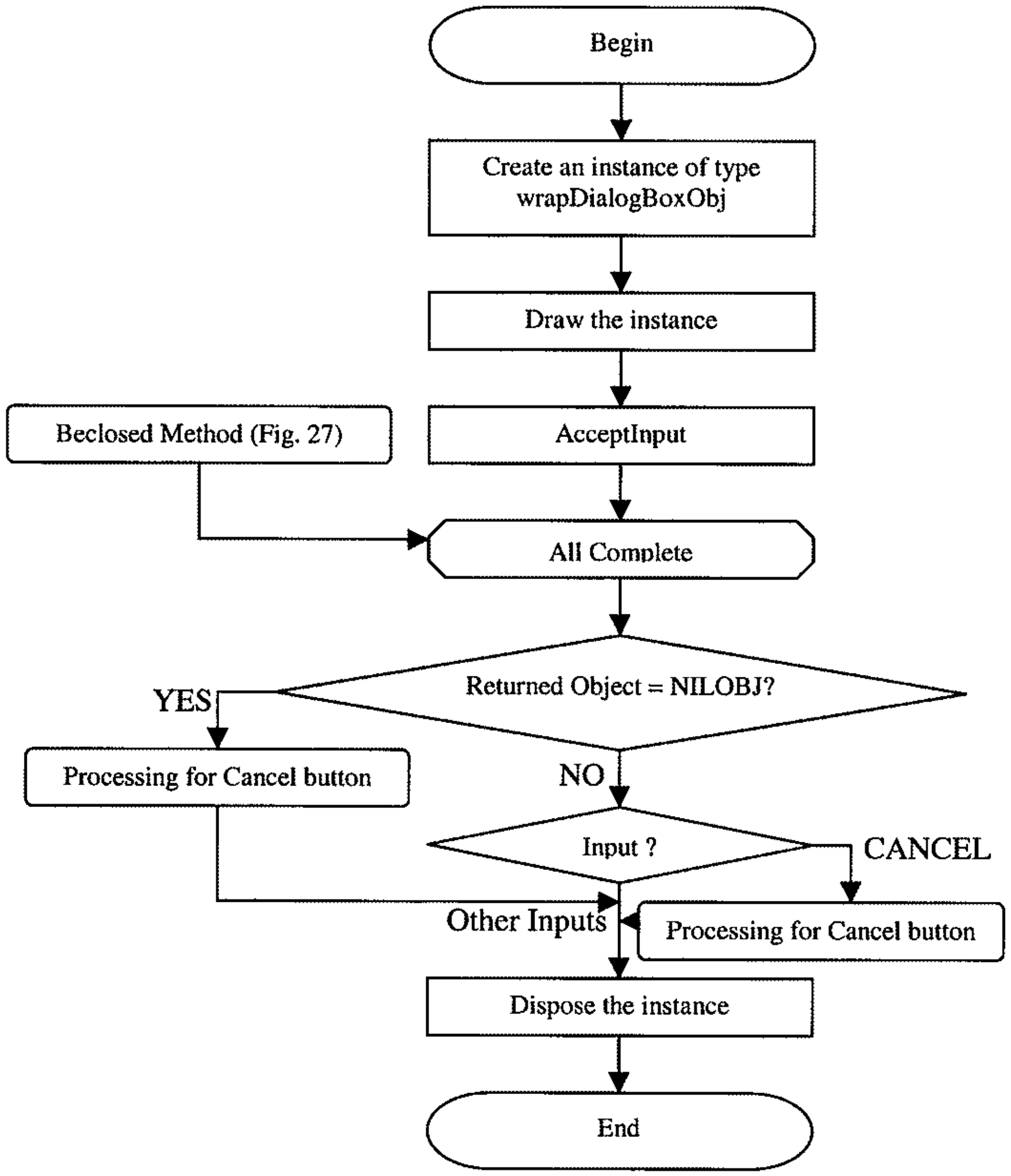

Begin
Create an instance of type wrapDialogBoxObj
Draw the instance
Beclosed Method (Fig. 27)
AcceptInput
All Complete
Returned Object = NILOBJ?
YES
Processing for Cancel button
NO
Input ?
CANCEL
Other Inputs
Processing for Cancel button
Dispose the instance
End

Fig. 30 Modified representation of Fig. B.4

Solution for Fig. B.5:

From Fig. B.5 we can see that these types of instances are not accepting any input from a user. So this kind of dialog boxes are to be left alone.

2. Solution for instances of objects derived from DialogBoxObj object:

As dialog boxes have an object type that is inherited from DialogBoxObj, we can simply overwrite the BeClosed method in the inherited objects itself.

Solution for Fig. B.6:

From Fig. B.6a, we can see that this type of dialog box is not referenced once after the cancel button is pressed. Therefore, just copying the code for the cancel button into the body of BeClosed method is sufficient to solve the problem. The code for the cancel button can be achieved from the dialog boxes BeSelected method under the "cancel" option. From Fig. B.6b, we can see that the functionality of cancel button is to dispose the instance, so we dispose the instance in BeClosed method as shown in Fig. 31. Fig. 32 shows the modified representation of Fig. B.6a.

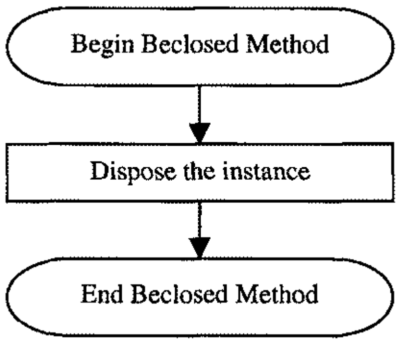


Fig. 31 Representation of BeClosed method using "Dispose"

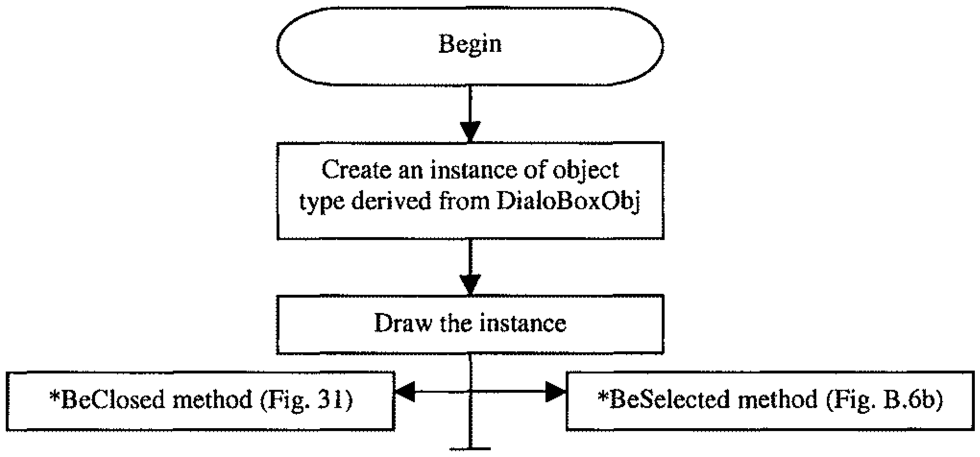

Begin
Create an instance of object type derived from DialoBoxObj
Draw the instance
*BeClosed method (Fig. 31)
*BeSelected method (Fig. B.6b)

Fig. 32 Modified representation of Fig. B.6a

Solution for Fig. B.7:

From Fig. B.7, we can see that these dialog boxes have two different types of behavior. The two different behaviors are shown in Figures B.8a and B.9a, respectively. The similarity between these figures is that in both cases the instances are referenced after the AcceptInput method returns, so we need to:

1) Erase the instance in BeClosed method, as shown in Fig. 27.
2) Check the returned object type from AcceptInput method for NILOBJ type.

The difference between Figures B.8a and B.9a is that the instances are disposed in Fig. B.8a but are not in Fig. B.9a. We can see that this difference between Figures B.8a and B.9a is due to the difference in the behavior of the cancel button in the corresponding BeSelected methods, shown in Figures B.8b and B.9b respectively. From Fig. B.8b, we can see that the function of the cancel button is to do nothing. From Fig. B.9b, we can see that the functionality of the cancel button is to check for a closing condition and if the condition is satisfied the instance is removed or else an error message is displayed. So, in order to make the functionality of **X** button the same as that of cancel button, in the case of Fig. B.8a, it is sufficient to erase the instance in BeClosed method and check the returned object type from Acceptinput method for NILOBJ type in the loop of Fig. B.8a. However, in the case of Fig. B.9a, we need to place the functionality of cancel button within the loop if the returned object type from Acceptinput method is of type NILOBJ. Figures 33 and 34 show the modified representations of Figures B.8a and B.9a, respectively.

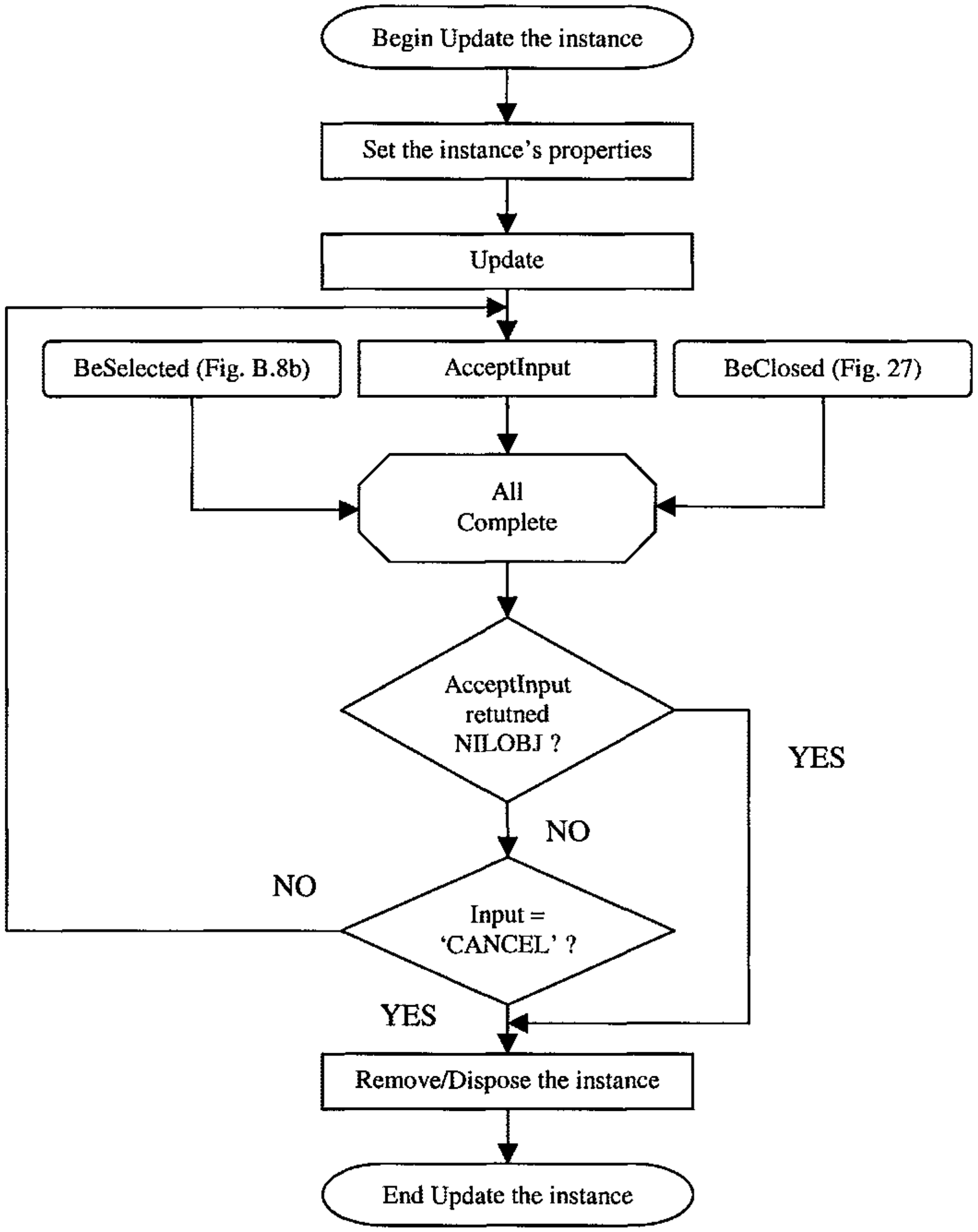


Fig. 33 Modified representation of Fig. B.8a

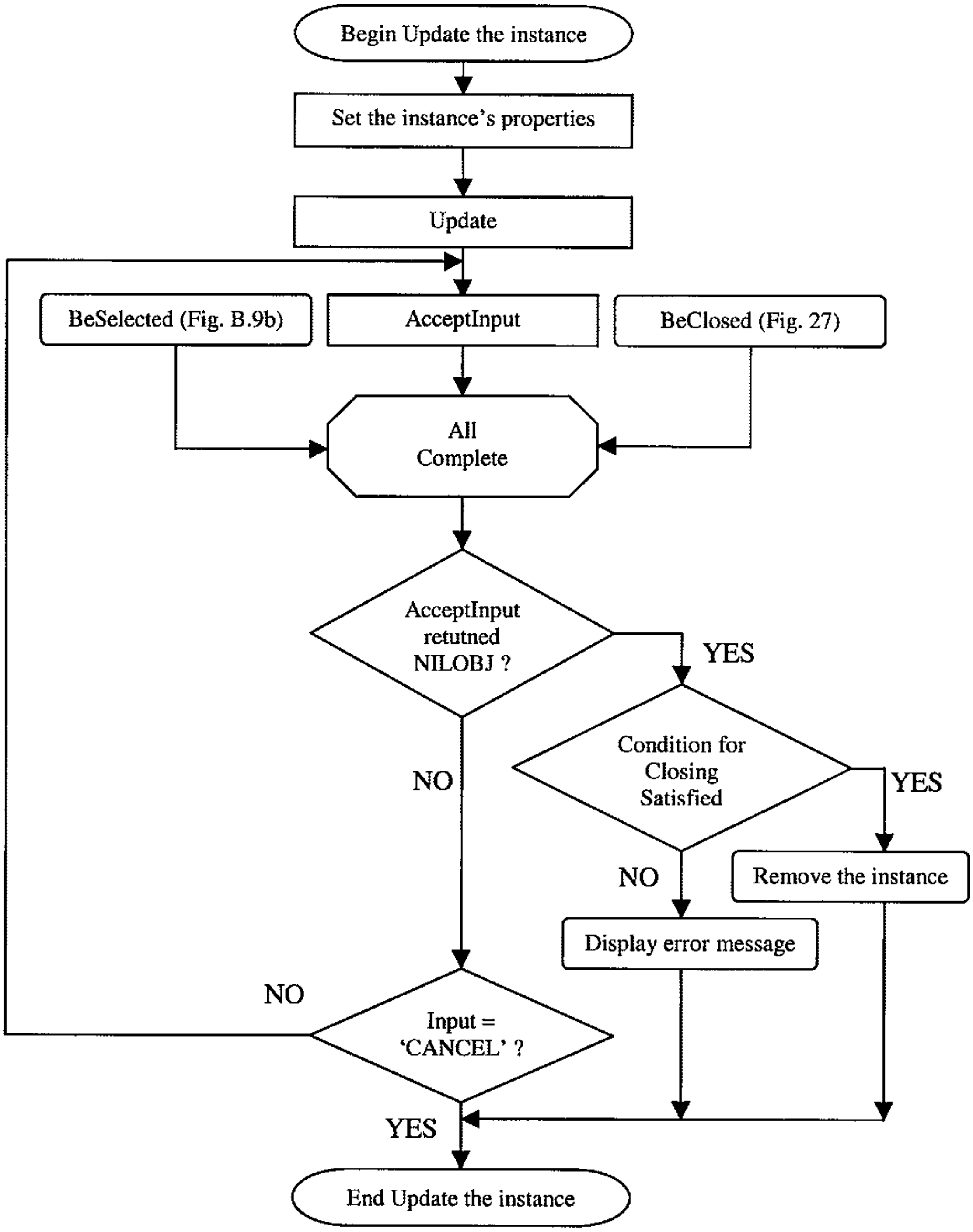


Fig. 34 Modified representation of Fig. B.9a

Step 6

The models developed above satisfy the condition mentioned in step 4 by making the dialog boxes disappear whenever a user clicks on the close button of a dialog box. The models developed above also makes sure that the functionality of the close button and the corresponding cancel button is same.

Step 7

Now we can convert the flow diagrams shown in Figures 27 to 34 into MODSIM III constructs using the notations developed. The codes for Figures 27 to 34 are shown in Figures C.4 through C.9.

Step 8

The modified code has been executed, and we found that all dialog boxes are closing when clicked on respective close buttons. Also, dialog boxes whose life cycles are controlled by the underlying process are not closing by their close buttons.

SECTION 5

CONCLUSIONS

This section summarizes the important achievements of the developed flowcharts model for pre-existing MODSIM software systems and later discusses drawbacks and possible enhancements.

5.1 Achievements

This section discusses the benefits of the flowchart model developed in Section 3.

1) Flowcharts help to understand the underlying processes that are responsible for accomplishing various tasks by the software.
2) The logical separation of processes helps to find the part of code that is most likely to have a bug.
3) The logical processes help to understand the side effects, if any, of the proposed solution to any given problem.
4) As whole processes are represented using flowcharts, it is quiet easy to understand the underlying model of any process.
5) As needed, the flowcharts do carry the information about the programming language environment and thus enable designers to model their solution accordingly.
6) The application of the developed method for modification consistently makes the flowcharts represent the recent models of the processes in the software.
7) If the initial flowcharts are developed properly, incorporation of any changes in the flowcharts into the software processes is mostly straightforward.

8) The developed method for modification does include all the three key phases of software development, namely analysis, design and implementation. The method is also iterative and incremental.

Previously when there was no model for debugging, the conceptual starting point to look for defective code was the first executable statement of the software. As one keeps working on the software, a good guess as to which module of the software may contain the bug can be made. To identify the problem causing code segment, one has to check all the possible defective code segments; this is mainly due to the message passing ability between various objects. Even once one finds the potential problem causing code segment, the probability that the first fix is the best and most effective fix is low, as the proposed fix may affect the underlying model as one does not know the entire model the software follows for its processes. However, once the model is established there is no need to go through the code first. Instead, one goes through the developed flowchart structure looking for the possible process model that may contain the bug. The process model can be easily guessed from the definition of the problem. Thus, the developed model is providing a relationship between the bugs and the PORTSIM process models. This relationship saves lot of time in finding the problem causing code. Once the process model is found, it is easy to pin point the problem causing structure and thus the code. Once the problem causing structure is found, one can propose the necessary changes to fix the bug without affecting the underlying model. It is the total picture of the underlying related process models that enables the bugs to be fixed effectively.

5.2 Drawbacks and Possible Enhancements

This section discusses drawbacks in the method and possible solutions.

1) The effectiveness of the method depends on how well the initial flowcharts were developed for each process. Conceptually, one could develop an automated method for abstracting the flowchart structure from the code and then allow the developer to fill in the resulting flowchart constructs according to the code.
2) Process models were developed when the requirement to work on that process came, so complete understanding of all the processes in the software is not available. At present, one does not know the complete Graphic User Interface (GUI) model and the complete memory semantics within the software. To solve this problem, one can develop models for all the processes within the software.
3) The flowchart constructs developed for MODSIM III language constructs do not include all the available MODSIM III constructs, like monitors, thus restricting the designers to the subset of available MODSIM III constructs. Development of flowchart constructs after a thorough study of the MODSIM III language manual can eliminate these restrictions.
4) There is currently no direct connection between the code and the flowcharts to further facilitate the code identification process. Inserting comments around code segments would provide a link between the code and the flowchart constructs. The comments should refer to the particular flowchart construct that is representing the code segment.

The developed model was useful in making modifications to the PORTSIM code by abstracting documentation of the software in the form of process flow. This helps to locate bugs in the code easily. The documentation captures the language constructs inherent to MODSIM, thus acting as a common platform between designers and programmers. The documentation helps to understand the underlying models of various processes and also the relation between these processes. This helps to make modifications in the code to fix bugs effectively. The model is currently being used by the Virginia Modeling, Analysis and Simulation Center (VMASC) and Argonne National Labs to continue PORTSIM development.

BIBLIOGRAPHY

1. Ian Sommerville, *Software Engineering*. Addison-Wesley Pub. Co., 1995.
2. Roger S. Pressman, *Software Engineering A Practioner's Approach*. McGraw-Hill International Editions, 1997.
3. Jeremy L. Rosenberger, *Sam's Teach Yourself Corba In 14 Days*. Techmedia, 1998.
4. Grady Booch, *Object Solutions Managing the Object-Oriented Project*. Addison-Wesley Pub. Co., 1998.
5. CACI Products Company. MODSIM Reference Manual. CACI Products Company, La Jolla, CA, 1997.
6. CACI Products Company. SIMGRAPHICS II User's Manual. CACI Products Company, La Jolla, CA, 1997.
7. "UML v1.3 Documentation", Rational Software Corporation, 2000. http://www.rational.com/uml/resources/documentation.
8. Boris Feldman, "UML FAQ", 2000. http://www.uml-zone.com/umlfaq.asp.
9. Jacobson, "Object-Oriented Software Engineering", 2000. http://wwwis.cs.utwente.nl:8080/dmrg/OODOC/oodoc/oo-12.html.
10. "Summary of OMT", ICON Computing, Inc., 1995. http://www.iconcomp.com/papers/comp/comp_56.html.
11. Nevins, M., Macal, C. and Joines, J., "A Discrete-Event Simulation Model for Seaport Operations", *Simulation,* vol. 70, no.4, pp.213-223, April 1998.
12. Nevins, M., Macal, C. and Joines, J., "PORTSIM: An Object-Oriented Port Simulation." *Proceedings of the 1995 Summer Computer Simulation Conference,* pp. 160-165, 1995.

13. Leathrum, J. and Joines, J., "Modeling Requirements in the Management of Simulation Projects", *Proceedings of the 1998 National Conference of the American Society for Engineering Management*, pp.37-42, October 1998.

14. Leathrum, J., R. Mielke, M. Meyer, J. Joines, C. Macal, and M. Nevins, "Stragies for Integrating Commercial and Military Port Simulations", *Proceedings of the 1998 National Conference of the American Society for Engineering Management*, pp.98-105, October 1997.

APPENDIX A

PORTSIM PROCESS FLOW

As a whole PORTSIM can be divided into three phases:

1) An initialization phase.

2) A simulation phase.

3) An output phase.

Fig. A.1 represents the program flow of PORTSIM at macro level.

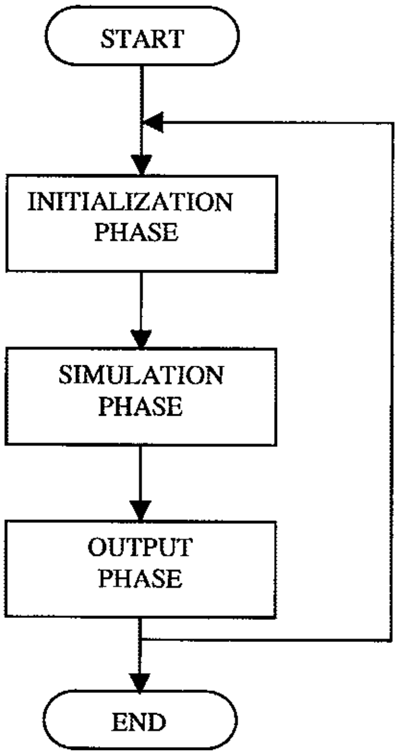


Fig. A.1

A.1 Initialization phase

It is in this phase that we prepare PORTSIM to run the scenario we want. For simplicity and ease of understanding, this section deals only with POE-Standalone type. We can either create a new scenario in which PORTSIM relies heavily on the underlying database to get required data or load an existing scenario in which PORTSIM reads data from the scenario file. To clearly understand all parts of initialization phase, we will look into the process of creating a new scenario.

As soon as we start PORTSIM, it is connected to the underlying database. When we decide to create a new scenario, PORTSIM provides a dialog box, which guides us in creating a new scenario. The various parameters that we have to decide upon are a scenario name, a port and a terminal name, a force name and a list of ships upon which the selected force has to embark. Fig. A.2 represents the flow of program control within PORTSIM during the creation of new scenario.

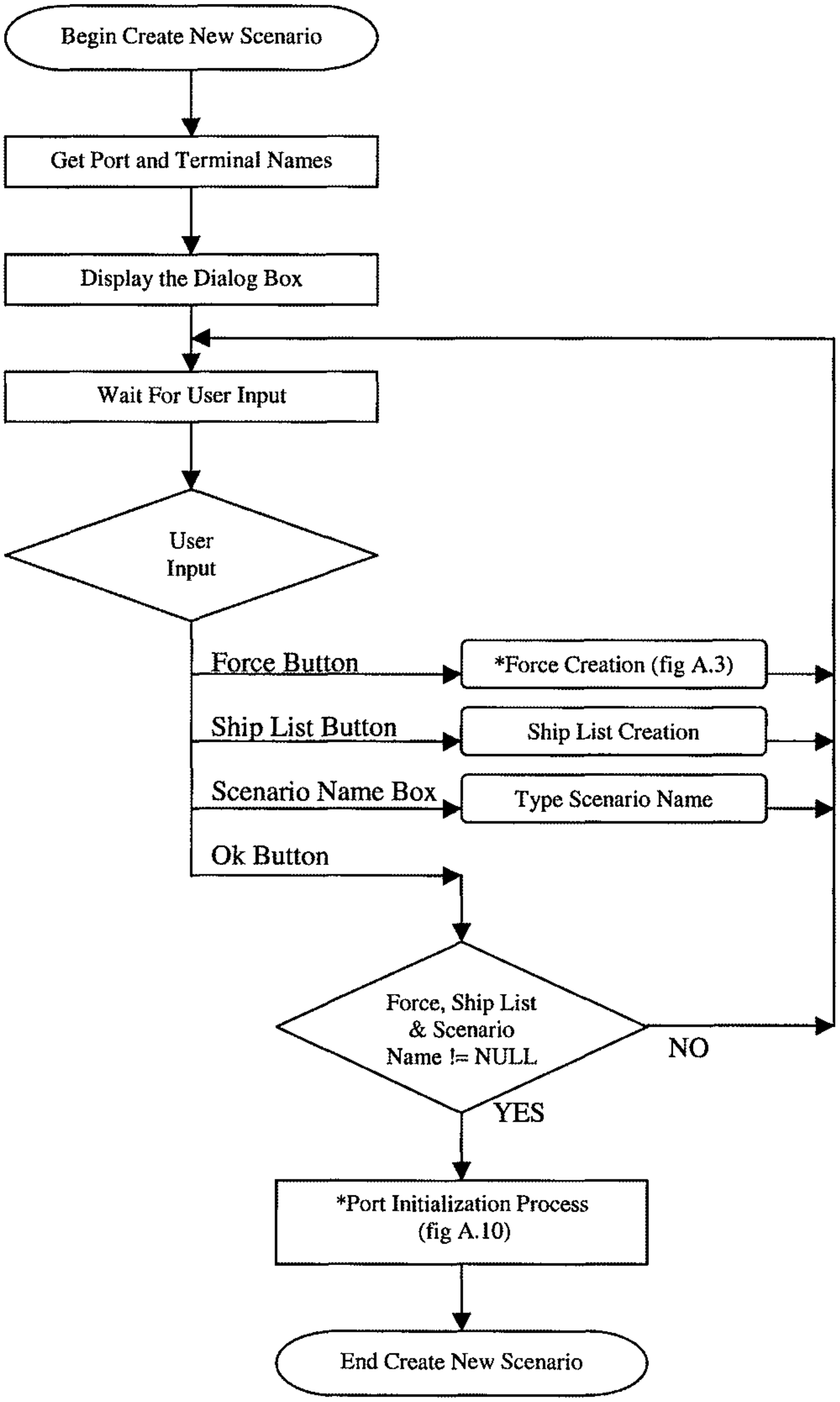

Begin Create New Scenario
Get Port and Terminal Names
Display the Dialog Box
Wait For User Input
User Input
Force Button
*Force Creation (fig A.3)
Ship List Button
Ship List Creation
Scenario Name Box
Type Scenario Name
Ok Button
Force, Ship List & Scenario Name != NULL
NO
YES
*Port Initialization Process (fig A.10)
End Create New Scenario

Fig. A.2

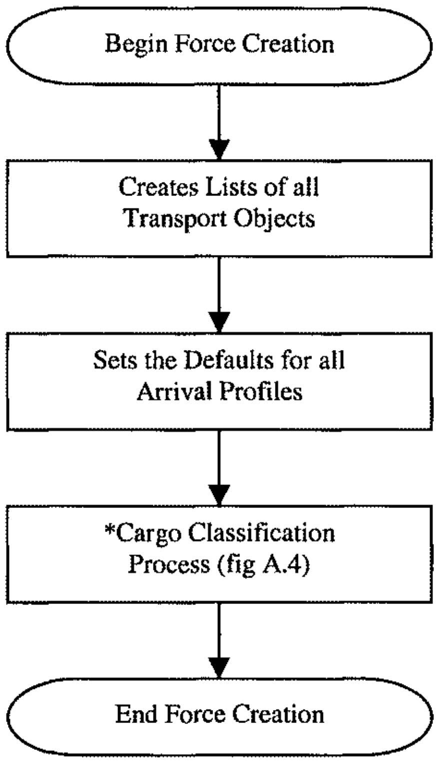

Begin Force Creation
Creates Lists of all Transport Objects
Sets the Defaults for all Arrival Profiles
*Cargo Classification Process (fig A.4)
End Force Creation

Fig. A.3

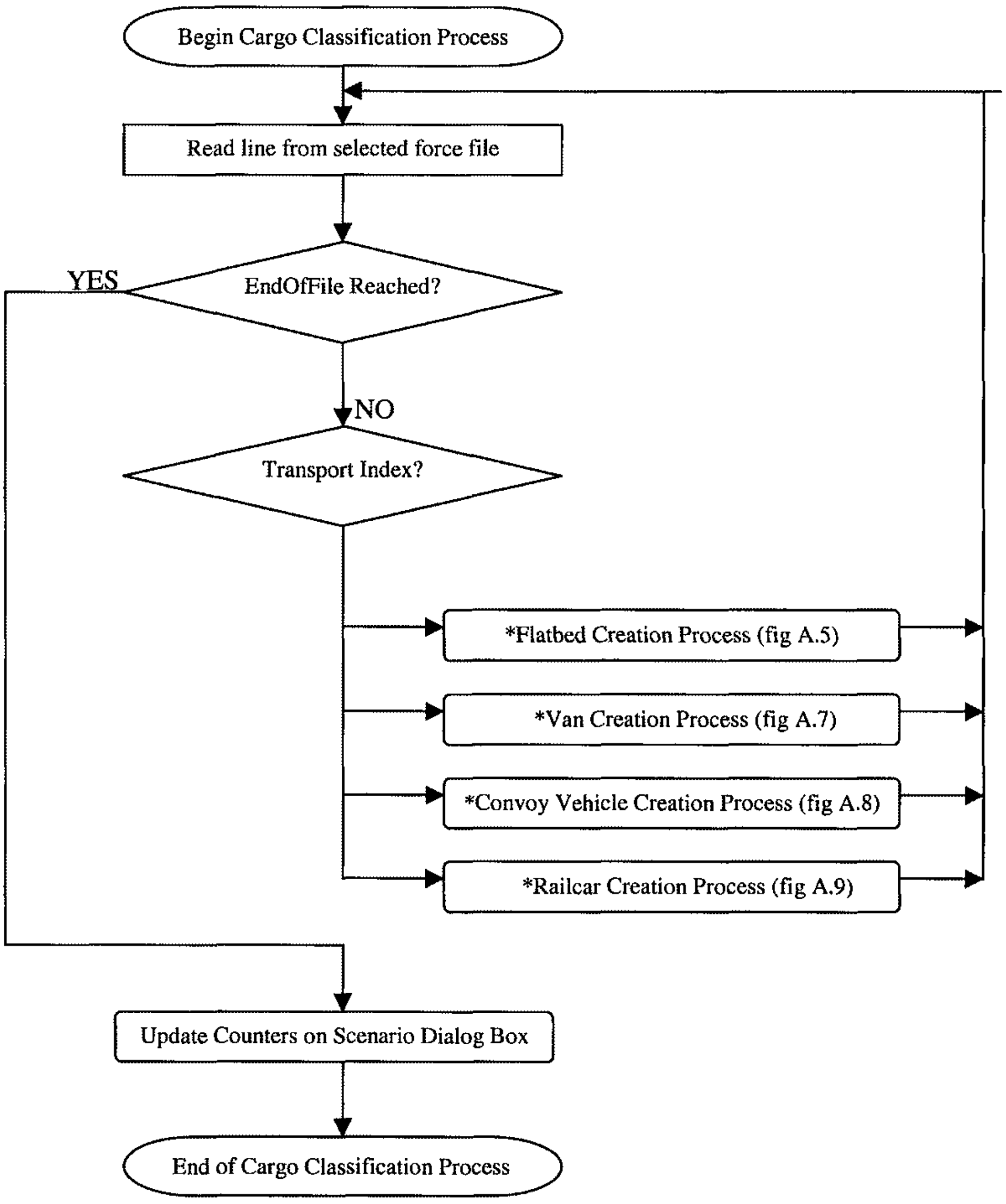

Begin Cargo Classification Process
Read line from selected force file
EndOfFile Reached?
YES
NO
Transport Index?
*Flatbed Creation Process (fig A.5)
*Van Creation Process (fig A.7)
*Convoy Vehicle Creation Process (fig A.8)
*Railcar Creation Process (fig A.9)
Update Counters on Scenario Dialog Box
End of Cargo Classification Process

Fig. A.4

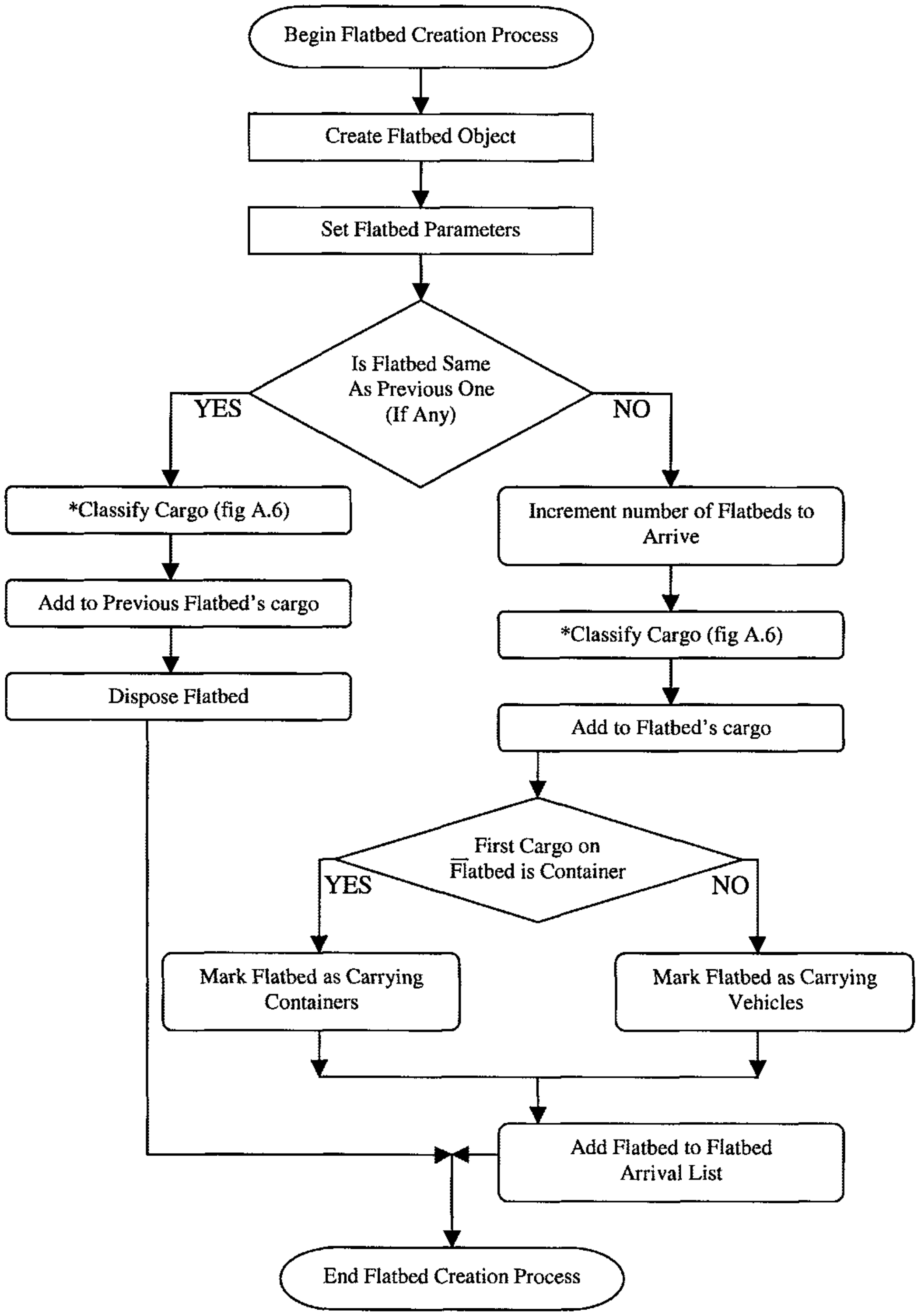

Begin Flatbed Creation Process
Create Flatbed Object
Set Flatbed Parameters
Is Flatbed Same As Previous One (If Any)
YES
NO
*Classify Cargo (fig A.6)
Add to Previous Flatbed's cargo
Dispose Flatbed
Increment number of Flatbeds to Arrive
*Classify Cargo (fig A.6)
Add to Flatbed's cargo
First Cargo on Flatbed is Container
YES
NO
Mark Flatbed as Carrying Containers
Mark Flatbed as Carrying Vehicles
Add Flatbed to Flatbed Arrival List
End Flatbed Creation Process

Fig. A.5

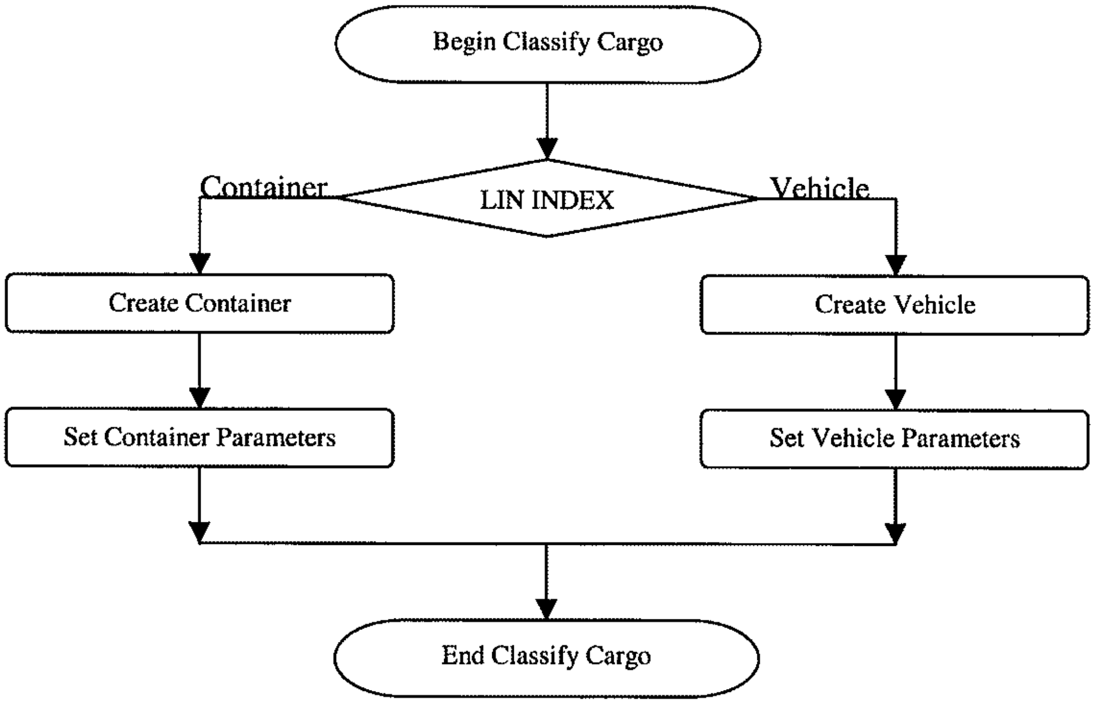


Fig. A.6

Fig. A.6 shows how containers and vehicles are differentiated in PORTSIM. This type of index-based classification is used for vehicles and containers only and not for pallets, as both flatbeds and railcars are capable of carrying both vehicles and containers. Pallets are carried by vans only.

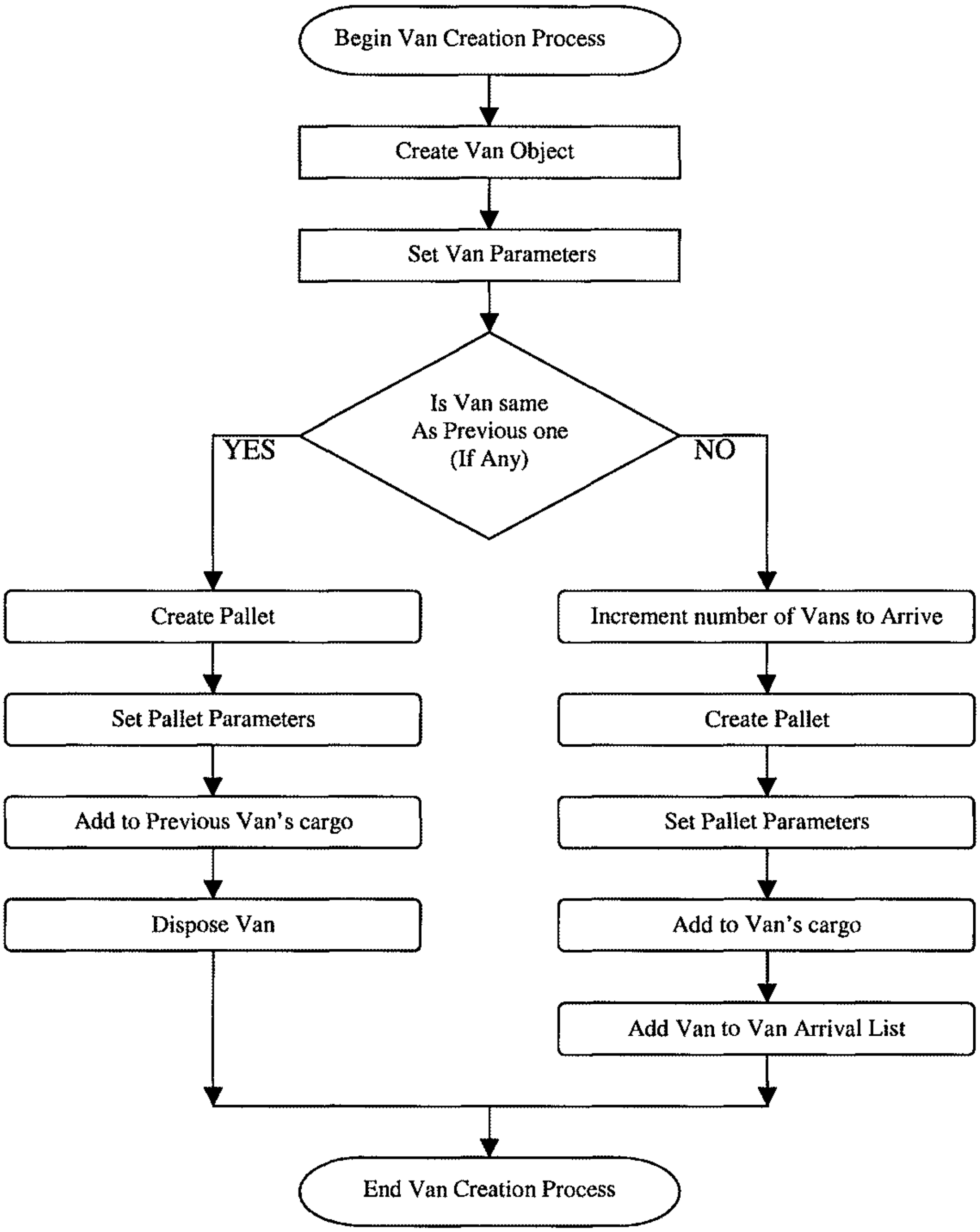

Begin Van Creation Process
Create Van Object
Set Van Parameters
Is Van same As Previous one (If Any)
YES
NO
Create Pallet
Set Pallet Parameters
Add to Previous Van's cargo
Dispose Van
Increment number of Vans to Arrive
Create Pallet
Set Pallet Parameters
Add to Van's cargo
Add Van to Van Arrival List
End Van Creation Process

Fig. A.7

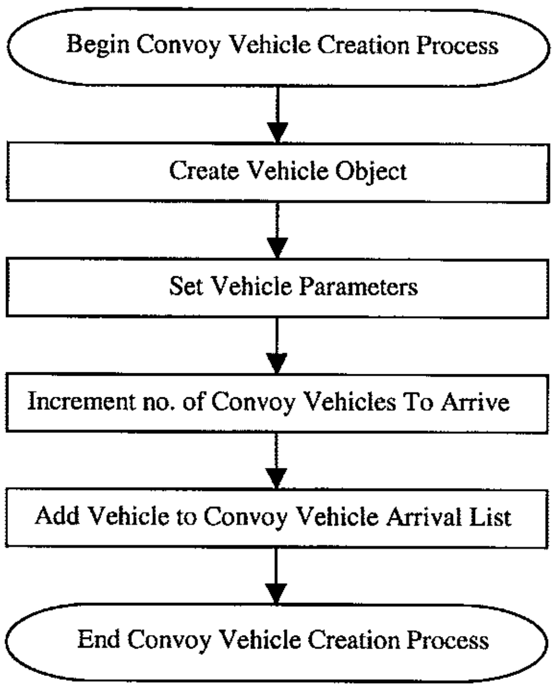

Begin Convoy Vehicle Creation Process
Create Vehicle Object
Set Vehicle Parameters
Increment no. of Convoy Vehicles To Arrive
Add Vehicle to Convoy Vehicle Arrival List
End Convoy Vehicle Creation Process

Fig. A.8

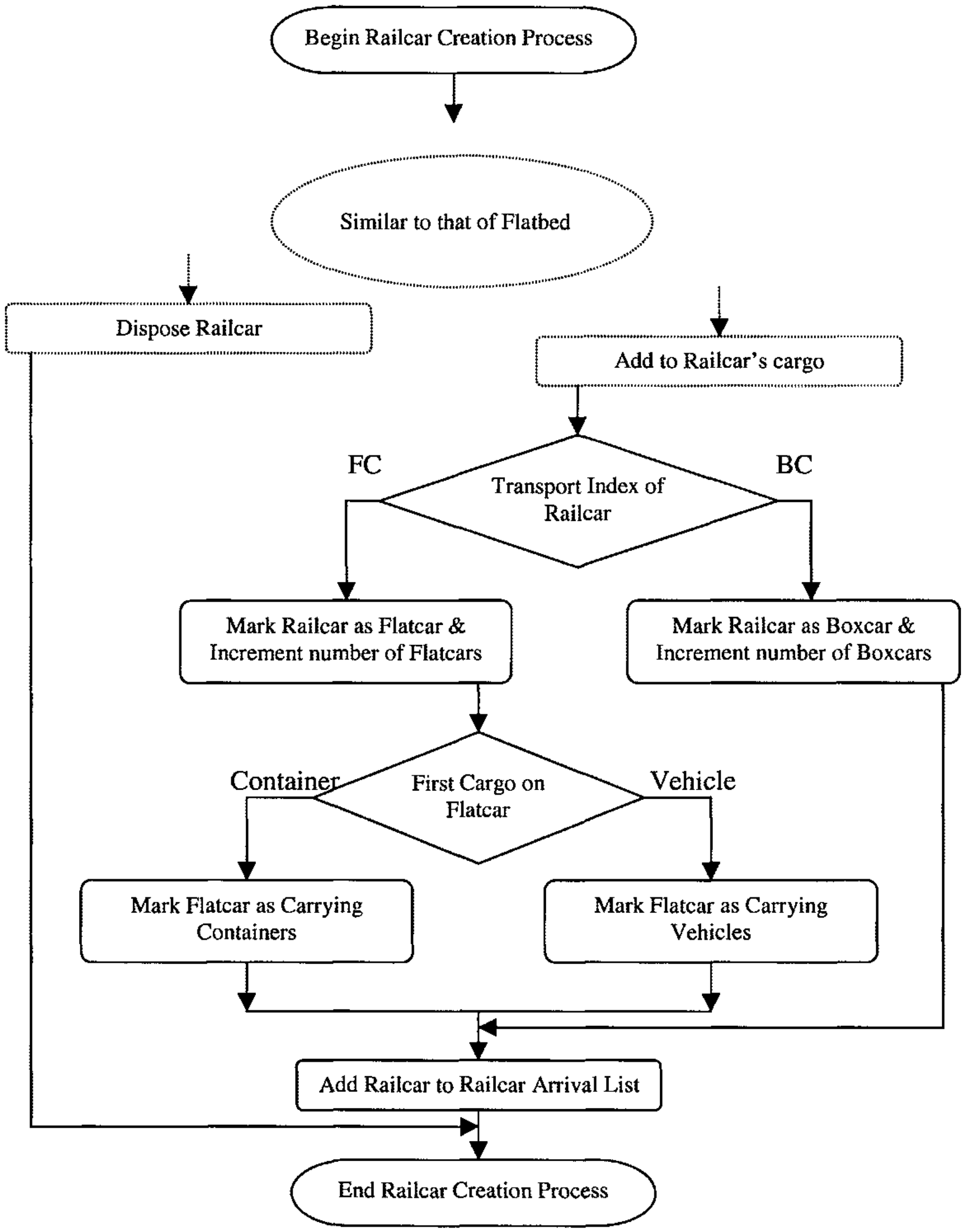

Begin Railcar Creation Process
Similar to that of Flatbed
Dispose Railcar
Add to Railcar's cargo
FC
Transport Index of Railcar
BC
Mark Railcar as Flatcar & Increment number of Flatcars
Mark Railcar as Boxcar & Increment number of Boxcars
Container
First Cargo on Flatcar
Vehicle
Mark Flatcar as Carrying Containers
Mark Flatcar as Carrying Vehicles
Add Railcar to Railcar Arrival List
End Railcar Creation Process

Fig. A.9

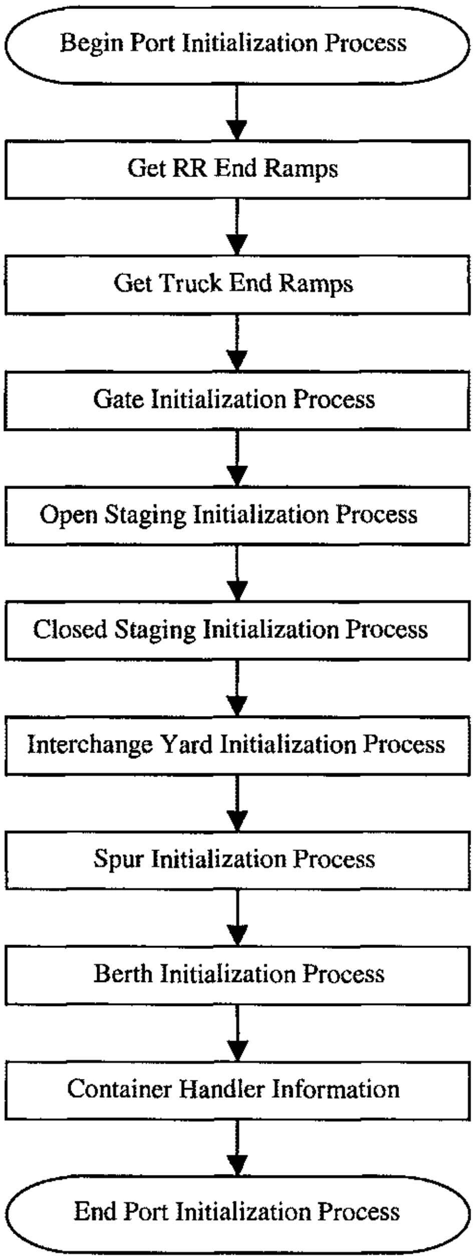

Begin Port Initialization Process
Get RR End Ramps
Get Truck End Ramps
Gate Initialization Process
Open Staging Initialization Process
Closed Staging Initialization Process
Interchange Yard Initialization Process
Spur Initialization Process
Berth Initialization Process
Container Handler Information
End Port Initialization Process

Fig. A.10

Each process mentioned above can be generalized as shown in Fig. A.11.

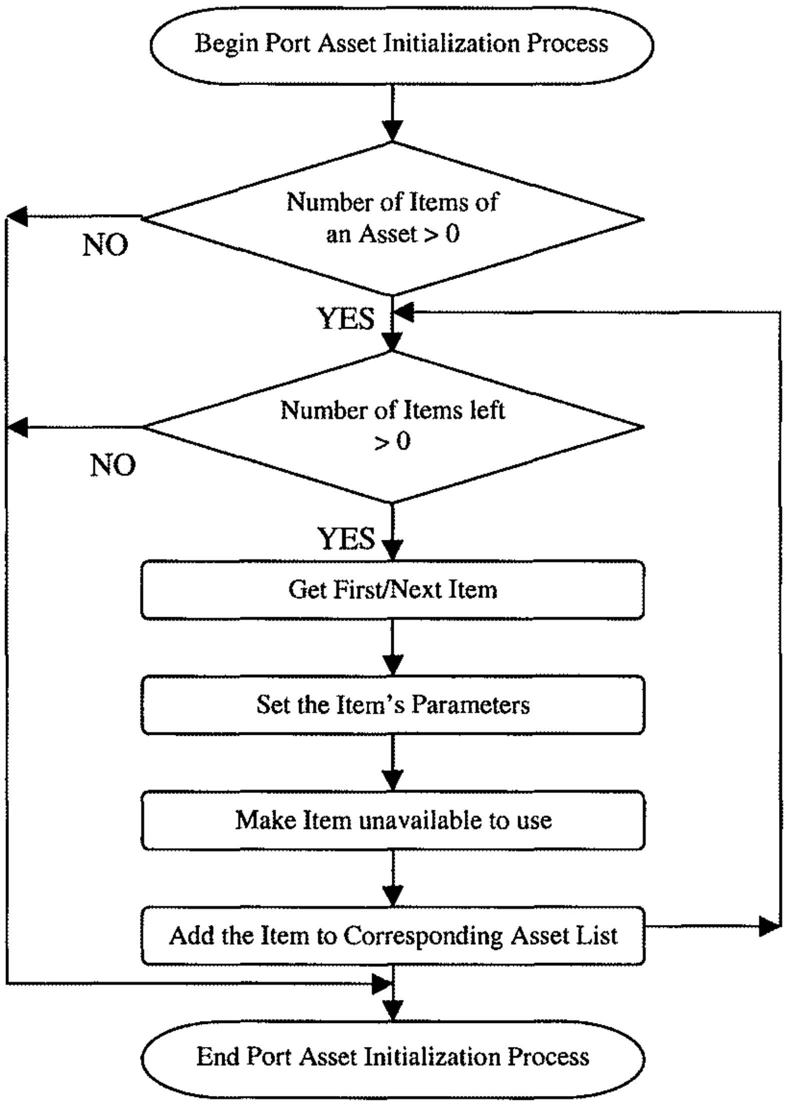

Begin Port Asset Initialization Process
Number of Items of an Asset > 0
NO
YES
Number of Items left > 0
NO
YES
Get First/Next Item
Set the Item's Parameters
Make Item unavailable to use
Add the Item to Corresponding Asset List
End Port Asset Initialization Process

Fig. A.11

NOTE: After all of this initialization is done, PORTSIM checks if there is any other scenario with same name. If there is, it asks us whether to overwrite the previous scenario or not. Now comes the important part of this phase. As we can see from Fig. A.11, all the port assets are made unavailable for military use. It is our duty to decide which assets are to be made available for military use. After this, PORTSIM is ready to simulate the created scenario.

A.2 Simulation Phase

It is in this phase that we watch the simulation run to completion. In most cases, there is little interaction with the software. Fig. A.12 shows the sequence of processes that PORTSIM goes through during this phase.

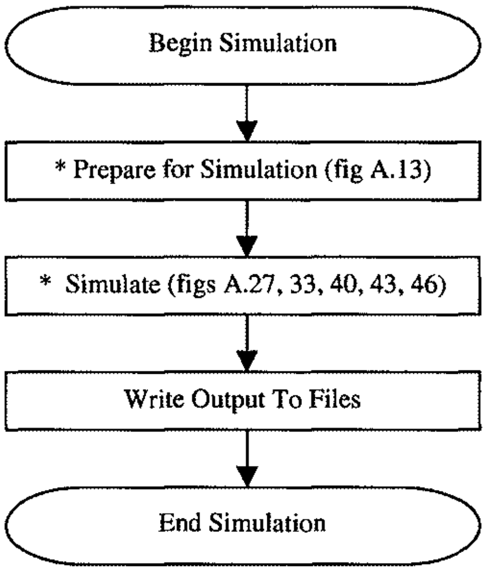

Begin Simulation
* Prepare for Simulation (fig A.13)
* Simulate (figs A.27, 33, 40, 43, 46)
Write Output To Files
End Simulation

Fig. A.12

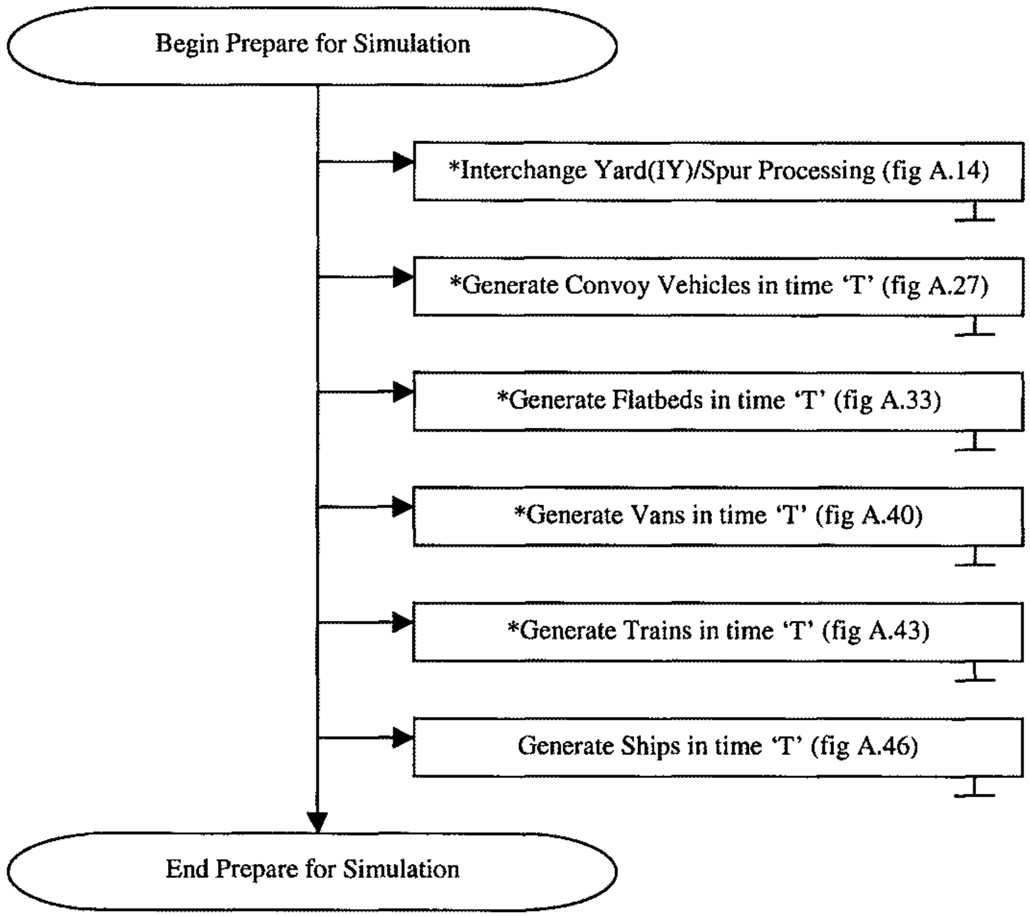

Begin Prepare for Simulation
*Interchange Yard(IY)/Spur Processing (fig A.14)
*Generate Convoy Vehicles in time 'T' (fig A.27)
*Generate Flatbeds in time 'T' (fig A.33)
*Generate Vans in time 'T' (fig A.40)
*Generate Trains in time 'T' (fig A.43)
Generate Ships in time 'T' (fig A.46)
End Prepare for Simulation

Fig. A.13

The time 'T' in the above figure is with respect to the start of simulation time. If the value of 'T' is one, that method is executed one minute after the start of simulation.

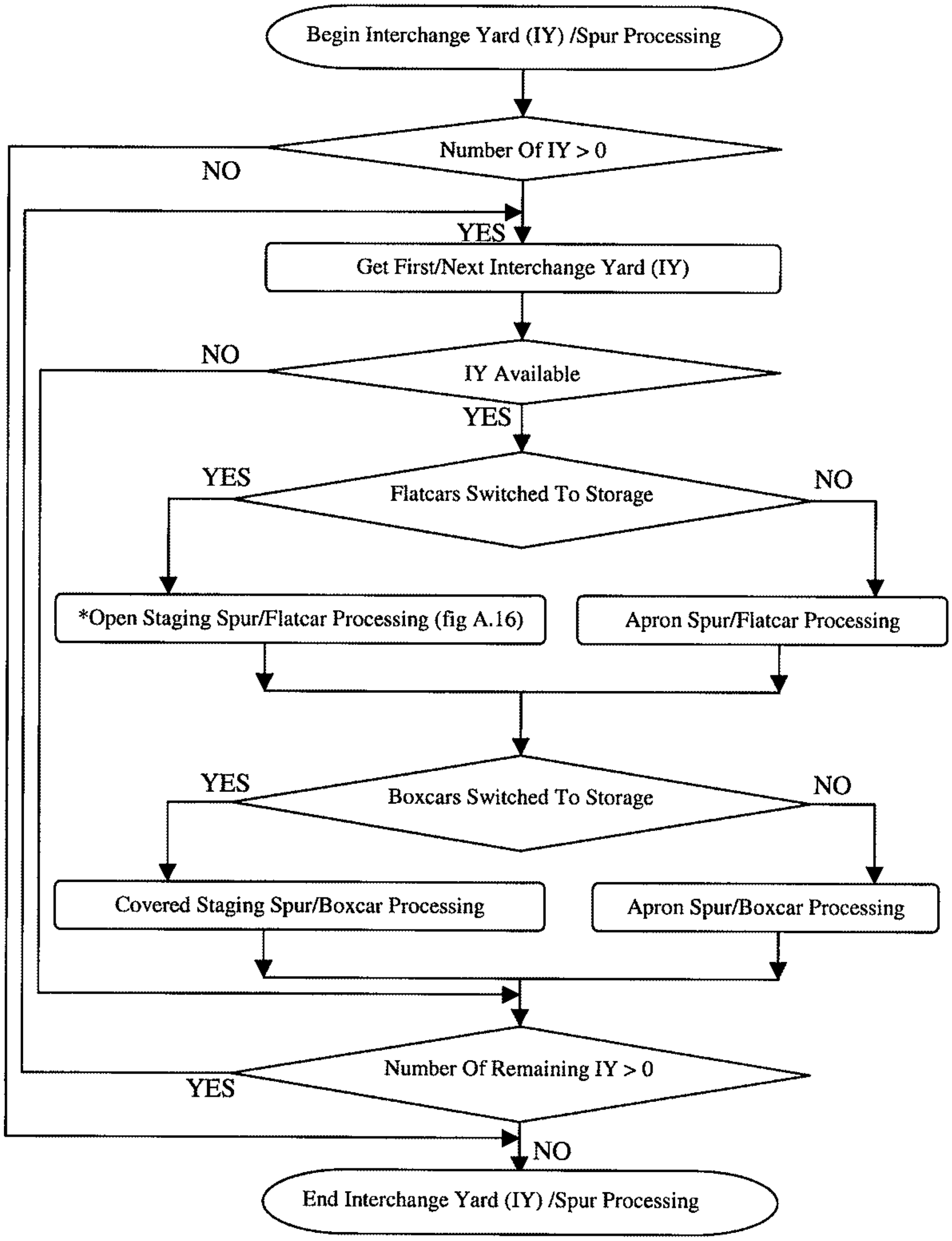

Begin Interchange Yard (IY) /Spur Processing
Number Of IY > 0
NO
YES
Get First/Next Interchange Yard (IY)
NO
IY Available
YES
YES
Flatcars Switched To Storage
NO
*Open Staging Spur/Flatcar Processing (fig A.16)
Apron Spur/Flatcar Processing
YES
Boxcars Switched To Storage
NO
Covered Staging Spur/Boxcar Processing
Apron Spur/Boxcar Processing
Number Of Remaining IY > 0
YES
NO
End Interchange Yard (IY) /Spur Processing

Fig. A.14

All the above mentioned 'spur type/railcar type processing' can be generalized as shown in Fig. A.15.

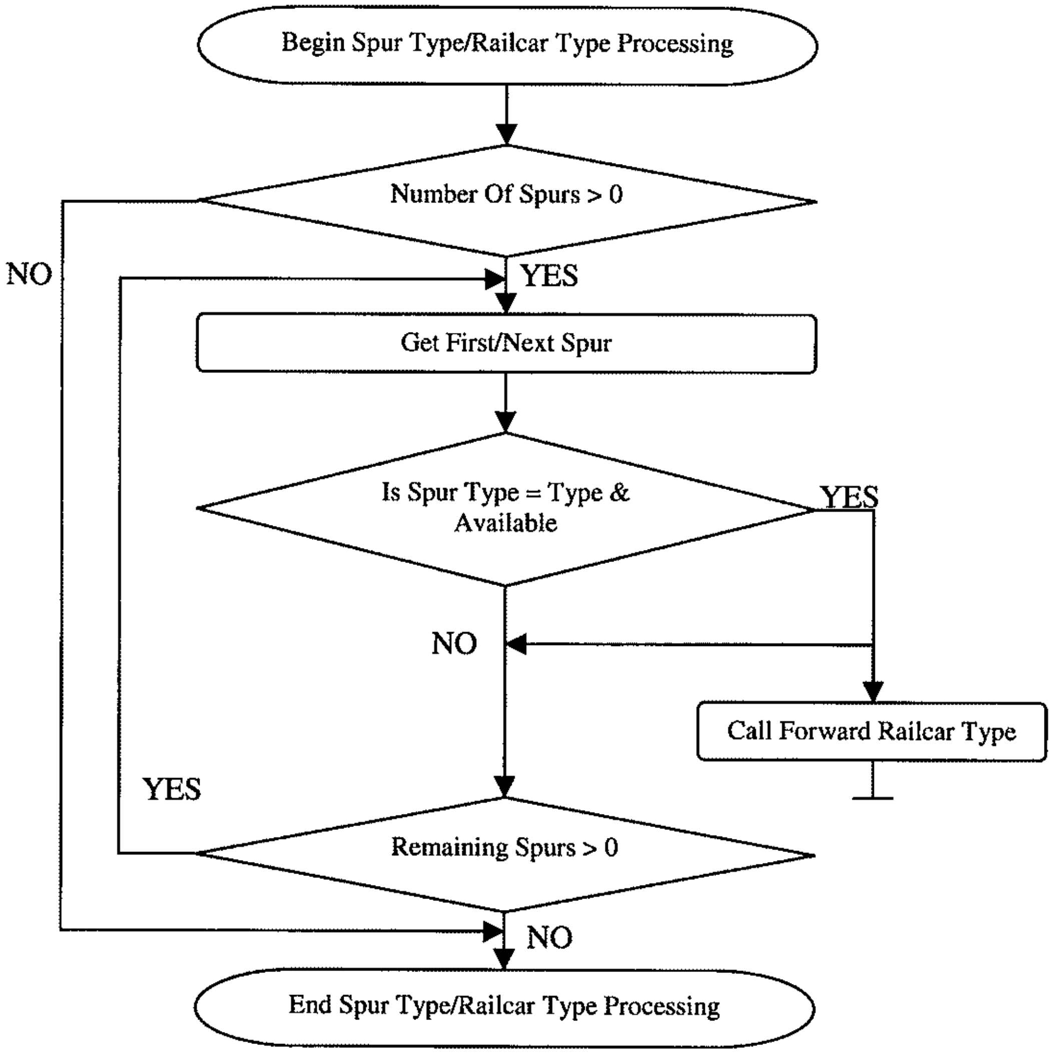

Begin Spur Type/Railcar Type Processing
Number Of Spurs > 0
NO
YES
Get First/Next Spur
Is Spur Type = Type & Available
YES
NO
Call Forward Railcar Type
YES
Remaining Spurs > 0
NO
End Spur Type/Railcar Type Processing

Fig. A.15

As an example, 'Open Staging Spur/Flatcar Processing' can be represented as shown in the following figure.

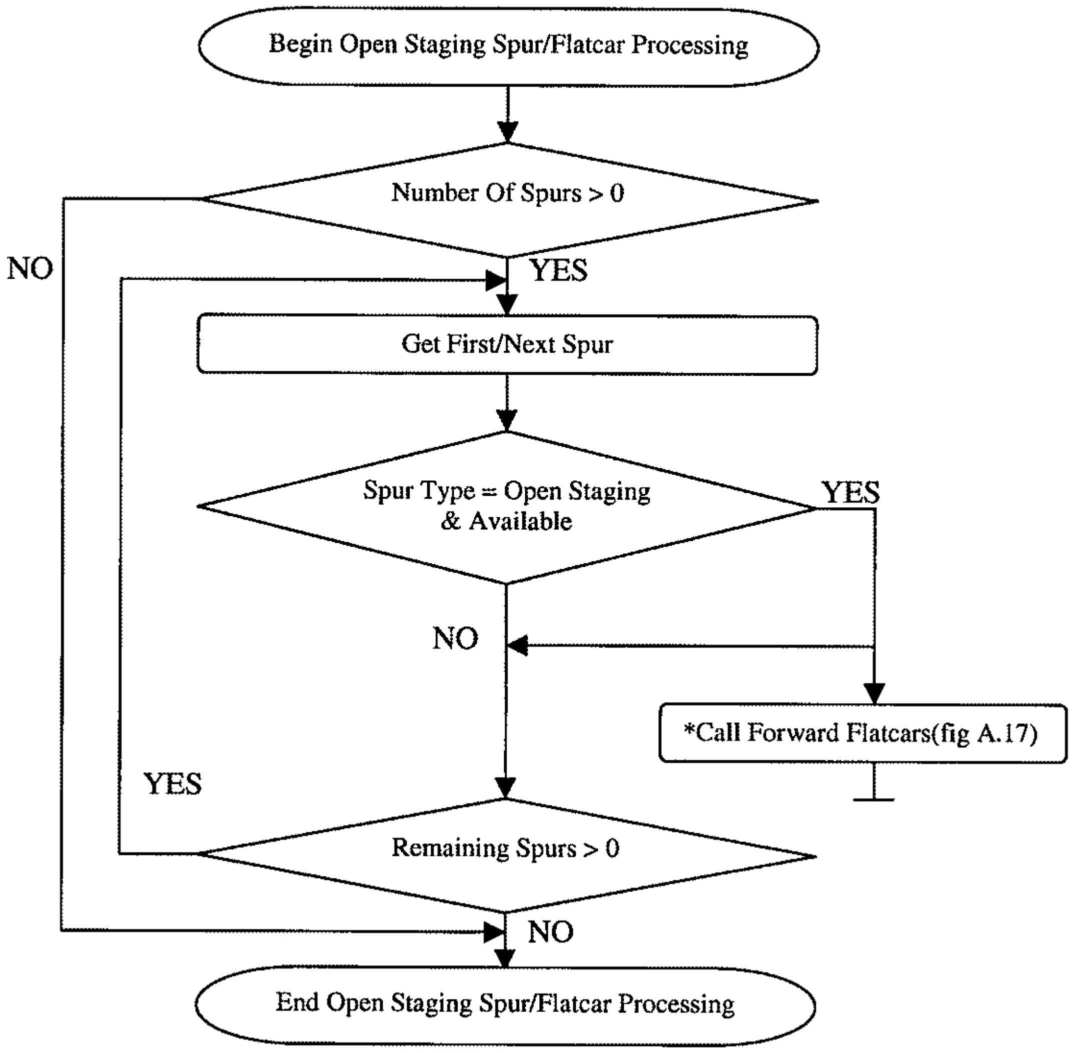

Begin Open Staging Spur/Flatcar Processing
Number Of Spurs > 0
NO
YES
Get First/Next Spur
Spur Type = Open Staging & Available
YES
NO
*Call Forward Flatcars(fig A.17)
YES
Remaining Spurs > 0
NO
End Open Staging Spur/Flatcar Processing

Fig. A.16

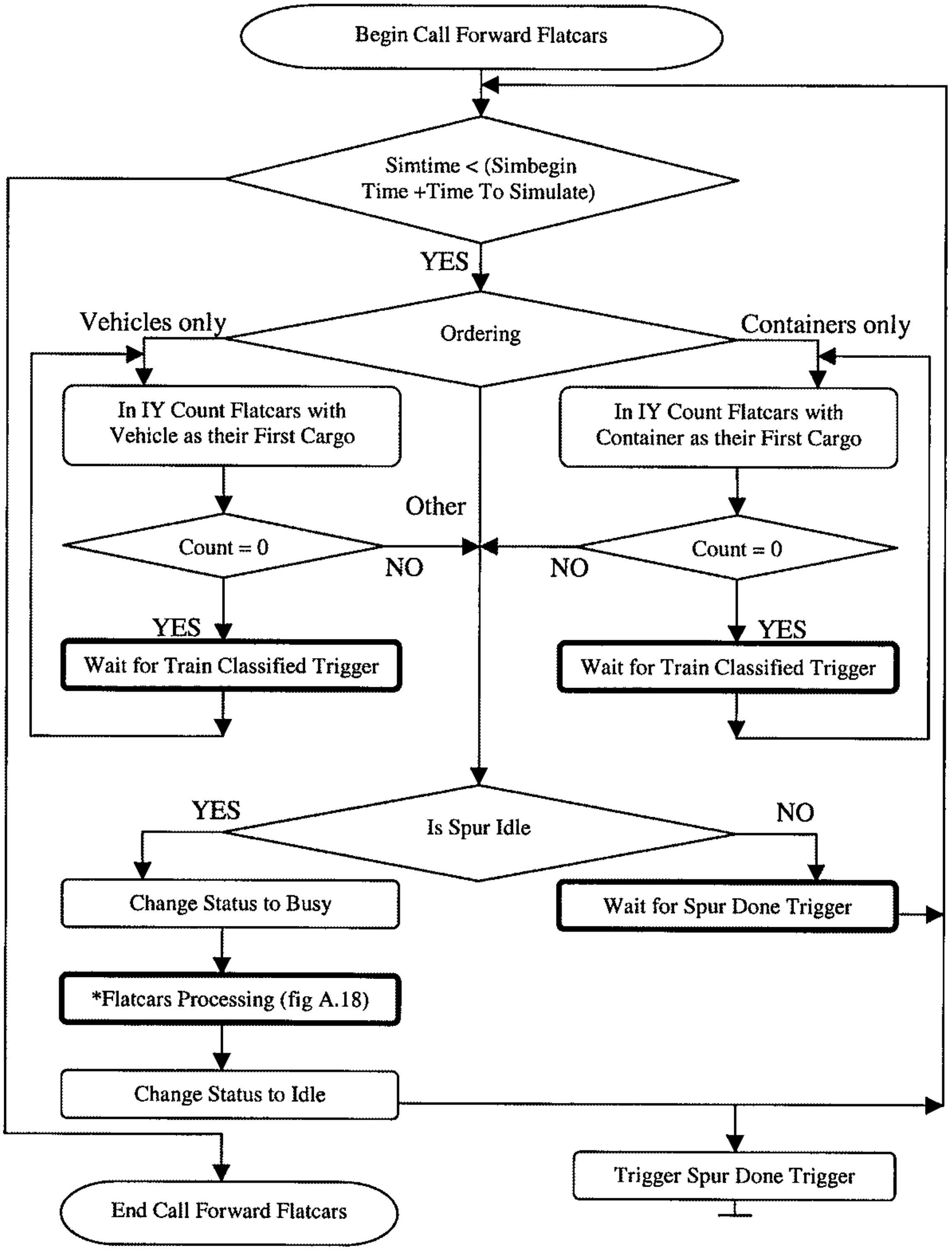

Begin Call Forward Flatcars
Simtime < (Simbegin Time +Time To Simulate)
YES
Ordering
Vehicles only
Containers only
In IY Count Flatcars with Vehicle as their First Cargo
In IY Count Flatcars with Container as their First Cargo
Other
Count = 0
NO
YES
Wait for Train Classified Trigger
Count = 0
NO
YES
Wait for Train Classified Trigger
Is Spur Idle
YES
NO
Change Status to Busy
Wait for Spur Done Trigger
*Flatcars Processing (fig A.18)
Change Status to Idle
Trigger Spur Done Trigger
End Call Forward Flatcars

Fig. A.17

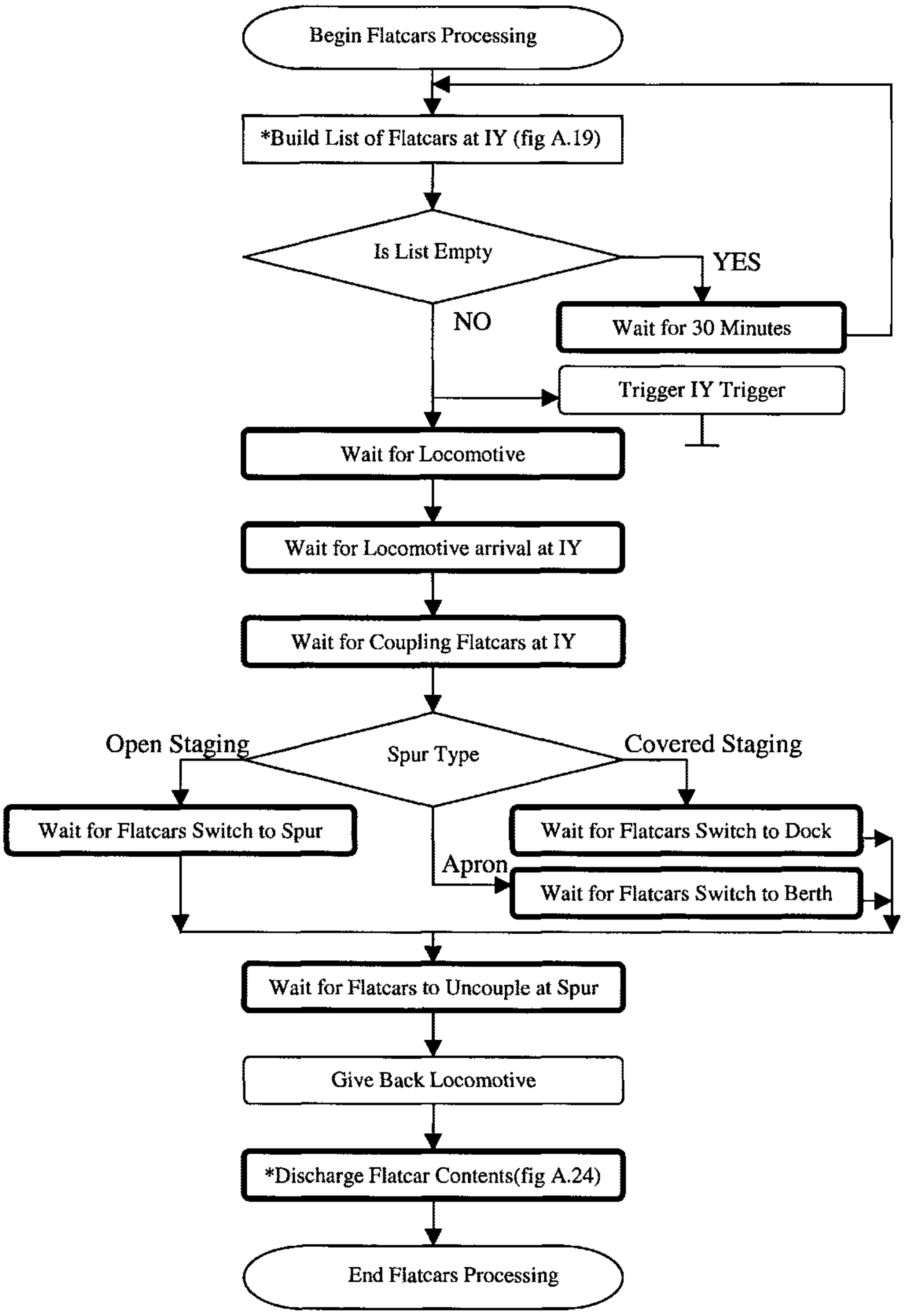

Begin Flatcars Processing
*Build List of Flatcars at IY (fig A.19)
Is List Empty
YES
NO
Wait for 30 Minutes
Trigger IY Trigger
Wait for Locomotive
Wait for Locomotive arrival at IY
Wait for Coupling Flatcars at IY
Spur Type
Open Staging
Covered Staging
Apron
Wait for Flatcars Switch to Spur
Wait for Flatcars Switch to Dock
Wait for Flatcars Switch to Berth
Wait for Flatcars to Uncouple at Spur
Give Back Locomotive
*Discharge Flatcar Contents(fig A.24)
End Flatcars Processing

Fig. A.18

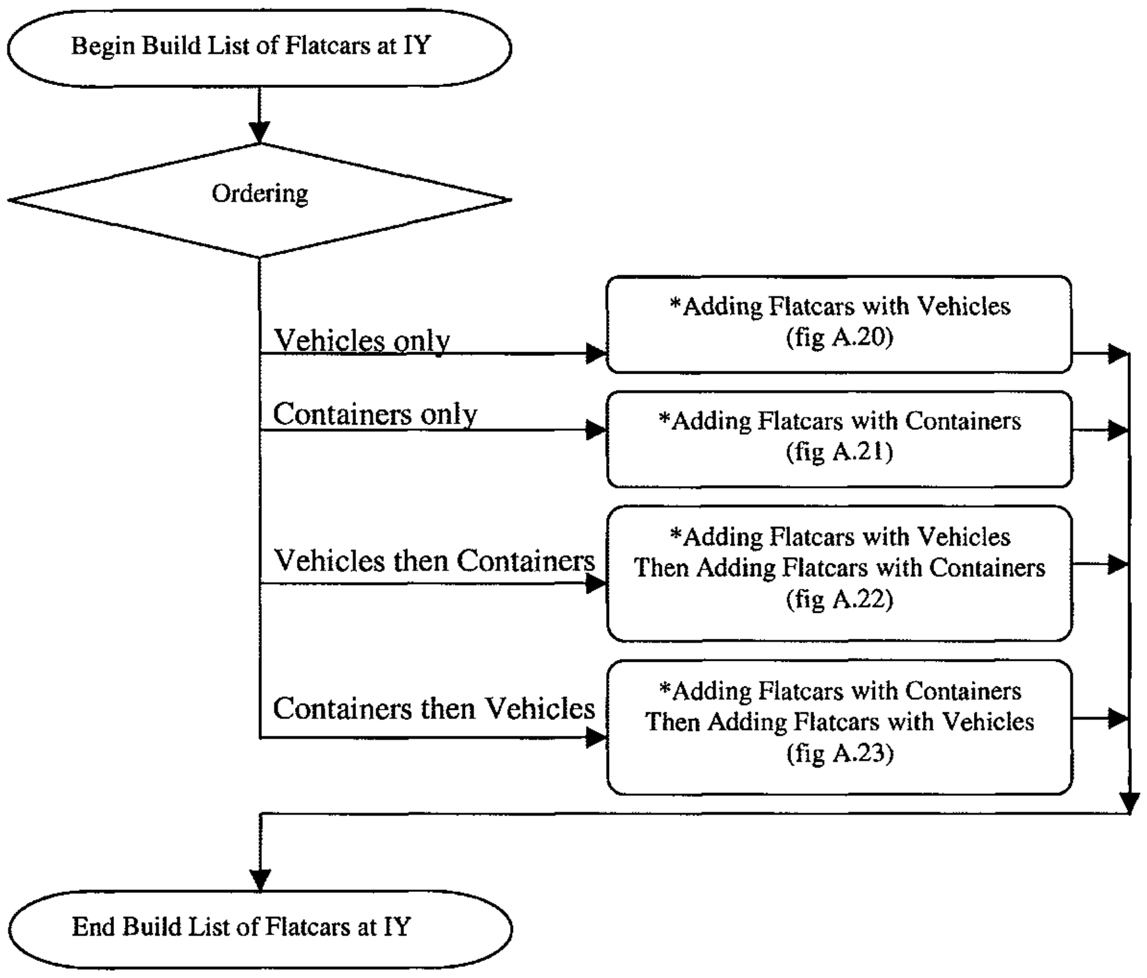

Begin Build List of Flatcars at IY
Ordering
Vehicles only
Containers only
Vehicles then Containers
Containers then Vehicles
*Adding Flatcars with Vehicles (fig A.20)
*Adding Flatcars with Containers (fig A.21)
*Adding Flatcars with Vehicles Then Adding Flatcars with Containers (fig A.22)
*Adding Flatcars with Containers Then Adding Flatcars with Vehicles (fig A.23)
End Build List of Flatcars at IY

Fig. A.19

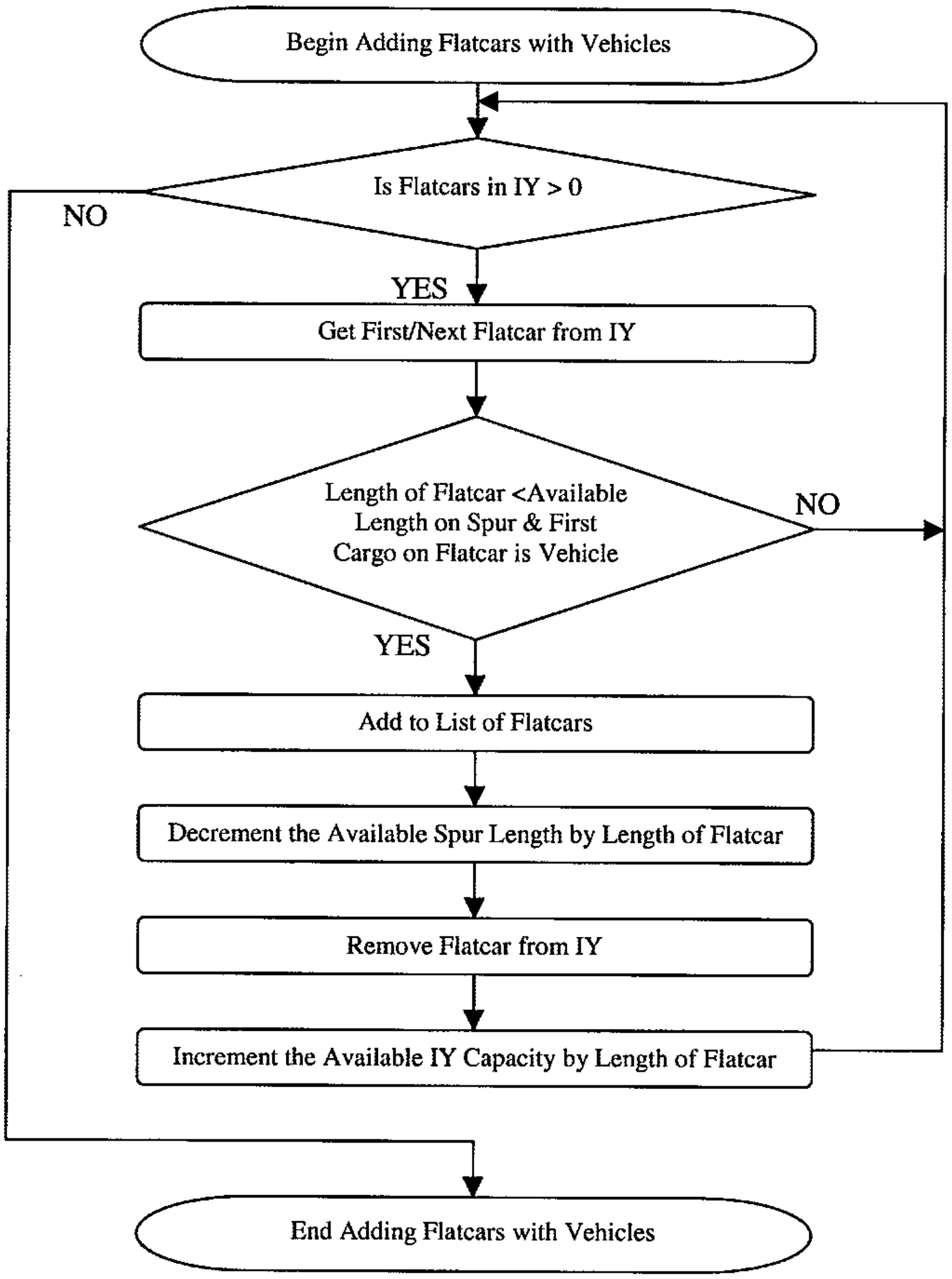

Begin Adding Flatcars with Vehicles
Is Flatcars in IY > 0
NO
YES
Get First/Next Flatcar from IY
Length of Flatcar <Available Length on Spur & First Cargo on Flatcar is Vehicle
NO
YES
Add to List of Flatcars
Decrement the Available Spur Length by Length of Flatcar
Remove Flatcar from IY
Increment the Available IY Capacity by Length of Flatcar
End Adding Flatcars with Vehicles

Fig. A.20

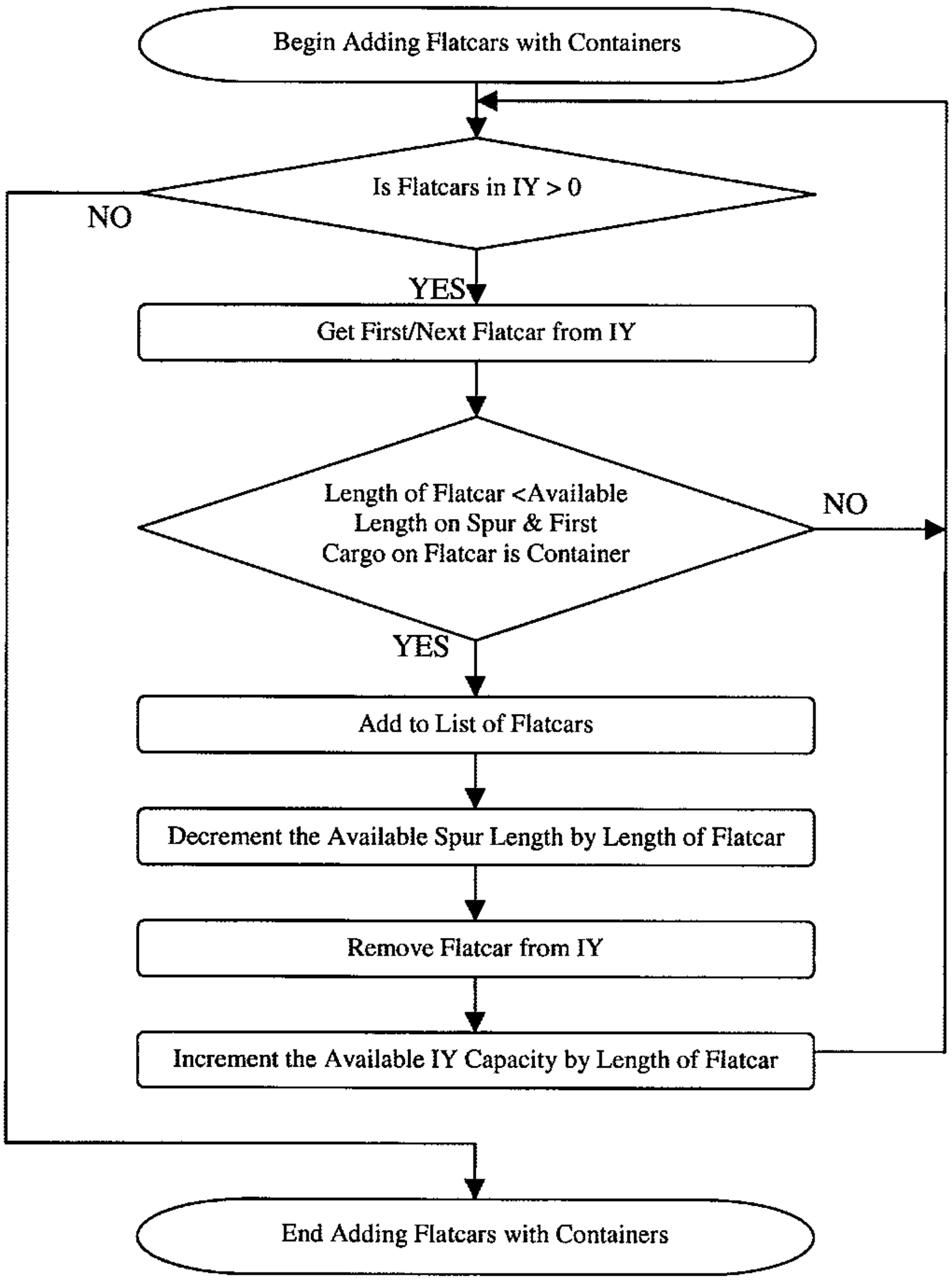

Begin Adding Flatcars with Containers
Is Flatcars in IY > 0
NO
YES
Get First/Next Flatcar from IY
Length of Flatcar <Available Length on Spur & First Cargo on Flatcar is Container
NO
YES
Add to List of Flatcars
Decrement the Available Spur Length by Length of Flatcar
Remove Flatcar from IY
Increment the Available IY Capacity by Length of Flatcar
End Adding Flatcars with Containers

Fig. A.21

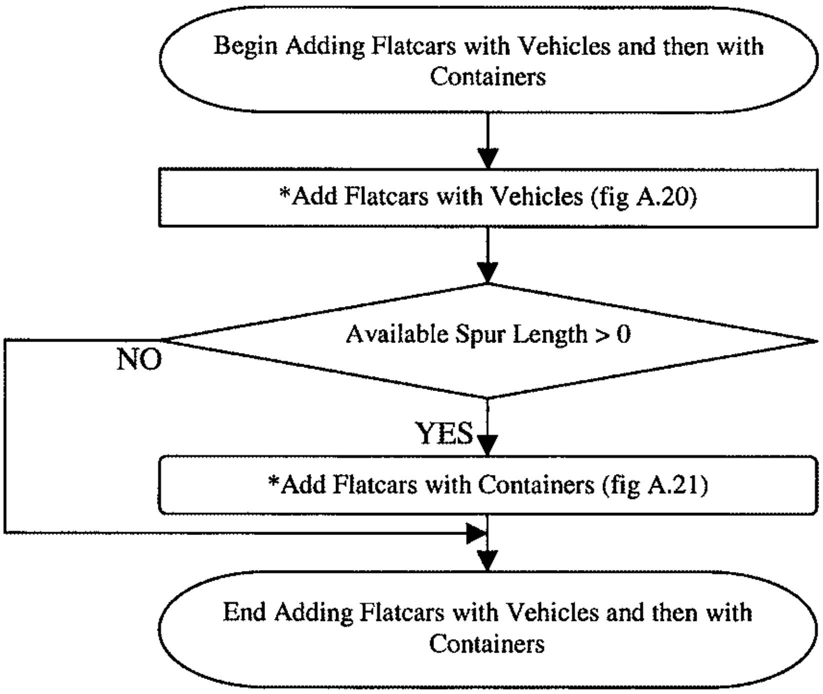

Begin Adding Flatcars with Vehicles and then with Containers
*Add Flatcars with Vehicles (fig A.20)
Available Spur Length > 0
NO
YES
*Add Flatcars with Containers (fig A.21)
End Adding Flatcars with Vehicles and then with Containers

Fig. A.22

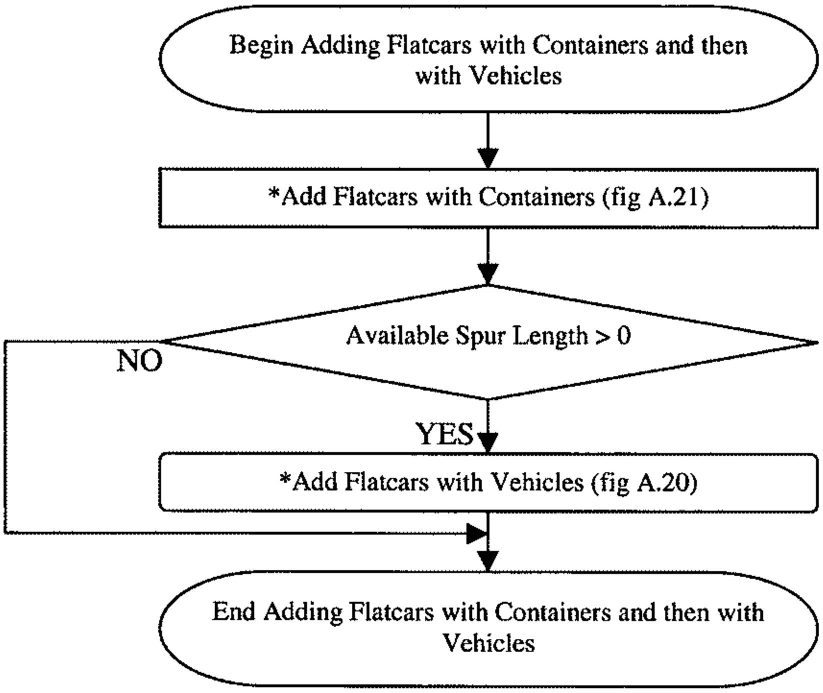

Begin Adding Flatcars with Containers and then with Vehicles
*Add Flatcars with Containers (fig A.21)
Available Spur Length > 0
NO
YES
*Add Flatcars with Vehicles (fig A.20)
End Adding Flatcars with Containers and then with Vehicles

Fig. A.23

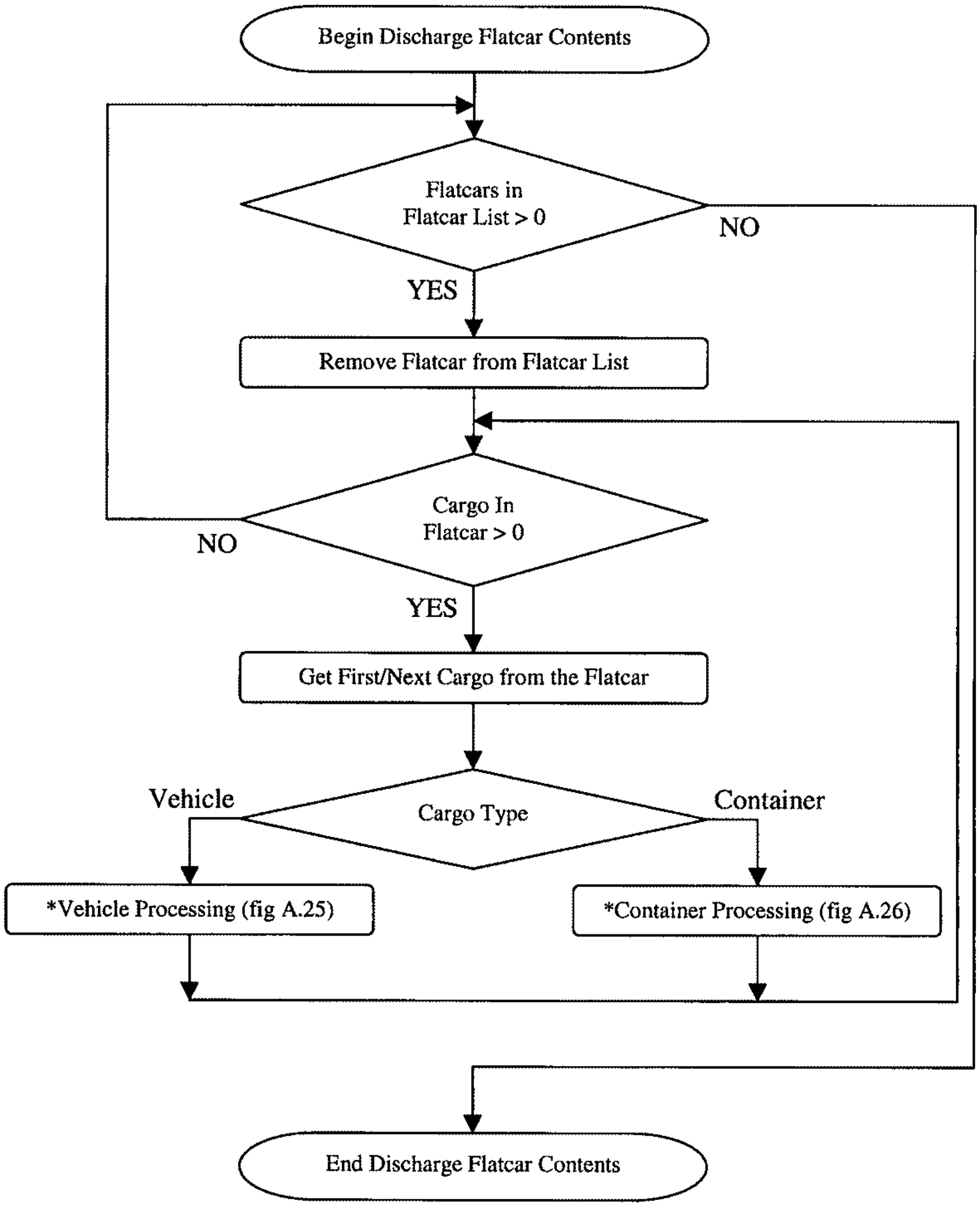

Begin Discharge Flatcar Contents
Flatcars in Flatcar List > 0
NO
YES
Remove Flatcar from Flatcar List
Cargo In Flatcar > 0
NO
YES
Get First/Next Cargo from the Flatcar
Cargo Type
Vehicle
Container
*Vehicle Processing (fig A.25)
*Container Processing (fig A.26)
End Discharge Flatcar Contents

Fig. A.24

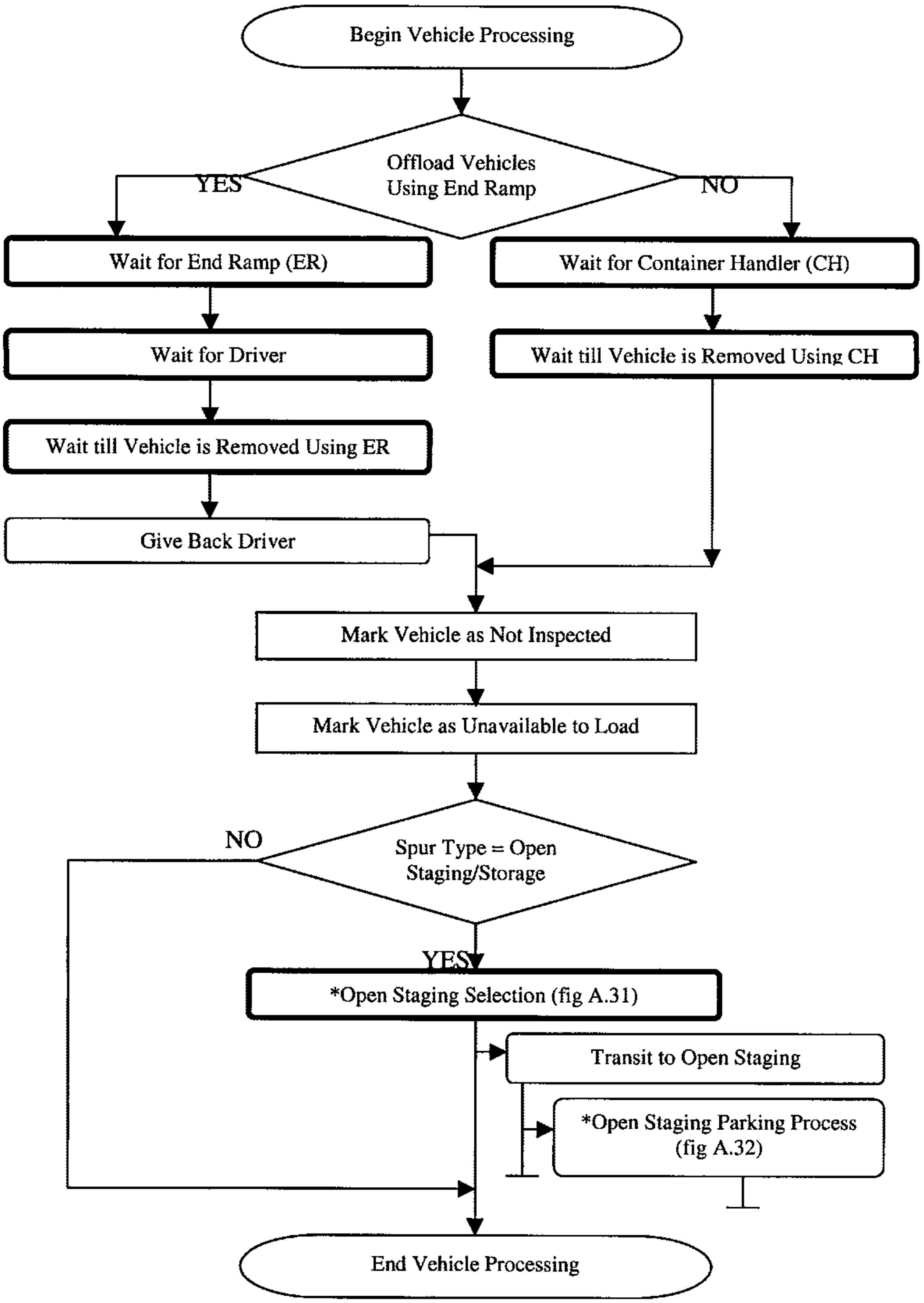

Begin Vehicle Processing
Offload Vehicles Using End Ramp
YES
NO
Wait for End Ramp (ER)
Wait for Driver
Wait till Vehicle is Removed Using ER
Give Back Driver
Wait for Container Handler (CH)
Wait till Vehicle is Removed Using CH
Mark Vehicle as Not Inspected
Mark Vehicle as Unavailable to Load
Spur Type = Open Staging/Storage
NO
YES
*Open Staging Selection (fig A.31)
Transit to Open Staging
*Open Staging Parking Process (fig A.32)
End Vehicle Processing

Fig. A.25

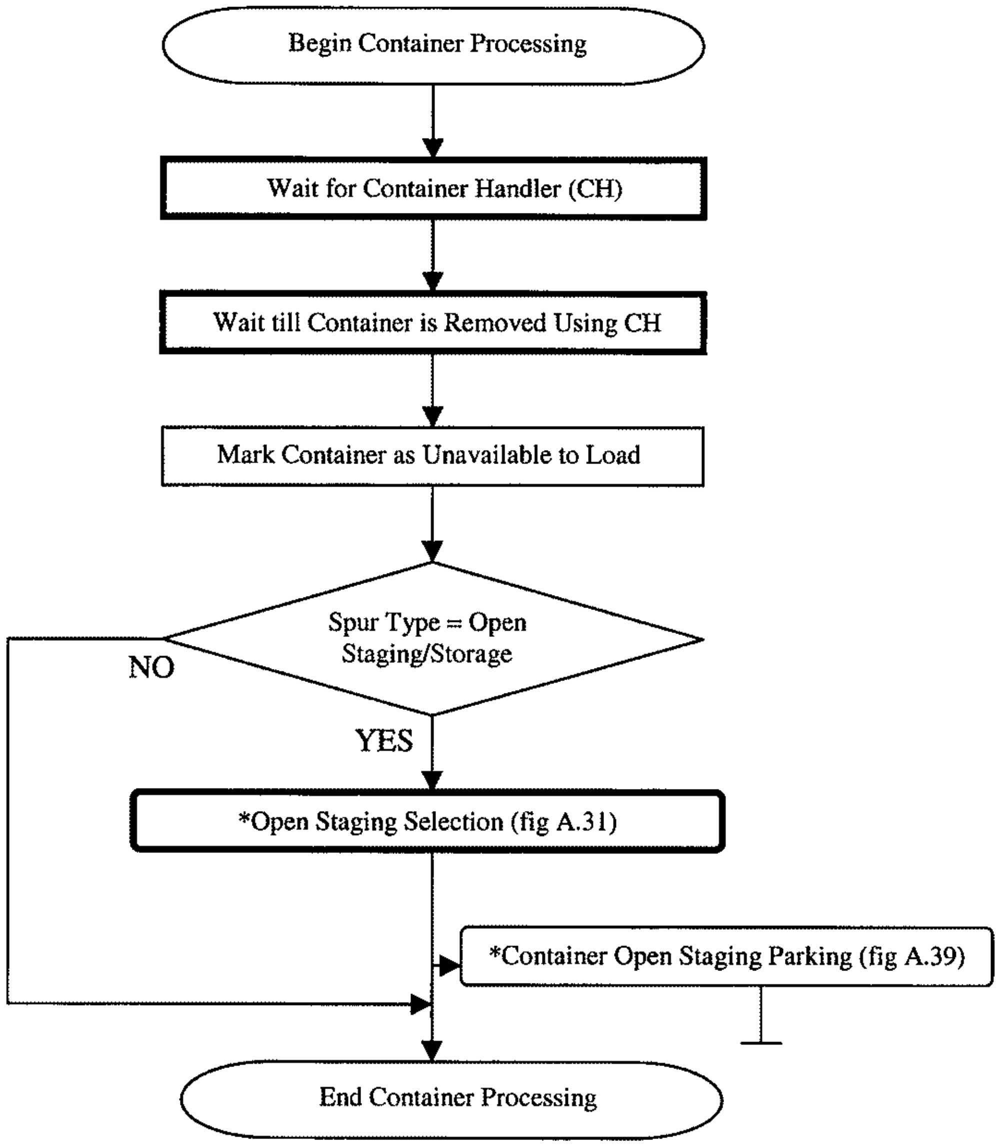

Begin Container Processing
Wait for Container Handler (CH)
Wait till Container is Removed Using CH
Mark Container as Unavailable to Load
Spur Type = Open Staging/Storage
NO
YES
*Open Staging Selection (fig A.31)
*Container Open Staging Parking (fig A.39)
End Container Processing

Fig. A.26

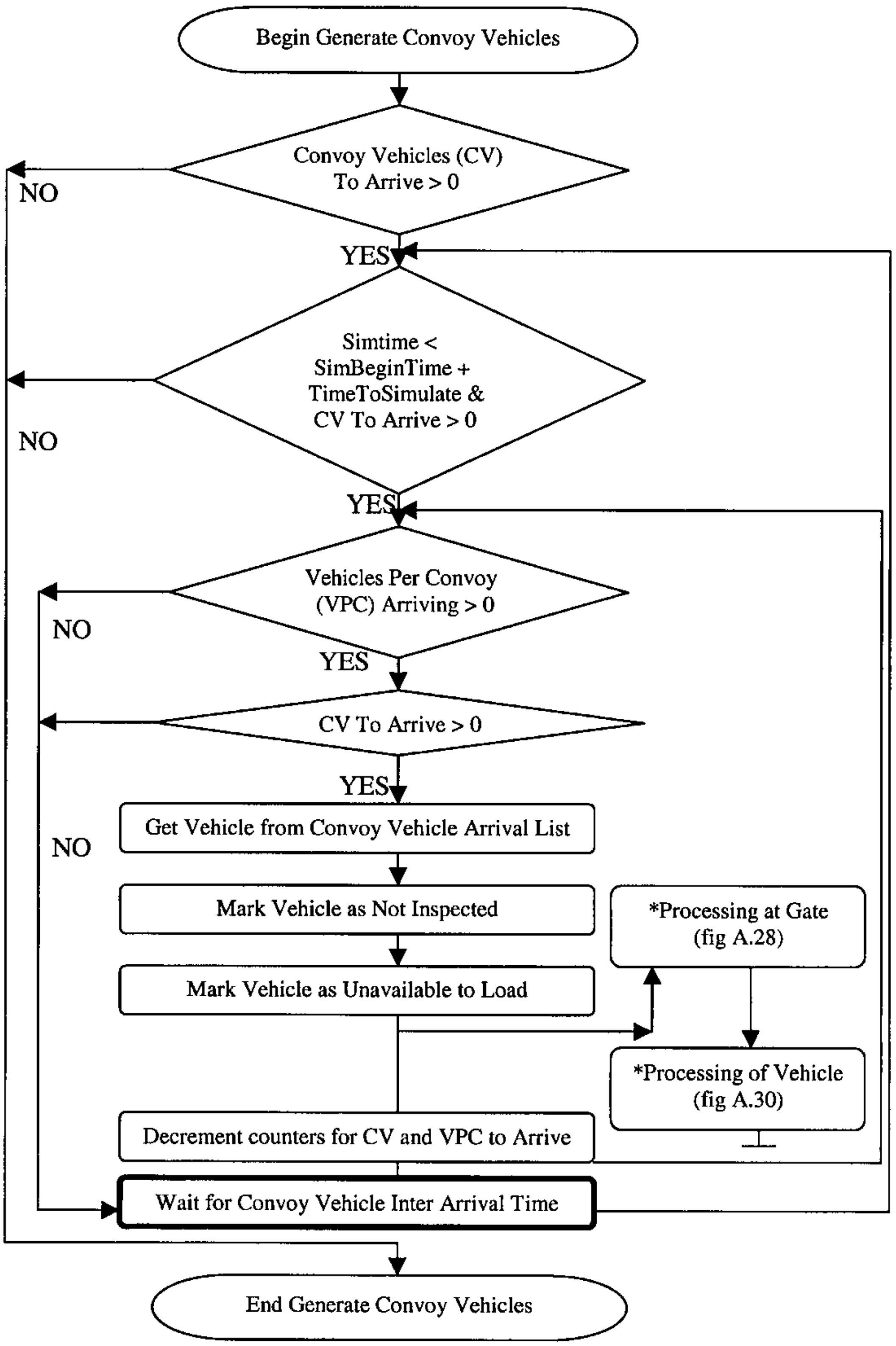

Begin Generate Convoy Vehicles
Convoy Vehicles (CV) To Arrive > 0
NO
YES
Simtime < SimBeginTime + TimeToSimulate & CV To Arrive > 0
NO
YES
Vehicles Per Convoy (VPC) Arriving > 0
NO
YES
CV To Arrive > 0
YES
Get Vehicle from Convoy Vehicle Arrival List
NO
Mark Vehicle as Not Inspected
*Processing at Gate (fig A.28)
Mark Vehicle as Unavailable to Load
*Processing of Vehicle (fig A.30)
Decrement counters for CV and VPC to Arrive
Wait for Convoy Vehicle Inter Arrival Time
End Generate Convoy Vehicles

Fig. A.27

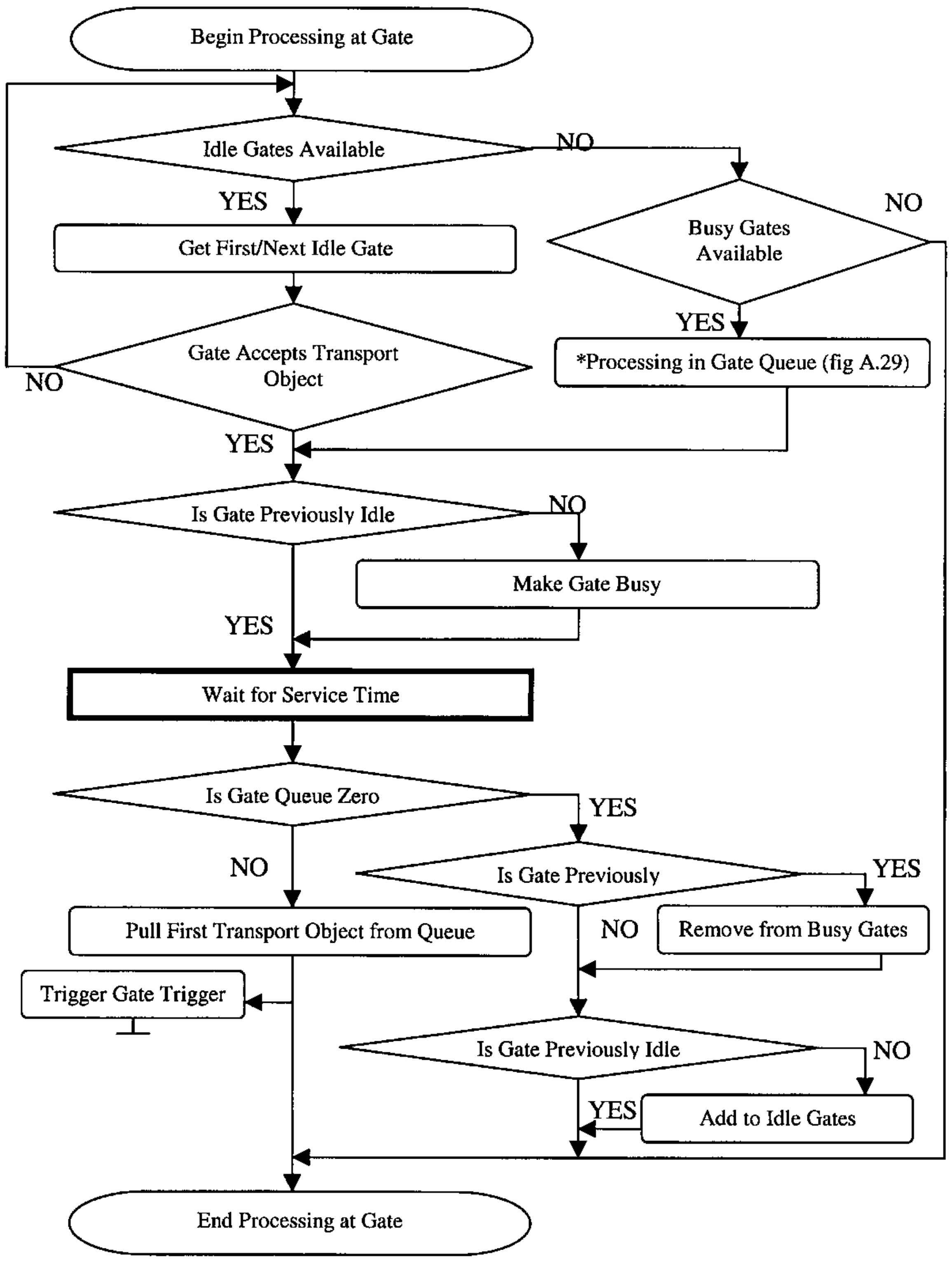

Begin Processing at Gate
Idle Gates Available
NO
YES
Get First/Next Idle Gate
Busy Gates Available
NO
YES
Gate Accepts Transport Object
NO
*Processing in Gate Queue (fig A.29)
YES
Is Gate Previously Idle
NO
Make Gate Busy
YES
Wait for Service Time
Is Gate Queue Zero
YES
NO
Is Gate Previously
YES
Pull First Transport Object from Queue
NO
Remove from Busy Gates
Trigger Gate Trigger
Is Gate Previously Idle
NO
YES
Add to Idle Gates
End Processing at Gate

Fig. A.28

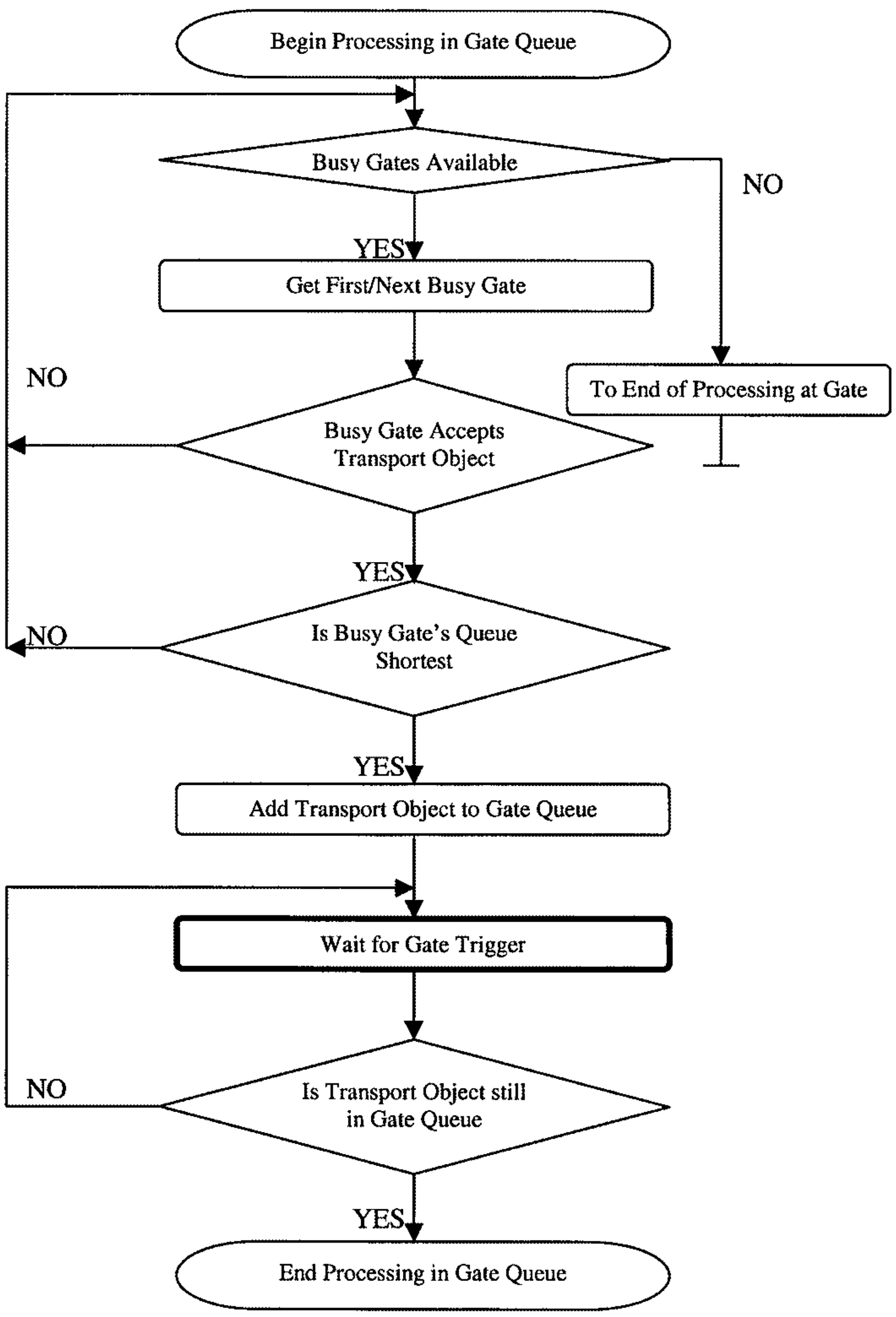

Begin Processing in Gate Queue
Busy Gates Available
NO
YES
Get First/Next Busy Gate
To End of Processing at Gate
NO
Busy Gate Accepts Transport Object
YES
NO
Is Busy Gate's Queue Shortest
YES
Add Transport Object to Gate Queue
Wait for Gate Trigger
NO
Is Transport Object still in Gate Queue
YES
End Processing in Gate Queue

Fig. A.29

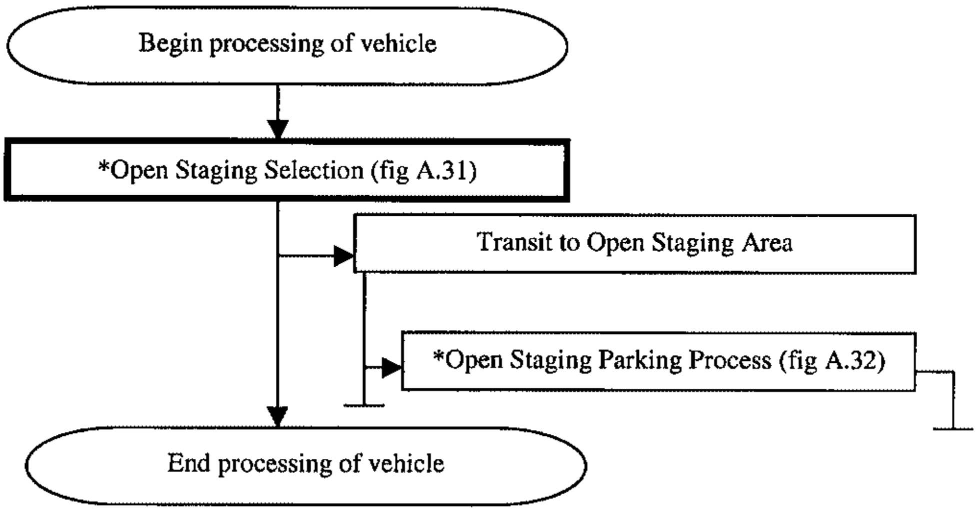

Begin processing of vehicle
*Open Staging Selection (fig A.31)
Transit to Open Staging Area
*Open Staging Parking Process (fig A.32)
End processing of vehicle

Fig. A.30

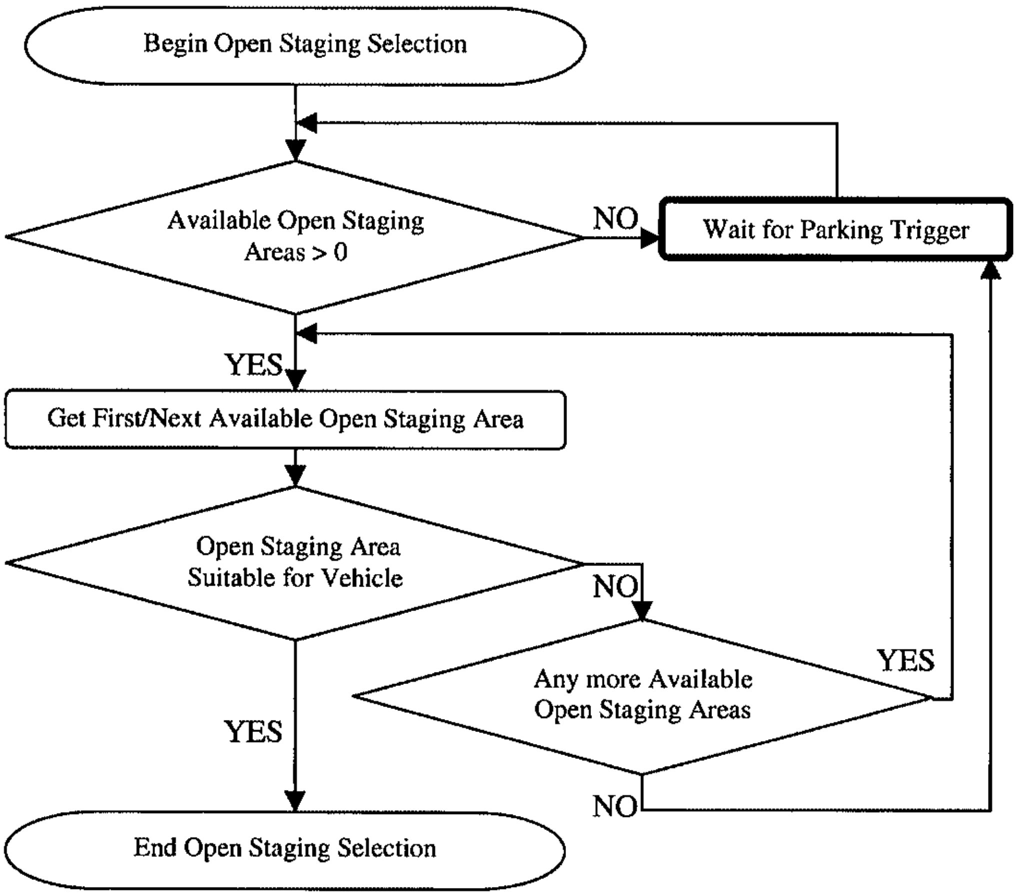

Begin Open Staging Selection
Available Open Staging Areas > 0
NO
Wait for Parking Trigger
YES
Get First/Next Available Open Staging Area
Open Staging Area Suitable for Vehicle
NO
YES
Any more Available Open Staging Areas
YES
NO
End Open Staging Selection

Fig. A.31

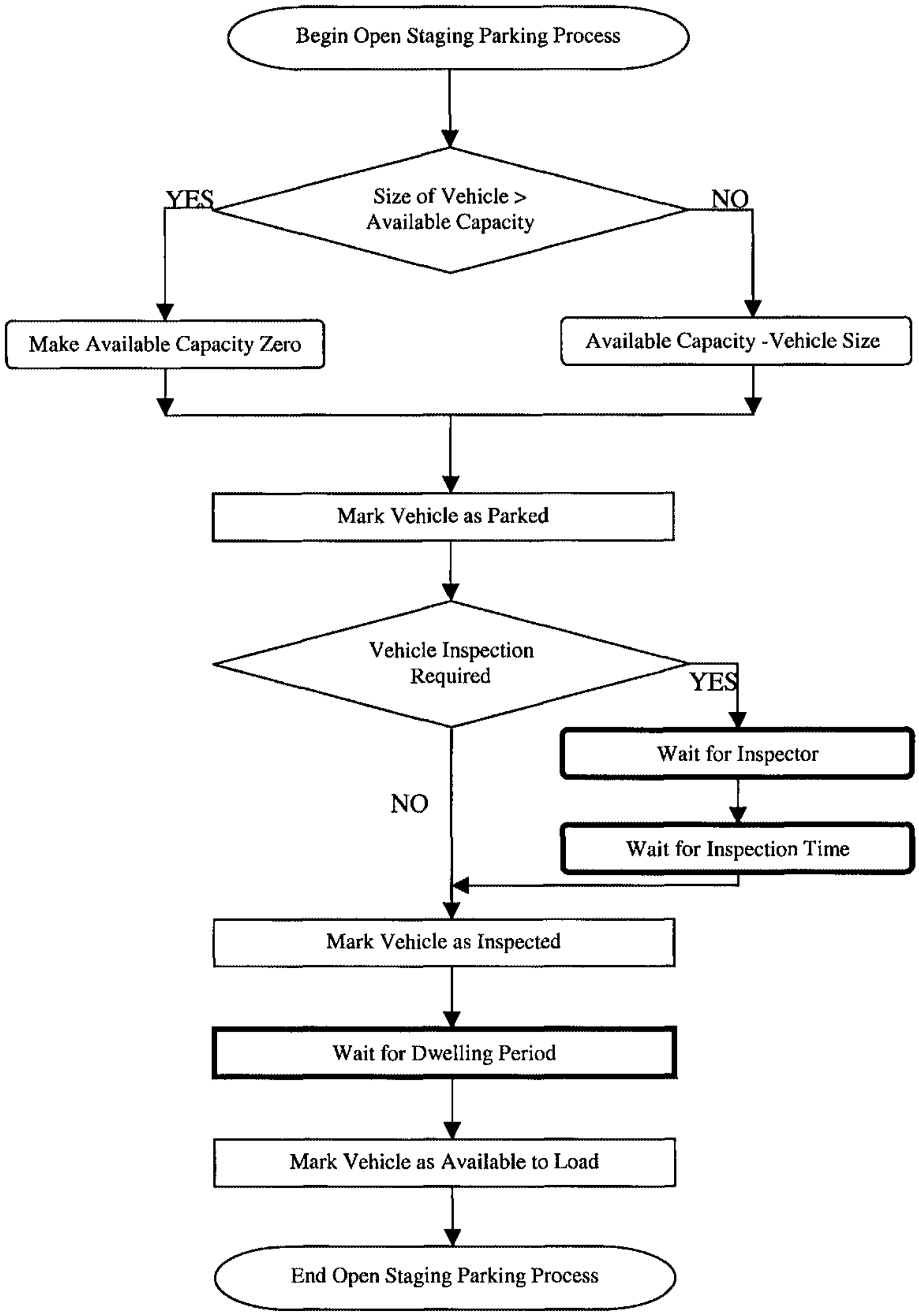

Begin Open Staging Parking Process
Size of Vehicle > Available Capacity
YES
NO
Make Available Capacity Zero
Available Capacity -Vehicle Size
Mark Vehicle as Parked
Vehicle Inspection Required
YES
NO
Wait for Inspector
Wait for Inspection Time
Mark Vehicle as Inspected
Wait for Dwelling Period
Mark Vehicle as Available to Load
End Open Staging Parking Process

Fig. A.32

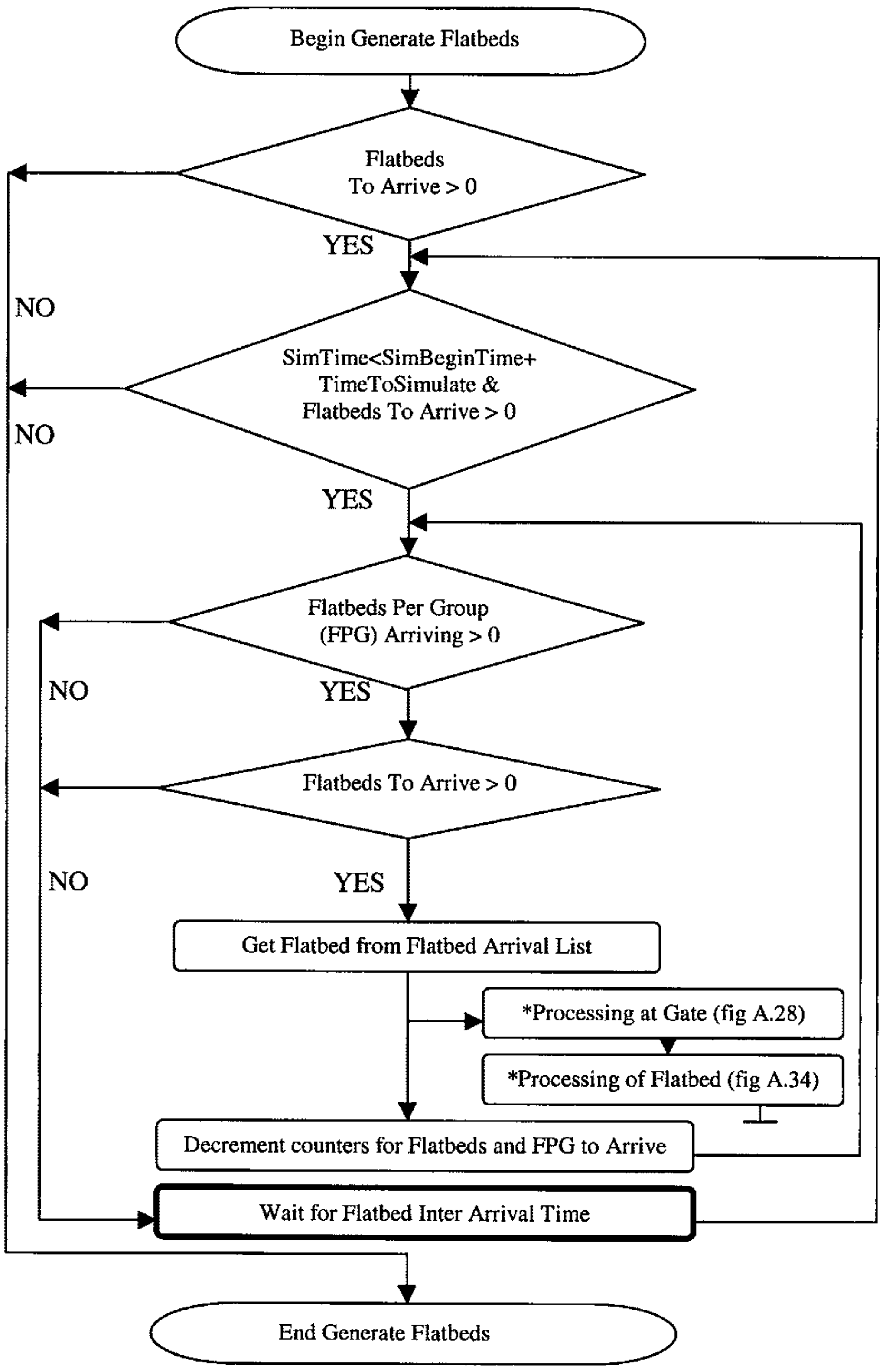

Begin Generate Flatbeds
Flatbeds To Arrive > 0
YES
NO
SimTime<SimBeginTime+ TimeToSimulate & Flatbeds To Arrive > 0
NO
YES
Flatbeds Per Group (FPG) Arriving > 0
NO
YES
Flatbeds To Arrive > 0
NO
YES
Get Flatbed from Flatbed Arrival List
*Processing at Gate (fig A.28)
*Processing of Flatbed (fig A.34)
Decrement counters for Flatbeds and FPG to Arrive
Wait for Flatbed Inter Arrival Time
End Generate Flatbeds

Fig. A.33

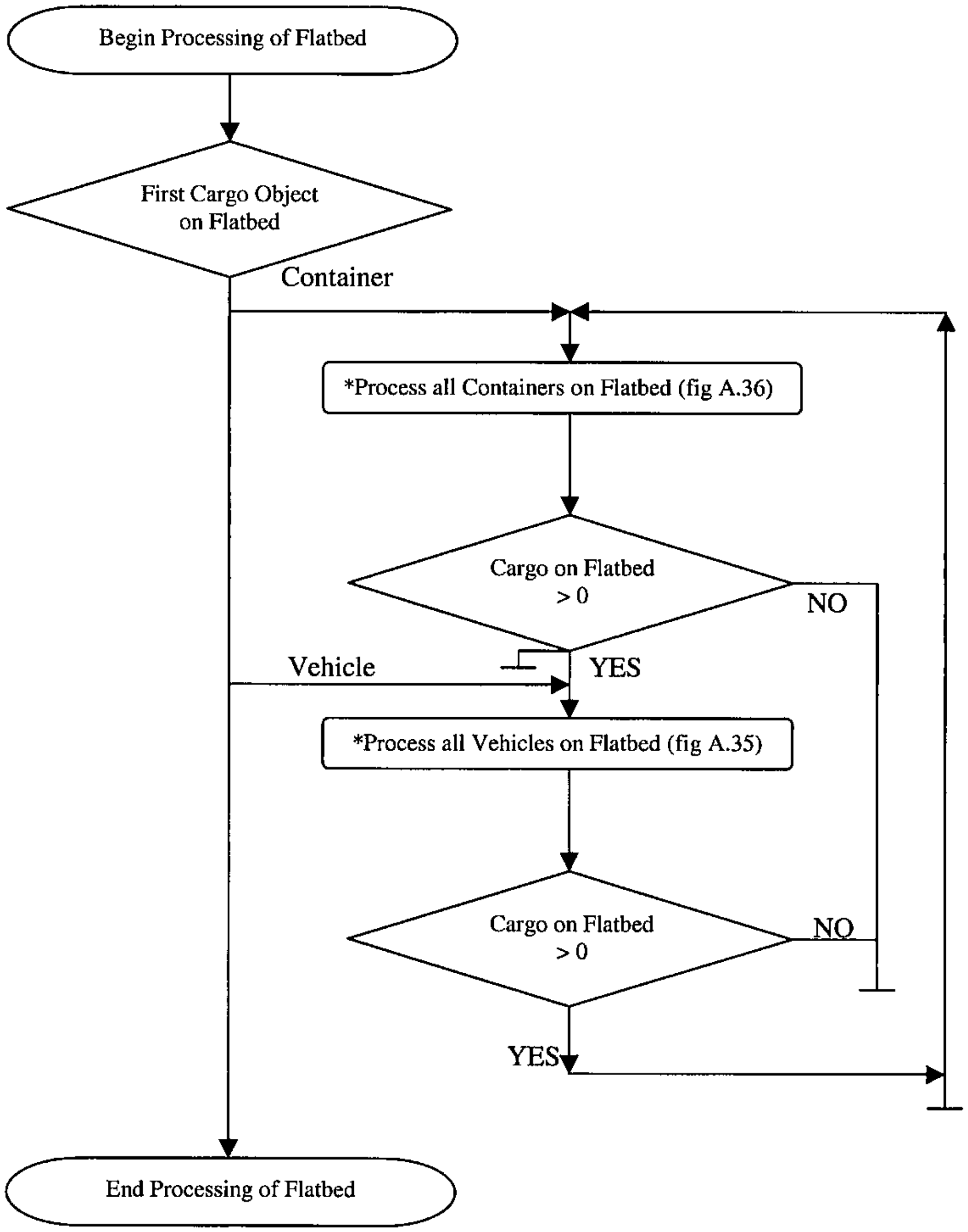

Begin Processing of Flatbed
First Cargo Object on Flatbed
Container
*Process all Containers on Flatbed (fig A.36)
Cargo on Flatbed > 0
NO
YES
Vehicle
*Process all Vehicles on Flatbed (fig A.35)
Cargo on Flatbed > 0
NO
YES
End Processing of Flatbed

Fig. A.34

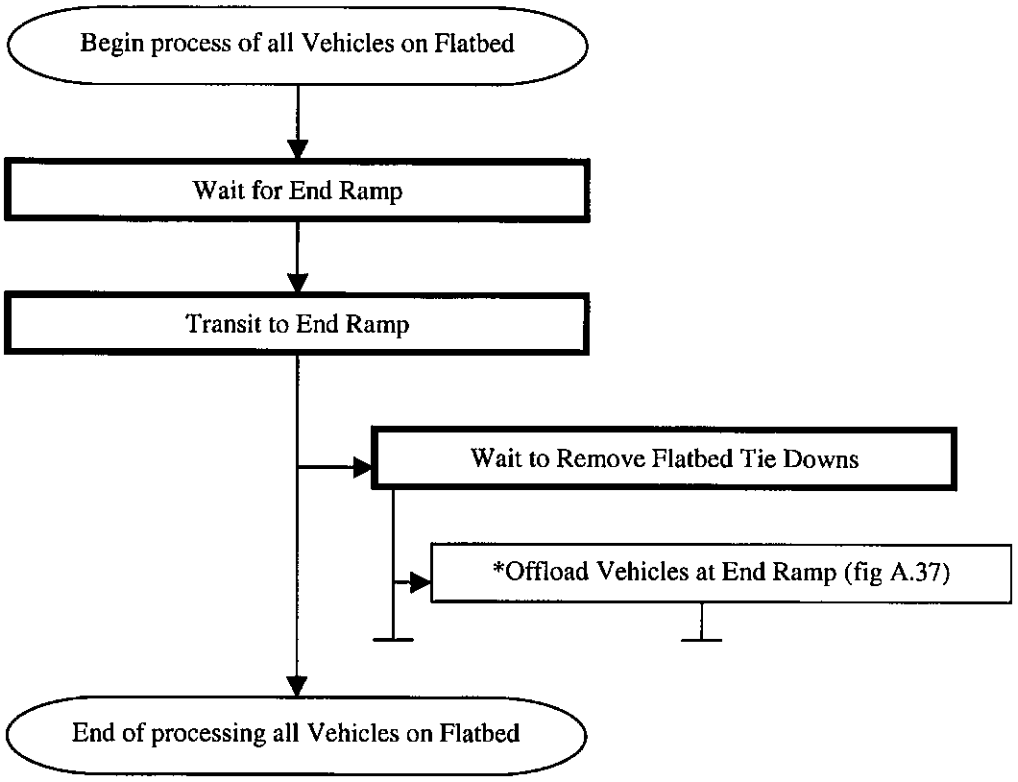

Begin process of all Vehicles on Flatbed
Wait for End Ramp
Transit to End Ramp
Wait to Remove Flatbed Tie Downs
*Offload Vehicles at End Ramp (fig A.37)
End of processing all Vehicles on Flatbed

Fig. A.35

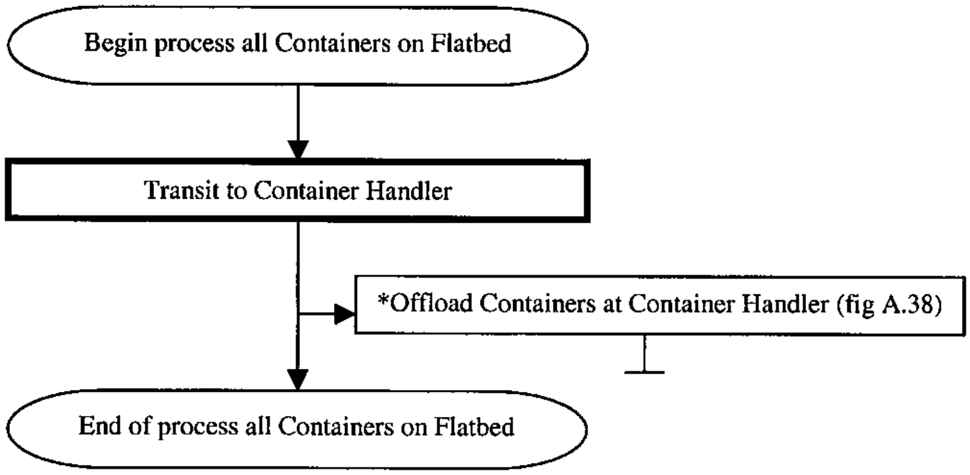

Begin process all Containers on Flatbed
Transit to Container Handler
*Offload Containers at Container Handler (fig A.38)
End of process all Containers on Flatbed

Fig. A.36

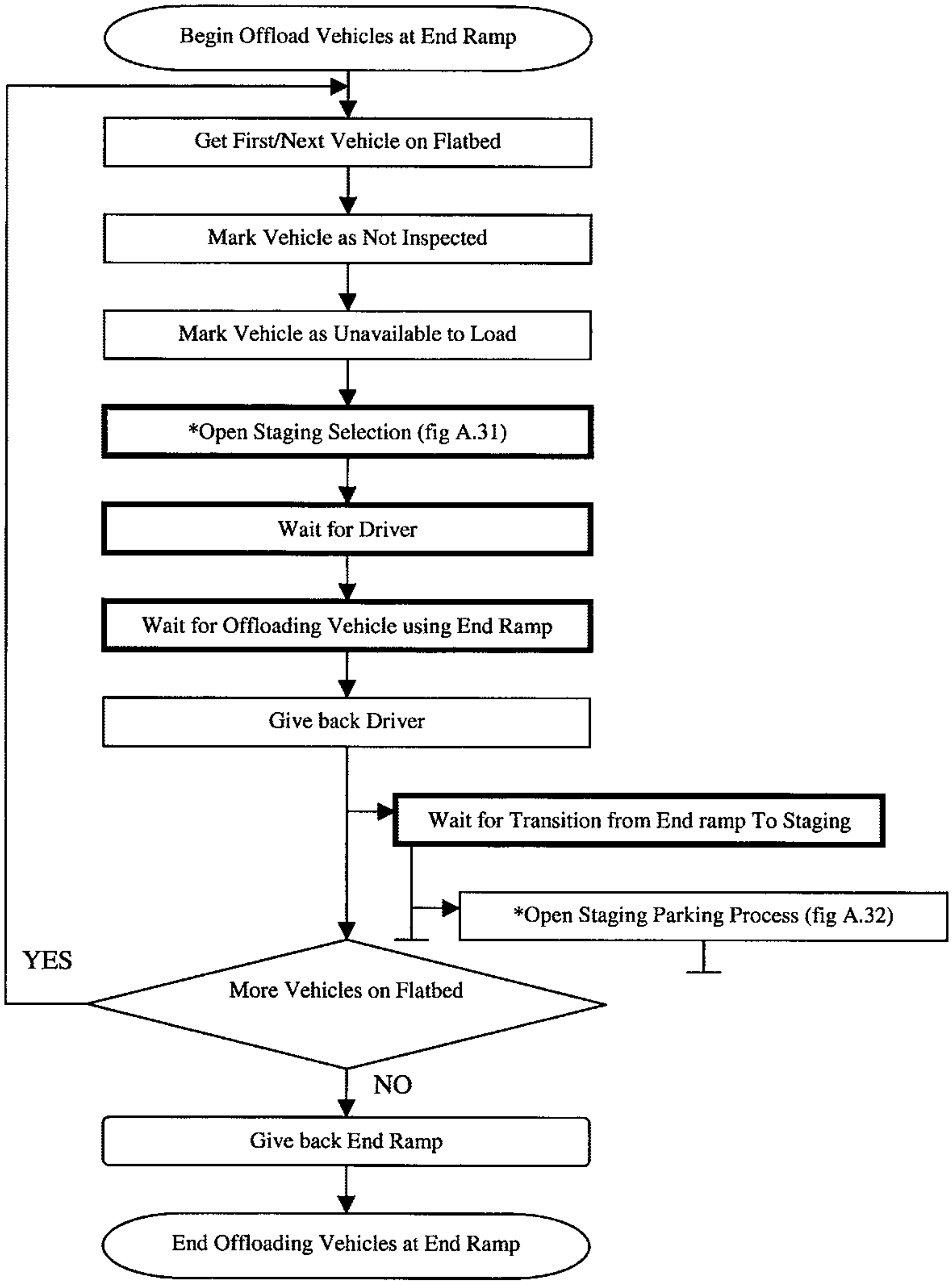

Begin Offload Vehicles at End Ramp
Get First/Next Vehicle on Flatbed
Mark Vehicle as Not Inspected
Mark Vehicle as Unavailable to Load
*Open Staging Selection (fig A.31)
Wait for Driver
Wait for Offloading Vehicle using End Ramp
Give back Driver
Wait for Transition from End ramp To Staging
*Open Staging Parking Process (fig A.32)
YES
More Vehicles on Flatbed
NO
Give back End Ramp
End Offloading Vehicles at End Ramp

Fig. A.37

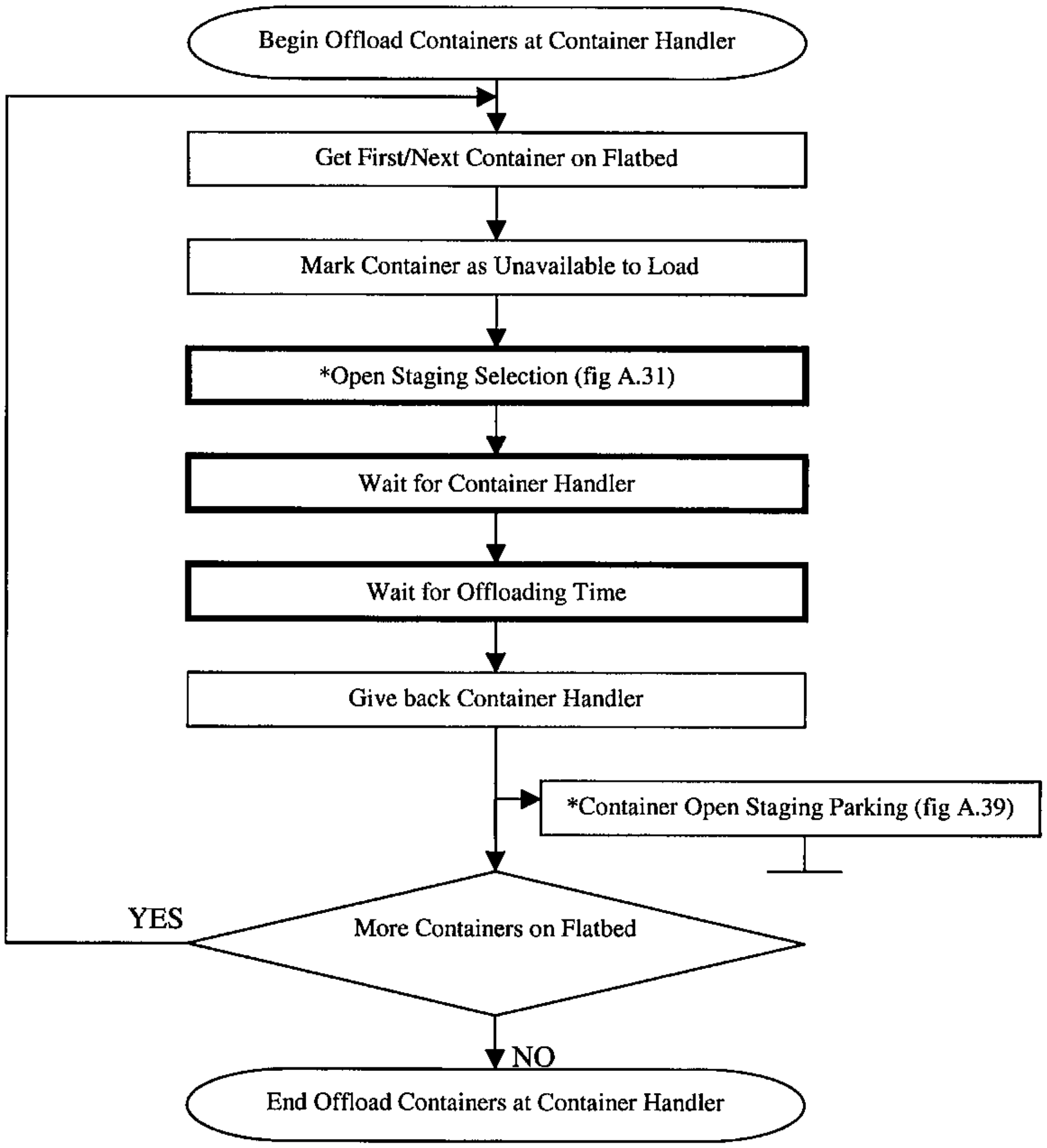

Begin Offload Containers at Container Handler
Get First/Next Container on Flatbed
Mark Container as Unavailable to Load
*Open Staging Selection (fig A.31)
Wait for Container Handler
Wait for Offloading Time
Give back Container Handler
*Container Open Staging Parking (fig A.39)
More Containers on Flatbed
YES
NO
End Offload Containers at Container Handler

Fig. A.38

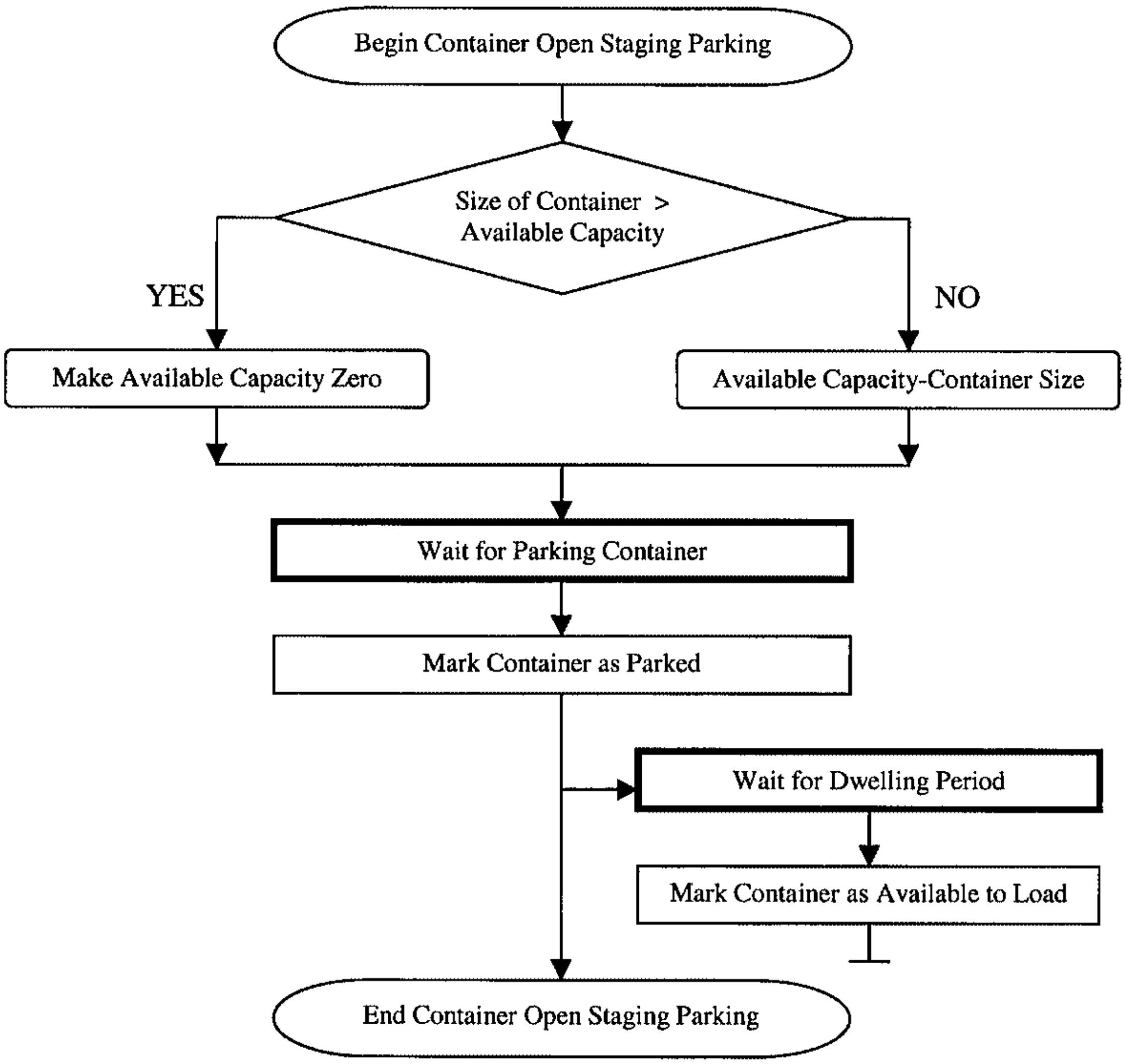

Begin Container Open Staging Parking
Size of Container > Available Capacity
YES
NO
Make Available Capacity Zero
Available Capacity-Container Size
Wait for Parking Container
Mark Container as Parked
Wait for Dwelling Period
Mark Container as Available to Load
End Container Open Staging Parking

Fig. A.39

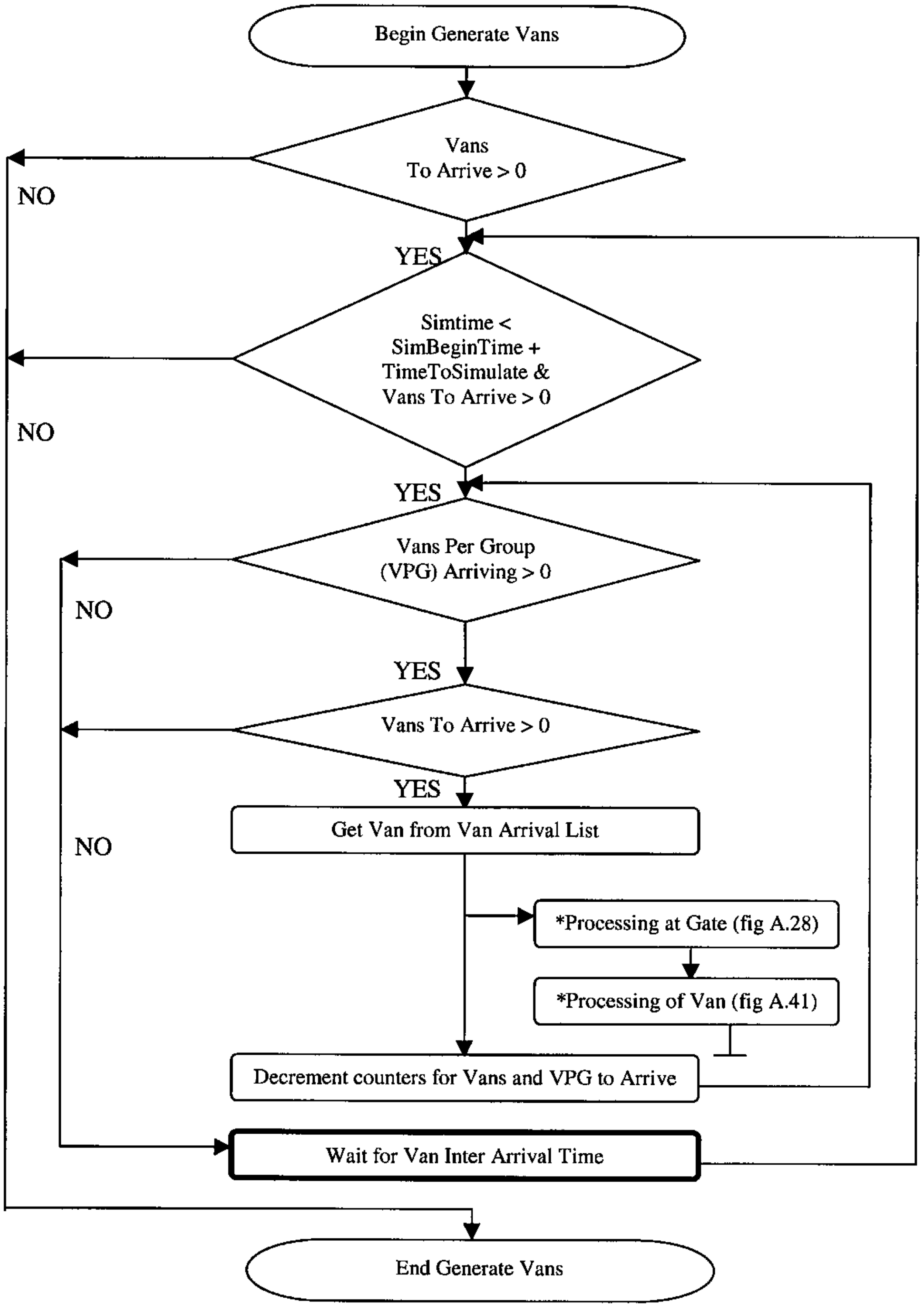

Begin Generate Vans
Vans To Arrive > 0
NO
YES
Simtime < SimBeginTime + TimeToSimulate & Vans To Arrive > 0
NO
YES
Vans Per Group (VPG) Arriving > 0
NO
YES
Vans To Arrive > 0
YES
NO
Get Van from Van Arrival List
*Processing at Gate (fig A.28)
*Processing of Van (fig A.41)
Decrement counters for Vans and VPG to Arrive
Wait for Van Inter Arrival Time
End Generate Vans

Fig. A.40

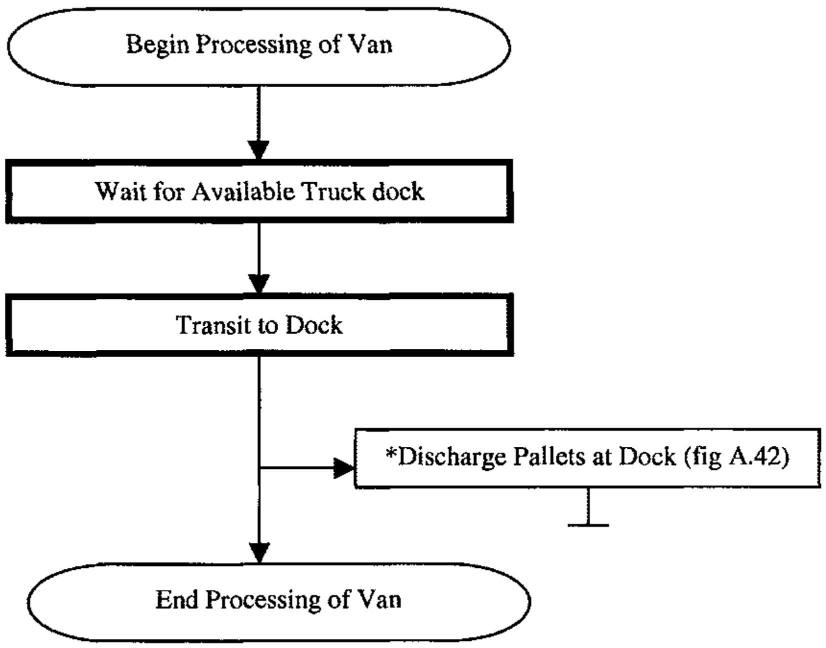

Begin Processing of Van
Wait for Available Truck dock
Transit to Dock
*Discharge Pallets at Dock (fig A.42)
End Processing of Van

Fig. A.41

In Fig. 42, the program flow for "Covered staging area selection" process is similar to that of "Open staging selection," and the program flow for "Covered staging area parking" process is similar to that of "Container Open staging parking".

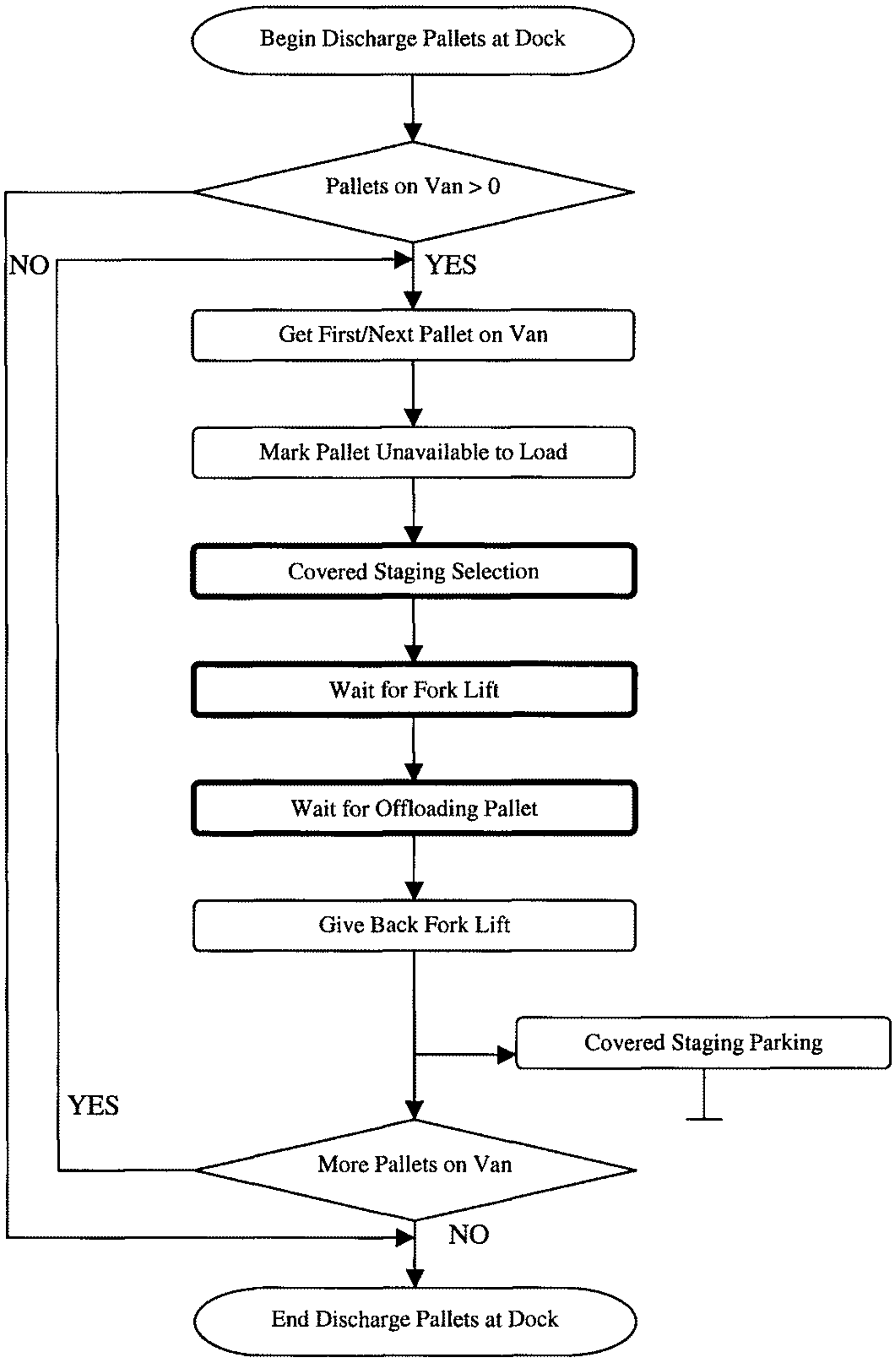
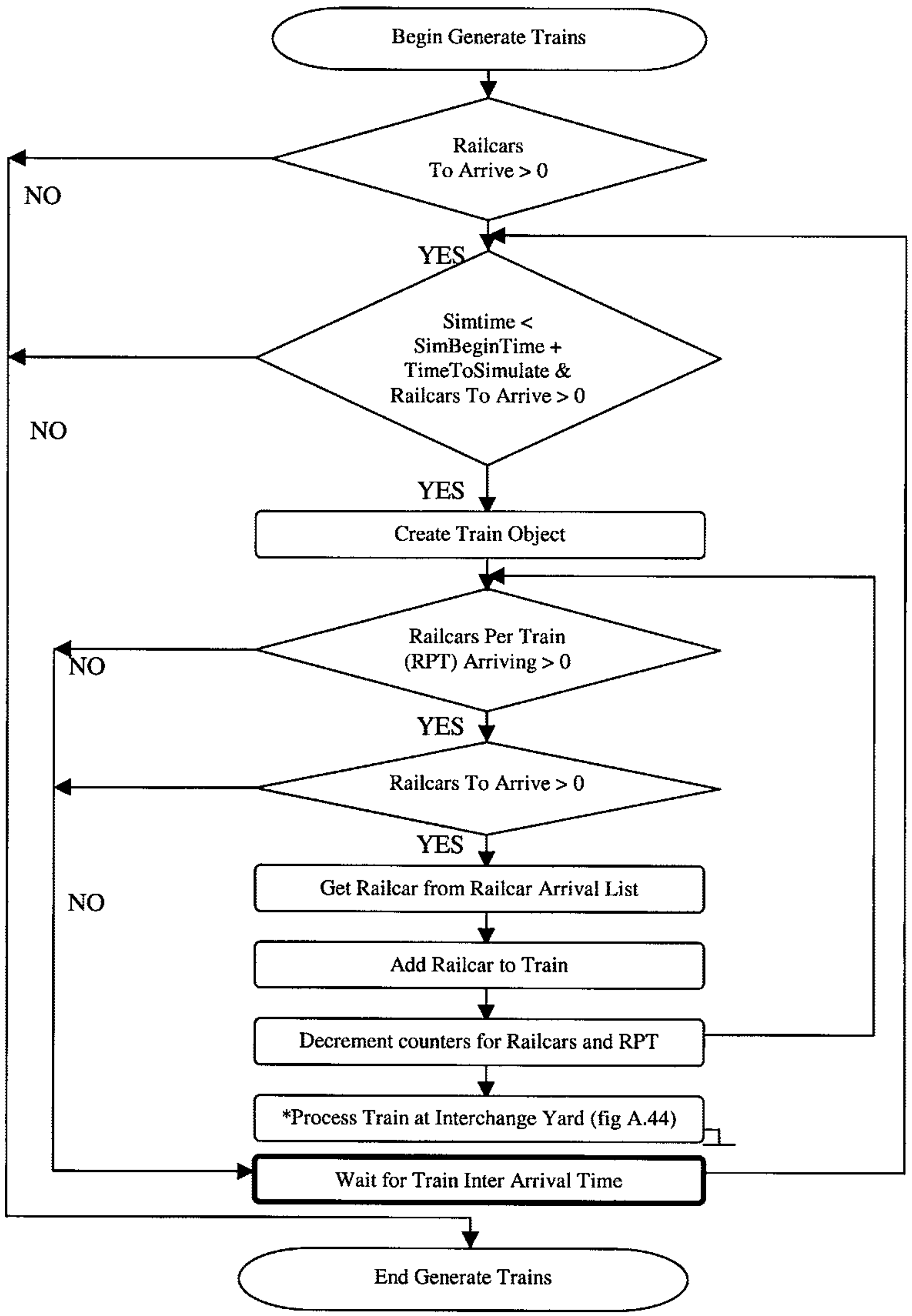

Begin Discharge Pallets at Dock
Pallets on Van > 0
NO
YES
Get First/Next Pallet on Van
Mark Pallet Unavailable to Load
Covered Staging Selection
Wait for Fork Lift
Wait for Offloading Pallet
Give Back Fork Lift
Covered Staging Parking
YES
More Pallets on Van
NO
End Discharge Pallets at Dock

Fig. A.42



Begin Generate Trains
Railcars To Arrive > 0
NO
YES
Simtime < SimBeginTime + TimeToSimulate & Railcars To Arrive > 0
NO
YES
Create Train Object
Railcars Per Train (RPT) Arriving > 0
NO
YES
Railcars To Arrive > 0
YES
NO
Get Railcar from Railcar Arrival List
Add Railcar to Train
Decrement counters for Railcars and RPT
*Process Train at Interchange Yard (fig A.44)
Wait for Train Inter Arrival Time
End Generate Trains

Fig. A.43

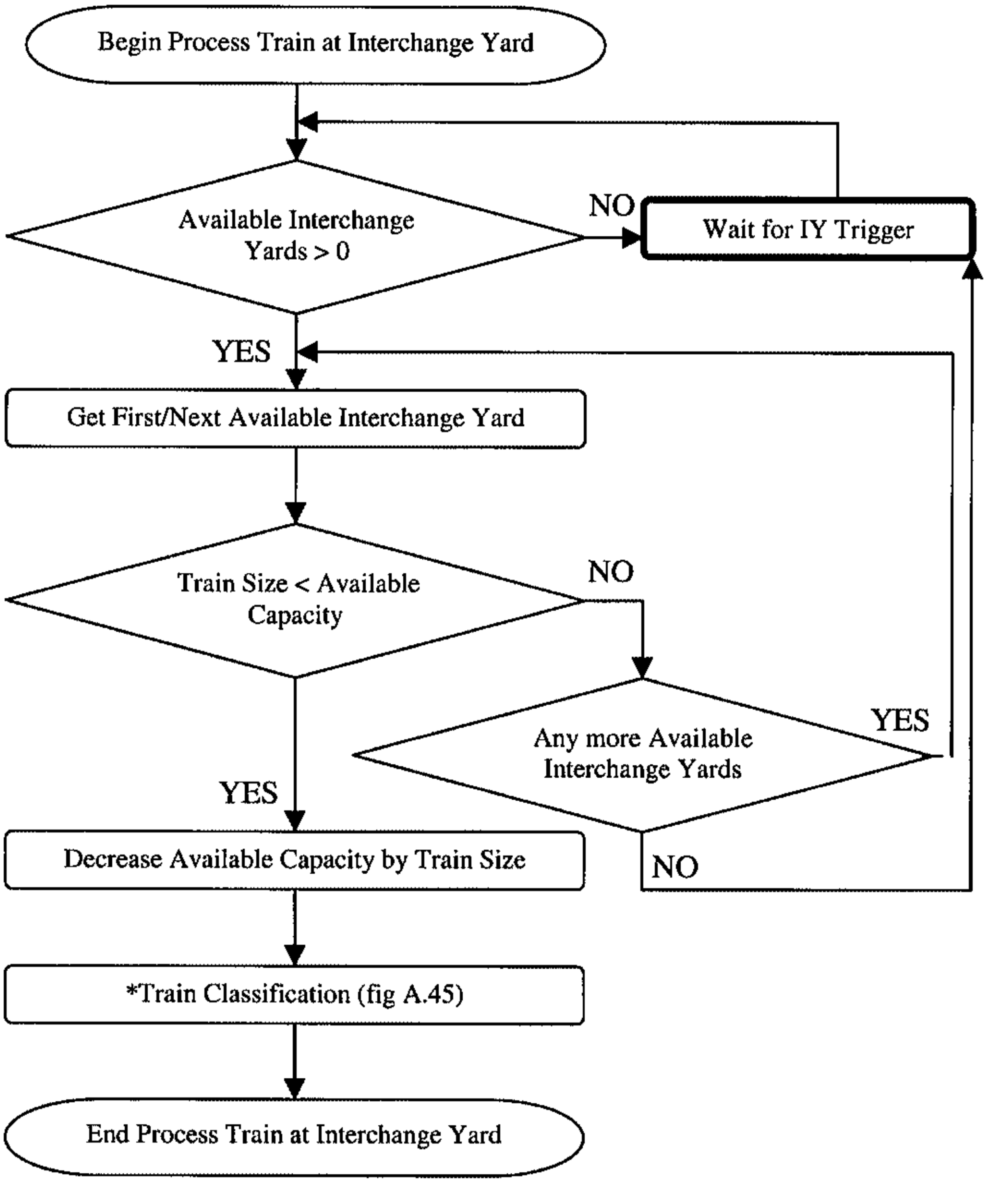

Begin Process Train at Interchange Yard
Available Interchange Yards > 0
NO
Wait for IY Trigger
YES
Get First/Next Available Interchange Yard
Train Size < Available Capacity
NO
Any more Available Interchange Yards
YES
YES
Decrease Available Capacity by Train Size
NO
*Train Classification (fig A.45)
End Process Train at Interchange Yard

Fig. A.44

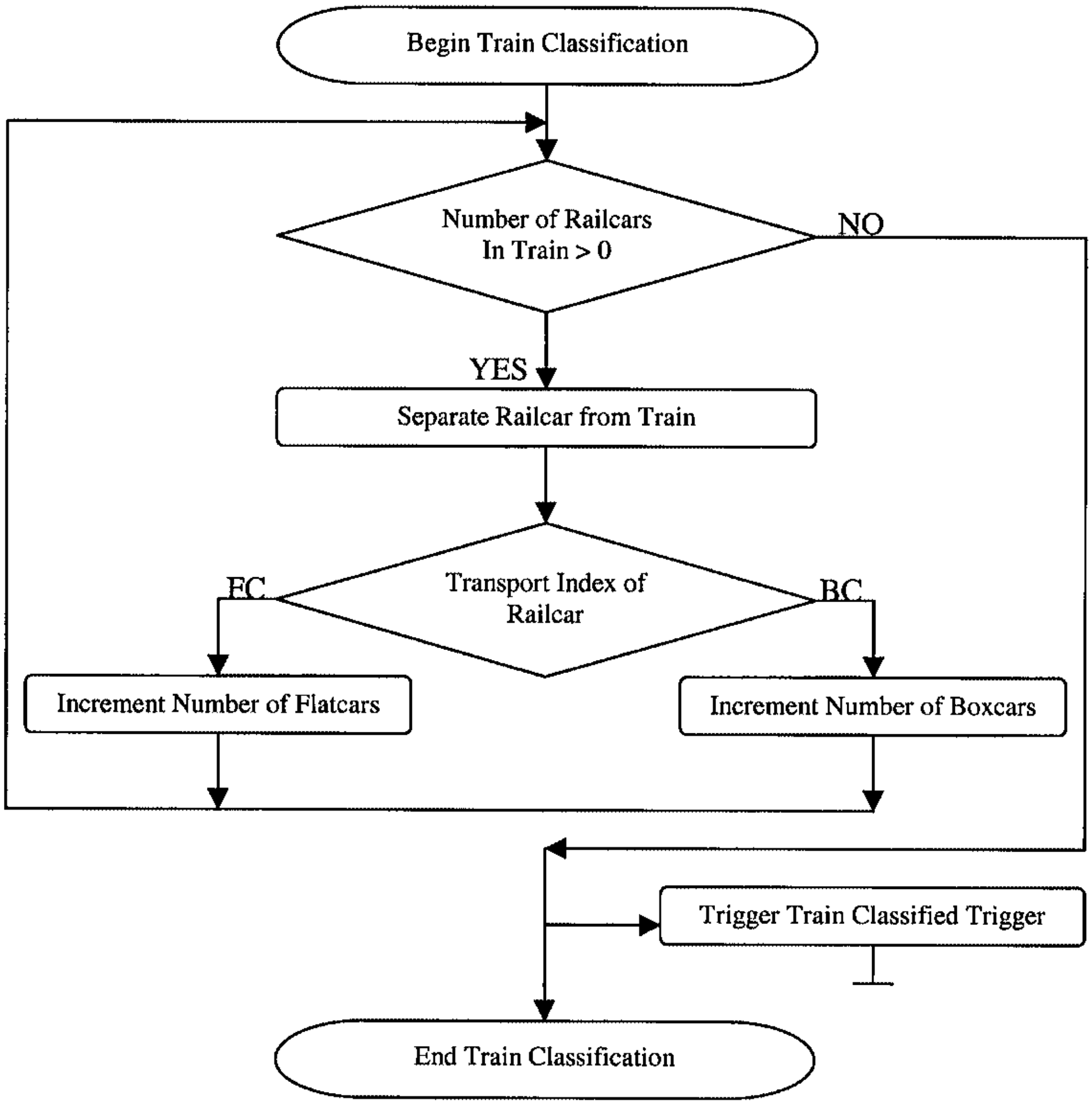

Begin Train Classification
Number of Railcars In Train > 0
NO
YES
Separate Railcar from Train
Transport Index of Railcar
FC
BC
Increment Number of Flatcars
Increment Number of Boxcars
Trigger Train Classified Trigger
End Train Classification

Fig. A.45

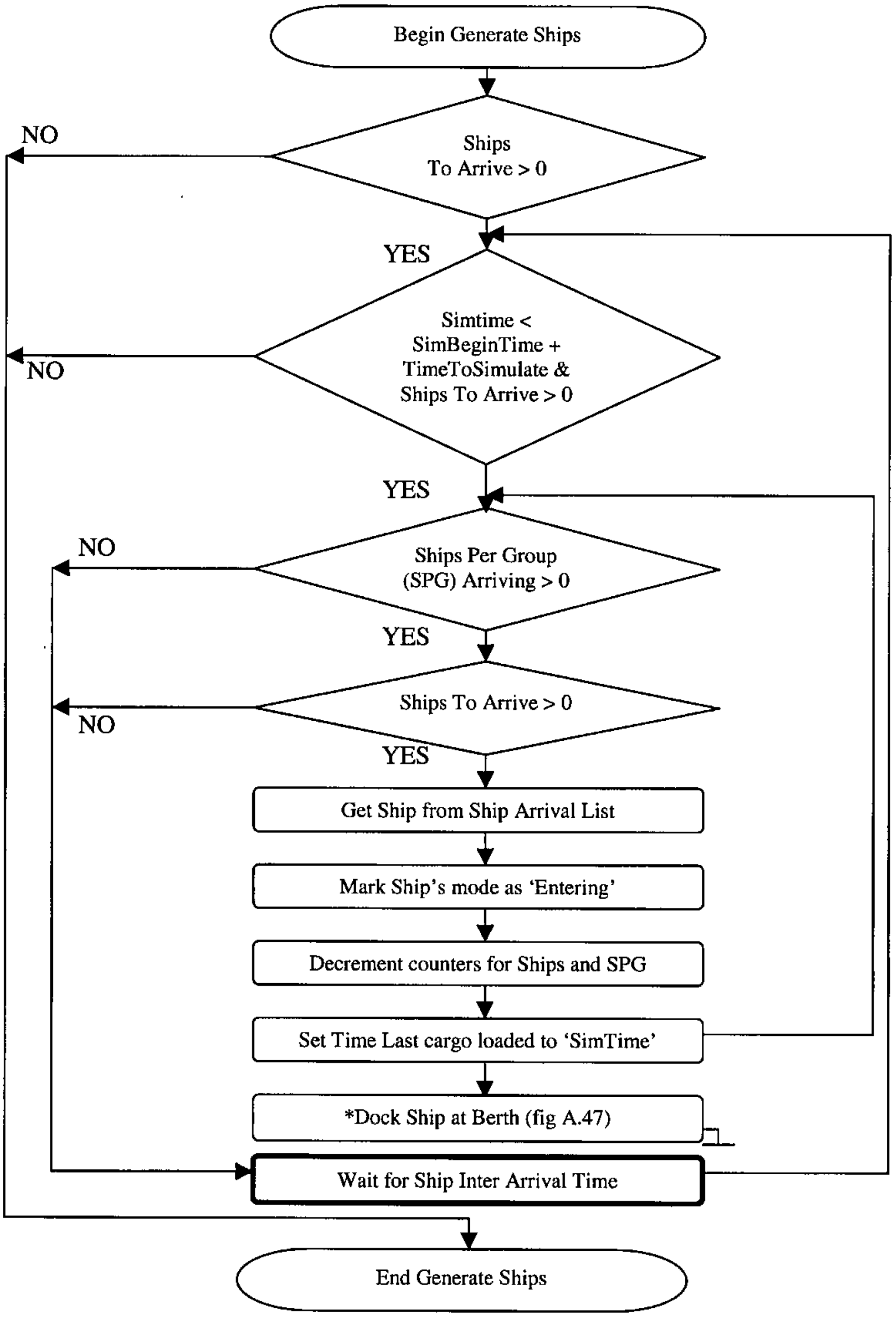

Begin Generate Ships
Ships To Arrive > 0
NO
YES
Simtime < SimBeginTime + TimeToSimulate & Ships To Arrive > 0
NO
YES
Ships Per Group (SPG) Arriving > 0
NO
YES
Ships To Arrive > 0
NO
YES
Get Ship from Ship Arrival List
Mark Ship's mode as 'Entering'
Decrement counters for Ships and SPG
Set Time Last cargo loaded to 'SimTime'
*Dock Ship at Berth (fig A.47)
Wait for Ship Inter Arrival Time
End Generate Ships

Fig. A.46

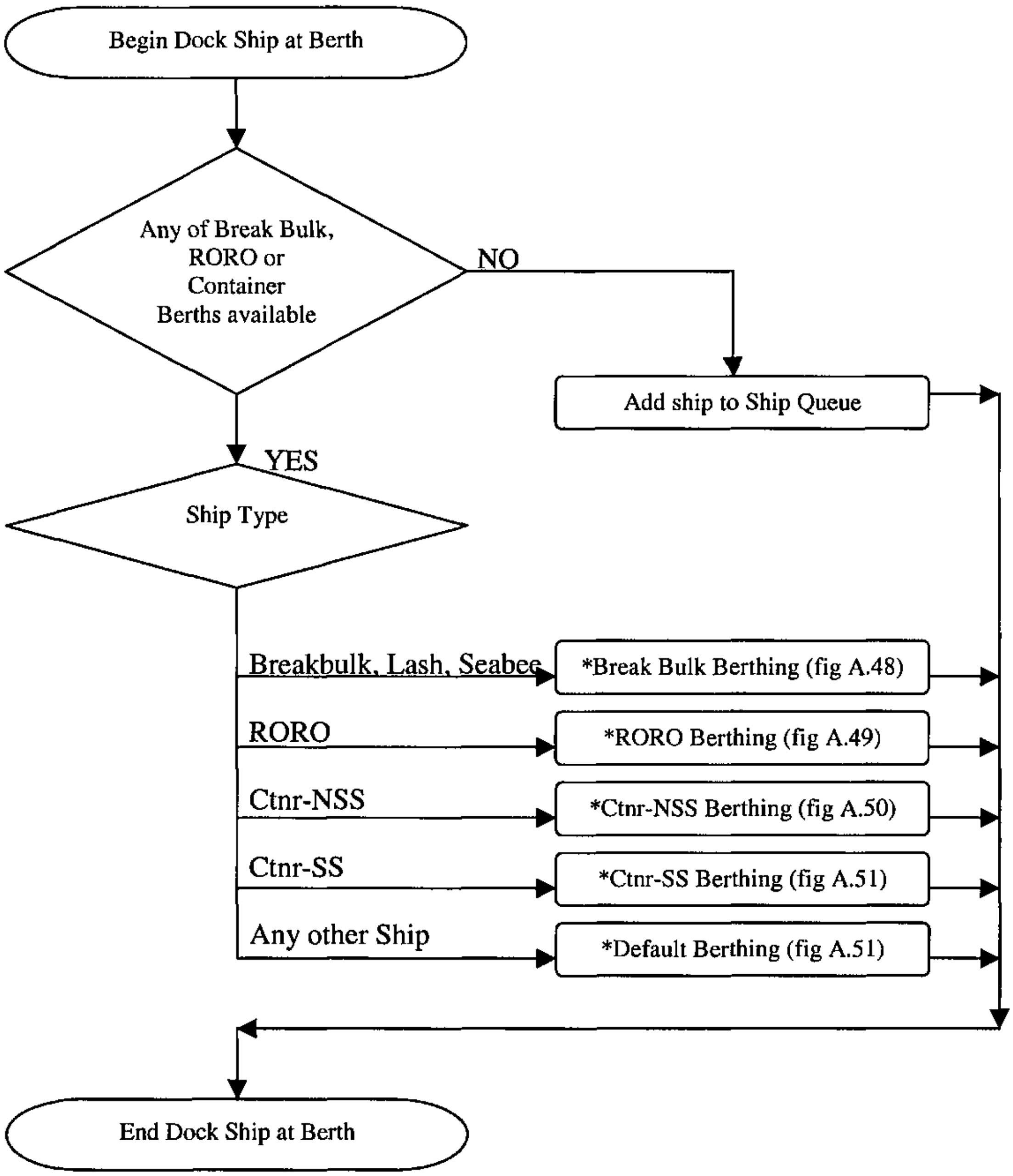

Begin Dock Ship at Berth
Any of Break Bulk, RORO or Container Berths available
NO
Add ship to Ship Queue
YES
Ship Type
Breakbulk, Lash, Seabee
*Break Bulk Berthing (fig A.48)
RORO
*RORO Berthing (fig A.49)
Ctnr-NSS
*Ctnr-NSS Berthing (fig A.50)
Ctnr-SS
*Ctnr-SS Berthing (fig A.51)
Any other Ship
*Default Berthing (fig A.51)
End Dock Ship at Berth

Fig. A.47

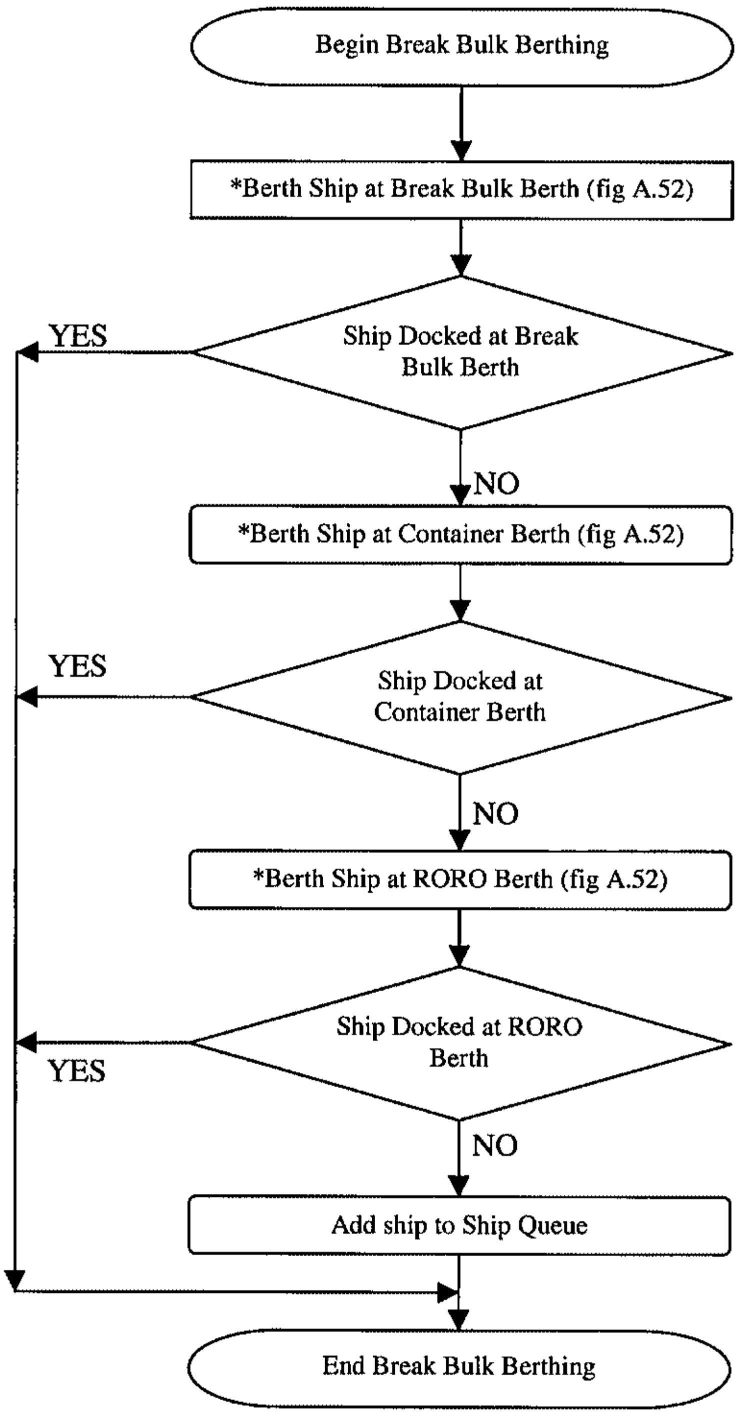

Begin Break Bulk Berthing
*Berth Ship at Break Bulk Berth (fig A.52)
Ship Docked at Break Bulk Berth
YES
NO
*Berth Ship at Container Berth (fig A.52)
Ship Docked at Container Berth
YES
NO
*Berth Ship at RORO Berth (fig A.52)
Ship Docked at RORO Berth
YES
NO
Add ship to Ship Queue
End Break Bulk Berthing

Fig. A.48

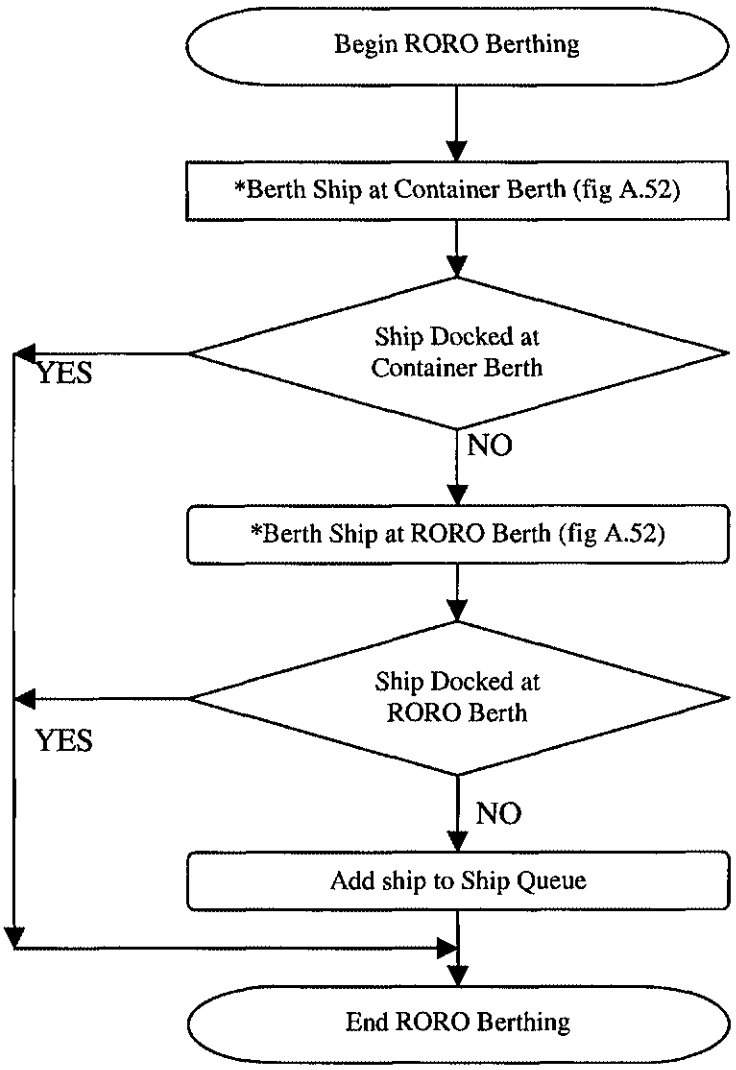

Begin RORO Berthing
*Berth Ship at Container Berth (fig A.52)
Ship Docked at Container Berth
YES
NO
*Berth Ship at RORO Berth (fig A.52)
Ship Docked at RORO Berth
YES
NO
Add ship to Ship Queue
End RORO Berthing

Fig. A.49

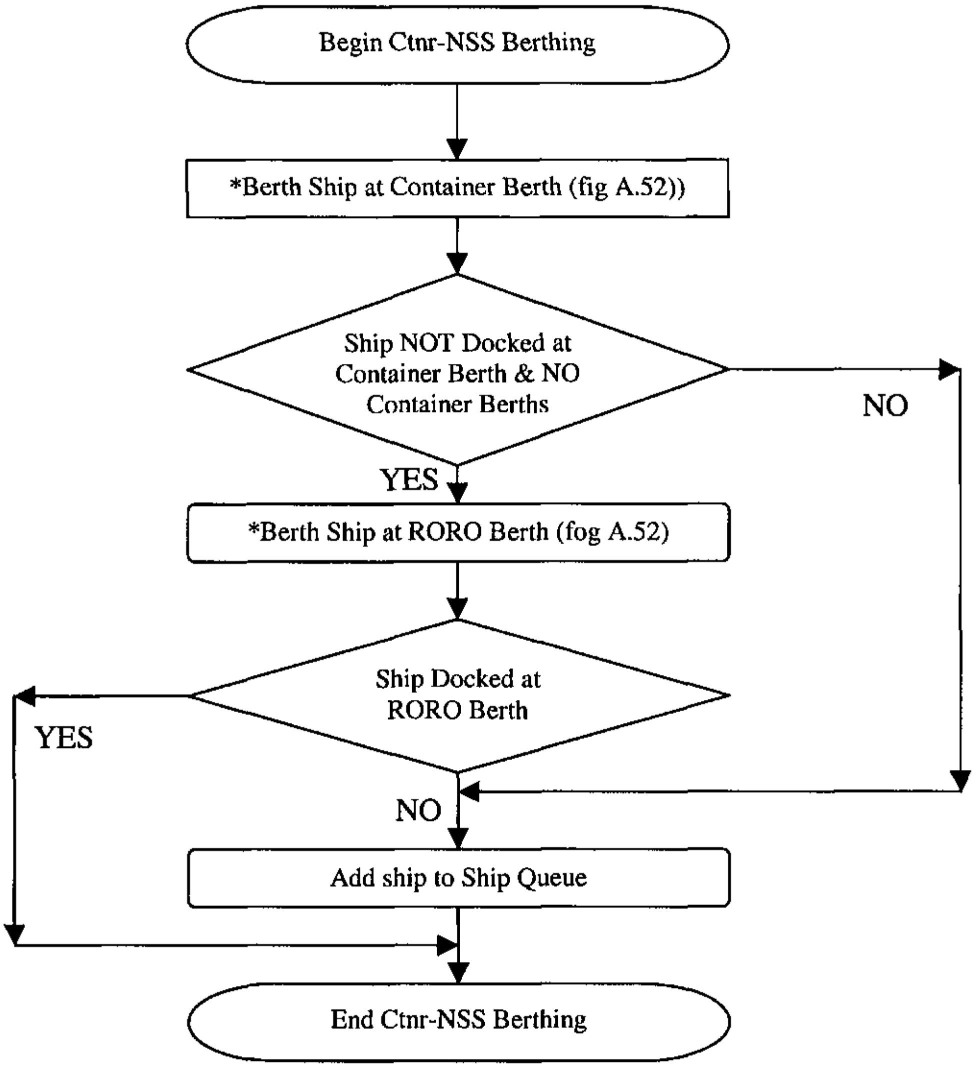

Begin Ctnr-NSS Berthing
*Berth Ship at Container Berth (fig A.52))
Ship NOT Docked at Container Berth & NO Container Berths
NO
YES
*Berth Ship at RORO Berth (fog A.52)
Ship Docked at RORO Berth
YES
NO
Add ship to Ship Queue
End Ctnr-NSS Berthing

Fig. A.50

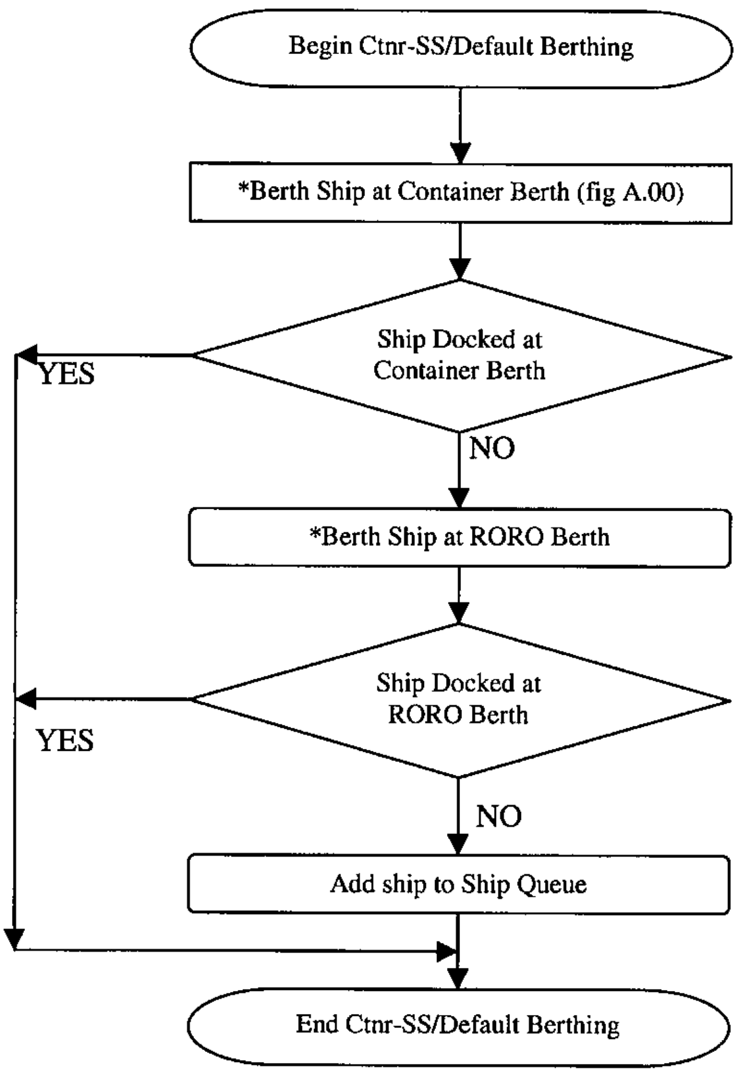

Begin Ctnr-SS/Default Berthing
*Berth Ship at Container Berth (fig A.00)
Ship Docked at Container Berth
YES
NO
*Berth Ship at RORO Berth
Ship Docked at RORO Berth
YES
NO
Add ship to Ship Queue
End Ctnr-SS/Default Berthing

Fig. A.51

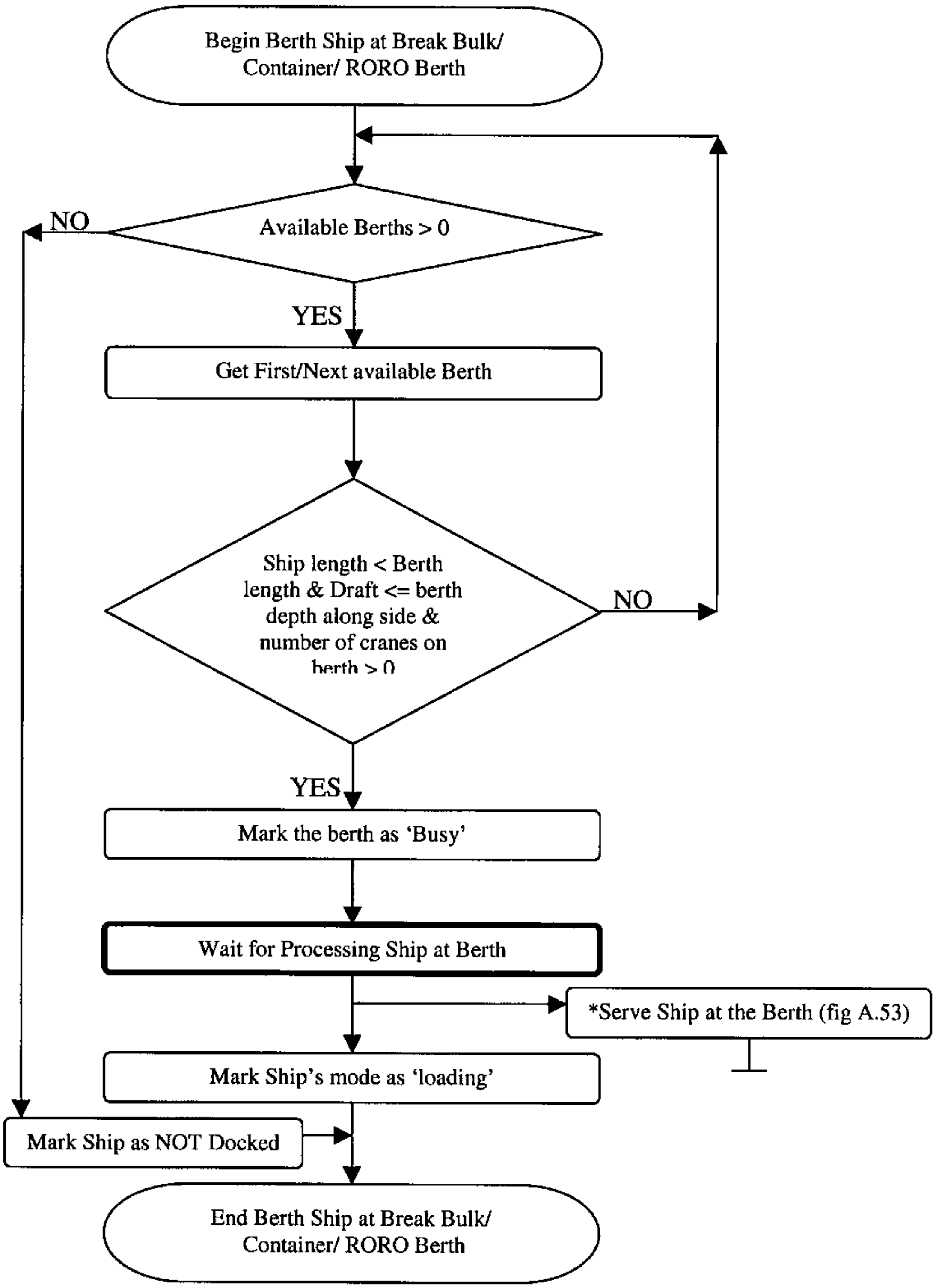

Begin Berth Ship at Break Bulk/ Container/ RORO Berth
Available Berths > 0
NO
YES
Get First/Next available Berth
Ship length < Berth length & Draft <= berth depth along side & number of cranes on berth > 0
NO
YES
Mark the berth as 'Busy'
Wait for Processing Ship at Berth
*Serve Ship at the Berth (fig A.53)
Mark Ship's mode as 'loading'
Mark Ship as NOT Docked
End Berth Ship at Break Bulk/ Container/ RORO Berth

Fig. A.52

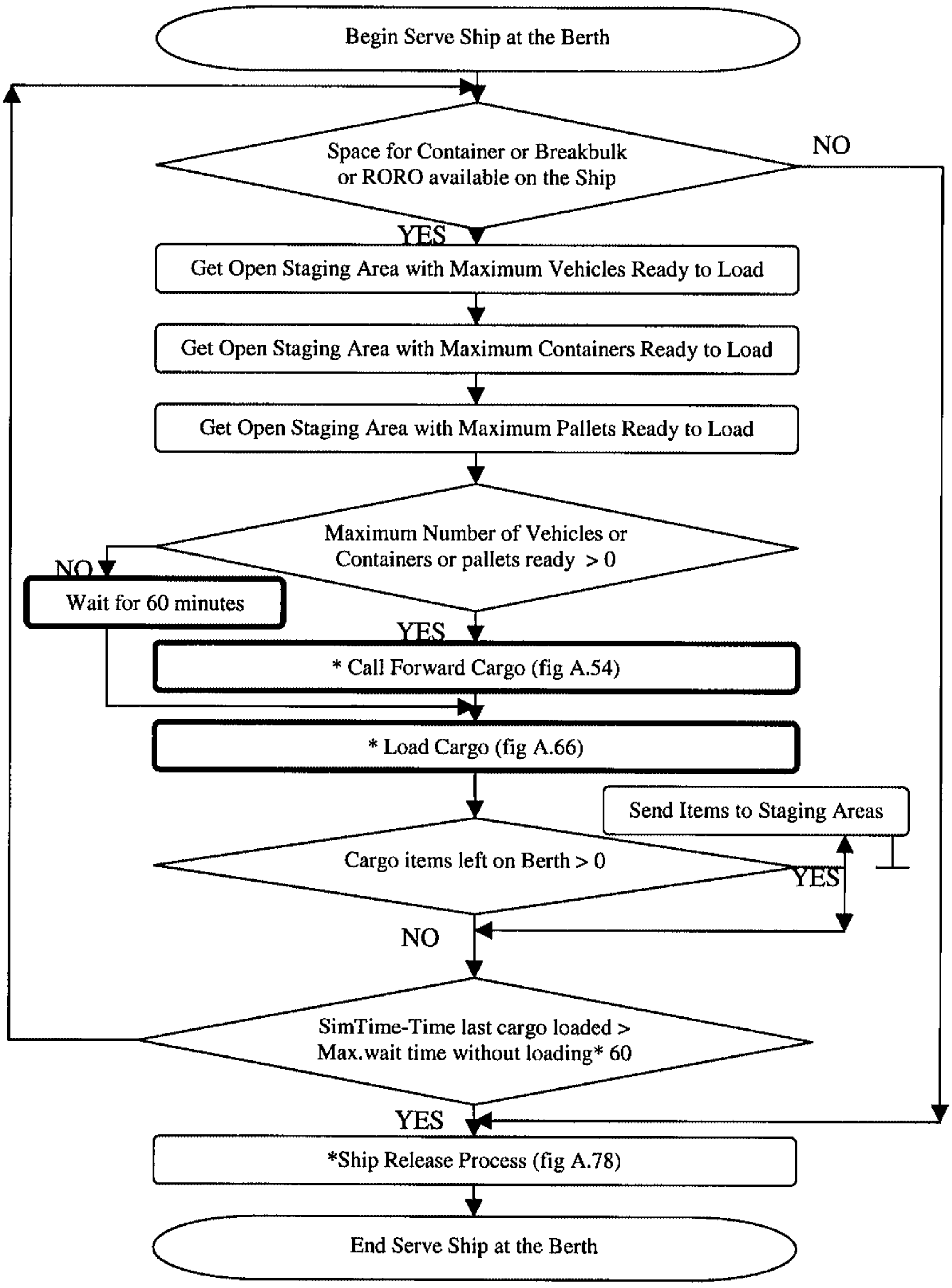
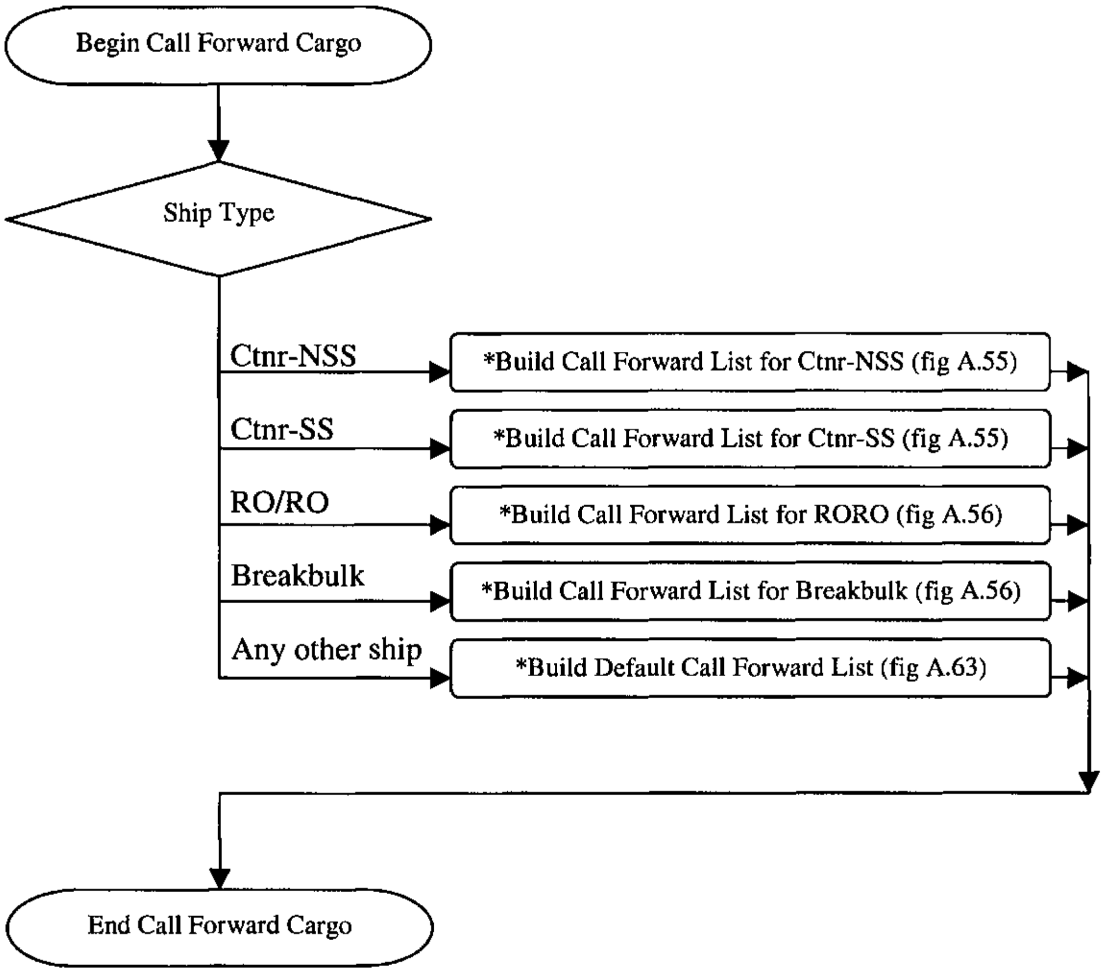

Begin Serve Ship at the Berth
Space for Container or Breakbulk or RORO available on the Ship
NO
YES
Get Open Staging Area with Maximum Vehicles Ready to Load
Get Open Staging Area with Maximum Containers Ready to Load
Get Open Staging Area with Maximum Pallets Ready to Load
Maximum Number of Vehicles or Containers or pallets ready > 0
NO
Wait for 60 minutes
YES
* Call Forward Cargo (fig A.54)
* Load Cargo (fig A.66)
Send Items to Staging Areas
Cargo items left on Berth > 0
YES
NO
SimTime-Time last cargo loaded > Max.wait time without loading* 60
YES
*Ship Release Process (fig A.78)
End Serve Ship at the Berth

Fig. A.53



Begin Call Forward Cargo
Ship Type
Ctnr-NSS
Ctnr-SS
RO/RO
Breakbulk
Any other ship
*Build Call Forward List for Ctnr-NSS (fig A.55)
*Build Call Forward List for Ctnr-SS (fig A.55)
*Build Call Forward List for RORO (fig A.56)
*Build Call Forward List for Breakbulk (fig A.56)
*Build Default Call Forward List (fig A.63)
End Call Forward Cargo

Fig. A.54

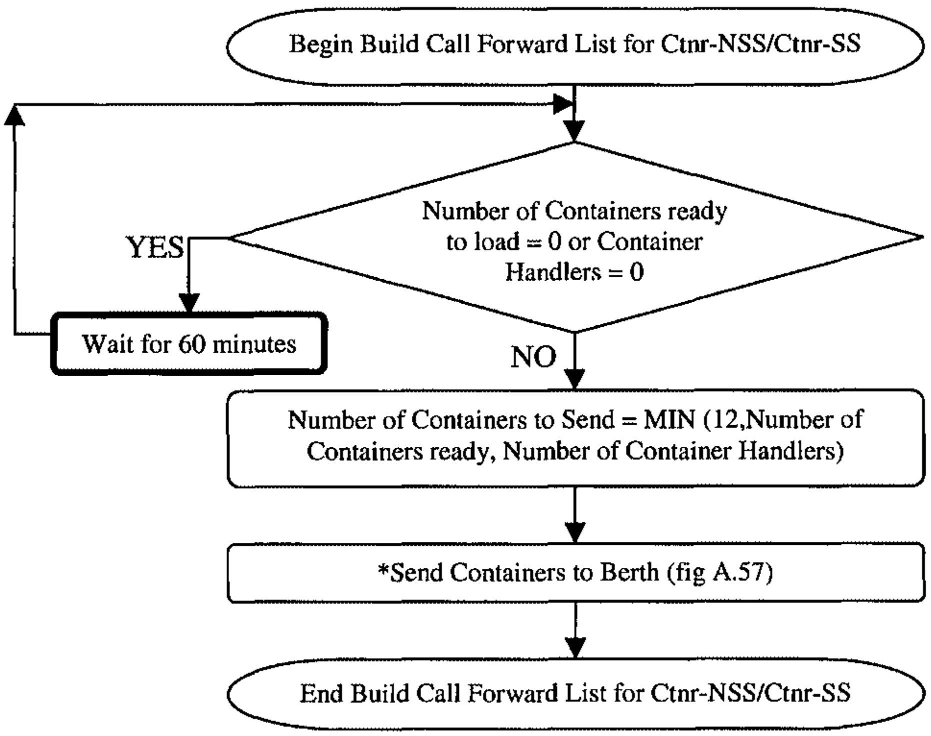

Begin Build Call Forward List for Ctnr-NSS/Ctnr-SS
Number of Containers ready to load = 0 or Container Handlers = 0
YES
Wait for 60 minutes
NO
Number of Containers to Send = MIN (12,Number of Containers ready, Number of Container Handlers)
*Send Containers to Berth (fig A.57)
End Build Call Forward List for Ctnr-NSS/Ctnr-SS

Fig. A.55

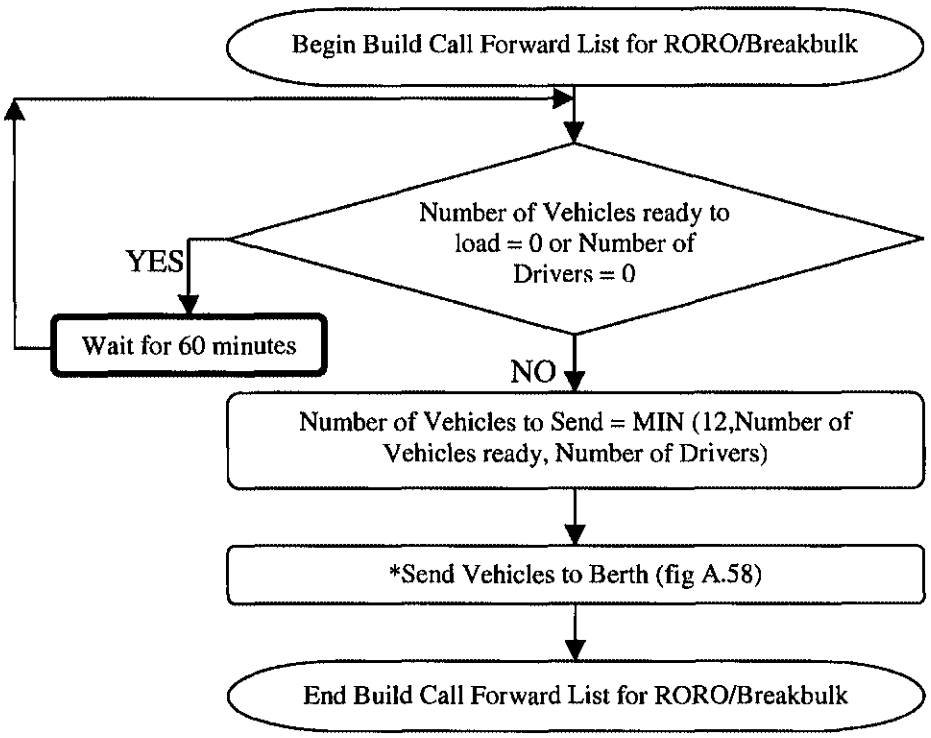

Begin Build Call Forward List for RORO/Breakbulk
Number of Vehicles ready to load = 0 or Number of Drivers = 0
YES
Wait for 60 minutes
NO
Number of Vehicles to Send = MIN (12,Number of Vehicles ready, Number of Drivers)
*Send Vehicles to Berth (fig A.58)
End Build Call Forward List for RORO/Breakbulk

Fig. A.56

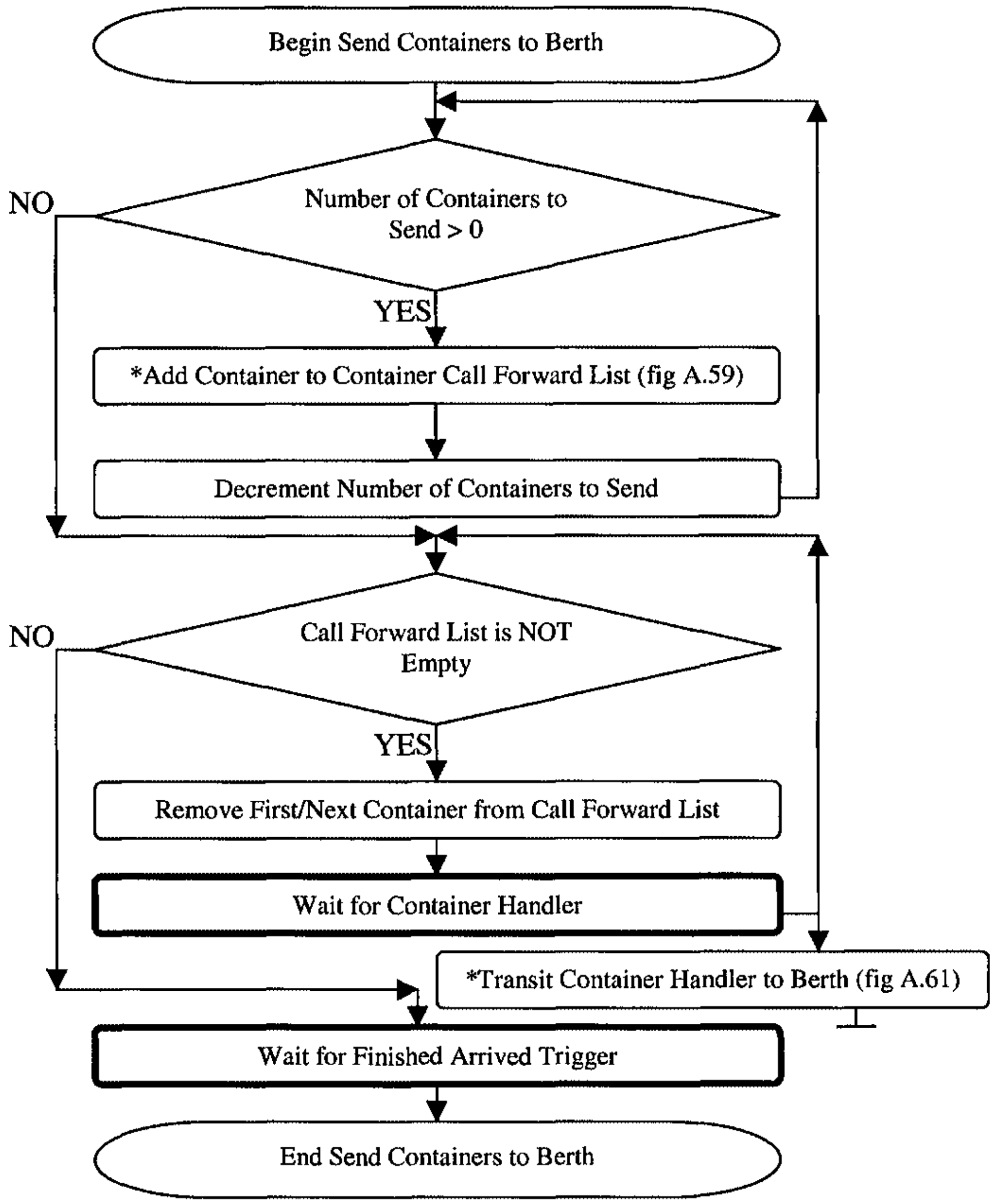

Begin Send Containers to Berth
Number of Containers to Send > 0
NO
YES
*Add Container to Container Call Forward List (fig A.59)
Decrement Number of Containers to Send
Call Forward List is NOT Empty
NO
YES
Remove First/Next Container from Call Forward List
Wait for Container Handler
*Transit Container Handler to Berth (fig A.61)
Wait for Finished Arrived Trigger
End Send Containers to Berth

Fig. A.57

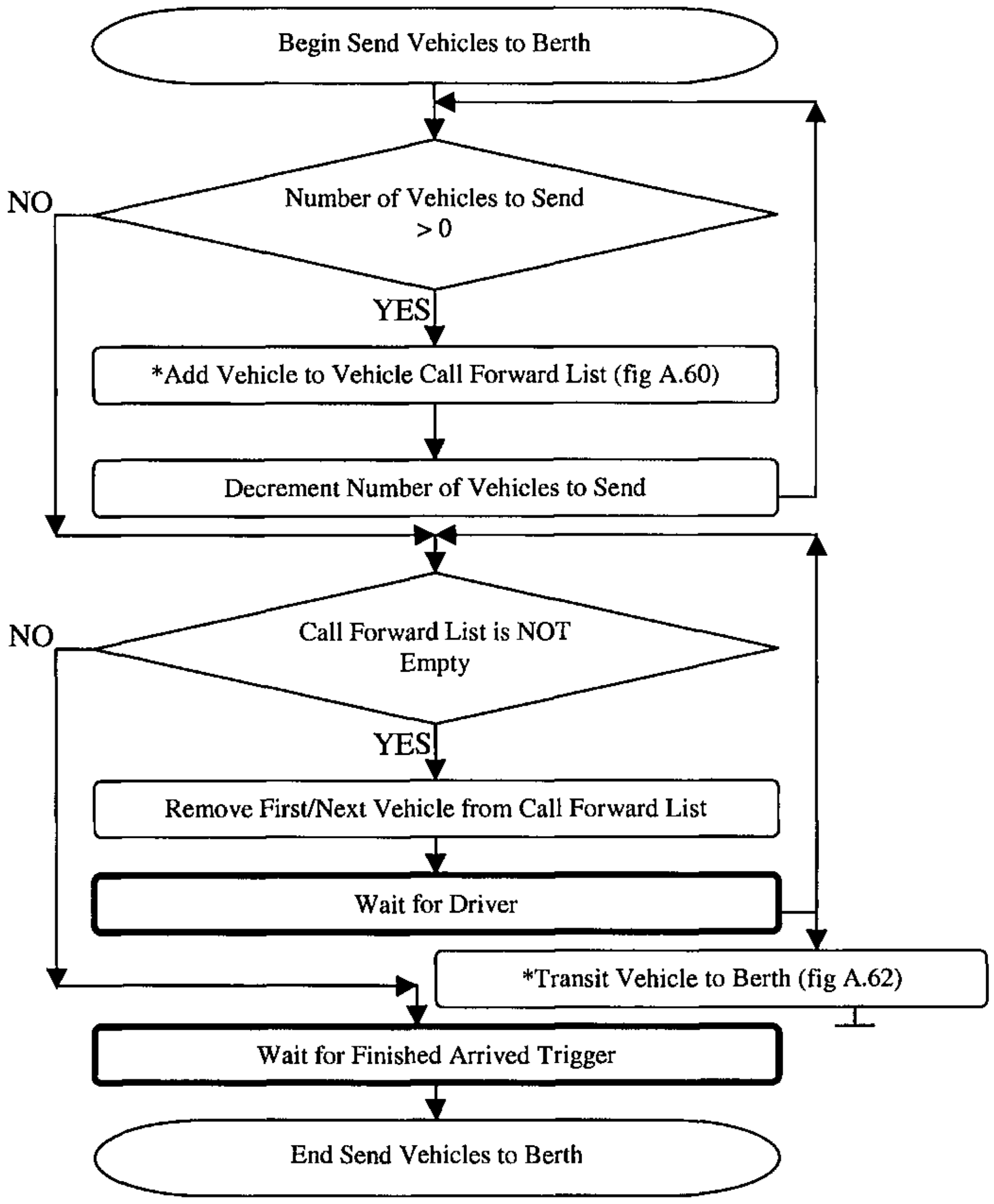

Begin Send Vehicles to Berth
NO
Number of Vehicles to Send > 0
YES
*Add Vehicle to Vehicle Call Forward List (fig A.60)
Decrement Number of Vehicles to Send
NO
Call Forward List is NOT Empty
YES
Remove First/Next Vehicle from Call Forward List
Wait for Driver
*Transit Vehicle to Berth (fig A.62)
Wait for Finished Arrived Trigger
End Send Vehicles to Berth

Fig. A.58

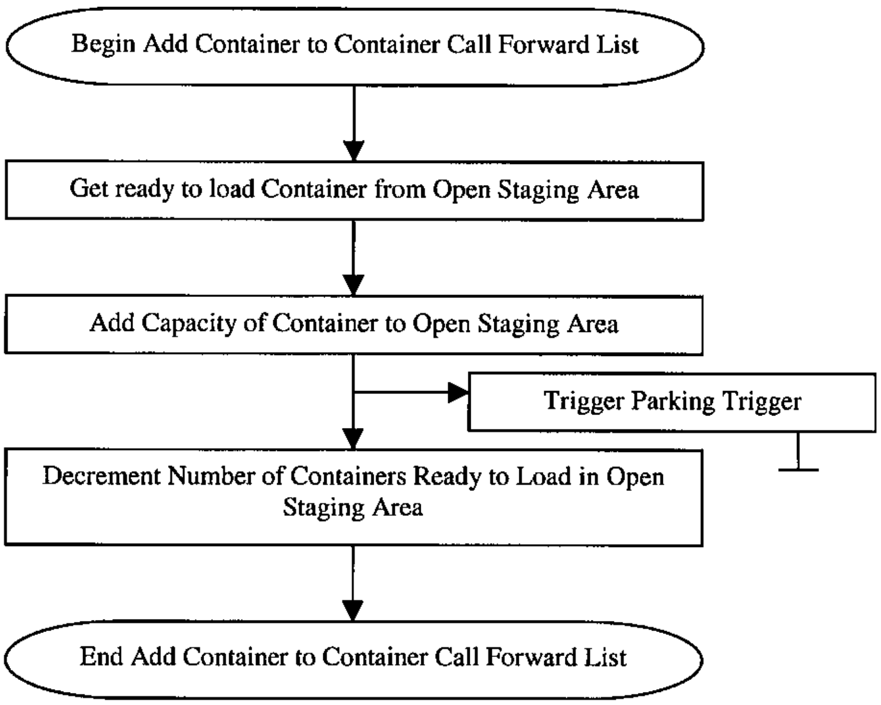

Begin Add Container to Container Call Forward List
Get ready to load Container from Open Staging Area
Add Capacity of Container to Open Staging Area
Trigger Parking Trigger
Decrement Number of Containers Ready to Load in Open Staging Area
End Add Container to Container Call Forward List

Fig. A.59

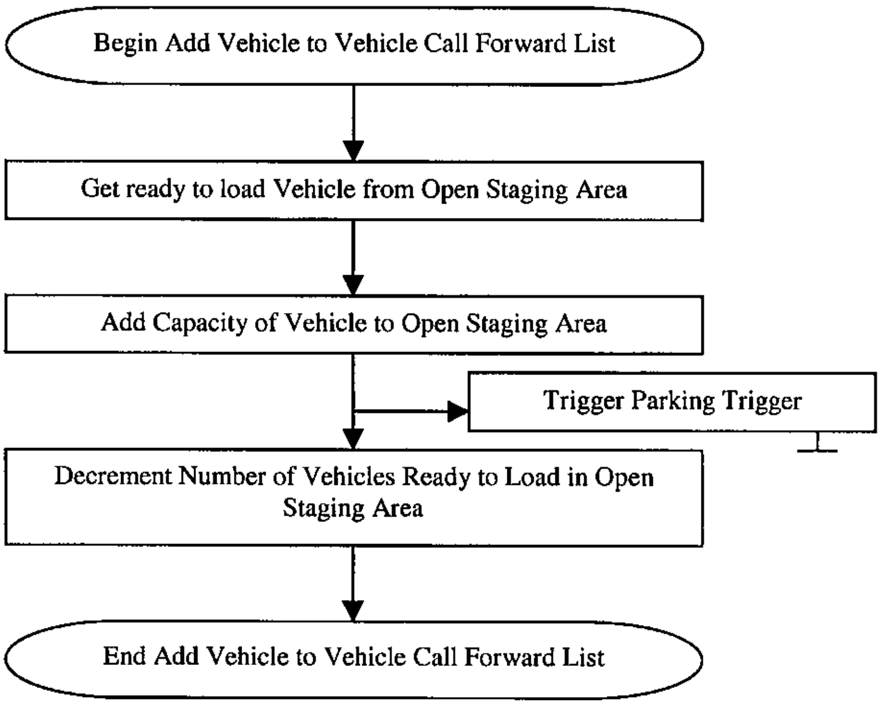

Begin Add Vehicle to Vehicle Call Forward List
Get ready to load Vehicle from Open Staging Area
Add Capacity of Vehicle to Open Staging Area
Trigger Parking Trigger
Decrement Number of Vehicles Ready to Load in Open Staging Area
End Add Vehicle to Vehicle Call Forward List

Fig. A.60

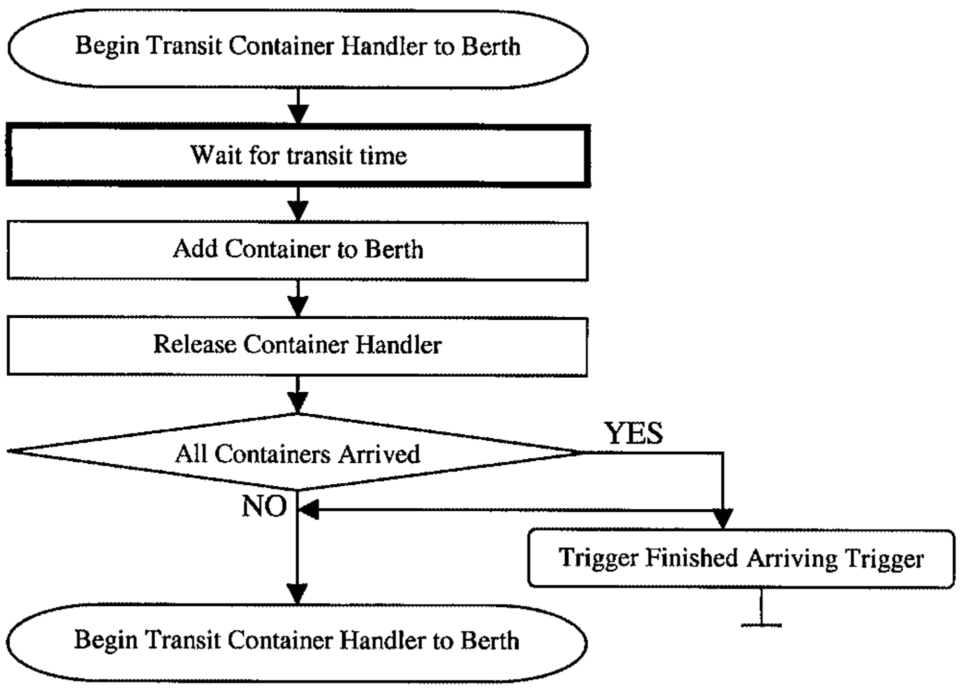

Begin Transit Container Handler to Berth
Wait for transit time
Add Container to Berth
Release Container Handler
All Containers Arrived
YES
NO
Trigger Finished Arriving Trigger
Begin Transit Container Handler to Berth

Fig. A.61

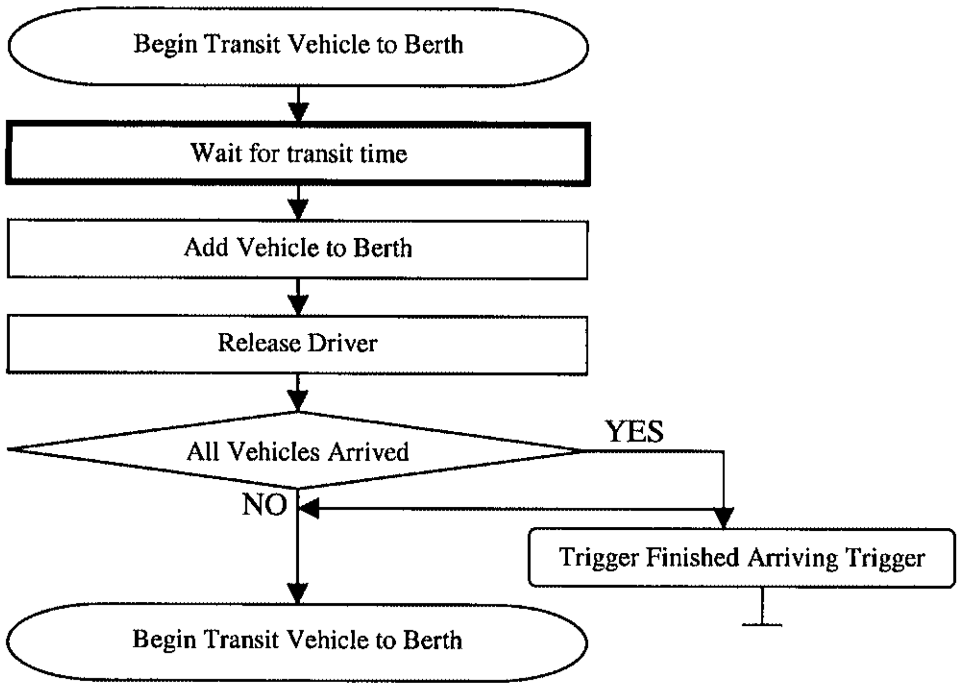
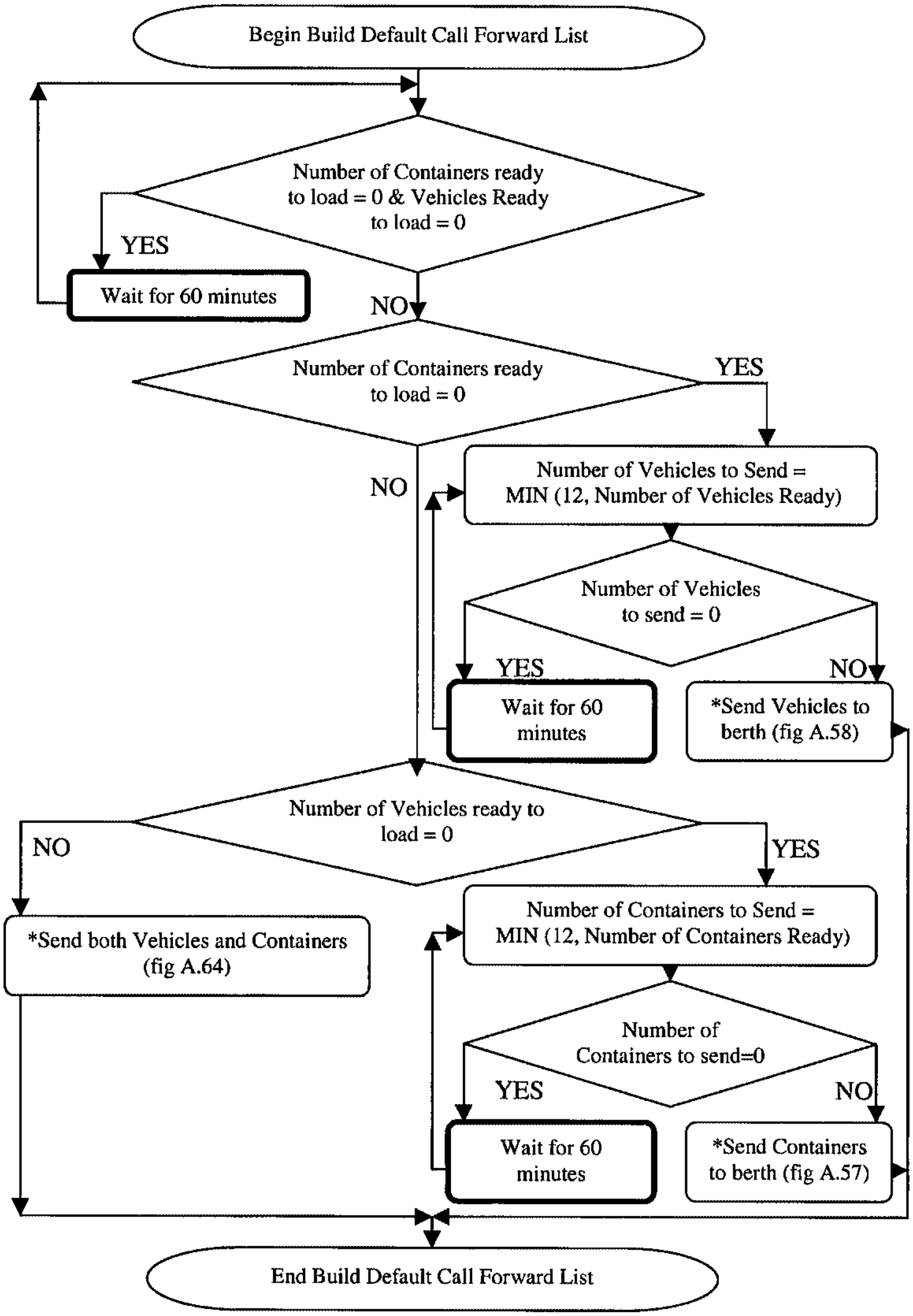

Begin Transit Vehicle to Berth
Wait for transit time
Add Vehicle to Berth
Release Driver
All Vehicles Arrived
YES
NO
Trigger Finished Arriving Trigger
Begin Transit Vehicle to Berth

Fig. A.62



Begin Build Default Call Forward List
Number of Containers ready to load = 0 & Vehicles Ready to load = 0
YES
Wait for 60 minutes
NO
Number of Containers ready to load = 0
YES
NO
Number of Vehicles to Send = MIN (12, Number of Vehicles Ready)
Number of Vehicles to send = 0
YES
NO
Wait for 60 minutes
*Send Vehicles to berth (fig A.58)
Number of Vehicles ready to load = 0
NO
YES
*Send both Vehicles and Containers (fig A.64)
Number of Containers to Send = MIN (12, Number of Containers Ready)
Number of Containers to send=0
YES
NO
Wait for 60 minutes
*Send Containers to berth (fig A.57)
End Build Default Call Forward List

Fig. A.63

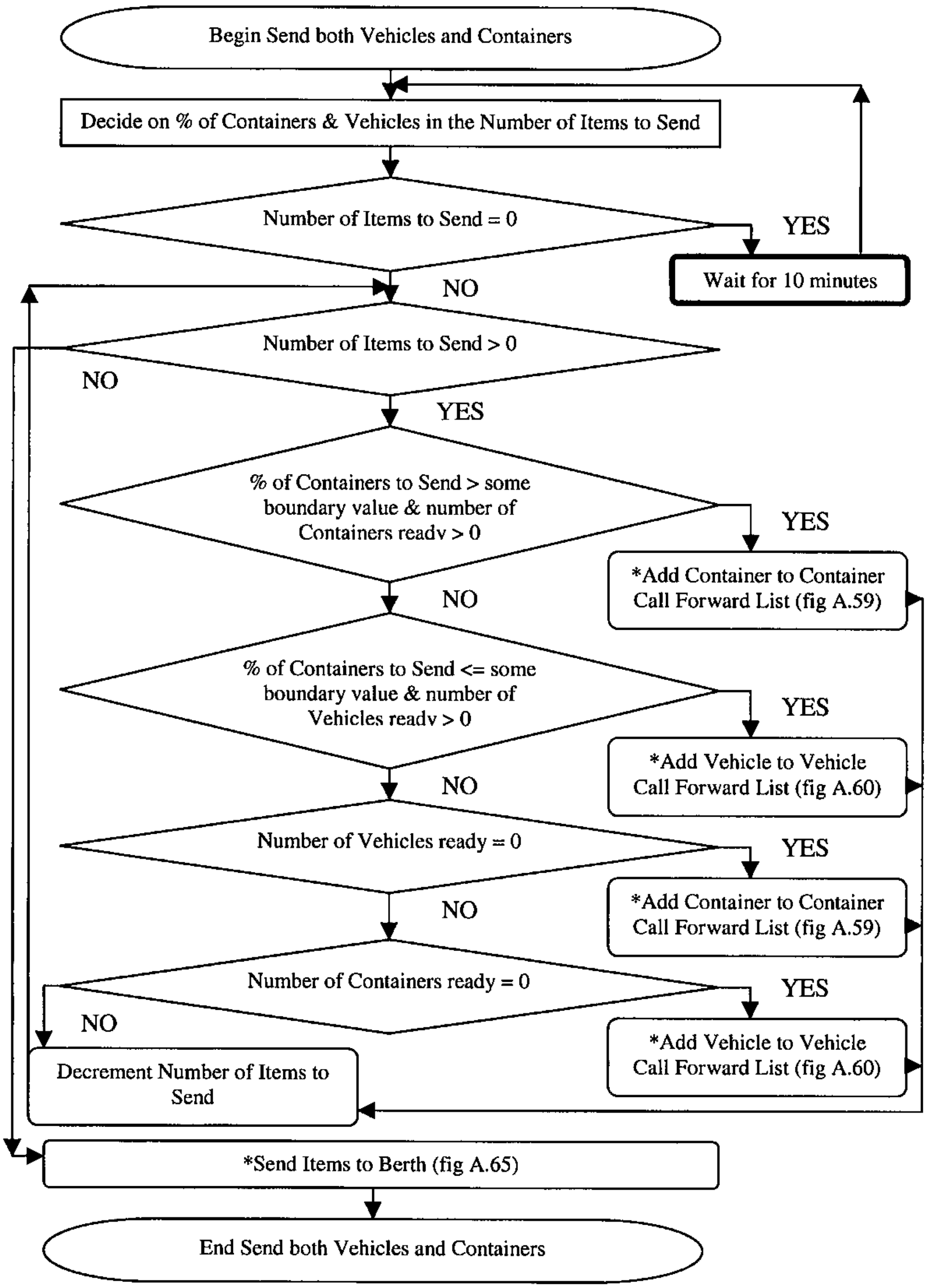

Begin Send both Vehicles and Containers
Decide on % of Containers & Vehicles in the Number of Items to Send
Number of Items to Send = 0
YES
Wait for 10 minutes
NO
Number of Items to Send > 0
NO
YES
% of Containers to Send > some boundary value & number of Containers ready > 0
YES
*Add Container to Container Call Forward List (fig A.59)
NO
% of Containers to Send <= some boundary value & number of Vehicles ready > 0
YES
*Add Vehicle to Vehicle Call Forward List (fig A.60)
NO
Number of Vehicles ready = 0
YES
*Add Container to Container Call Forward List (fig A.59)
NO
Number of Containers ready = 0
YES
NO
*Add Vehicle to Vehicle Call Forward List (fig A.60)
Decrement Number of Items to Send
*Send Items to Berth (fig A.65)
End Send both Vehicles and Containers

Fig. A.64

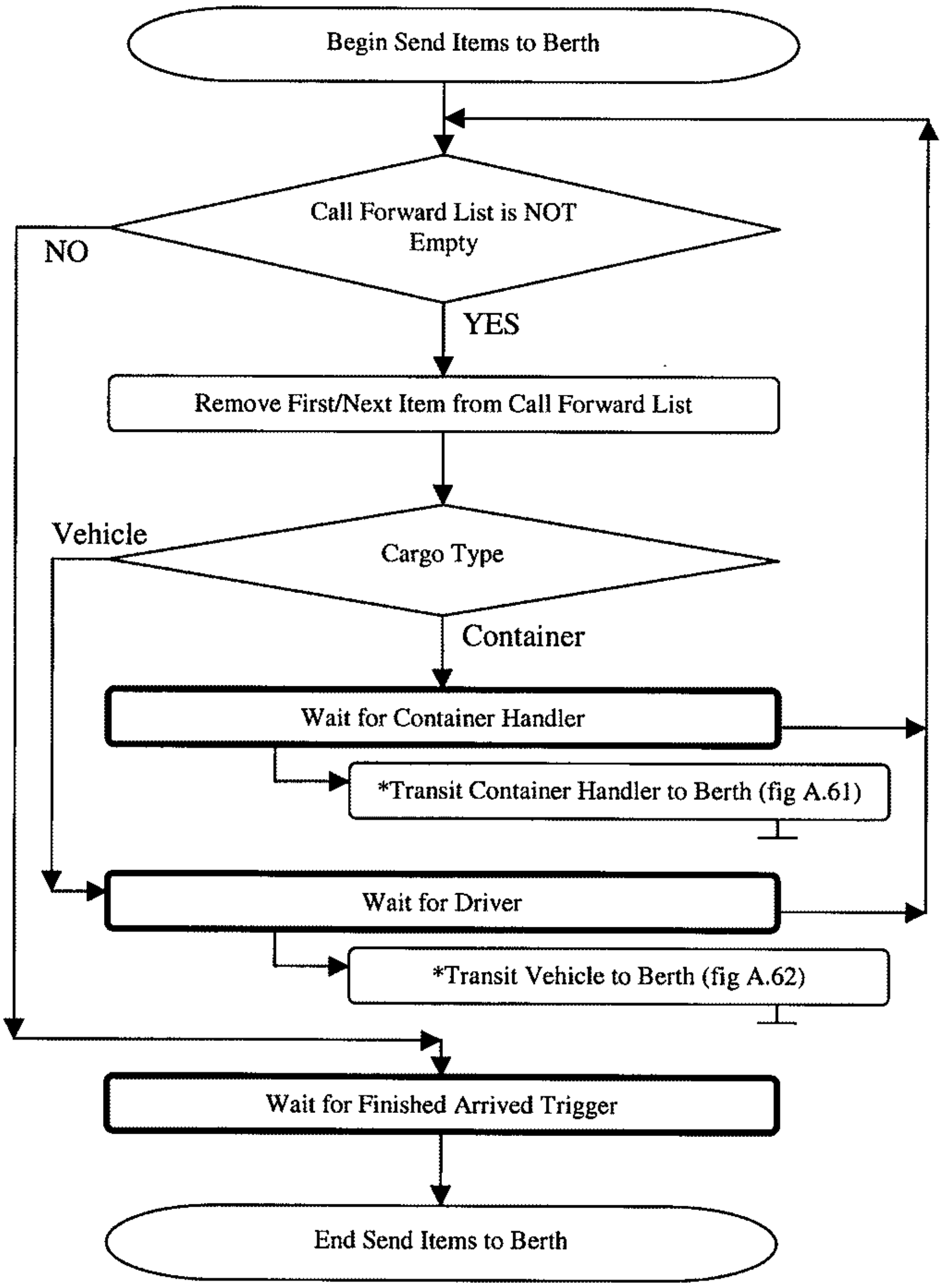

Begin Send Items to Berth
Call Forward List is NOT Empty
NO
YES
Remove First/Next Item from Call Forward List
Vehicle
Cargo Type
Container
Wait for Container Handler
*Transit Container Handler to Berth (fig A.61)
Wait for Driver
*Transit Vehicle to Berth (fig A.62)
Wait for Finished Arrived Trigger
End Send Items to Berth

Fig. A.65

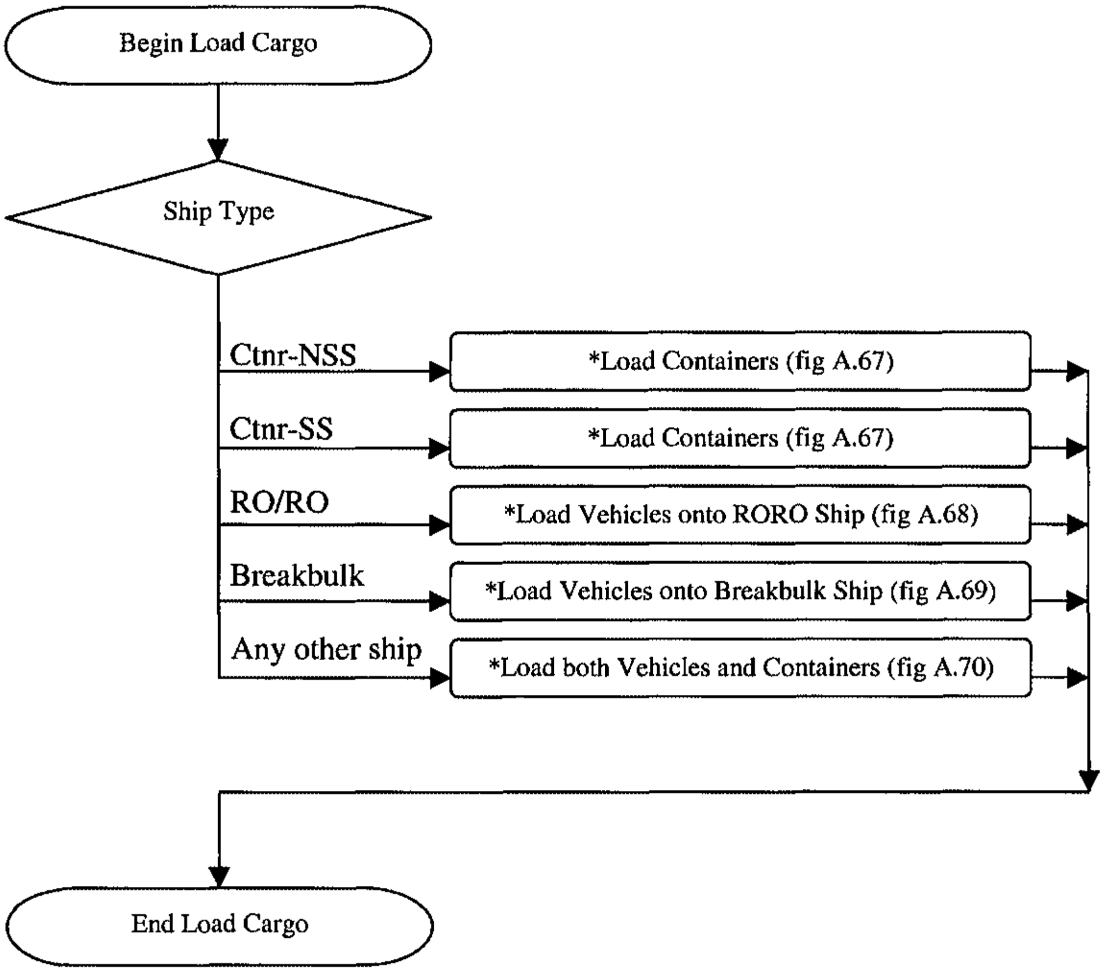

Begin Load Cargo
Ship Type
Ctnr-NSS
Ctnr-SS
RO/RO
Breakbulk
Any other ship
*Load Containers (fig A.67)
*Load Containers (fig A.67)
*Load Vehicles onto RORO Ship (fig A.68)
*Load Vehicles onto Breakbulk Ship (fig A.69)
*Load both Vehicles and Containers (fig A.70)
End Load Cargo

Fig. A.66

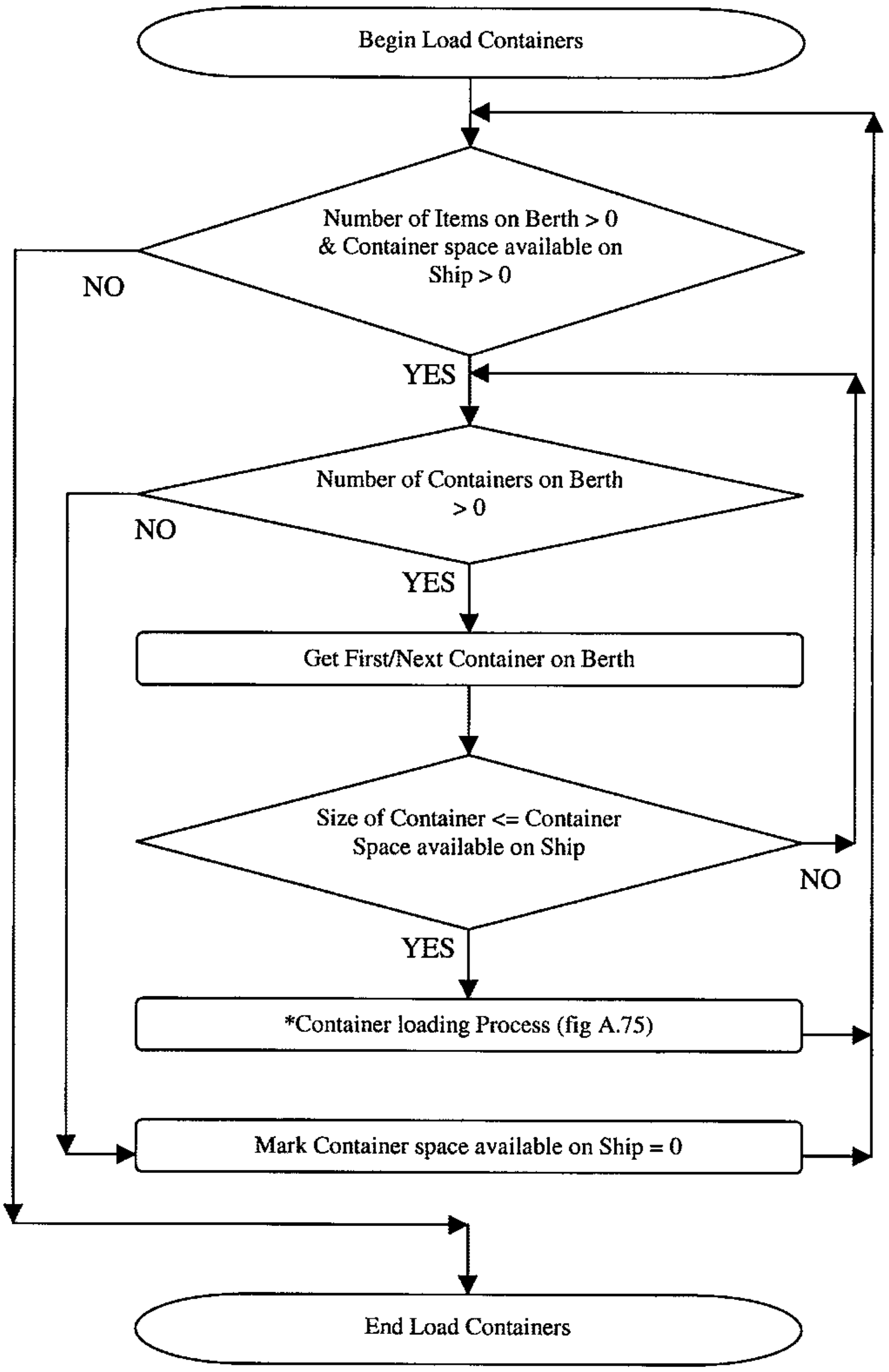

Begin Load Containers
Number of Items on Berth > 0 & Container space available on Ship > 0
NO
YES
Number of Containers on Berth > 0
NO
YES
Get First/Next Container on Berth
Size of Container <= Container Space available on Ship
NO
YES
*Container loading Process (fig A.75)
Mark Container space available on Ship = 0
End Load Containers

Fig. A.67

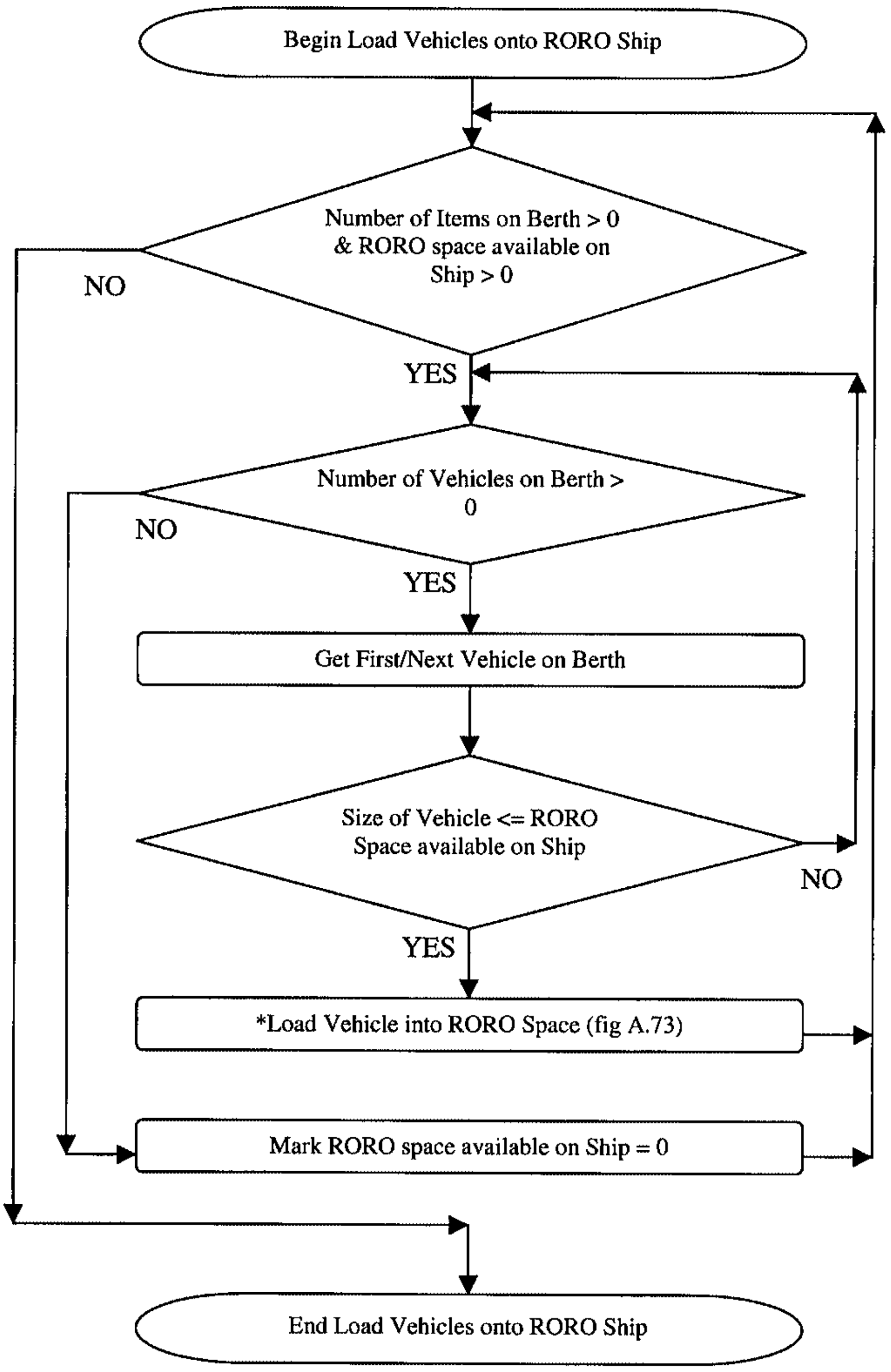

Begin Load Vehicles onto RORO Ship
Number of Items on Berth > 0 & RORO space available on Ship > 0
NO
YES
Number of Vehicles on Berth > 0
NO
YES
Get First/Next Vehicle on Berth
Size of Vehicle <= RORO Space available on Ship
NO
YES
*Load Vehicle into RORO Space (fig A.73)
Mark RORO space available on Ship = 0
End Load Vehicles onto RORO Ship

Fig. A.68

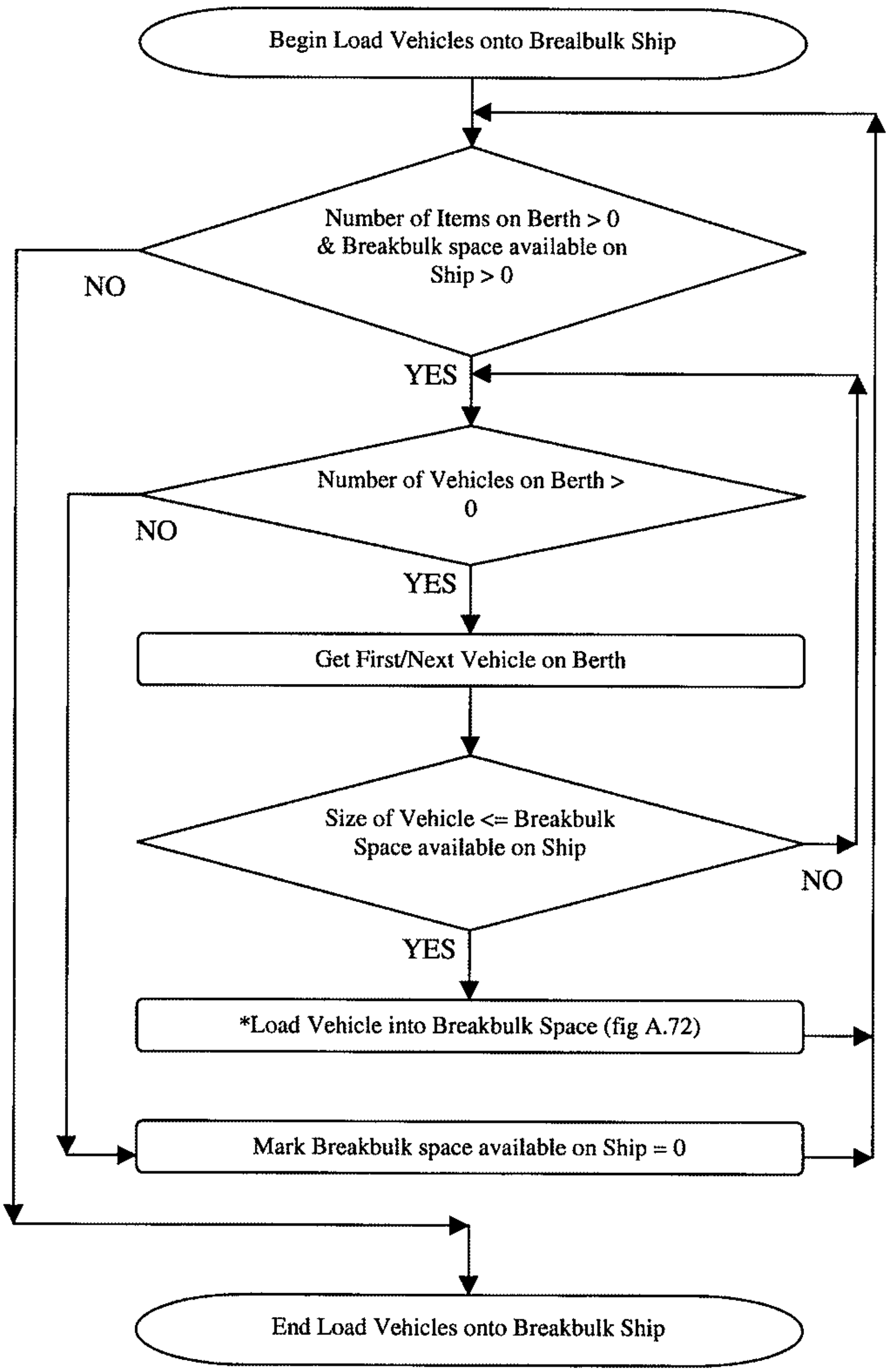

Begin Load Vehicles onto Brealbulk Ship
Number of Items on Berth > 0 & Breakbulk space available on Ship > 0
NO
YES
Number of Vehicles on Berth > 0
NO
YES
Get First/Next Vehicle on Berth
Size of Vehicle <= Breakbulk Space available on Ship
NO
YES
*Load Vehicle into Breakbulk Space (fig A.72)
Mark Breakbulk space available on Ship = 0
End Load Vehicles onto Breakbulk Ship

Fig. A.69

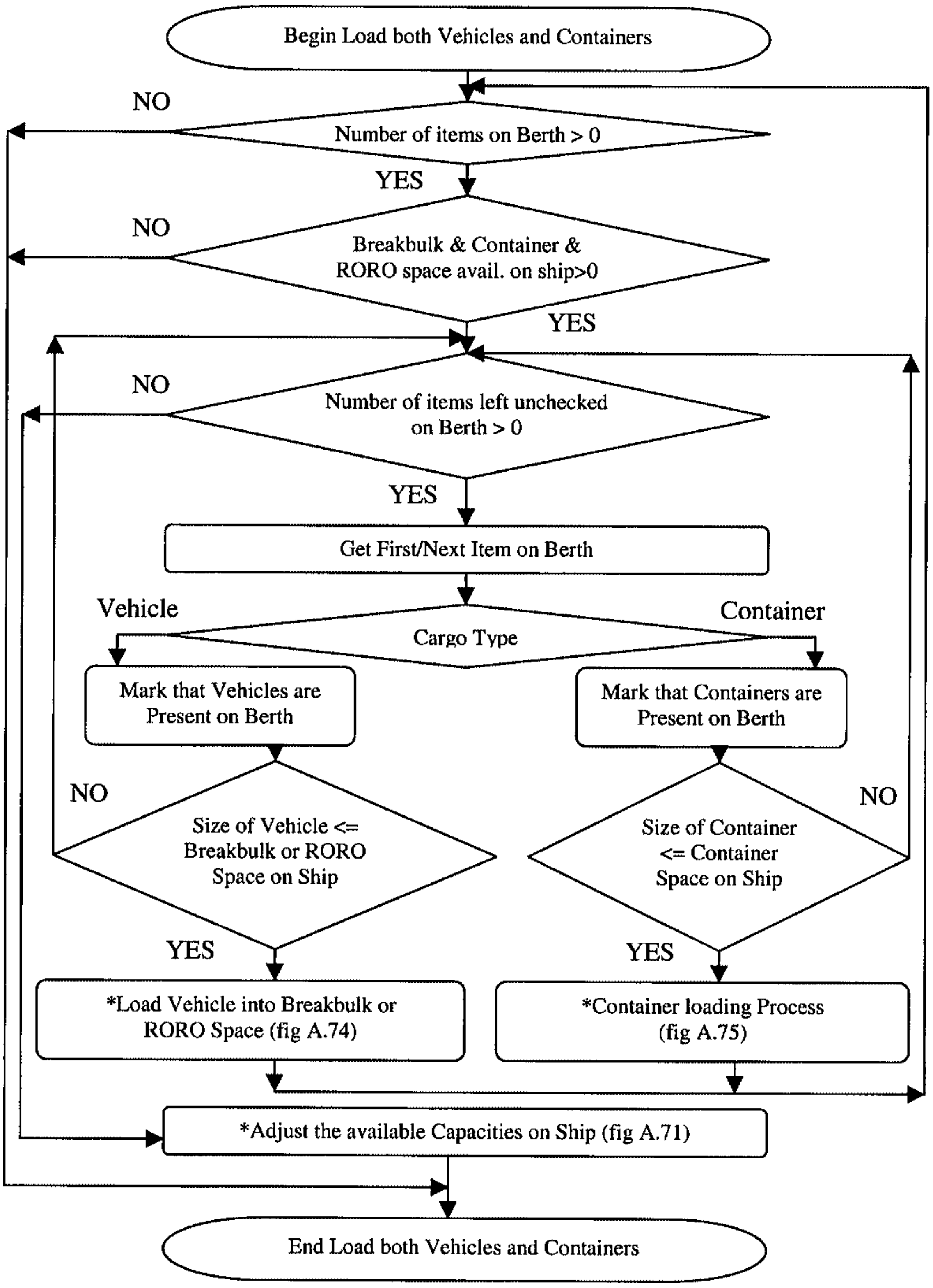

Begin Load both Vehicles and Containers
Number of items on Berth > 0
NO
YES
Breakbulk & Container & RORO space avail. on ship>0
NO
YES
Number of items left unchecked on Berth > 0
NO
YES
Get First/Next Item on Berth
Cargo Type
Vehicle
Container
Mark that Vehicles are Present on Berth
Mark that Containers are Present on Berth
Size of Vehicle <= Breakbulk or RORO Space on Ship
NO
YES
Size of Container <= Container Space on Ship
NO
YES
*Load Vehicle into Breakbulk or RORO Space (fig A.74)
*Container loading Process (fig A.75)
*Adjust the available Capacities on Ship (fig A.71)
End Load both Vehicles and Containers

Fig. A.70

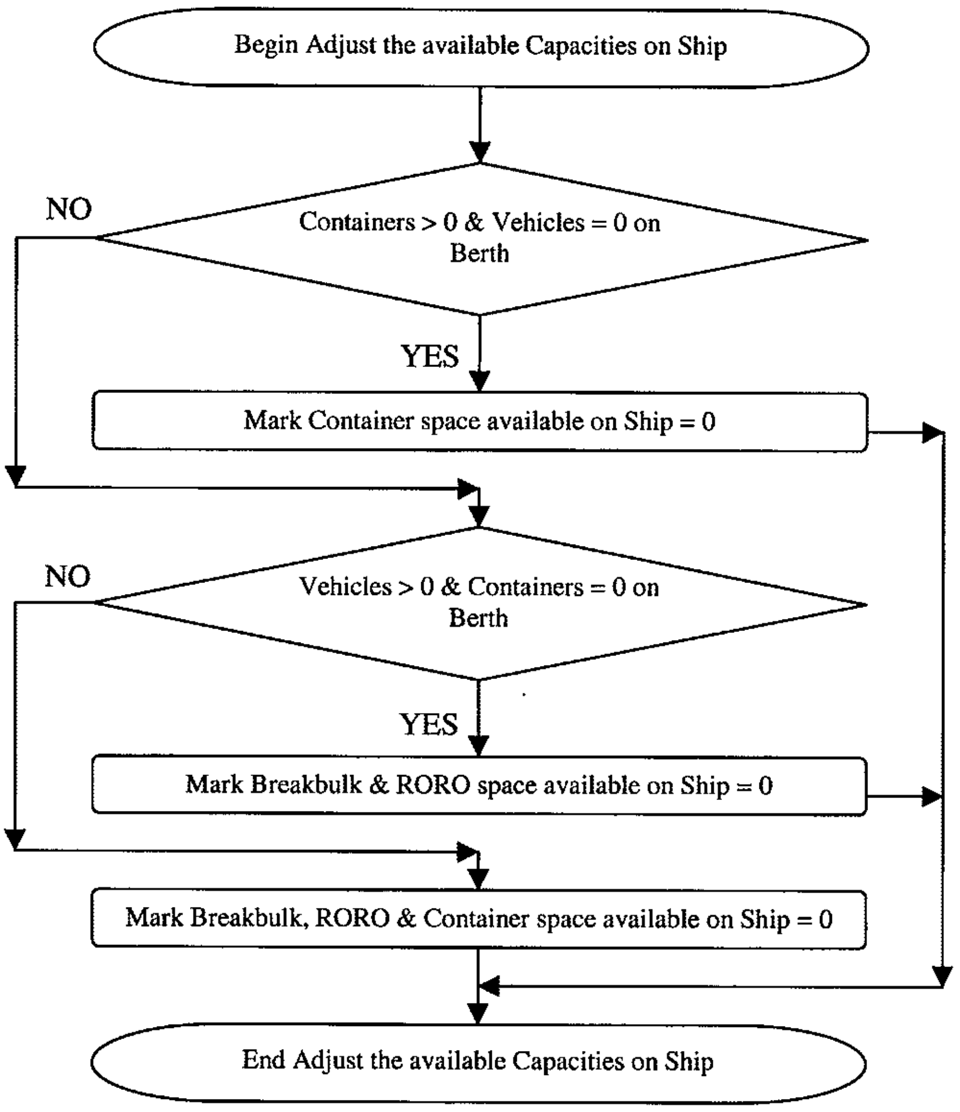

Begin Adjust the available Capacities on Ship
Containers > 0 & Vehicles = 0 on Berth
NO
YES
Mark Container space available on Ship = 0
Vehicles > 0 & Containers = 0 on Berth
NO
YES
Mark Breakbulk & RORO space available on Ship = 0
Mark Breakbulk, RORO & Container space available on Ship = 0
End Adjust the available Capacities on Ship

Fig. A.71

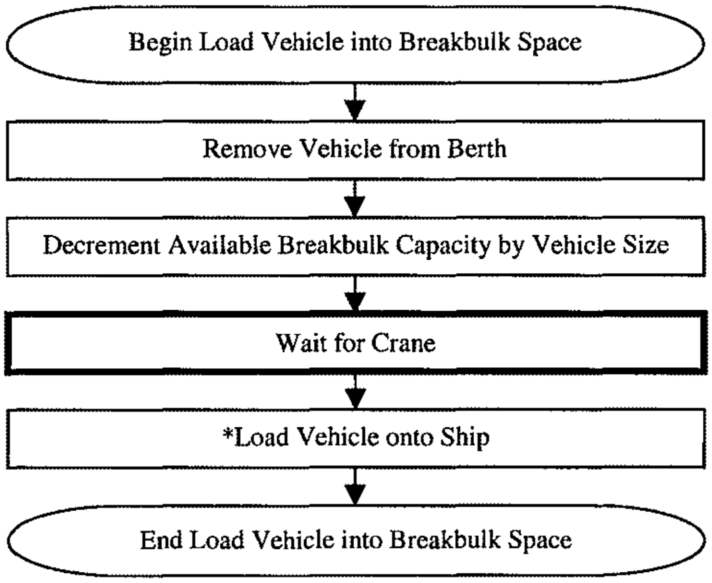

Begin Load Vehicle into Breakbulk Space
Remove Vehicle from Berth
Decrement Available Breakbulk Capacity by Vehicle Size
Wait for Crane
*Load Vehicle onto Ship
End Load Vehicle into Breakbulk Space

Fig. A.72

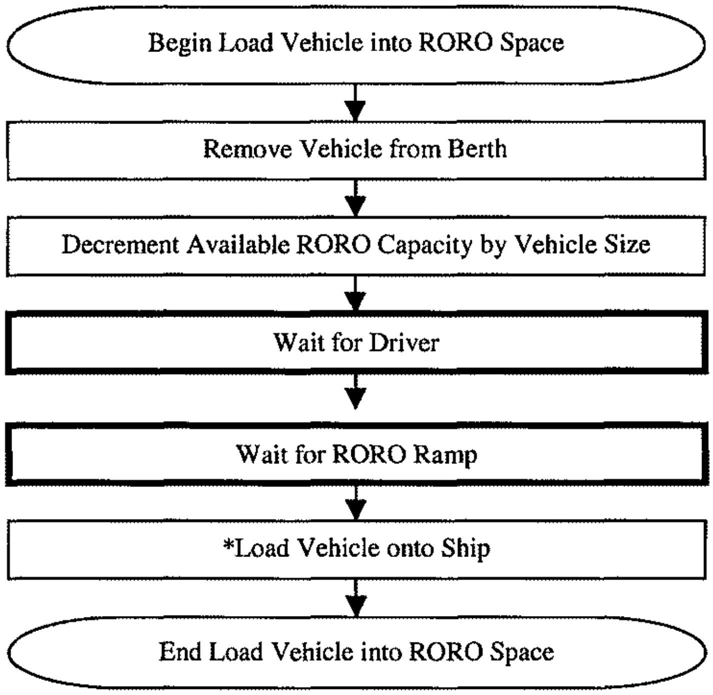

Begin Load Vehicle into RORO Space
Remove Vehicle from Berth
Decrement Available RORO Capacity by Vehicle Size
Wait for Driver
Wait for RORO Ramp
*Load Vehicle onto Ship
End Load Vehicle into RORO Space

Fig. A.73

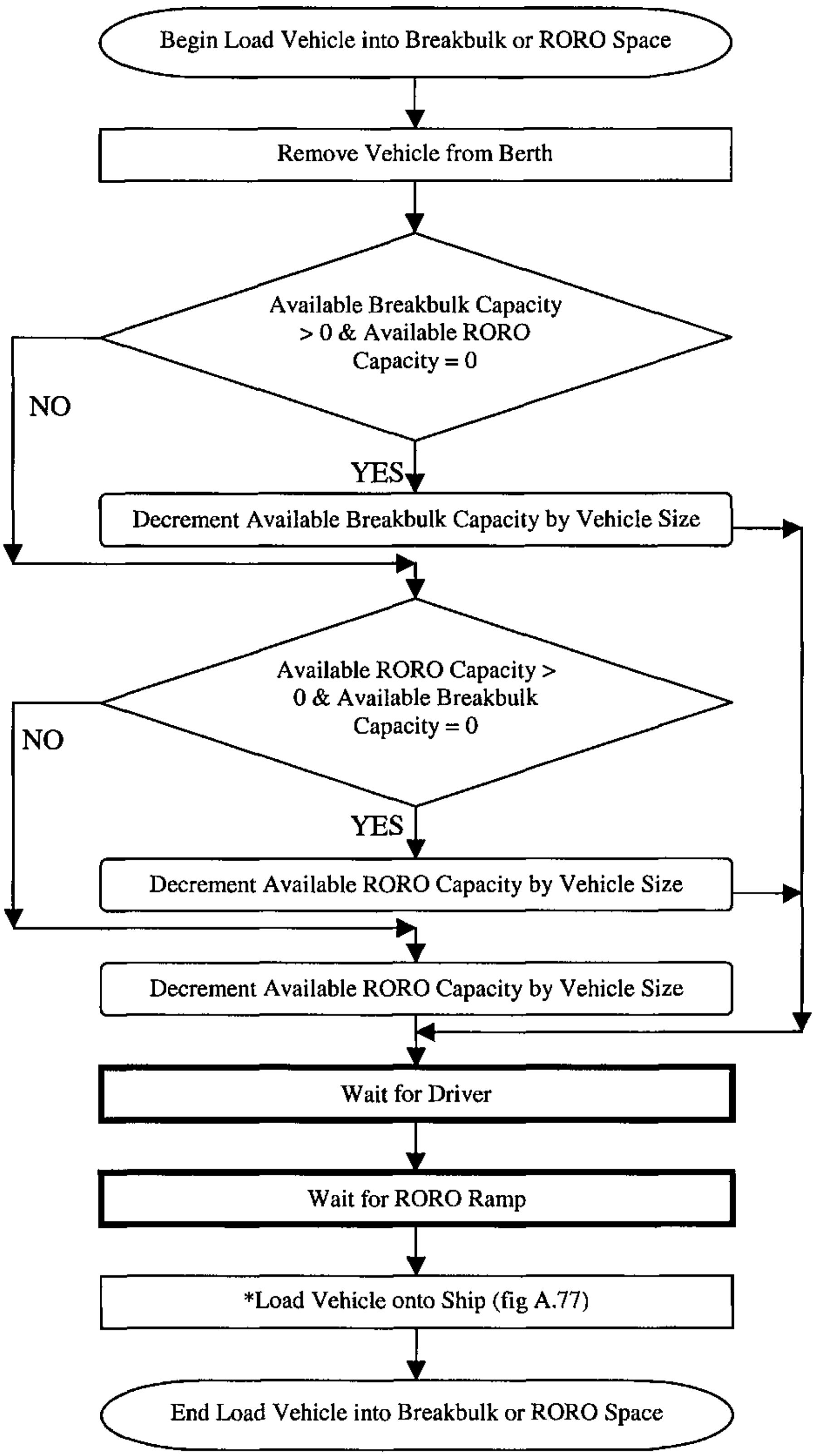

Begin Load Vehicle into Breakbulk or RORO Space
Remove Vehicle from Berth
Available Breakbulk Capacity > 0 & Available RORO Capacity = 0
NO
YES
Decrement Available Breakbulk Capacity by Vehicle Size
Available RORO Capacity > 0 & Available Breakbulk Capacity = 0
NO
YES
Decrement Available RORO Capacity by Vehicle Size
Decrement Available RORO Capacity by Vehicle Size
Wait for Driver
Wait for RORO Ramp
*Load Vehicle onto Ship (fig A.77)
End Load Vehicle into Breakbulk or RORO Space

Fig. A.74

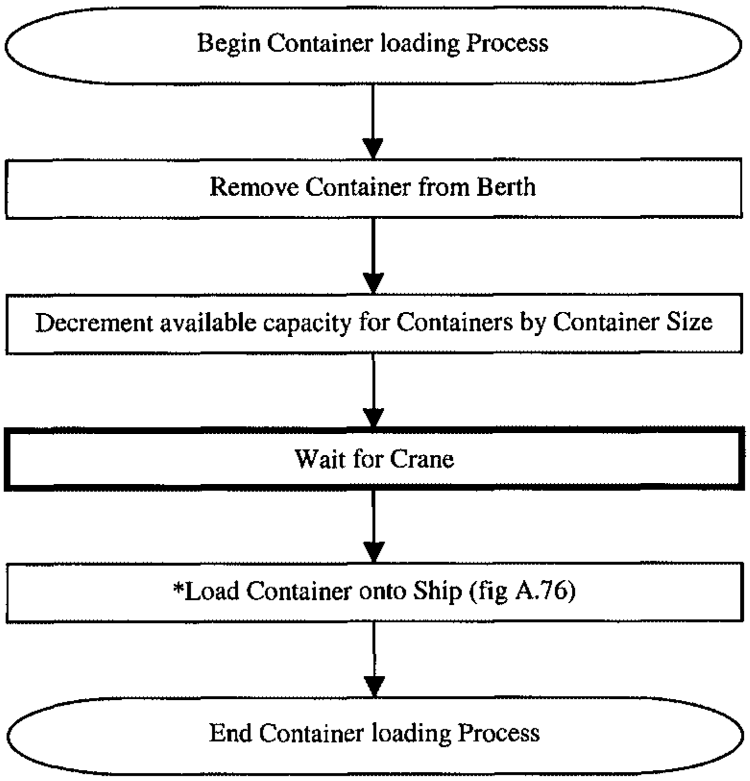

Begin Container loading Process
Remove Container from Berth
Decrement available capacity for Containers by Container Size
Wait for Crane
*Load Container onto Ship (fig A.76)
End Container loading Process

Fig. A.75

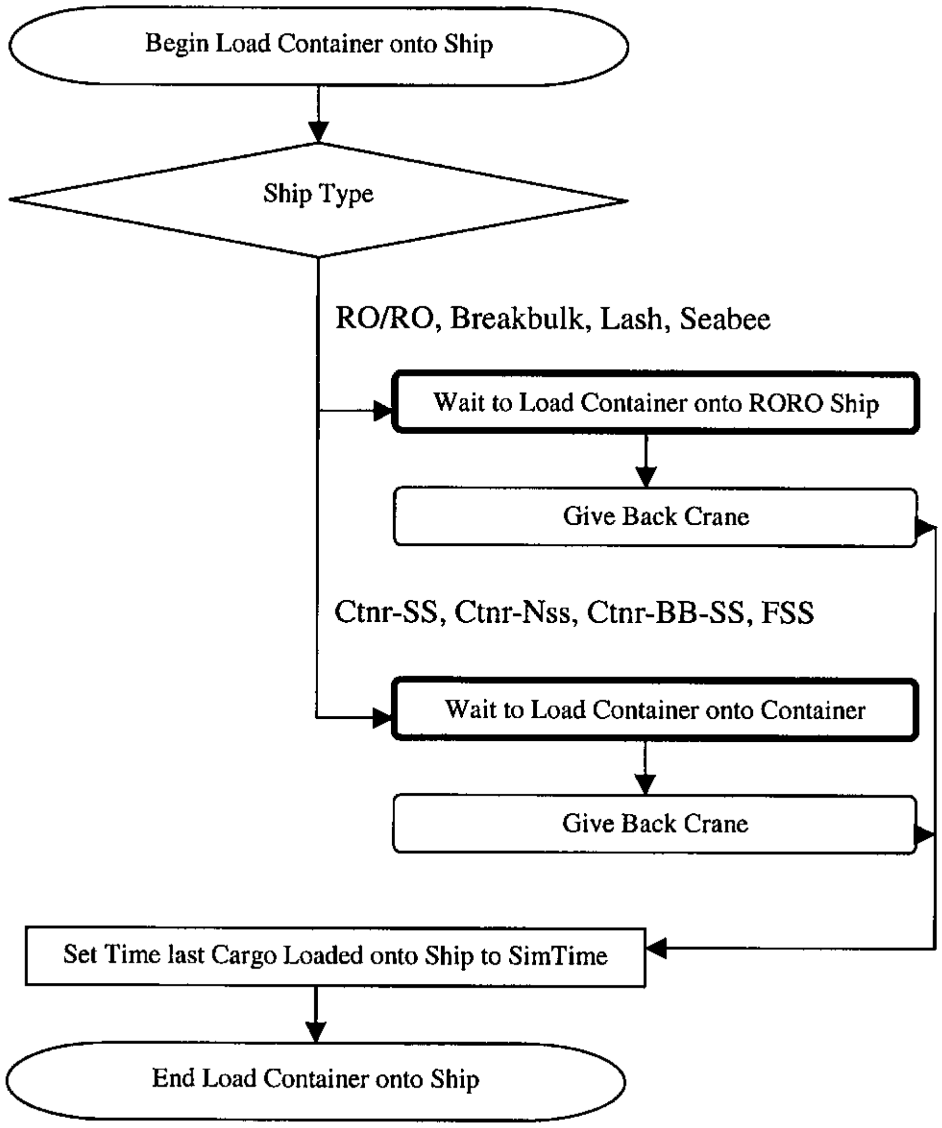

Begin Load Container onto Ship
Ship Type
RO/RO, Breakbulk, Lash, Seabee
Wait to Load Container onto RORO Ship
Give Back Crane
Ctnr-SS, Ctnr-Nss, Ctnr-BB-SS, FSS
Wait to Load Container onto Container
Give Back Crane
Set Time last Cargo Loaded onto Ship to SimTime
End Load Container onto Ship

Fig. A.76

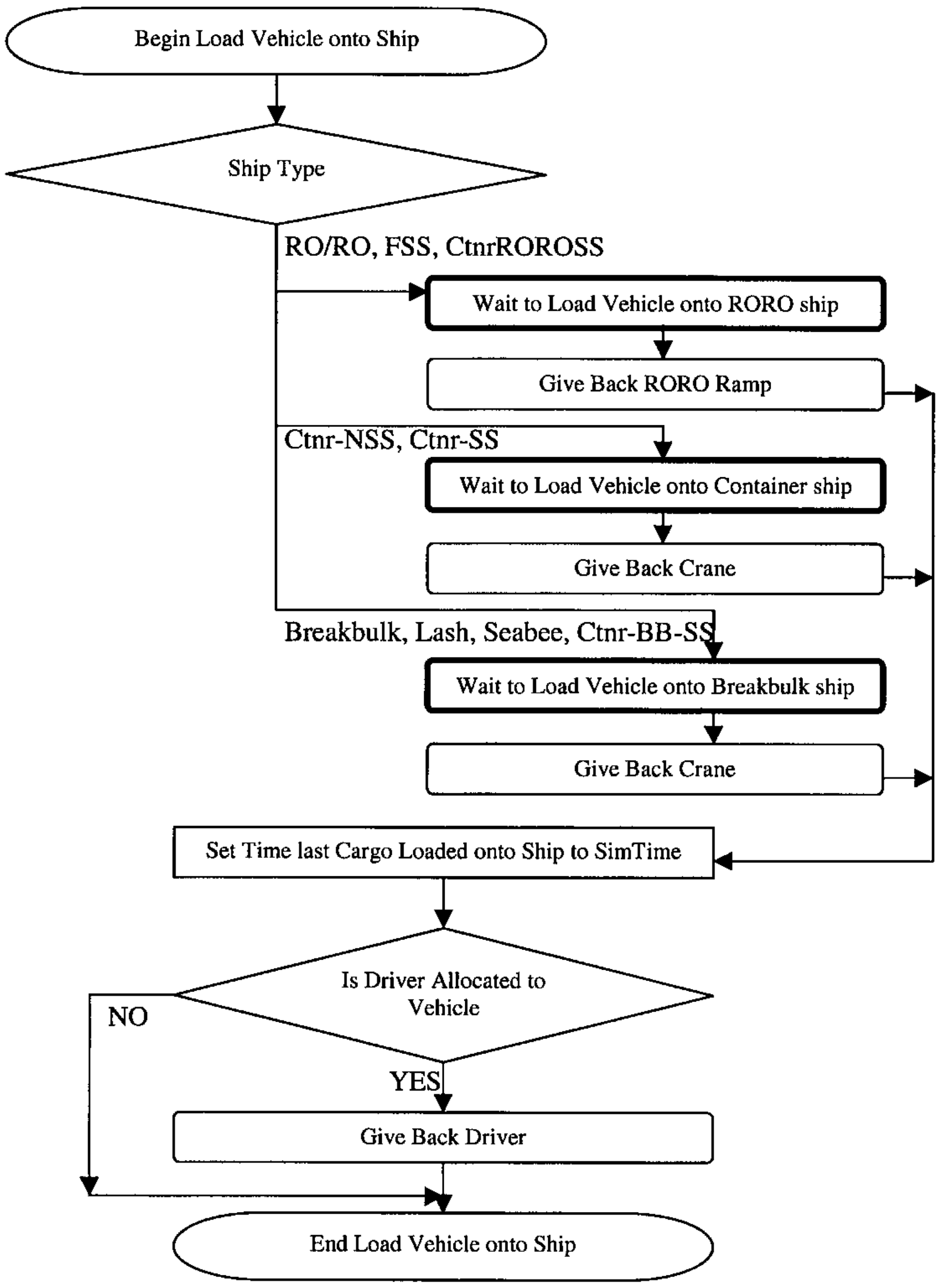

Begin Load Vehicle onto Ship
Ship Type
RO/RO, FSS, CtnrROROSS
Wait to Load Vehicle onto RORO ship
Give Back RORO Ramp
Ctnr-NSS, Ctnr-SS
Wait to Load Vehicle onto Container ship
Give Back Crane
Breakbulk, Lash, Seabee, Ctnr-BB-SS
Wait to Load Vehicle onto Breakbulk ship
Give Back Crane
Set Time last Cargo Loaded onto Ship to SimTime
Is Driver Allocated to Vehicle
NO
YES
Give Back Driver
End Load Vehicle onto Ship

Fig. A.77

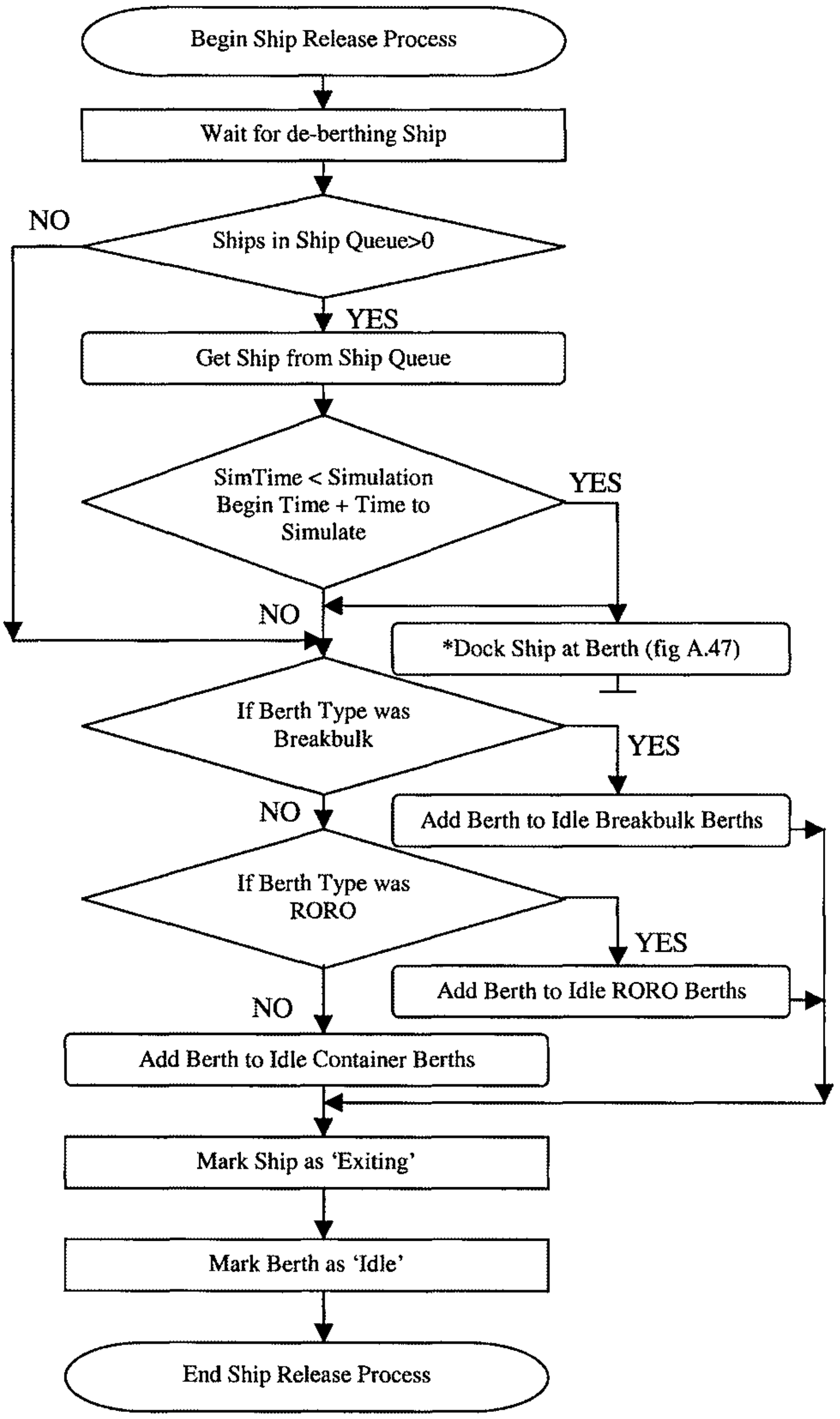

Begin Ship Release Process
Wait for de-berthing Ship
Ships in Ship Queue>0
NO
YES
Get Ship from Ship Queue
SimTime < Simulation Begin Time + Time to Simulate
YES
NO
*Dock Ship at Berth (fig A.47)
If Berth Type was Breakbulk
YES
NO
Add Berth to Idle Breakbulk Berths
If Berth Type was RORO
YES
NO
Add Berth to Idle RORO Berths
Add Berth to Idle Container Berths
Mark Ship as 'Exiting'
Mark Berth as 'Idle'
End Ship Release Process

Fig. A.78

APPENDIX B

LIFE CYCLES OF GRAPHICAL WINDOW OBJECTS IN PORTSIM

The two important graphical window objects that are used extensively as a means of user interface are WindowObj and DialogBoxObj.

B.1 Life Cycle of instances of type WindowObj

This section shows the generic representation of life cycle of instances of type WindowObj used in PORTSIM.

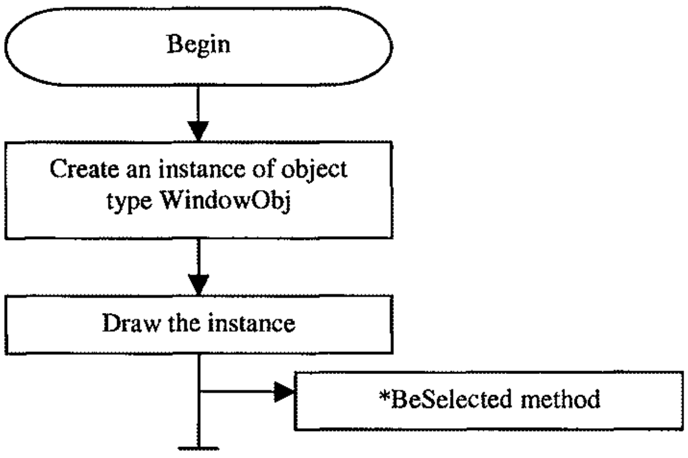


Fig. B.1 Generic representation of Life Cycle of instances of WindowObj

B.2 Life Cycles of instances of type DialogBoxObj

There are two types of dialog box objects used in PORTSIM. One is of DialogBoxObj object type that SIMGRAPHICS provides and another is of object types that are derived from DialogBoxObj.

B.2.1. Life Cycles of instances of type DialogBoxObj

This section shows the generic representation of the life cycles of instances of type DialogBoxObj object.

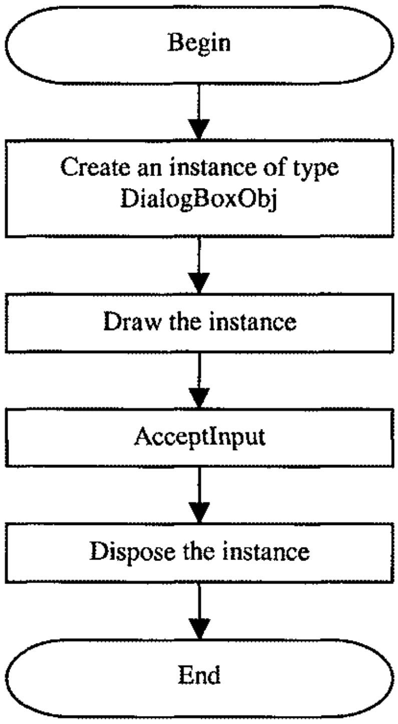


Fig. B.2 Life Cycle of instances of type DialogBoxObj with no processing depending on input

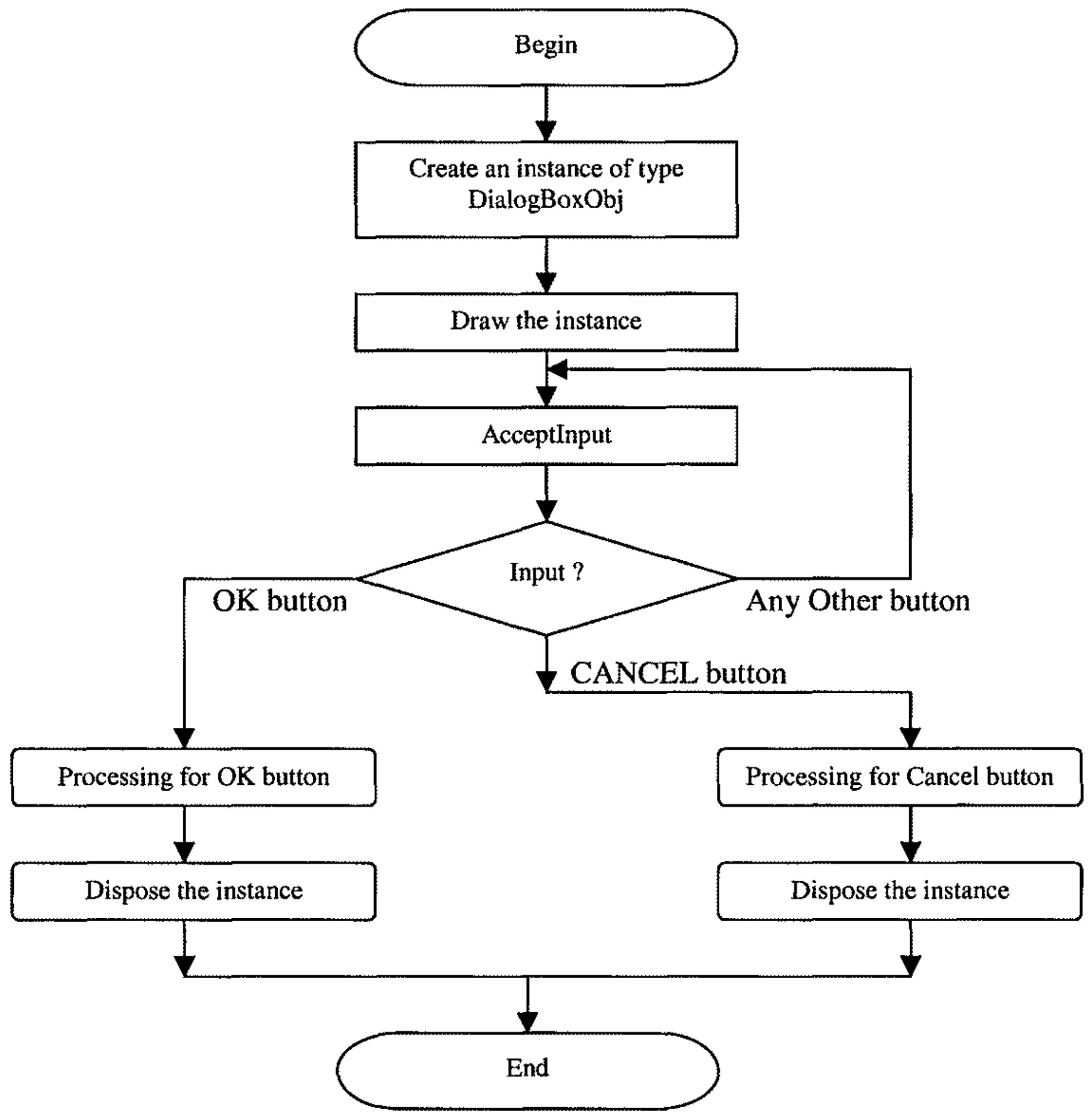


Fig. B.3 Life Cycle of instances of type DialogBoxObj with a loop and with unique processing for 'Ok' and 'Cancel' input

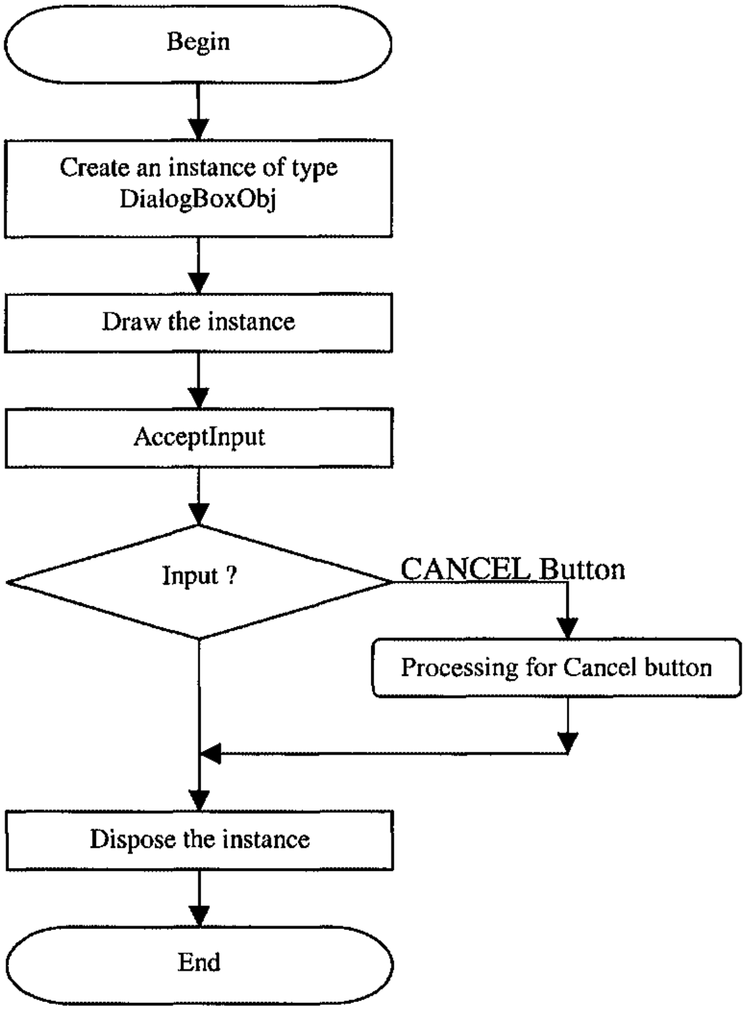


Fig. B.4 Life Cycle of instances of type DialogBoxObj with processing for ‘Cancel’ input alone

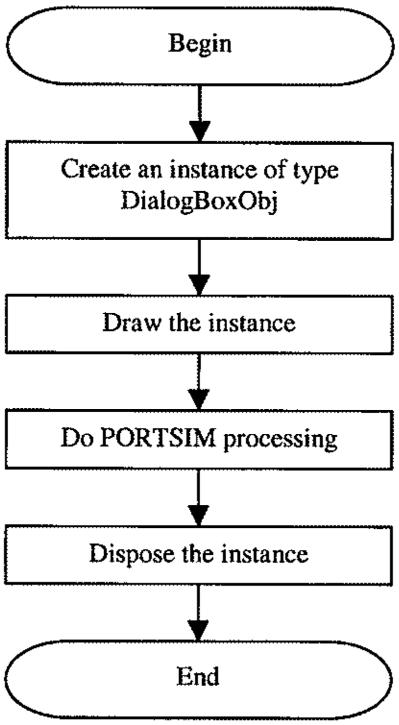


Fig. B.5 Life Cycle of instances of type DialogBoxObj with no provision for accepting input

B.2.2. Life Cycles of instances of Object types that are derived from DialogBoxObj

This section shows the generic representation of the life cycles of instances of dialog boxes of object types that are derived from DialogBoxObj object.

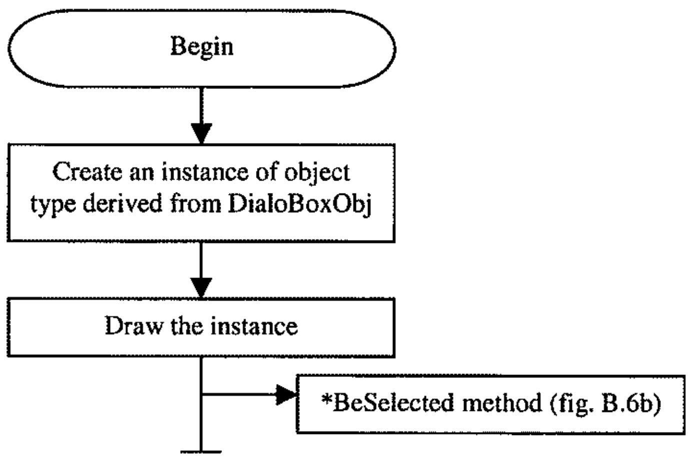

Begin
Create an instance of object type derived from DialoBoxObj
Draw the instance
*BeSelected method (fig. B.6b)

Fig. B.6a Life Cycle of instances of type derived from DialogBoxObj using BeSelected method to accept input

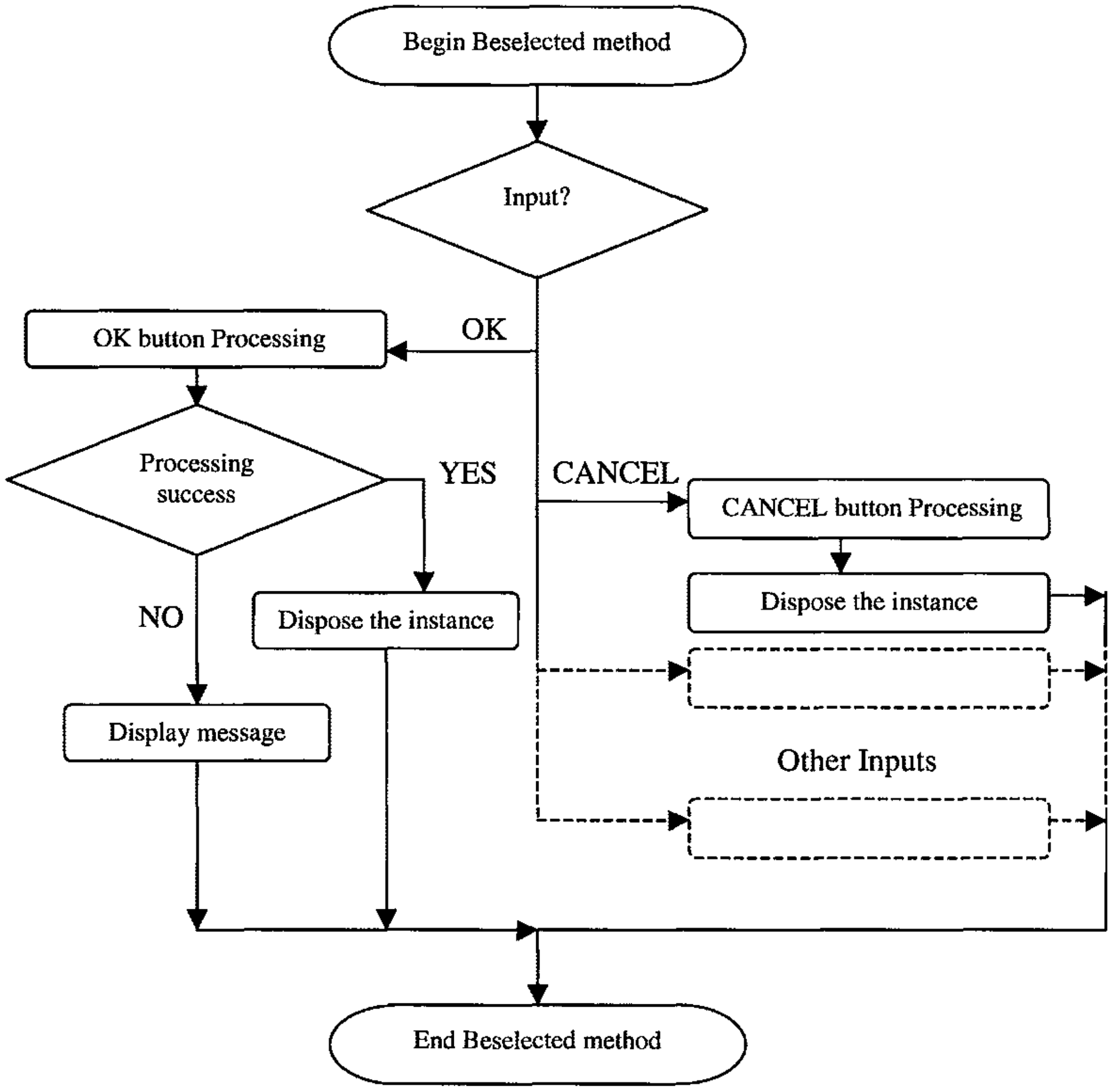


Fig. B.6b Generic representation of Program Flow within the BeSelected method of object types shown in Fig. B.6a

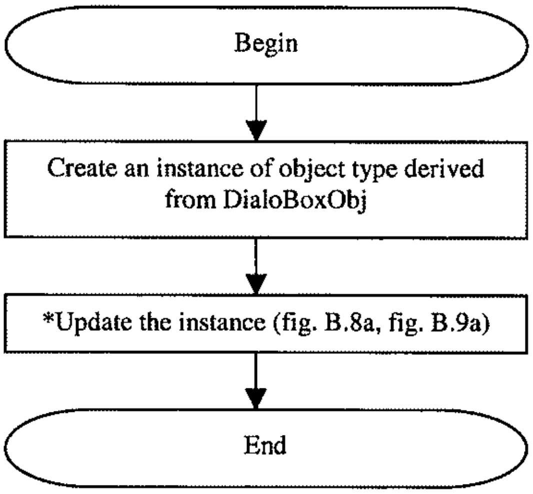

Begin
Create an instance of object type derived from DialoBoxObj
*Update the instance (fig. B.8a, fig. B.9a)
End

Fig. B.7 Life Cycle of instances of type derived from DialogBoxObj using 'Update' function

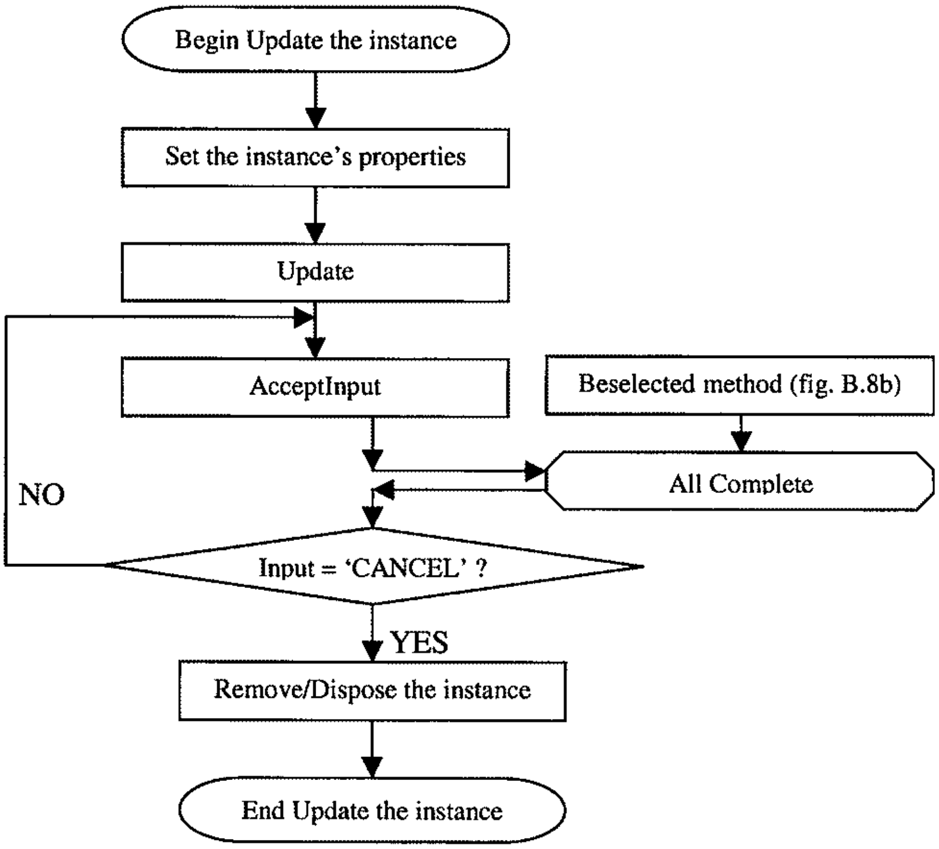

Begin Update the instance
Set the instance's properties
Update
AcceptInput
Beselected method (fig. B.8b)
All Complete
NO
Input = 'CANCEL' ?
YES
Remove/Dispose the instance
End Update the instance

Fig. B.8a One of the generic representations of 'Update' functions shown in Fig. B.7

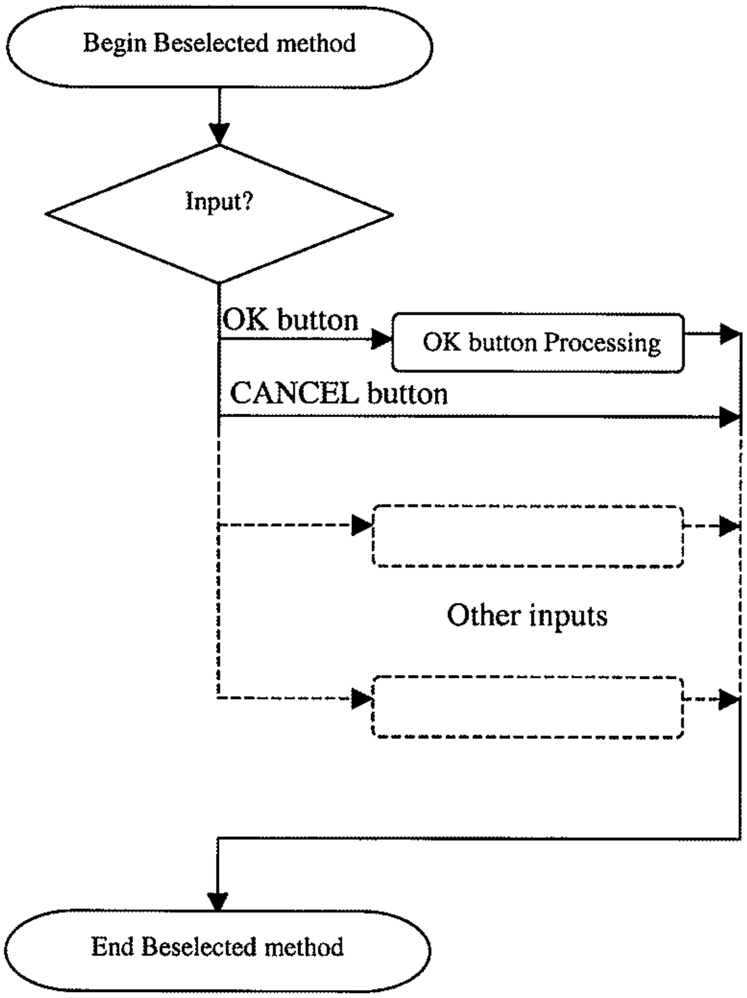


Fig. B.8b Generic representation of Program Flow within the BeSelected method of objects having 'Update' function as shown in Fig. B.8a

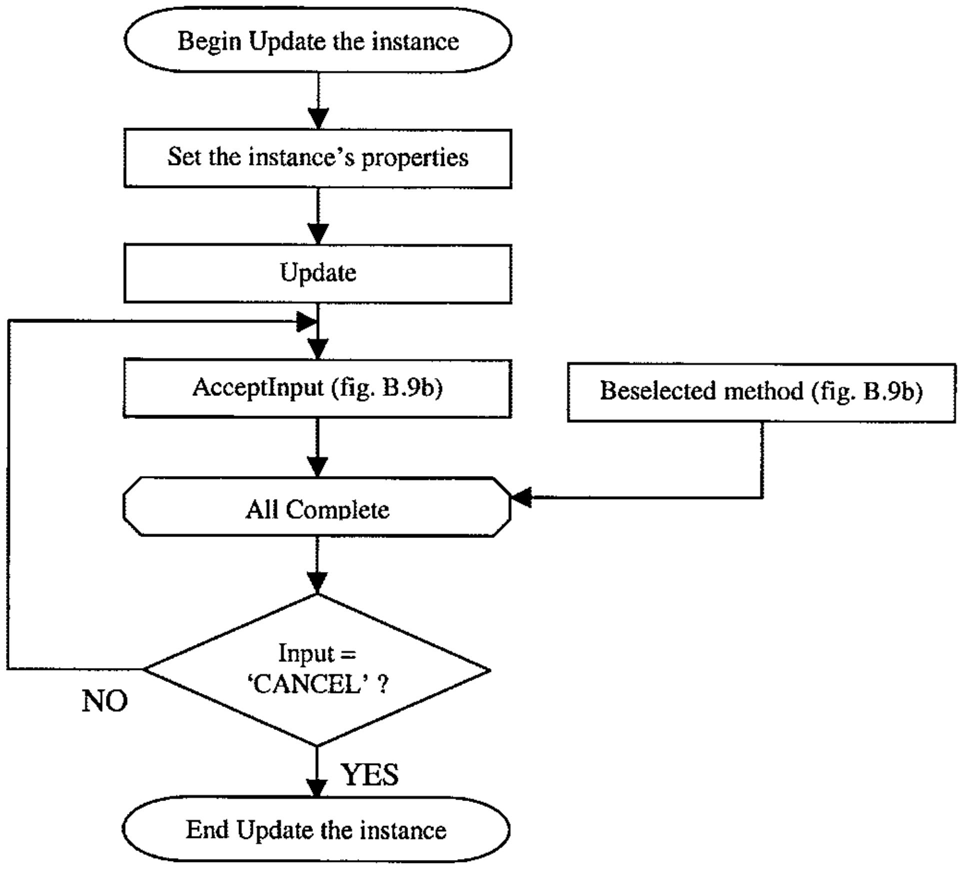


Fig. B.9a One of the generic representations of 'Update' functions shown in Fig. B.7

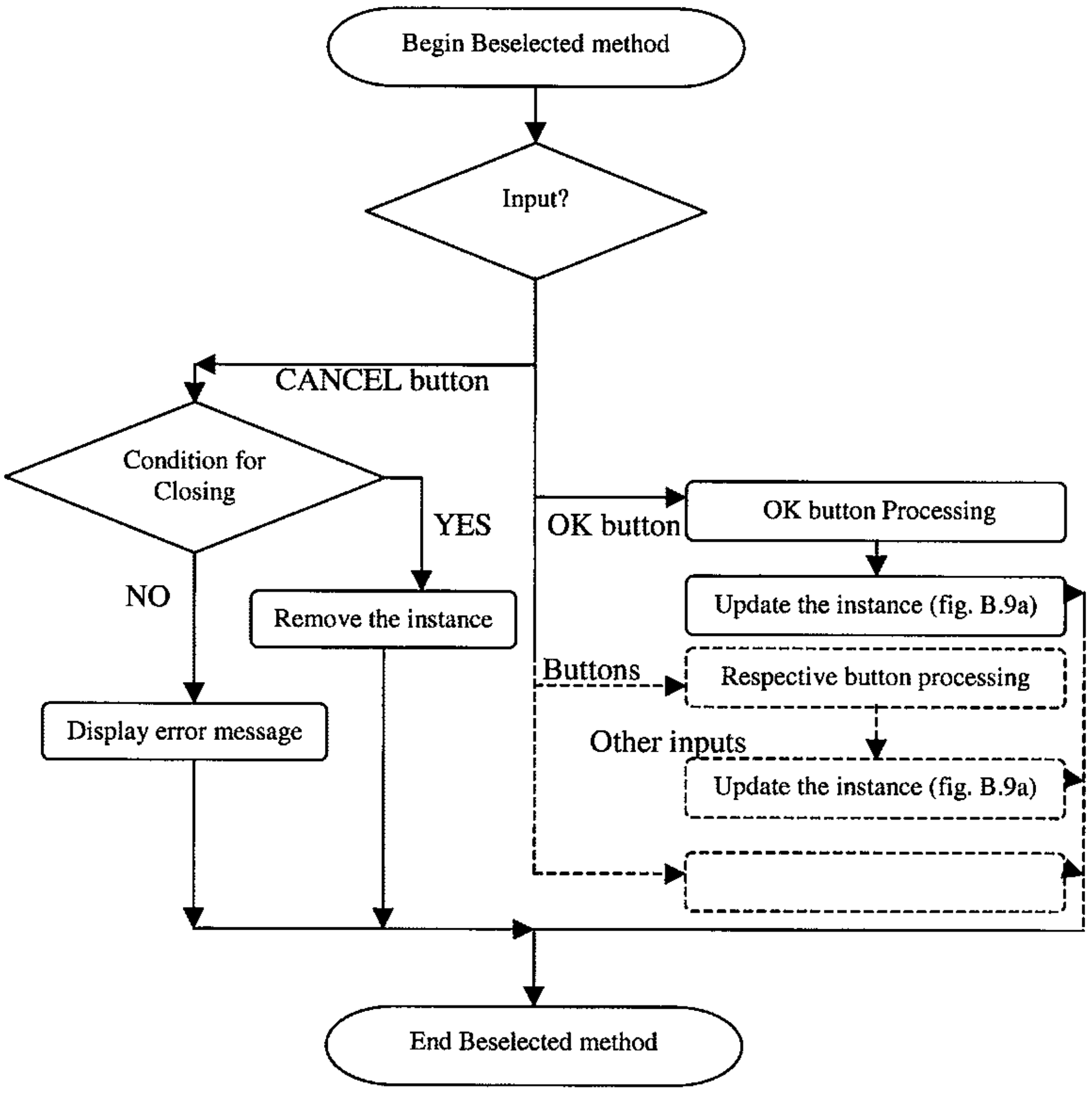


Fig. B.9b Generic representation of Program Flow within the BeSelected method of objects having 'Update' function as shown in Fig. B.9a

APPENDIX C

PORTSIM CODE FOR RELATED FLOW DIAGRAMS

The following Figures represent the actual PORTSIM code for modified flow diagrams, except Fig. C.1, of Appendix A and Appendix B in Sections 3 and 4.

```
TELL METHOD callForwardFlatcars(INinterchangeYard:interchangeYardObj;IN
ranGen : RandomObj);
VAR
 lo,hi    : REAL;
 temp     : REAL;
 stringOfFlatcars : trainObj;
 railcar          : railcarObj;
 tangentLengthAvail : INTEGER;
 len                  : INTEGER;
 j                    : INTEGER;
 tempTime             : REAL;
 svcTime              : REAL;
 typeNeeded           : BOOLEAN;
 numFlatcarsCarryingVehicles      : INTEGER;
 numFlatcarsCarryingContainers    : INTEGER;
 cargo                            : cargoObj;
BEGIN
 WHILE (SimTime() <= (simBeginTime + timeToSimulate))
  {If no flatcars in interchange yard,}
  {wait until a train has been classified in the
  interchange yard}
  {This signifies that railcars are available to call
  forward}
  IF (ordering = "Vehicles/Unit Equipment Only")
   IF (poeMode)
    {Vehicles/Unit Equipment only accepted}
    numFlatcarsCarryingVehicles := ASK interchangeYard TO
                                   countFlatcarsCarryingVehicles();
    WHILE (numFlatcarsCarryingVehicles = 0)
     WAIT FOR trainClassifiedTrigger TO Fire;
     ON INTERRUPT
     END WAIT;
     numFlatcarsCarryingVehicles := ASK interchangeYard TO
                                    countFlatcarsCarryingVehicles();
    END WHILE;
   END IF;
  ELSIF (ordering = "Containers Only")
   IF (poeMode)
    {Containers only accepted}
    numFlatcarsCarryingContainers := ASK interchangeYard TO
    countFlatcarsCarryingContainers();
    WHILE (numFlatcarsCarryingContainers = 0)
     WAIT FOR trainClassifiedTrigger TO Fire;
     ON INTERRUPT
     END WAIT;
   numFlatcarsCarryingContainers := ASK interchangeYard TO
   countFlatcarsCarryingContainers();
  END WHILE;
 END IF;
ELSE
 {Both Vehicles/Unit Equipment and Containers are accepted}
END IF;
IF (status = "idle")
 {Set status to busy}
 ASK SELF TO setStatus("busy");
 tempTime := SimTime();
 svcTime := 0.0;
 {Build the appropriate stringOfFlatcars to send to spur}
 NEW(stringOfFlatcars);
 WHILE (stringOfFlatcars.numberIn = 0)
  ASK SELF TO buildStringOfFlatcar(interchangeYard,stringOfFlatcars);
  IF (stringOfFlatcars.numberIn = 0)
   {OUTPUT("Waiting for more items to be ready in
   emptyRailcarStorage");}
   WAIT DURATION 30.0;
   END WAIT;
  END IF;
 END WHILE;
 {Let the system know that more items can be put in interchangeYard}
 TELL IYTrigger TO Trigger;
 {Obtain a locomotive resource and transit it to the IY}
 WAIT FOR port.locomotiveQ TO Give(stringOfFlatcars,1);
  {Processing Time to switch locomotive to IY}
  IF (useProcessDistributionsForTransitTimes)
  lo := switchSpurToIYTime.actualValue-
        switchSpurToIYStdDev.actualValue;
  hi := switchSpurToIYTime.actualValue +
        switchSpurToIYStdDev.actualValue;
  temp := ranGen.UniformReal(lo,hi);
  IF (temp < 0.0)
   temp := 0.0;
  END IF;
ELSE
 {Calculate the time based upon speed and distance of route}
 {NEED TO ADD THIS}
 {Until this is added, make transit time 0.0}
 temp := 0.0;
END IF;
 WAIT DURATION temp;
 END WAIT;

 {Processing Time to couple flatcars at IY}
 lo := coupleAtIYTime.actualValue - coupleAtIYStdDev.actualValue;
 hi := coupleAtIYTime.actualValue + coupleAtIYStdDev.actualValue;
 temp := ranGen.UniformReal(lo,hi);
 IF (temp < 0.0)
  temp := 0.0;
 END IF;
 WAIT DURATION temp;
 END WAIT;

 IF (type = "OPEN STAGING") OR (type = "Open Staging") OR
```

```
   (type = "OPEN STORAGE") OR (type = "Open Storage")
  {Processing Time to switch flatcars to spur}
  IF (useProcessDistributionsForTransitTimes)
   lo := switchToSpurTime.actualValue -
         switchToSpurStdDev.actualValue;
   hi := switchToSpurTime.actualValue +
         switchToSpurStdDev.actualValue;
   temp := ranGen.UniformReal(lo,hi);
   IF (temp < 0.0)
    temp := 0.0;
   END IF;
  ELSE
   {Calculate the time based upon speed and distance of route}
   {NEED TO ADD THIS}
   {Until this is added, make transit time 0.0}
   temp := 0.0;
  END IF;
  WAIT DURATION temp;
  END WAIT;
 ELSIF (type = "COVERED STAGING") OR (type = "Covered Staging")
  {Processing Time to switch flatcars to dock}
  IF (useProcessDistributionsForTransitTimes)
   lo := switchToDockTime.actualValue -
         switchToDockStdDev.actualValue;
   hi := switchToDockTime.actualValue +
         switchToDockStdDev.actualValue;
   temp := ranGen.UniformReal(lo,hi);
   IF (temp < 0.0)
    temp := 0.0;
   END IF;
  ELSE
   {Calculate the time based upon speed and distance of route}
   {NEED TO ADD THIS}
   {Until this is added, make transit time 0.0}
   temp := 0.0;
  END IF;
  WAIT DURATION temp;
  END WAIT;
 ELSE
  {type = "APRON" or type = "Apron"}
  {Processing Time to switch flatcars to Berth}
  IF (useProcessDistributionsForTransitTimes)
   lo := switchToBerthTime.actualValue -
         switchToBerthStdDev.actualValue;
   hi := switchToBerthTime.actualValue +
         switchToBerthStdDev.actualValue;
   temp := ranGen.UniformReal(lo,hi);
   IF (temp < 0.0)
    temp := 0.0;
   END IF;
  ELSE
   {Calculate the time based upon speed and distance of route}
   {NEED TO ADD THIS}
   {Until this is added, make transit time 0.0}
   temp := 0.0;
  END IF;
  WAIT DURATION temp;
```

```
    END WAIT;
   END IF;
   {Processing Time to uncouple flatcars at spur}
   lo := uncoupleAtSpurTime.actualValue -
         uncoupleAtSpurStdDev.actualValue;
   hi := uncoupleAtSpurTime.actualValue +
         uncoupleAtSpurStdDev.actualValue;
   temp := ranGen.UniformReal(lo,hi);
   IF (temp < 0.0)
    temp := 0.0;
   END IF;
   WAIT DURATION temp;
   END WAIT;
  END WAIT;
  {Make the locomotive available again}
  ASK port.locomotiveQ TO TakeBack(stringOfFlatcars,1);
  IF (poeMode)
   {Wait for vehicles/containers to be discharged from flatcars}
   WAIT FOR SELF TO dischargeFlatcarContents(stringOfFlatcars,ranGen);
   END WAIT;
   DISPOSE(stringOfFlatcars);
   {Cycle out empty flatcars}
   {NOTE: This code needs to be added}
  END IF;
  {Set status "idle" to signify that spur is ready for next string}
  {of railcars}
  ASK SELF TO setStatus("idle");
  {Update operational svcTime parameters}
  svcTime := SimTime() - tempTime;
  totalBusyTime := totalBusyTime + svcTime;
  IF svcTime > maxServiceTime
   maxServiceTime := svcTime;
  END IF;

  TELL spurDoneTrigger TO Trigger;
 ELSE
  WAIT FOR spurDoneTrigger TO Fire;
  END WAIT;
 END IF;
END WHILE;
END METHOD {callForwardFlatcars};
```

Fig. C.1 PORTSIM code for "callForwardFlatcars" method

```
TELL METHOD callForwardFlatcars(INinterchangeYard:interchangeYardObj;IN
ranGen : RandomObj);
VAR
 lo,hi    : REAL;
 temp     : REAL;
 stringOfFlatcars : trainObj;
 railcar          : railcarObj;
 tangentLengthAvail : INTEGER;
 len                : INTEGER;
 j                  : INTEGER;
 tempTime           : REAL;
 svcTime            : REAL;
 typeNeeded         : BOOLEAN;
 numFlatcarsCarryingVehicles      : INTEGER;
 numFlatcarsCarryingContainers    : INTEGER;
 cargo                            : cargoObj;
BEGIN
 WHILE (SimTime() <= (simBeginTime + timeToSimulate))
  {If no flatcars in interchange yard,}
  {wait until a train has been classified in the
  interchange yard}
  {This signifies that railcars are available to call
  forward}
  IF (ordering = "Vehicles/Unit Equipment Only")
   IF (poeMode)
    {Vehicles/Unit Equipment only accepted}
    numFlatcarsCarryingVehicles := ASK interchangeYard TO
                              countFlatcarsCarryingVehicles();
    WHILE (numFlatcarsCarryingVehicles = 0)
     WAIT FOR trainClassifiedTrigger TO Fire;
     ON INTERRUPT
     END WAIT;
     numFlatcarsCarryingVehicles := ASK interchangeYard TO
                              countFlatcarsCarryingVehicles();
    END WHILE;
   END IF;
  ELSIF (ordering = "Containers Only")
   IF (poeMode)
    {Containers only accepted}
    numFlatcarsCarryingContainers := ASK interchangeYard TO
    countFlatcarsCarryingContainers();
    WHILE (numFlatcarsCarryingContainers = 0)
     WAIT FOR trainClassifiedTrigger TO Fire;
     ON INTERRUPT
     END WAIT;
     numFlatcarsCarryingContainers := ASK interchangeYard TO
     countFlatcarsCarryingContainers();
    END WHILE;
   END IF;
  ELSE
   {Both Vehicles/Unit Equipment and Containers are accepted}
  END IF;
  IF (status = "idle")
   {Set status to busy}
   {ASK SELF TO setStatus("busy");} {Removed}
   tempTime := SimTime();
 svcTime := 0.0;
 {Build the appropriate stringOfFlatcars to send to spur}
 NEW(stringOfFlatcars);
 WHILE (stringOfFlatcars.numberIn = 0)
  IF(status = "idle") {"IF" construct added}
   ASK SELF TO
          buildStringOfFlatcars(interchangeYard,stringOfFlatcars);
  END IF;
  IF (stringOfFlatcars.numberIn = 0)
   {OUTPUT("Waiting for more items to be ready in
   emptyRailcarStorage");}
   WAIT DURATION 30.0;
   END WAIT;
  END IF;
 END WHILE;
 {Let the system know that more items can be put in interchangeYard}
 TELL IYTrigger TO Trigger;
 ASK SELF TO setStatus("busy"); {Changed status here}
 {Obtain a locomotive resource and transit it to the IY}
 WAIT FOR port.locomotiveQ TO Give(stringOfFlatcars,1);
  {Processing Time to switch locomotive to IY}
  IF (useProcessDistributionsForTransitTimes)
  lo := switchSpurToIYTime.actualValue-
        switchSpurToIYStdDev.actualValue;
  hi := switchSpurToIYTime.actualValue +
        switchSpurToIYStdDev.actualValue;
  temp := ranGen.UniformReal(lo,hi);
  IF (temp < 0.0)
   temp := 0.0;
  END IF;
ELSE
 {Calculate the time based upon speed and distance of route}
 {NEED TO ADD THIS}
 {Until this is added, make transit time 0.0}
 temp := 0.0;
END IF;
 WAIT DURATION temp;
 END WAIT;

 {Processing Time to couple flatcars at IY}
 lo := coupleAtIYTime.actualValue - coupleAtIYStdDev.actualValue;
 hi := coupleAtIYTime.actualValue + coupleAtIYStdDev.actualValue;
 temp := ranGen.UniformReal(lo,hi);
 IF (temp < 0.0)
  temp := 0.0;
 END IF;
 WAIT DURATION temp;
 END WAIT;

 IF (type = "OPEN STAGING") OR (type = "Open Staging") OR
    (type = "OPEN STORAGE") OR (type = "Open Storage")
  {Processing Time to switch flatcars to spur}
  IF (useProcessDistributionsForTransitTimes)
   lo := switchToSpurTime.actualValue -
         switchToSpurStdDev.actualValue;
   hi := switchToSpurTime.actualValue +
        switchToSpurStdDev.actualValue;
  temp := ranGen.UniformReal(lo,hi);
  IF (temp < 0.0)
   temp := 0.0;
  END IF;
 ELSE
  {Calculate the time based upon speed and distance of route}
  {NEED TO ADD THIS}
  {Until this is added, make transit time 0.0}
  temp := 0.0;
 END IF;
 WAIT DURATION temp;
 END WAIT;
ELSIF (type = "COVERED STAGING") OR (type = "Covered Staging")
 {Processing Time to switch flatcars to dock}
 IF (useProcessDistributionsForTransitTimes)
  lo := switchToDockTime.actualValue -
        switchToDockStdDev.actualValue;
  hi := switchToDockTime.actualValue +
        switchToDockStdDev.actualValue;
  temp := ranGen.UniformReal(lo,hi);
  IF (temp < 0.0)
   temp := 0.0;
  END IF;
 ELSE
  {Calculate the time based upon speed and distance of route}
  {NEED TO ADD THIS}
  {Until this is added, make transit time 0.0}
  temp := 0.0;
 END IF;
 WAIT DURATION temp;
 END WAIT;
ELSE
 {type = "APRON" or type = "Apron"}
 {Processing Time to switch flatcars to Berth}
 IF (useProcessDistributionsForTransitTimes)
  lo := switchToBerthTime.actualValue -
        switchToBerthStdDev.actualValue;
  hi := switchToBerthTime.actualValue +
        switchToBerthStdDev.actualValue;
  temp := ranGen.UniformReal(lo,hi);
  IF (temp < 0.0)
   temp := 0.0;
  END IF;
 ELSE
  {Calculate the time based upon speed and distance of route}
  {NEED TO ADD THIS}
  {Until this is added, make transit time 0.0}
  temp := 0.0;
 END IF;
 WAIT DURATION temp;
 END WAIT;
END IF;
{Processing Time to uncouple flatcars at spur}
lo := uncoupleAtSpurTime.actualValue -
      uncoupleAtSpurStdDev.actualValue;
hi := uncoupleAtSpurTime.actualValue +
          uncoupleAtSpurStdDev.actualValue;
    temp := ranGen.UniformReal(lo,hi);
    IF (temp < 0.0)
     temp := 0.0;
    END IF;
    WAIT DURATION temp;
    END WAIT;
   END WAIT;
   {Make the locomotive available again}
   ASK port.locomotiveQ TO TakeBack(stringOfFlatcars,1);
   IF (poeMode)
    {Wait for vehicles/containers to be discharged from flatcars}
    WAIT FOR SELF TO dischargeFlatcarContents(stringOfFlatcars,ranGen);
    END WAIT;
    DISPOSE(stringOfFlatcars);
    {Cycle out empty flatcars}
    {NOTE: This code needs to be added}
   END IF;
   {Set status "idle" to signify that spur is ready for next string}
   {of railcars}
   ASK SELF TO setStatus("idle");
   {Update operational svcTime parameters}
   svcTime := SimTime() - tempTime;
   totalBusyTime := totalBusyTime + svcTime;
   IF svcTime > maxServiceTime
    maxServiceTime := svcTime;
   END IF;

   TELL spurDoneTrigger TO Trigger;
  ELSE
   WAIT FOR spurDoneTrigger TO Fire;
   END WAIT;
  END IF;
 END WHILE;
 END METHOD {callForwardFlatcars};
```

Fig. C.2 Modified PORTSIM code for "callForwardFlatcars" method

```
WAITFOR METHOD callForwardCargo (IN stream1 : RandomObj; IN myBerth :
         berthObj;IN vehStagingArea : stagingObj;IN contStagingArea :
         stagingObj;IN palletStagingArea : stagingObj);
{This method calls forward 12 items from the staging area at a time}
{If less than 12 items are available to be call forwarded from the
staging}
{area, then this method will take as many as are available.}
VAR
 cont                 : containerObj;
 veh                  : vehicleObj;
 pallet               : palletObj;
 cargo                : ANYOBJ;
 tempveh              : vehicleObj;
 selectedRoute        : routeObj;
 tempreal             : REAL;
 i,j                  : INTEGER;
 cnt                  : INTEGER;
 numItems             : INTEGER; {# of items in staging available to be
                                  loaded}
 contPct,vehPct,palletPct    : REAL;
 flag                 : BOOLEAN;
 callForwardList      : myQueueObj;
BEGIN
 CASE typedesc
  WHEN "Ctnr-NSS":          {Fig. 24}
   {Code before modification
   WHILE (contStagingArea.numContainersReady = 0) OR
         (numContainerHandlers = 0)
    WAIT DURATION 60.0
    END WAIT;
   END WHILE;}
   IF (contStagingArea.numContainersReady = 0) OR {"WHILE" construct}
      (numContainerHandlers = 0)                  {replaced with "IF"}
    WAIT DURATION 60.0
    END WAIT;
   END IF;
   {Checked again after waiting for 60 minutes}
   IF NOT ((contStagingArea.numContainersReady = 0) OR
         (numContainerHandlers = 0))
    numItems := contStagingArea.numContainersReady;
    IF (numItems > numContainerHandlers) OR (numItems > 12)
     IF (numContainerHandlers > 12)
      numItems := 12;
     ELSE
      numItems := numContainerHandlers;
     END IF;
    END IF;

    {Obtain the containers that have been dwelled and are ready to
     load,}
    {discharge them from the staging area, and then transit them to
     berth.}
    {Code for sending Containers to Berth}
   END IF;
  WHEN "Ctnr-SS":           {Fig. 24}
   {Code before modification
   WHILE (contStagingArea.numContainersReady = 0) OR
```

```
        (numContainerHandlers = 0)
     WAIT DURATION 60.0
     END WAIT;
    END WHILE;}
    IF (contStagingArea.numContainersReady = 0) OR {"WHILE" construct}
       (numContainerHandlers = 0)                  {replaced with "IF"}
     WAIT DURATION 60.0
     END WAIT;
    END IF;
    {Checked again after waiting for 60 minutes}
    IF NOT ((contStagingArea.numContainersReady = 0) OR
          (numContainerHandlers = 0))
     numItems := contStagingArea.numContainersReady;
     IF (numItems > numContainerHandlers) OR (numItems > 12)
      IF (numContainerHandlers > 12)
       numItems := 12;
      ELSE
       numItems := numContainerHandlers;
      END IF;
     END IF;

     {Obtain the containers that have been dwelled and are ready to
      load,}
     {discharge them from the staging area, and then transit them to
      berth.}
     {Code for sending Containers to Berth}
    END IF;
   WHEN "RO/RO":           {Fig. 25}
    {Code before modification
    WHILE (vehStagingArea.numVehiclesReady = 0) OR (numDrivers = 0)
     WAIT DURATION 60.0
     END WAIT;
    END WHILE;}
    {"WHILE" construct replaced with "IF"}
    IF (vehStagingArea.numVehiclesReady = 0) OR (numDrivers = 0)
     WAIT DURATION 60.0
     END WAIT;
    END IF;
    {Checked again after waiting for 60 units}
    IF NOT ((vehStagingArea.numVehiclesReady = 0) OR (numDrivers = 0))
     i := 0; numItems := 0;
     numItems := vehStagingArea.numVehiclesReady;
     IF (numItems > numDrivers) OR (numItems > 12)
      IF (numDrivers > 12)
       numItems := 12;
      ELSE
       numItems := numDrivers;
      END IF;
     END IF;
     {Obtain the vehicles that have been inspected and dwelled and are
      ready to load,}
     {discharge them from the staging area, and then transit them to
      berth.}
     {Code for sending Vehicles to Berth}
    END IF;
   WHEN "Breakbulk":       {Fig. 25}
    {Code before modification
```

```
WHILE (vehStagingArea.numVehiclesReady = 0) OR (numDrivers = 0)
 WAIT DURATION 60.0
 END WAIT;
END WHILE;}
{"WHILE" construct replaced with "IF"}
IF (vehStagingArea.numVehiclesReady = 0) OR (numDrivers = 0)
 WAIT DURATION 60.0
 END WAIT;
END IF;
{Checked again after waiting for 60 units}
IF NOT ((vehStagingArea.numVehiclesReady = 0) OR (numDrivers = 0))
 numItems := vehStagingArea.numVehiclesReady;
 IF (numItems > numDrivers) OR (numItems > 12)
  IF (numDrivers > 12)
   numItems := 12;
  ELSE
   numItems := numDrivers;
  END IF;
 END IF;

 {Obtain the vehicles that have been inspected and dwelled and are
  ready to load,}
 {discharge them from the staging area, and then transit them to
  berth.}
 {Code for sending Vehicles to Berth}
END IF;
OTHERWISE                 {Fig. 26}
 {Need to have a mechanism for discharging both vehicles and
  containers.}
 {Use a distribution that will discharge a percentage of vehicles
  equal to}
 {the percentage of overall area in the ship it is supposed to load.}
 {Code before modification
 WHILE ((vehStagingArea.numVehiclesReady = 0) AND
        (contStagingArea.numContainersReady = 0))
  WAIT DURATION 60.0
  END WAIT;
 END WHILE;}
 {"WHILE" construct replaced with "IF"}
 IF ((vehStagingArea.numVehiclesReady = 0) AND
     (contStagingArea.numContainersReady = 0))
  WAIT DURATION 60.0
  END WAIT;
 END IF;

 {Checked again after waiting for 60 units}
 IF NOT ((vehStagingArea.numVehiclesReady = 0) AND
         (contStagingArea.numContainersReady = 0))
  IF (contStagingArea.numContainersReady  = 0)
   {OUTPUT("Trying to send vehicles only");}
   {Send only vehicles}
   {Limit the number of vehicles to call forward by:}
   {1) # of vehicles ready, 2) 12 maximum, 3) # of drivers}
   numItems := vehStagingArea.numVehiclesReady;
   numItems := MINOF(numItems,12);
   {numItems := MINOF(numItems,numDrivers);}
   WHILE (numItems = 0)
```

```
      WAIT DURATION 60.0;
      END WAIT;
      {Limit the number of vehicles to call forward by:}
      {1) # of vehicles ready, 2) 12 maximum, 3) # of drivers}
      numItems := vehStagingArea.numVehiclesReady;
      numItems := MINOF(numItems,12);
     END WHILE;

     {Obtain the vehicles that have been inspected and dwelled and are
      ready to load,}
     {discharge them from the staging area, and then transit them to
      berth.}
     {Code for sending Vehicles to berth}
    ELSIF (vehStagingArea.numVehiclesReady = 0)
    {OUTPUT("Trying to send containers only");}
    {Limit the number of vehicles to call forward by:}
    {1) # of containers ready, 2) 12 maximum, 3) # of container
        handlers}
    numItems := contStagingArea.numContainersReady;
    numItems := MINOF(numItems,12);
    WHILE (numItems = 0)
     WAIT DURATION 60.0;
     END WAIT;
     numItems := contStagingArea.numContainersReady;
     numItems := MINOF(numItems,12);
     {numItems := MINOF(numItems,numContainerHandlers);}
    END WHILE;

    {Obtain the containers that have been dwelled and are ready to
     load,}
    {discharge them from the staging area, and then transit them to
     berth.}
    {Code for sending Containers to Berth}
   ELSE
    {Code for sending both containers and vehicles to the berth}
   END IF;
  END IF;
 END CASE;
END METHOD {callForwardCargo};
```

Fig. C.3 Modified PORTSIM code for "callForwardCargo"

```
DEFINITION MODULE aanimate;
 ...............
 wrapDialogBoxObj = OBJECT(DialogBoxObj);
  OVERRIDE
   ASK METHOD BeClosed;
 END OBJECT {wrapDialogBoxObj};
 ...............
END MODULE {aanimate}

IMPLEMENTATION MODULE aanimate;
 ...............
 OBJECT wrapDialogBoxObj;
  ASK METHOD BeClosed;    {Fig. 27}
   BEGIN
    ASK SELF TO Erase;
  END METHOD;
 END OBJECT;

 OBJECT XXXXX;
  ASK METHOD XXXXX;
   VAR
    dialogBox        : DialogBoxObj;
    wdialogBox       : wrapDialogBoxObj;
    ...............
   BEGIN
    ...............
    NEW(wdialogBox);      {Fig. 28}
    ASK wdialogBox TO
            LoadFromLibrary(library,"transportModesErrorDialog");
    ASK portsimMainWin TO AddGraphic(wdialogBox);
    item := ASK wdialogBox TO AcceptInput();
    ASK portsimMainWin TO RemoveThisGraphic(wdialogBox);
    DISPOSE(wdialogBox);
    ...............
  END METHOD;
 END OBJECT;
 ...............
END MODULE.
```

Fig. C.4 PORTSIM code for Figures 27 and 28

```
IMPLEMENTATION MODULE aanimate;
 ...............
 OBJECT XXXXX;
  ASK METHOD XXXXX;
   VAR
    dialogBox         : DialogBoxObj;
    wdialogBox2       : wrapDialogBoxObj;
    ...............
   BEGIN
    ...............
    NEW (wdialogBox2);
    ASK portsimMainWin TO AddGraphic(wdialogBox2);
    REPEAT
     item3 := ASK wdialogBox2 TO AcceptInput();
     IF item3 = NILOBJ
      EXIT;
     END IF;
    UNTIL (ASK item3 ReferenceName = "ok") OR
          (ASK item3 ReferenceName = "cancel");
    IF item3 <> NILOBJ
     IF (ASK item3 ReferenceName = "ok")
      ...............
      ASK portsimMainWin TO RemoveThisGraphic(wdialogBox2);
      ...............
     ELSE
      {Remove this dialog box}
      ASK portsimMainWin TO RemoveThisGraphic(wdialogBox2);
      DISPOSE(wdialogBox2);
     END IF;
    ELSE
     ASK portsimMainWin TO RemoveThisGraphic(wdialogBox2);
     DISPOSE(wdialogBox2);
   END IF;
   ...............
  END METHOD;
 END OBJECT;
 ...............
END MODULE.
```

Fig. C.5 PORTSIM code for Fig. 29

```
IMPLEMENTATION MODULE aanimate;
 ...............
 OBJECT XXXXX;
  ASK METHOD XXXXX;
   VAR
    dialogBox        : DialogBoxObj;
    wdialogBox       : wrapDialogBoxObj;
    ...............
   BEGIN
    ...............
    NEW(wdialogBox);
    ASK wdialogBox TO LoadFromLibrary(library,"scenarioExistsDialog");
    ASK portsimMainWin TO AddGraphic(wdialogBox);
    ASK wdialogBox TO Draw;
    button := ASK wdialogBox TO AcceptInput();
    IF button <> NILOBJ
     IF ASK button ReferenceName = "cancel"
      flag := FALSE;
     END IF;
    ELSE
     flag := FALSE;
    END IF;
    DISPOSE(wdialogBox);
   ...............
  END METHOD;
 END OBJECT;
 ...............
END MODULE.
```

Fig. C.6 PORTSIM code for Fig. 30

```
DEFINITION MODULE aanimate;
 ...............
 shipRemoveDialogBoxObj = OBJECT(DialogBoxObj);
 OVERRIDE
  ASK METHOD BeSelected;
  ASK METHOD BeClosed;
 END OBJECT {shipRemoveDialogBoxObj};
 ...............
END MODULE.
```

```
IMPLEMENTATION MODULE aanimate;
 ...............

 OBJECT shipRemoveDialogBoxObj;
  ASK METHOD BeSelected;
  VAR
   ...............
  BEGIN
   cnt := ASK LastPicked Id;
   CASE cnt
     WHEN 1: {ok}
      ...............
      ASK portsimMainWin TO RemoveThisGraphic(SELF);
      DISPOSE(SELF);
     WHEN .........
     WHEN 2: {cancel}
      ASK portsimMainWin TO RemoveThisGraphic(SELF);
      DISPOSE(SELF);
     OTHERWISE
   END CASE;
  END METHOD {BeSelected};

  ASK METHOD BeClosed;   {Fig. 31}
  BEGIN
   ASK portsimMainWin TO RemoveThisGraphic(SELF);
   DISPOSE(SELF);
  END METHOD;
 END OBJECT {shipRemoveDialogBoxObj};

 OBJECT scenarioDialogBoxObj;
  ASK METHOD BeSelected;
  VAR
   ...............
  BEGIN
   CASE ASK LastPicked Id
    WHEN .........
    WHEN 12: {removeShipButton}
     NEW(shipRemoveDialogBox);     {Fig. 32}
     ASK shipRemoveDialogBox TO
                  LoadFromLibrary(library,"shipRemoveDialog");
     {Initialize ship list}
     shipTable := ASK SELF Child("shipTable",10);
     shipListBox := ASK shipRemoveDialogBox Child("shipList",5);
     NEW(shipTableItem);
     shipTableItem := ASK shipTable FirstGraphic();
     WHILE shipTableItem <> NILOBJ
      NEW(shipListBoxItem);
      ASK shipListBoxItem TO SetText(shipTableItem.Label);
      ASK shipListBox TO AddGraphic(shipListBoxItem);
      shipTableItem := ASK shipTable NextGraphic(shipTableItem);
     END WHILE;
     ASK portsimMainWin TO AddGraphic(shipRemoveDialogBox);
     ASK shipRemoveDialogBox TO Draw;
    WHEN .........
   END CASE;
  END METHOD;
 END OBJECT {scenarioDialogBoxObj};
 ...............
END MODULE.
```

Fig. C.7 PORTSIM code for Figures 31 and 32

```
DEFINITION MODULE aanimate;
 ...............
 gateParamsDialogBoxObj = OBJECT(DialogBoxObj);
  ASK METHOD updateGateParamsDialog(IN Num : INTEGER);
  OVERRIDE
   ASK METHOD BeSelected;
   ASK METHOD BeClosed;
 END OBJECT {gateParamsDialogBoxObj};
 ...............
END MODULE.

IMPLEMENTATION MODULE aanimate;
 ...............
 OBJECT gateParamsDialogBoxObj;
  ASK METHOD updateGateParamsDialog(IN Num : INTEGER);
  VAR
   ...............
  BEGIN
   ...............
   ASK SELF TO Update;
   REPEAT
    item := ASK SELF TO AcceptInput();
    IF item = NILOBJ
     EXIT;
    END IF;
   UNTIL (ASK item ReferenceName = "cancel");
   IF (ASK portsimMainWin IncludesGraphic(SELF))
       ASK portsimMainWin TO RemoveThisGraphic(SELF);
   END IF;
  END METHOD;

  ASK METHOD BeClosed;
  BEGIN
   ASK SELF TO Erase;
  END METHOD;
 END OBJECT {gateParamsDialogBoxObj};
 ...............
END MODULE.
```

Fig. C.8 PORTSIM code for Fig. 33

```
DEFINITION MODULE aanimate;
 ...............
 stagingParamsDialogBoxObj = OBJECT(DialogBoxObj);
  ASK METHOD updateStagingParamsDialog(IN Num : INTEGER);
  OVERRIDE
   ASK METHOD BeSelected;
   ASK METHOD BeClosed;
  END OBJECT {stagingParamsDialogBoxObj};
 ...............
END MODULE.

IMPLEMENTATION MODULE aanimate;
 ...............
 OBJECT stagingParamsDialogBoxObj;
  ASK METHOD updateStagingParamsDialog(IN Num : INTEGER);
  VAR
   ...............
  BEGIN
   ...............
   REPEAT
    item := ASK SELF TO AcceptInput();
    IF item = NILOBJ
     {Ensure at least one staging area is available before allowing
      user to exit}
     IF (availStagingAreas.numberIn < 0)
      NEW(wdialogBox);
      ASK wdialogBox TO
            LoadFromLibrary(library,"selectStagingErrorDialog");
      ASK portsimMainWin TO AddGraphic(wdialogBox);
      item := ASK wdialogBox TO AcceptInput();
      ASK portsimMainWin TO RemoveThisGraphic(wdialogBox);
      DISPOSE(wdialogBox);
     ELSE
      IF (ASK portsimMainWin IncludesGraphic(SELF))
       ASK portsimMainWin TO RemoveThisGraphic(SELF);
      END IF;
     END IF;
     EXIT;
    END IF;
   UNTIL (ASK item ReferenceName = "cancel");
  END METHOD {updateStagingParamsDialog};

  ASK METHOD BeSelected;
  VAR
   ...............
  BEGIN
   CASE name
    WHEN "ok":
     ...............
    WHEN "cancel":
     {Ensure at least one staging area is available before allowing
      user to exit}
     IF (availStagingAreas.numberIn < 0)
      NEW(wdialogBox);
      ASK wdialogBox TO
            LoadFromLibrary(library,"selectStagingErrorDialog");
      ASK portsimMainWin TO AddGraphic(wdialogBox);
      item := ASK wdialogBox TO AcceptInput();
      ASK portsimMainWin TO RemoveThisGraphic(wdialogBox);
      DISPOSE(wdialogBox);
     ELSE
      IF (ASK portsimMainWin IncludesGraphic(SELF))
       ASK portsimMainWin TO RemoveThisGraphic(SELF);
      END IF;
     END IF;
    WHEN .........
   END CASE;
  END METHOD {BeSelected};

  ASK METHOD BeClosed;
  BEGIN
    ASK SELF TO Erase;
  END METHOD;
 END OBJECT {stagingParamsDialogBoxObj};
 ...............
END MODULE.
```

Fig. C.9 PORTSIM code for Fig. 34

CURRICULUM VITA
For
MURALI K ADATRAO

NAME: Murali K Adatrao

DATE OF BIRTH: August 21, 1974

DEGREES:

Bachelor of Technology (Electronics and Communication Engineering), Jawaharlal Nehru Technological University, College of Engineering, Hyderabed, Andhra Pradesh, India, July 1996