2009

An Integrated Simulation Environment Combining Process-Driven and Event-Driven Models

Vishnu Sharma Kesaraju
Wright State University

Follow this and additional works at: http://corescholar.libraries.wright.edu/etd_all
Part of the Engineering Commons

Repository Citation

This Dissertation is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu.
AN INTEGRATED SIMULATION ENVIRONMENT COMBINING
PROCESS-DRIVEN AND EVENT-DRIVEN MODELS

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

By

VISHNU S. KESARAJU
M.S., Wright State University, 2005

2009
Wright State University
I HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER MY SUPERVISION BY VISHNU S. KESARAJU, ENTITLED AN INTEGRATED SIMULATION ENVIRONMENT COMBINING PROCESS-DRIVEN AND EVENT-DRIVEN MODELS BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy.

______________________________
Frank W. Ciarallo, Ph.D.
Dissertation Director

______________________________
Ramana V. Grandhi, Ph.D.
Director, Ph.D in Engineering Program

______________________________
Joseph F. Thomas, Jr., Ph.D
Dean, School of Graduate Studies

Committee on Final Examination

______________________________
Frank W. Ciarallo, Ph.D.

______________________________
S. Narayanan, Ph.D.

______________________________
Raymond R. Hill, Ph.D.

______________________________
Thomas Hartrum, Ph.D.

______________________________
Marcus B. Perry, Ph.D.
ABSTRACT


A simulation framework that integrates process-driven and event-driven approaches offers a powerful combination of tools to the modeler. In process-driven simulation models, the system can be represented by block diagrams or system networks through which entities flow to mimic real life system objects. In event-driven models, the system can be represented by event graphs, which focus on the abstraction of the event rather than on observable physical entities. In this research, a simulation environment is proposed to integrate both the approaches, i.e. process and event. The main purpose of this research is to mitigate complexity of large models through process orientation, while retaining the control over the attributes, variables and the logic through event orientation. Discrete event simulation is often taught to the students at either the event level or the process level. A simulation tool that effectively preserves both the levels would be useful from the simulation education perspective.

An important feature of standard event graphs is parameterization of the event vertices, allowing similar model sub-graphs to be combined together as a generic sub-graph distinguished by parameter values. A framework based on an Integrated Entity/Event ($IE^3$) approach has been further enhanced to explicitly represent entities at the event-driven level. The integrated simulation framework works towards attenuation of the abstraction involved in parameter passing. The solution lies in explicitly passing the entities through the event-driven model. Event parameters are replaced by entity attributes. The usage of entities in the event-driven layer serves two purpose, a) reduces
the abstraction by manipulating entity objects instead of working with parameters as in a programming language, and b) gives the intuitive feel of process-driven models to modelers at the event level, which enhances the appeal of the event-driven models.

The advantage of using the entity attributes in the $IE^2$ model is that the similar model sub-graphs can be combined together as a generic sub-graph distinguished by entity attribute values. At the event level, entities are handled as objects in a way that is analogous to their treatment in the process models. The attributes of an entity are defined by the modeler, enabling the flexibility and explicit handling of entities at the event level. Instead of passing information as event parameters to other nodes as in a programming language, the $IE^2$ model defines them explicitly as attributes of entities that are associated with events as they are scheduled.

The contributions of the $IE^2$ simulation framework can only be realized through a decent interface. The essential elements discussed by Kuljis (1996) were considered in the research as guidelines for constructing user interface for the $IE^2$ simulation framework. Though Buss et al. (2002) attempted to integrate process-driven and event-driven approaches on the user interface level; the interface for $IE^2$ model is different by explicitly defining the role of process- and event-driven models in the $IE^2$ simulation framework. In order to measure the benefits of the $IE^2$ simulation framework as standard simulation software, it has to be tested against current modeling frameworks. An experiment has been conducted to test the features of the $IE^2$ software vis-à-vis pure process-driven models. The test results showed that the average performance of the $IE^2$ simulation framework is better than the pure process-driven models.
The research has successfully integrated two different models i.e. process- and event-driven, in the simulation framework as hierarchical layers. The simulation framework is designed to handle the processing of entities and events. A formal relationship among process-driven models, event-driven models and resident entities, like resources and queues, has been established. This formalism enables the DES (Discrete Event Simulation) models in the integrated simulation framework to be more accurate and elegant by using both process- and event-driven components in a logically consistent way. In an effort to build models that accurately represent real-world structure, this ability is critical.
CONTENTS

CHAPTER 1 INTRODUCTION.................................................................1
  1.1 Background ..............................................................................1
  1.2 Simulation Framework Overview ............................................4
  1.3 Resource Layer .......................................................................5
  1.4 Process Components in IE² Framework ...................................8
  1.5 Enhanced Modeling Using Entities ..........................................9
  1.6 Example ...................................................................................9
  1.7 Interface ..................................................................................11
  1.8 Experiment .............................................................................13
  1.9 Contributions of the Research .................................................14
  1.10 Future Work ..........................................................................15
  1.11 Organization of the Report .....................................................16

CHAPTER 2 BACKGROUND AND LITERATURE REVIEW ..................20
  2.1 Introduction ............................................................................20
  2.2 Simulation ..............................................................................22
  2.3 Simulation Models ...................................................................25
  2.4 Discrete-Event System Simulation ..........................................26
    2.4.1 Components of a System ..................................................27
      2.4.1.1 Entity .......................................................................28
      2.4.1.2 Events .....................................................................28
      2.4.1.3 State Variables ..........................................................29
      2.4.1.4 Attributes ................................................................29
2.4.1.5 Event List/Event Calendar/Future Event List (FEL) ........................................ 30
2.4.1.6 Entity List ........................................................................................................ 30
2.4.1.6 Activity ............................................................................................................ 30
2.4.1.7 Delay ............................................................................................................... 31
2.4.1.8 Resources ....................................................................................................... 31
2.4.1.9 Queues ............................................................................................................ 31
2.4.1.10 Simulation Clock .......................................................................................... 31
2.4.2 Simulation World Views .................................................................................... 32
  2.4.2.1 Event Scheduling or Event-Driven Models .................................................... 32
  2.4.2.2 Process-Driven ............................................................................................... 33
  2.4.2.3 Activity Scanning .......................................................................................... 35
2.5 Simulation Software ............................................................................................... 36
  2.5.1 Arena ................................................................................................................ 37
  2.5.2 Automod .............................................................................................................. 38
  2.5.3 Extend ................................................................................................................ 39
  2.5.4 Promodel ............................................................................................................ 41
  2.5.5 Event-driven Model: Sigma ............................................................................. 41
  2.5.6 Process-driven Model: Summary ..................................................................... 42
Chapter 3 INTEGRATED ENTITY/EVENT (IE²) SIMULATION FRAMEWORK .... 45
  3.1 Simulation Framework Overview ........................................................................ 45
    3.1.1 IE² Integrator .................................................................................................. 45
    3.1.2 IE² model ....................................................................................................... 46
  3.2 Simulation Model Components: Resource & Queue ............................................ 47
# LIST OF FIGURES

Figure 1: Integrated Simulation Framework Overview ........................................... 4

Figure 2: Entity-Event Interaction in an Integrated $E^2$ Model ................................ 5

Figure 3: States of Entities in different layers of $IE^2$ model .................................. 6

Figure 4: Non-Preemptive Failures with an Event-Entity Node in $IE^2$ Model ............... 7

Figure 5: User Interface for the $IE^2$ Simulation Framework .................................... 13

Figure 6: Main Event-Scheduling Algorithm (Schruben et al., 2006) ............................ 33

Figure 7: Process-Driven Algorithm ........................................................................... 34

Figure 8: Event Graph for Simple Queuing System ................................................... 42

Figure 9: Process Model for Simple Queuing System ................................................ 43

Figure 10: Integrated Simulation Framework Overview .............................................. 45

Figure 11: Entity-Event Interaction in an $IE^2$ Model .............................................. 47

Figure 12: States of Entities in different layers of $IE^2$ model ................................... 48

Figure 13: Standard Event Graph Segment with Condition (i), Delay Time $t$, and Parameter Passing ................................................................. 52

Figure 14: $IE^2$ model segment with 2 events .......................................................... 54

Figure 15: Process Blocks, Event Node, and Connectors Supported by the Integrated Entity/Event ($IE^2$) Environment ................................................................. 58

Figure 16: Arrival Process, Multi-Server Queue, and Queue Model LEGOS (Buss et al., 2002) ........................................................................................................... 59

Figure 17: Event-Driven Model Embedded in an EventGraph Block ............................ 60

Figure 18: $IE^2$ Model Representation of Multiple Server Queue ................................. 61

Figure 19: $IE^2$ model built in the $IE^2$ Software .................................................... 61
Figure 41: Visual Studio 2005 .................................................................113
Figure 42: WinForms Interface for Process- and Event-driven layers ..........114
Figure 43: User Interface for the $IE^2$ Simulation Framework ..................117
Figure 44: Interface for the Event-Driven layer of the $IE^2$ Simulation Framework ....117
Figure 45: Non-Preemptive Failures in the Process-Driven Environment ...........124
Figure 46: Non-Preemptive Failures in the $IE^2$ Environment (a) & (b) ...............125
Figure 47: Illustration of the Blocking Scenario ........................................139
Figure 48: Overlapping Resources to Solve the Blocking Issue .....................140
Figure 49: 3-Tier $IE^2$ Simulation Framework ........................................140
LIST OF CHARTS

Chart 1: Expertise of the subjects in DES, Process- and Event-driven models ..........123
Chart 2: Subjects’ Comprehension of the Simulation Problem and Entity-Event Interaction .............................................................................................................126
Chart 3: User/Development Experience of the Subjects ........................................127
Chart 4: Effectiveness/Elegance of the Process and $IE^2$ Model with Increasing Complexity .............................................................................................................128
Chart 5: Verification of the Simulation Logic, Process Vs $IE^2$ Models .................129

LIST OF TABLES

Table 1: Technical Hurdles in the Dissertation Research ........................................112
Table 2: Tables in the SQL Server Database ............................................................116
Table 3: Simulation Software Features I (Banks, 1996) .............................................120
Table 4: Simulation Software Features II (Banks, 1996) ..........................................120

LIST OF APPENDIX

A 1: Node Class and Sub Classes INITNode, INITEntNode ....................................146
A 2: Block Class and Sub Classes: Create, Delay, EventGraph, Dispose, Seize, and Release ....................................................................................................................147
A 3: Class Associations: Block Class .......................................................................148
A 4: Class Associations: Entity and EntityList Classes .........................................149
A 5: Class Associations: Event and EventList Classes .........................................149
A 6: Class Associations: ScheduleArc Class ..........................................................149
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 7: Class Associations: <em>Simulation</em> Class</td>
<td>150</td>
</tr>
<tr>
<td>A 8: Class Associations: <em>Model</em> Class</td>
<td>151</td>
</tr>
<tr>
<td>A 9: Class Associations: <em>Node</em> Class</td>
<td>152</td>
</tr>
<tr>
<td>A 10: DESCRIPTION OF THE SIMULATION PROBLEM</td>
<td>153</td>
</tr>
<tr>
<td>A 11: POST-TEST QUESTIONNAIRE</td>
<td>155</td>
</tr>
<tr>
<td>A 12: DATABASE TABLES</td>
<td>157</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to express my gratitude towards my enthusiastic dissertation advisor Dr. Frank W. Ciarallo during this work. I would cherish every moment of my association with my advisor over the past half decade. This work has been carried out with the support from the DAGSI and BIE department. I take this opportunity to thank the DAGSI and BIE department for providing me the opportunity to finish my doctoral education. I am grateful to India and United States of America, for all the opportunities and paving the path to discover myself.

I am grateful to all my friends from Wright State University, especially Suman Niranjan, Pavan Paindla and Praveen Timmapuram, and my roommates, Harsha, Rakesh, Amit, Uday, Vijay, and Chandu for being the surrogate family during the many years I stayed there and for their continued moral support thereafter.

Finally, I am forever indebted to my parents, sisters, brothers-in-law for their understanding, endless patience and encouragement when it was most needed.
CHAPTER 1 INTRODUCTION

1.1 Background

Simulation used as a tool to analyze as well as to experiment with various strategies makes it a very important component of the decision making process. The applications of simulation can be found in various fields of study from health care systems to military applications, manufacturing to computational systems biology. The significance of simulation as a component of any important decision making process is emerging along with the growing potential of computers. The conduciveness of answering the “what if?” question, is the major reason for its widespread application in many science or technology projects. A simulation can be defined as the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system (Shannon, 1998).

A model is a representation of a system of interest, used to gain insights and investigate the system under new operating conditions. Simulation models can be classified in a number of dimensions. Kelton et al. (2004) classifies the simulation models along three dimensions i.e. static vs. dynamic, continuous vs. discrete, and deterministic vs. stochastic. A discrete-event simulation model is discrete, dynamic and stochastic in nature. Discrete-event simulation is the modeling of a system in which the state of the system changes only at a discrete set of points in time.

Modelers opt for one of several world views while developing a simulation model, the most popular being event scheduling, process interaction, and activity scanning. The
event scheduling or event-driven world view concentrates on the events and their effect on the state of the system, the process interaction world view focuses on the processes, entities and their lifecycle in the system, while activity scanning concentrates on the activities of the model and the conditions that allow them to begin.

Simulation software has evolved from more general purpose procedural languages such as FORTRAN into simulation packages and environments with a more intuitive graphical user interface and templates. Nance (1996) classifies the history of simulation software into periods of Search, Advent, Formative, Expansion, Consolidation and Regeneration, and Integrated Environments. The period of integrated environments has seen the growth of simulation environments with graphical user interfaces, animations, input-output analyzers, web-based simulations, and customized built-in templates for supply chain management, call centers, manufacturing etc. There is a need for a simulation framework that carefully amalgamates the process-driven interaction, which is graphically intuitive as in other simulation software and the event-driven approach that can model any complicated system with ease.

Almost all of the popular simulation packages use the process-driven worldview and have common characteristics or features such as a graphical user interface, animation, an input-output analyzer, and simulation optimization tools. However, the details of elements of a model, building a model, or statistical analysis tools vary between environments. Simulation software currently available in the market do not allow the modeling of detailed logic easily and the underlying model of the system i.e. activity, event, process, or some variation, is not apparent.
The common simulation packages have many useful features: The Arena (Bapat et al., 2003) standard edition has the capabilities for analyzing all types of systems and for more detailed models of discrete and continuous systems. In addition to the standard features, such as resources, queuing, process logic etc, Arena templates includes modules focused on specific aspects of manufacturing and material-handling systems. The AutoMod (Rohrer, 2003) product suite includes, AutoMod for model building and a simulation execution environment, AutoStat for statistical analysis including optimization, AutoView for 3D animation with AVI support, Model Communication Module (MCM) protocols for linking to third party software. Extend (Krahl, 2003) software is built on a message-based simulation engine that supports the block diagram approach to model building. Extend’s blocks can be easily configurable and combined to model very complex systems. ProModel (Harrell et al., 2003) was designed to model manufacturing systems ranging from small jobs, to large mass production. Apart from the modeling elements for general purpose models, ProModel provides programming capabilities for special situations. Event graphs were first introduced by Schruben (1983). Based upon the event graphs, Sigma, an event-driven simulation tool was developed. The building blocks of event scheduling or event-driven models (e.g. Sigma) are event procedures (Schruben et al., 2006). Event procedures update the state of the system, schedule other events, and/or cancel events.

In process-driven simulation models, the system can be represented by block diagrams, or system networks, through which entities flow to mimic real life system objects. In event-driven models, the system can be represented by event graphs, which focus on the abstraction of the event rather than on observable physical entities. In this
research, a simulation environment is proposed to integrate both the approaches i.e. process and event. The main purpose of this research is to mitigate complexity of the large models through process orientation, while retaining the control over the attributes, variables and the logic through event orientation. Discrete event simulation is often taught to the students at either the event level or the process level. A simulation tool that effectively preserves both the levels would be useful from the simulation education perspective.

1.2 Simulation Framework Overview

The proposed integrated simulation framework (Figure 1) has two main components: an $IE^2$ (Integrated Entity/Event) Integrator and an $IE^2$ model. One of the main goals for the design of the framework is to preserve the elegantly simple logic to process events in the simulation engine, even when the processing of entities is taking place simultaneously. The main functions of the $IE^2$ Integrator, as a core of the simulation engine, are a) Establish an effective communication between the process-driven and event-driven model components, and b) Efficient handling of entities and events for better coordination of the hierarchical layers of the model. The components and interaction in the framework are shown in Figure 1.

![Figure 1: Integrated Simulation Framework Overview](image-url)
The $IE^2$ model provides hierarchical modeling capabilities with process-driven and event-driven components as the upper and lower layers, respectively. Process-driven models created by the user are a collection of appropriate, interconnected process blocks. In order to hierarchically embed event-driven models within a process-driven model, the $IE^2$ model includes an ‘EventGraph’ block (Figure 2). The ‘EventGraph’ block behaves as a regular process block as well as a workspace for containing an event-driven model.

![Figure 2: Entity-Event Interaction in an Integrated $E^2$ Model](image)

### 1.3 Resource Layer

An important feature of the $IE^2$ model is that it explicitly models the entity flow taking place at the event level. This feature augments the capabilities of a simulation modeler by making some aspects of process logic available in the event layer and vice versa. However, the two layers are different when they interact with resident entities like resources, queues etc. Chapter 3 discusses the intermediate layer that handles queues and resources for the process- and event-driven models. This flexibility provided by the intermediate layer, reduces the level of modeling abstraction at the event graph level, and leads to a more seamless, $IE^2$ model that spans the two levels. However, an entity can change its state depending upon the layer in which it is active. In previous implementations of the event-driven paradigm (e.g. Sigma) resource and queue objects
have not been represented explicitly, and had to be abstractly defined by the model developer.

The resource layer in the $IE^2$ simulation framework manages the resident entities like resources, queues etc. The resource layer has been designed to be a separate layer from the process- and event-driven layers (Figure 3). The main aim of the resource layer is to hold the components (in this case, queues and resources) that are common to both the layers and to establish a protocol for communication between the layers.

![Diagram of states of entities in different layers of $IE^2$ model](image)

**Figure 3: States of Entities in different layers of $IE^2$ model**

The functions of the resource layer can be summarized as follows:

- Control access to the resident entities, i.e. resource, queue
- Effective management of the global and local resident entities in large simulation models

Resources can be in one of three states: *idle*, *busy*, and *breakdown*. When a resource is not processing an entity, then the resource is said to be in the *idle* state. On the other hand, if the resource is busy with an entity, then the resource is said to be in *busy* state. Failures are random events that cause the resource or servers to become unavailable. When a failure occurs then the resource state will be updated to the *breakdown* state.
Whenever a failure of a resource occurs and it is *idle*, then the state of the resource changes from *idle* to *breakdown*. If the resource is *busy* in processing an entity and a failure has occurred, then the resource can respond in two ways, *preemptive* and *non-preemptive*. In the $IE^2$ simulation framework, we demonstrate a server model with *preemptive* and *non-preemptive* failures using two different blocks at the process level. These blocks are differentiated by different underlying logic at the event graph level.

Two pre-built *EventGraph* blocks are included in the framework to model the *preemptive* and *non-preemptive* (Figure 4) failures. Because these blocks are constructed using the event graph formalism, a user has direct access to the design and implementation of these process blocks. This enables a user to interact with a process block when that is all that is required. It also enables a user to modify detailed logic easily when that is required. For example, a user may want to do this if their system requires a non-standard implementation of the pre-emption logic.

![Diagram](image.png)

**Figure 4:** Non-Preemptive Failures with an Event-Entity Node in $IE^2$ Model
The implementation of this logic in the resource layer demonstrates the power of the $IE^2$ framework to allow understanding of both simulation modeling concepts (event vs. process) and practical modeling concepts (preemptive vs. non-preemptive) in the same environment. Typically, this issue would be dealt with in a language such as Arena by navigating a dialog box with different options, without any underlying understanding of what was fundamentally changing in the model.

1.4 Process Components in $IE^2$ Framework

The $IE^2$ framework provides a set of basic process blocks for building process-driven models. In addition to that, a sub-model block is also provided for building multiple layers of hierarchical simulation models. These process components are built on the existing simulation framework namely, queues, resources, blocks, etc. The purpose of building the process-driven model components, in an integrated framework, is to augment the modeling capabilities of a user. However, an elegant and accurate model can only be realized by the modelers’ acumen in using the event- and process-driven models appropriately. In the succeeding chapters, the process components like, Create, Seize, Delay, Release, Dispose, Decision, Batch, Separate, and Sub-Model are explained in detail. In particular, their implementation is guided by a philosophy that values a consistency with logic that represents the physical world. Rather than creating a software object that only has the appropriate inputs and outputs, the event-driven logic for these components is constructed with an emphasis on mirroring real-world internal logic. Typical options available in each of these blocks are also incorporated in these process-driven components and their construct is explained in Chapter 5.
1.5 Enhanced Modeling Using Entities

An important feature of standard event graphs is parameterization of the event vertices, allowing similar model sub-graphs to be combined together as a generic graph distinguished by parameter values (Schruben, 1983). The $IE^2$ framework based on an integrated entity/event approach is further enhanced to explicitly represent entities at the event-driven level. This feature has not been present in earlier definitions of event graphs. This addition to the integrated simulation framework helps to diminish the abstraction for the model builder involved in parameter passing. When defining the passing of a parameter, the model builder must think like a computer programmer rather than as a model builder. The solution to this issue lies in explicitly passing the entities through the event-driven model. Event parameters are replaced by entity attributes. The usage of entities in the event-driven layer serves two purposes, a) reduces the abstraction by manipulating entity objects instead of working with parameters as in a programming language, and b) gives the intuitive feel of process-driven models to modelers at the event level, which enhances the appeal of the event-driven models. The succeeding chapters present an event graph construct with limited entity logic, compares specific examples in traditional event graphs vis-à-vis the same problem in an $IE^2$ model, and explores the level of modeling detail obtained while reducing the level of abstraction.

1.6 Example

In order to illustrate both the process- and event-driven model components in an $IE^2$ model, a supply chain management system has been studied as an example. Evaluating the Impact of Retailer Gaming and Supplier Capacity Allocation on Supply Chain Costs (Vutukuru, 2006), focuses on a supplier-retailer supply chain, consisting of a single
supplier and three retailers. The model considers how partial information sharing has an impact on the supply chain costs with different allocation mechanisms on the supplier side vis-à-vis gaming behaviors on the retailer side. The model was originally implemented in Arena. Implementing allocation mechanisms in the Arena model is a complex task, and hence a more detailed language was used to develop the allocation logic and link it to Arena. VBA (Visual Basic for Applications) was employed to handle the allocation mechanism computations as VBA has an excellent interface with Arena and is inbuilt in Arena. Allocation mechanisms were coded in VBA and linked to the Arena model. VBA-based blocks are an integral part of Arena.

A comprehensive $IE^2$ model for retailer demand and cost sub-models is presented in chapter 4, to allow direct comparisons between a pure process model with VBA programming and an integrated entity/event model. The example demonstrates how the logic of allocation and entity flow can be more elegantly represented with the $IE^2$ model. It also demonstrates the well-structured interface defined for entities that interact with the process layer, transitioning to the event layer, and then returning to the process layer.

Comparing the standard Arena based process-driven model with an $IE^2$ model, the $IE^2$ model embodies the function of event parameterization through the entity attributes. Entity attributes are a natural construct for modelers familiar with process-based logic. The advantage of using the entity attributes in the $IE^2$ model is that similar model sub-graphs can be combined together as a generic sub-graph distinguished by the attribute values of entities flowing through them. At the event level, entities are handled as objects in a way that is analogous to their treatment in the process models. The attributes of an entity are defined by the modeler, enabling the flexibility and explicit handling of
entities at the event level. Instead of passing information as event parameters to other nodes as in a programming language, the $IE^2$ model defines them explicitly as attributes of entities that are associated with events as they are scheduled. This entity passing through the events in the event graph, gives the intuitive feel of the process-driven model to the modelers. This modeling of entity flow through the event graph enhances the appeal of event graphs to modelers with a process perspective, while retaining the power and flexibility of the event logic. At the process level, the modelers’ ability to model complex logic is enhanced without resorting to programming languages in a simulation model. One of the major objectives of the $IE^2$ model is to diminish the gap between real world processes, and their representation in the simulation environment, while not limiting itself to the graphical representation as in most commercially available process-driven simulation tools.

1.7 Interface

Three critical features of any simulation environment are model logic representation, model elements, and model verification & validation. The model logic representation is the first step towards the making the simulation engine accessible to non-specialists. Secondly, the model elements must provide a high level simulation layer for the non-expert users. Finally, both explicit and implicit support for model verification & validation in the simulation environment is important in debugging and ensuring that a model serves the purpose for which it was designed. Chapter 6 describes the prototype user interface for the $IE^2$ framework, focusing on the first two of these issues, while developing a software interface for the conceptual simulation environment for the $IE^2$. It indirectly supports the first issue through its use of the $IE^2$ multiple-layer approach, which
supports models that have a closer connection to the real system, and thus should be easier to verify and validate.

Model logic representation serves the objective by providing cognizable simulation objects in the graphical user interface and supporting the novice users through a natural and intuitive model building process. Apart from the process blocks and event nodes that are well-established in the simulation world, the fundamental model logic elements that are introduced in the \(IE^2\) framework are the EventGraph block and event-entity arc. The EventGraph block provides the event graph workspace, while event-entity arc allows explicit modeling of the entity in the event layer.

The important components of the interface for the process layer of the \(IE^2\) framework are the process block menu panel, work space, menu strip, animation, and variables display (Figure 5). The process block menu panel consists of commonly used process block i.e. Create, Delay, Seize, Release, Dispose, Assign and EventGraph. These blocks can be dragged and dropped onto the work space, just like in any other commercially available DES software. The animation part of the interface displays a very basic animation of the machine or server and its interaction with the entities. The variables display portion of the interface displays the values of the important variables as the simulation is running. The interface for the event-driven models is displayed when the modeler clicks the EventGraph for data entry. The important portions of the \(IE^2\) event-driven layer are the event node panel, menu strip, Enter, and Exit nodes. The event node panel consists of three types of nodes namely, regular node, initialization node, and initialization node with a entity tagged to it.
1.8 Experiment

An experiment was conducted to get a first assessment of the accomplishment of the objectives set forth at the onset of the research. In order to test the $IE^2$ simulation environment, a formal hypothesis was proposed that tests the effectiveness of the $IE^2$ simulation environment. A total of seven subjects volunteered for the experiment. Five of the seven subjects had experience in both the process- and event-driven models, one subject only in event-driven models and one subject only in process-driven models.

The experiment did not attempt an elaborate usability test of the $IE^2$ framework interface. The subjects’ responses on the user experience, on average, were equal for both the process and $IE^2$ models (Average = 4.14). However, the users’ comments at the end of the questionnaire were quite useful for further improvement of the framework and software. Some of the remarks made by the subjects reflect the objectives of the current
research and its relevance in the context of their simulation models. The hypothesis of this experiment was that the \( IE^2 \) model provides more effectiveness over the conventional process-driven and event-driven models. In the current context of the experiment, we can draw a limited conclusion that \( IE^2 \) models provide more effectiveness over the process- and event-driven models.

1.9 Contributions of the Research

This dissertation research makes the following contributions:

- Definition of the \( IE^2 \) framework based on three layers: Process-Driven, Event-Driven, and the Resource layer, with a structured set of interface between the layers.
- Extension of the event graph formalism to explicitly model entities at the event level.
- Unified handling of events and entities in the \( IE^2 \) Integrator of the simulation engine of the \( IE^2 \) framework.
- A set of basic process blocks for building process-driven models in the \( IE^2 \) framework. In addition to that, a sub-model block to build multiple layers or hierarchical simulation models.
- An example that demonstrate the benefits of structured entity-event interaction in the \( IE^2 \) model.
- User interface for the \( IE^2 \) model to explicitly build process-driven models without translation. The interface provides hierarchical modeling capabilities with
process-driven and event-driven components as the upper and lower layers, respectively.

- Limited experimental evidence evaluating the effectiveness of the $IE^2$ models over the conventional process- and event-driven models.
- A proof of concept of the $IE^2$ simulation framework that demonstrates and establishes the viability of integrating two different approaches to Discrete Event Simulation (DES).

1.10 Future Work

$IE^2$ simulation framework can be used to solve some of the popular modeling issues. In this research, $IE^2$ was used to model the failures (preemptive and non preemptive) template. Blocking would be another interesting modeling issue to be examined. Blocking in a tightly coupled system is a scenario in which the entities have to be allocated resources downstream before they can move on. Tightly coupled systems are the systems with a limited space for parts buffering between workstations (Kelton et al. 2004).

Another interesting direction for the current research would be to add an object-oriented modeling approach as the upper level of the hierarchical $IE^2$ simulation framework. Object-oriented approach has important properties like inheritance, polymorphism, and encapsulation. The 3-tier $IE^2$ simulation framework will have event-driven, process-driven, and object-oriented layers. The $IE^2$ framework with the object-oriented feature would give the simulation modelers much broader scope to model the DES models.
1.11 Organization of the Report

The remainder of this dissertation document is organized as follows:

Chapter 2 *Background and Literature Review* presents an overview of simulation as a tool in the decision making process and different fields of applications of simulation in various fields. To establish the position of discrete event simulation in the world of simulation, the classification of simulation as a study and important steps in a typical simulation study are presented. Important components and different world views or approaches for a discrete event model are then described. *Simulation Software* consists of an overview of popular discrete event simulation software available in market, focusing on the underlying framework as well as relevant features. Almost all of the simulation packages use the process-driven worldview and have common characteristics or features such as a graphical user interface, animation, an input-output analyzer, and simulation optimization tools. However, the details of elements of a model, building a model, or statistical analysis tools vary between environments.

Chapter 3 *Integrated Entity/Event (IE²) Simulation Framework* presents the important components present in the IE² framework. This chapter describes the implementation details of the framework as well as the details of the construction of the different components. A simulation environment is proposed to integrate both the approaches i.e. process and event. The main purpose of this research is to mitigate the complexity of the large models through process orientation, while retaining the control over the attributes, variables, and the logic through event orientation. *Enhanced Modeling Using Entities in an IE² Framework* describes the parameterization of sub models in the IE² framework. *Interface of the IE² framework* section focuses on the development of the software user
interface for the Integrated Entity/Event ($IE^2$) models. This chapter discusses the model
logic representations of the simulation components, and their relevance on how users
interact with the $IE^2$ framework.

Chapter 4 Example Models in $IE^2$ Framework describes the parameterization of sub
models in the $IE^2$ framework. This chapter explains how entity objects are used to extend
the definition of event graphs. It presents the event graph construct with limited entity
logic, compares specific examples in traditional event graphs vis-à-vis the same problem
in an $IE^2$ model, and explores the level of modeling detail obtained while reducing the
level of abstraction. An Arena based supply chain management problem has been studied
to demonstrate the capabilities of the integrated approach. This problem focuses on a
supplier-retailer supply chain, consisting of a single supplier and three retailers. The
model considers how partial information sharing has an impact on the supply chain costs,
with different allocation mechanisms on the supplier side vis-à-vis gaming behaviors on
the retailer side. The modeling of the problem in the $IE^2$ simulation framework highlights
the use of entities at the event level. This allows similar model sub-graphs of the process-
driven model to be combined together as a generic sub-graph, distinguished by the
attributes that entities carry into the event graph block.

Chapter 5 Proof of Concept: Software Architecture describes the implementation
details of the framework as well as the details of the construction of the different
components such as process components and the resource layer in the integrated
framework. This chapter discusses the intermediate layer that handles queues and
resources for the process- and event-driven models. This flexibility provided by the
intermediate layer, reduces the level of modeling abstraction at the event graph level, and
leads to a more seamless, $IE^2$ model that spans the two levels. However, the entities change their states depending upon the layer in which it resides, namely, transient, passive, and tagged. *Process Components* section gives a detailed overview of the implementation of the process components including blocks, the fundamental component of the process modeling layer. The $IE^2$ framework provides a set of basic process blocks for building process-driven models. In addition to that, a sub-model block is also provided for building multiple layers or hierarchical simulation models. These process components were built on the existing simulation framework namely, queues, resources, blocks, etc. The purpose of building the process-driven model components, in an integrated framework, is to augment the modeling capabilities of a modeler. In this chapter, the process components like, *Create, Seize, Delay, Release, Dispose, Decision, Batch, Separate,* and *Sub-Model* are explained in detail.

Chapter 6 *Proof of Concept: User Interface and Testing* discusses the software techniques and technologies used in this research. In order to demonstrate the concept of the $IE^2$ framework, and refine its ideas, a software implementation of the simulation engine and model components was necessary. Therefore, skills in software and user interface development also supported this research. This chapter describes some of the technical and software-oriented details that must be considered beyond the conceptual structure of the framework. A limited experiment with subjects using the $IE^2$ simulation environment was also discussed in this chapter. A formal hypothesis was proposed to test the effectiveness of the $IE^2$ simulation environment. This chapter discusses the results of the experiment and its limited conclusion that $IE^2$ models provide more effectiveness over the process- and event-driven models.
Chapter 7 *Conclusions, Contributions and Future Research* summarizes the dissertation research. The contributions made by the research are also summarized in this chapter. Finally, the scope of the $IE^2$ simulation framework in solving the modeling issues and the direction of the future research are discussed.
CHAPTER 2  BACKGROUND AND LITERATURE REVIEW

2.1  Introduction

In event-driven simulation models, a modeler has the ability to know the state of everything at any time and flexibility with regard to attributes, variables, and logic flow (Kelton et al., 2004). Alternatively, the ease of modeling a real system with a commonly understood flow chart approach using process blocks, has allowed process-driven models to be more accepted among the non-technically oriented simulation user community. The major reason for the popularity of the process-driven models is the direct mapping of the simulation logic to the model animation. The commercial effort to make simulation modeling methodology more accessible to non-programmers has created a separation between technically oriented and non-technically oriented simulation users (Healy et al., 1997). The separation discussed here is the inaccessibility of the state variables of the simulation model to the simulation user.

The overwhelming majority of discrete event simulation tools in use, as reported in a software survey in 2005 (Swain, 2005), are based on process driven models. The process-driven modeling framework’s main strength is drawn from the process analysis approach (Seppanen, 2005), which when applied in an orderly manner helps in analyzing the multiplicity of factors affecting a process. Simulation environments based on this approach are very effective in translating the analysts’ understanding of system structure into a model. Application-specific tools like LayOPT (Grajo, 1996) and general-purpose simulation packages like Extend (Krahl et al., 1997) use the process-driven approach.

Buss (2001) supports event-oriented simulation models over other world views such as process/resource. Even though the event-orientation makes simple models slightly
more complex, it provides more flexibility and modeling power than a pure process-oriented world view. Every model that can be represented in the process world view can also be represented in an event-oriented model, but the reverse is not true. In order to combine the advantages of both world views, Buss attempts to give the appearance of process-oriented model to event-oriented models. It is accomplished by aggregating some sections of an event graph model into process-oriented components. The resulting simulation environment (Simkit) uses the “listener pattern” from software engineering to implement the interoperability of simulation components.

Schruben et al. (2003) compared the process-driven approach and the event-driven approach, with respect to job-driven and resource-driven models. Using a semiconductor wafer manufacturing example model, the authors found that execution time decreased when the modeling methodology was event-driven instead of process-driven. In the entity-driven approach every job or entity must be represented explicitly through every step of the processing flow paths, this leads to high congestion in the simulation of the system and lower execution speed in large-scale models. However, the process-driven models are insensitive to the size of the model, when developed using a resource-driven approach. The results of the paper discuss the comparison of the run-time ratios in Arena and Sigma. The run-time ratios (Arena /Sigma) vary from 24.6 to 100 with traffic intensity varying from 0.10 to 1.43. The paper concludes by suggesting that more attention be paid to the advantages of different simulation modeling approaches.

Schruben et al. (2003) offers a small set of examples of how access to event-driven modeling offers more control for complex model building situations, and higher execution speeds with respect to both job and resource-driven approaches. Because of
these advantages, sophisticated, detail-oriented simulation modelers often favor access to the event-driven modeling approach. On the other hand, non-technically-oriented simulation users will typically opt for ease in model building, using templates of application-specific, process-driven components. Many developments have taken place to make environments based on the process-driven models more appealing. There have not been simulation tools that readily give users the advantages of both approaches, i.e. process-driven and event-driven. A simulation tool that elegantly and efficiently integrates the two approaches in its simulation engine is a requirement for providing the process- and event- modeling functionality to all classes of users. This research describes the design and implementation of a simulation engine that integrates the two approaches.

2.2 Simulation

Simulation used as a tool to analyze as well as to experiment with various strategies makes it a very important component of the decision making process. The applications of simulation can be found in various fields of study from health care systems to military applications, manufacturing to computational systems biology. In this chapter, simulation as an important tool is discussed in a general way. The discussion in this chapter focuses on discrete-event simulation, while discussing the various kinds of simulation methodologies. At the end of the chapter, the important components of manufacturing systems in the context of a simulation framework are discussed.

The significance of simulation as a component of any important decision making process is emerging along with the growing potential of the computers. The conduciveness of answering the “what if?” question, is the major reason for its widespread application in many science or technology projects. A simulation can be
defined as the process of designing a model of a real system, and conducting experiments with this model for the purpose of understanding the behavior of the system, and/or evaluating various strategies for the operation of the system (Shannon, 1998). As mentioned in the definition, the real systems are modeled so as to get insights about the system, and/or to improve the performance of the system by evaluating the measures of effectiveness.

Irrespective of the area of study in which it is used, simulation is heavily based upon mathematics, statistics and computer science. Often a real-world system is represented by a mathematical formulation that can be solved by differential calculus, probability theory, or some other mathematical techniques. The outputs that are derived from such models are used as measures of effectiveness to improve the system. In certain instances, where the real-world system is very complex to represent mathematically or if there exists no “closed-form solution”, numerical or computer-based simulation can be used to imitate the behavior of the system. The numerical data generated by the simulation of those complex systems can be statistically analyzed to make inferences. A major advantage of simulation over an analytical or mathematical method is its intuitive appeal to comprehend as well as to justify the model. The advantages and disadvantages of simulation have been discussed extensively in the literature such as by Pegden et al. (1995), and Shannon (1998).

The diverse areas of simulation application can be fathomed by examining the research work presented every year at the annual Winter Simulation Conference (WSC) (http://www.wintersim.org/). The following are the area of applications listed at the recent WSC:
Health Care: Utilizing Health Resources, Emergency Department Operations, Health Policy Analysis


Virtual reality and simulation


Risk Analysis: Pricing American Options, Risk Analysis, Efficient Simulation for Risk Management, Stochastic Programming in Risk Analysis

Military Applications: Visualization for Military Simulation, M&S Support to Future Combat Systems, Military Aerospace Application, Simulation Architectures

Logistics, Transportation, and Distribution: Rail Simulation, Transportation Simulation, Supply Chain, Manufacturing and Transport Systems, Transport and Data Collection, Air Transportation and Maritime Simulation

Computational Systems Biology: Exploiting Data Exchange and Data Base Technology for Computational Biology, Parameter Estimation and Optimization, Modularity and Composition, Complexity Reduction, Simulation Tools for Systems Biology
2.3 Simulation Models

A model is a representation of a system of interest, used to gain insights and investigate the system under new operating conditions. Simulation models can be classified in a number of dimensions. Kelton et al. (2004) classified the simulation models along three dimensions i.e. static vs. dynamic, continuous vs. discrete, and deterministic vs. stochastic. The static vs. dynamic category is based on the role of time in the model. Monte Carlo simulation can be cited as an example of static simulation models, for there is no significant role of time during the course of a simulation run. At same time, they can be described as stochastic models because of the randomness involved. While on the
other hand, in dynamic models the state of the system evolves as the time progresses during the simulation run.

In continuous-time models, the state of the system changes continuously over time, and is usually represented by differential equations. In a discrete event model, the changes in the state of the system occur at separate instances of time. As mentioned by Banks et al. (2005), even though discrete event and continuous-time models are defined in an analogous manner, a discrete event simulation model is not always used to model a discrete system nor is a continuous model always used for a continuous system. The characteristic of the system, and/or the objectives of the simulation study play a major role for the modelers to prefer the simulation model as discrete event or continuous-time.

Deterministic or stochastic models are defined, depending upon the absence or presence of randomness, respectively. Deterministic models have predetermined inputs and result in corresponding unique outputs. While, stochastic models have one or more inputs represented as random variables, and their resulting outputs are random. Depending upon the representation chosen for a simulation model, a modeler has to determine whether each input variable should be random or deterministic.

2.4 Discrete-Event System Simulation
A discrete-event simulation model is discrete, dynamic and stochastic in nature. As defined earlier, discrete-event simulation is the modeling of a system in which the state of the system changes only at a discrete set of points in time. Unlike analytical models, discrete-event simulation models are “run” rather than solved (Banks et al. 2005). The artificial history generated by the models is similar to that of the real life system and is analyzed to estimate measures of effectiveness. Past literature has extensively discussed
the set of steps to be followed to make a simulation study valid and thorough (Shannon, 1975; Gordon, 1978; Law and Kelton, 2000). The following steps are the highlights of a typical simulation study:

- Problem Definition and Formulation
- Project Goal and Planning
- Conceptual Model
- Input Data Collection
- Model Translation
- Verification and Validation
- Experimental Design
- Sensitivity Analysis
- Implementation & Documentation

2.4.1 Components of a System

Decision making is becoming increasingly difficult at the same pace as the increasing complexity of real world systems. In order to use discrete-event simulation as a handy tool in those scenarios, it is important to understand the concept of a system and its components. Banks et al. (2005) defines a system as a group of objects that are joined together in some regular interaction or interdependence towards the accomplishment of some purpose. The complexity arises from the interrelations and interactions among the various elements of the system (Shannon, 1975). It can be understood that in such a complex system, changing one aspect can influence or change other parts of the system. To model and analyze the system requires clearly defined terms for the components of the system. The following eleven components make up a discrete event simulation:
2.4.1.1 Entity

Kelton et al. (2004) defines entities as the objects of interest that move around, change status, affect and are affected by other entities and the state of the system, and affect the output performance measures. Further, Kelton et al. elaborates on entities by describing them as the dynamic objects of the simulation that are created, move around the model, and are disposed of as they leave the model. As in real-world systems, entities can have many independent copies or realizations of itself at a time. However, as described by Kelton et al., a simulation model can have fake or logic entities which do not correspond to any tangible or real world object. The purpose of their existence in the simulation model is to take care of certain modeling operations.

As mentioned in the example by Kelton et al. for logic entities, machine failures can be modeled as either preemptive (breakdown entity) or non-preemptive (maintenance entity) failure entities. Breakdown or maintenance entities are generated in the simulation model at the rate (as specified by the modeler) of machine failures rate. However, the breakdown entity models the breakdown of a machine or resource by changing the state of the machine to down, even if the machine or resource is in middle of processing any entity. A maintenance entity, on the other hand, changes the state of machine or resource to down or maintenance only if it is available (i.e. not processing any entity). In the $IE^2$ framework, entities are represented explicitly in an expanded definition of event graph.

2.4.1.2 Events

An instantaneous occurrence that changes the state of a system is called an event. Discrete event simulation, as per definition, models the evolution of a system through time by changing its state at discrete points of time. Event-driven simulation, discussed
later, uses events to build the structure of the model. Entities and events are the basic simulation elements around which the process- and event-driven approaches are created. The proposal research focuses on handling of entities and events while integrating the process- and event-driven approaches.

2.4.1.3 State Variables

State variables are the set of variables that describe the characteristics of a system at any given time, irrespective of the types of entities existing in the system (Kelton et al., 2004). Similar to the attributes of an entity, state variables are the characteristics of the system but they are not attached or tagged to any entity. Most of them can be accessed or changed by any entity in the system. State variables represent the state of the system that changes during the simulation run.

2.4.1.4 Attributes

Attributes are the properties or characteristics of an entity that distinguishes it from other entities. The attribute values of a particular entity are attached or tagged to that entity itself. The value of an attribute will usually vary across the entities, even if all the entities have the same set of attributes. Either the simulation model, by default, or a modeler defines the attributes of an entity. The values or states of an attribute are changed along with the state variables at discrete points of time. In the $IE^2$ framework, attributes of the entities are used to represent sub models as a generic model similar to the parameterization of event graphs.
2.4.1.5 Event List/Event Calendar/Future Event List (FEL)

Event list or Event Calendar or Future Event List (FEL), is the list of events to occur in the simulated future, ordered by time. The general functionality of the event list is to sort the events according to their order of time of occurrence. When the simulation logic calls for the current event to occur during the simulation run, the event list provides the event with smallest or earliest time possible. When an appropriate event is executed, it schedules new events and places them in the event list. After an event occurs, it no longer is found in the event list. The role of event list with respect to other components in a general simulation framework is further discussed in the section related to event-driven simulation.

2.4.1.6 Entity List

Similar to the event list, an entity list is the list of entities that are required to be processed by blocks in a model, indexed by time. However in $IE^2$ integrator, entity lists are used in the implementation of process-driven simulation models. During the simulation run, the entity with the earliest time is removed from the entity list and sent to the corresponding block in the model it is mapped to. The detailed functionality of the entity list is discussed in the succeeding sections.

2.4.1.6 Activity

Activity time is the duration of time of during which an activity takes place (Banks et al. 2005). For example, activity time could represent a service time, an inter arrival time or any other processing time. The duration of an activity can be either deterministic or random variable or a function depending upon certain variables.
2.4.1.7 Delay

Delay (Banks et al., 2005) is defined as duration of time of unspecified indefinite length, which is not known until it ends.

2.4.1.8 Resources

A resource is a source of supply or support. In manufacturing systems, a resource represents personnel, equipment, machines or a storage space. In a simulation model, an entity is allocated or can seize a unit of resource when it is available and can release the resource when finished (Kelton et al., 2004). Kelton et al. discusses how resources are given to an entity rather than the entity being assigned to the resource, since an entity can be simultaneously using multiple resources. In the $IE^2$ model, resources are placed in a resource layer, from which the blocks (process layer) or nodes (event layer) that utilize the resource are given access.

2.4.1.9 Queues

Queues are used in a simulation model to store entities that are waiting for a resource to be seized. Queues usually have an interface that allows the addition and removal of entities. In the $IE^2$ framework, queues are implemented as objects with this interface. Queues also require the ability to process entities in a FIFO, LIFO etc. order.

2.4.1.10 Simulation Clock

The simulation clock is a variable that keeps track of the simulated time during a simulation run. In a discrete event simulation system, the simulation clock records the time of the events rather than the conventional real time. It keeps track of the time from event to event and ignores the time in between, where nothing interesting happens from
the simulation perspective. Usually, the simulation clock works closely with the main simulation program and event list. When the main simulation program removes the next event or event with smallest time from the event list, the simulation clock updates the clock time and moves the time of the system to that time. In the $IE^2$ framework, the simulation clock follows the same logic as in a typical simulation programs, except that it keeps track of both the event and entity list through its main simulation program.

2.4.2 Simulation World Views

Modelers opt for a world view while developing a simulation model, the most popular being *event scheduling, process interaction, and activity scanning*. Event scheduling or event-driven concentrates on the events and their effect on the state of the system, process interaction world view focuses on the processes, entities and their lifecycle in the system, while activity scanning concentrates on the activities of the model and the conditions that allow them to begin.

2.4.2.1 Event Scheduling or Event-Driven Models

The building blocks of event scheduling or event-driven models (e.g. *Sigma*) are event procedures (Schruben et al., 2006). Event procedures update the state of the system, schedule other events, and/or cancel events. The event procedures that describe a discrete event system are executed by a main simulation program using a list of scheduled events. This list of scheduled events is called the Future Events List (FEL) and it contains all the events that are scheduled to occur in the future. The simulation program selects the event with smallest time from the FEL and advances the simulated time to the time of that event (Figure 6). When the event (i.e. procedure) is executed, typically the system state is changed and/or other events are scheduled, or canceled. The simulation of the event
scheduling or event-driven model operates by successively renaming an event from the future event list (FEL) and executing the corresponding procedures until the condition for simulation termination is encountered. In the $IE^2$ framework, the event scheduling logic has been implemented for the event-driven models; with events, event lists (FEL), event procedures, event scheduling as well as event canceling.

Figure 6: Main Event-Scheduling Algorithm (Schruben et al., 2006)

2.4.2.2 Process-Driven

In a process-driven model, the modelers concentrate on processes. A process is a life cycle of an entity as it flows through the system, demanding resources, queuing to wait for resources, and interacting with other entities etc (Banks et al., 2005). The interaction between entities is the most important part of the process-driven models. The simplest interaction as described by Banks et al., being the entity forced to wait in a queue because the resource it needs is busy with another entity. Process-driven models are very popular among simulation modelers in practice because of their intuitive appeal, and their ability to let modelers describe or represent the model in processes that have direct mapping to the real world. Generally, the underlying implementation of process-driven models, in most of the simulation packages, is based on event scheduling logic. The processes defined by the user are implemented using a time-sequenced list of events, delays, and
interactions with system components or state variables, (such as resources etc). The entities are stored in the lists whenever they face delays, causing execution of one or more processes to be suspended.

In the $IE^2$ framework, instead of implementing the process-driven model using event scheduling logic, an entity is handled according to the process it is associated with. The basic implementation of the process-driven algorithm is shown in Figure 7. This figure is drawn similarly to the event scheduling algorithm shown in Figure 6 to allow direct comparison. The system is initialized by giving initial values to the state variables and entity list. The top entity or entity with smallest time is removed from the entity list and the simulation clock is advanced to the time associated with that entity. The entity, which is removed from the list, is processed by implementing the block corresponding to the next step in the process. The blocks can add entities back into the entity list with a new time. The process interaction algorithm iterates, by removing the entities from the list and sending them to blocks to process, until the termination condition is encountered.

![Figure 7: Process-Driven Algorithm](image_url)
2.4.2.3 Activity Scanning

Activity scanning concentrates on the activities in a model, and the conditions that allow the activities to begin. At each time step, the conditions of each activity are checked and the activities that return a true value are executed. A disadvantage of activity scanning is that it requires repeated scanning of the conditions on all of the activities. This tends to slow the speed of the simulation. In order to overcome the disadvantages of the pure activity scanning approach, Banks et al. (2005) discusses the three-phase approach that combines some of the features of the event scheduling with activity scanning. This would avoid unnecessary scanning of activities as well as allow variable time advance.

In a three-phase approach, activities are broadly classified into two categories, i.e. B and C.
- B activities are the activities that are bound to occur, primary events and unconditional activities
- C activities are the activities or events that are bound to the Boolean value i.e. true or false, of a corresponding condition

Activity-driven models maintain a FEL, similar to event-scheduling, which contain only B-type activities. Scheduling of these B-type events or activities allows variable time advance. At each time advance or time step of these B-type event or activity, C-type activities are scanned to decide whether a C-type activity or event occurs. Simulation using the activity scanning approach proceeds by repeated execution of the following three phases repeatedly until the simulation is terminated.

**Phase A** – Remove the imminent events from the FEL and update the simulation clock using the time of the events or activity
**Phase B** – Execute all the B-type events or activities from the FEL

**Phase C** – Scan the conditions for all C-type activities or events and execute the activities that return true value for their corresponding condition

Activity scanning requires that the model builder have a detailed understanding of how all the processes in a system interact. It can be difficult to implement models of systems with highly decentralized processes using the activity scanning approach. This is because the real system is very much decentralized, whereas the model may require very centralized activity scanning logic. Because of this, the activity scanning approach can be very difficult for model builders to master.

### 2.5 Simulation Software

Simulation software has evolved from more general purpose procedural languages such as FORTRAN into focused simulation packages and environments with more intuitive graphical user interface and templates. The history of simulation software is well documented (Nance, 1996) by experts and the updates of the state-of-the-art in the simulation world can be found in numerous sources, one of the most popular being the annual WSC. Nance classifies the history of simulation software into periods of Search, Advent, Formative, Expansion, Consolidation and Regeneration, and Integrated Environments. The period of integrated environments has sent the growth of simulation environments with graphical user interfaces, animations, input-output analyzers, web-based simulations, and customized built-in templates for supply chain management, call centers, manufacturing etc. The IE² Simulation framework definitely belongs to the period of integrated environments, with careful amalgamation of the process-driven
interaction, which is graphically intuitive as in other simulation software and the event-driven approach that can model any complicated system with ease.

In order to get an overview or update of the simulation industry, this chapter discusses popular simulation packages that are available in the market. Almost all of the simulation packages use the process-driven worldview and have common characteristics or features such as a graphical user interface, animation, an input-output analyzer, and simulation optimization tools. However, the details of elements of a model, building a model, or statistical analysis tools vary between environments. When compared with the $IE^2$ simulation framework, software available in the market the underlying model of the system i.e. activity, event, process, or some variation, is not readily accessible to most classed of model builders.

2.5.1 **Arena**

The *Arena* (Bapat et al., 2003) suite of products, offered by Systems Modeling Corporation, includes *Arena* basic, standard, and professional editions. *Arena* boasts of the new enhancements in its latest version that include:

- OPC (Open Process Connectivity) technology to test the control system on a model rather than testing on the real system
- Templates of tank farms and batch processing operations
- *Arena* 3D player for 3D animations

The *Arena* basic edition targets at modeling business processes and represents process dynamics in a hierarchical flowchart. With built-in activity-based costing and robust system performance data, *Arena* allows users to conduct sensitivity analysis and choose the best possible configuration. *Arena* basic edition is closely integrated with Visio and
through Arena’s standard ActiveX and DAO (Data Access Object) interfaces and VBA (Visual Basic for Applications), corporate data can be incorporated directly into the simulation model.

The Arena standard edition has the capabilities for analyzing all types of system and for more detailed models of discrete and continuous systems. In addition to the standard features, such as resources, queuing, process logic etc, Arena templates includes modules focused on specific aspects of manufacturing and material-handling systems. All the supporting services for a successful simulation have been included in the standard editions. The supporting services include, input analyzer for selecting appropriate input distribution, built-in confidence intervals, Process and Output Analyzer to automate the comparison of different design alternatives.

Arena professional edition offers the customers with customized modeling tools called modules. Each of these editions offers library of modules called Application Solution Template (AST). These AST’s dictates the product’s target application i.e. types of systems it can effectively model, process representation etc. The professional edition of Arena offers an important feature of designing modules and adding them to the Arena templates.

2.5.2 Automod

The AutoMod (Rohrer, 2003) product suite include, AutoMod for model build and simulation execution environment, AutoStat for statistical analysis including optimization, AutoView for 3D animation with AVI support, Model Communication Module (MCM) protocols for linking to third party software. The main focus being manufacturing and material handling, AutoMod offers templates for material movement
called movement systems. The movement systems aid in defining movement of materials through path mover, conveyors, automated storage and retrieval systems, kinematic robots, bridge cranes, power and free chain conveyors, and tanks and pipes.

An AutoMod model consists of one or more systems organized in one or more sub models. A system can either be a process system, in which flow and control logic are defined, or a material movement system. Any number of movement systems can be defined in an AutoMod model, which connects the process systems. Loads (entities) move between processes or locations, and compete for resources. Loads are active entities in AutoMod and can be created using deterministic and probabilistic generation. Processes are the places where actions are performed or decisions are made. Resources in AutoMod are used to represent machines, operators, fixtures, containers, and other finite capacity objects. Apart from the default states, the user can define states that represent blocked, starved etc. In addition to that, state monitors can be defined to keep track of states of entities, vehicles, conveyors, or particular areas of a facility. Queues in AutoMod are both graphical and statistical element and, they can have user-defined capacity. Order List, on the other hand, is not a physical entity like queue but a logical element that provides a way to sort loads that have been delayed for some reason.

2.5.3 Extend

Originally released in 1988, the Extend (Krahl, 2003) family of products is offered by Imagine That, Inc. The products include:

- Extend CP, for continuous modeling
- Extend OR, discrete event modeling added to the continuous modeling of Extend CP
- *Extend Industry*, Adds an integrated database and high speed systems modeling to
  
  *Extend OR*
  
- *Extend Suite*, Adds Proof Animation and Stat::Fit for distribution fitting for
  
  *Extend Industry* package

*Extend* software is build on a message-based simulation engine that supports the block diagram approach to model building. *Extend’s* blocks can be easily configurable and combined to model very complex system. All the products in the *Extend* have common features like, drag and drop interface, inter process communication tools for communicating with other applications, hierarchical modeling capabilities, evolutionary optimization, and development environment for building custom components.

The *Extend* modeling environment consists of libraries of blocks, each of which represent a set of blocks characteristic of discrete event, plotter, electronics, or business process engineering. The active entities are called items, with attributes and priorities associated with them. The items and values are a kind of logical flow with *Extend* blocks. The second type of logical flow is called values, which will change over time. Items and values are connected from one block to another using lines, single lines for values and double lines for items.

The block development environment in *Extend* is its most powerful features. While the pre-built blocks are enough to build a decent model, the block environment provides the users the ability to expand the modeling capabilities to perform complicated tasks. *Extends* open architecture allows user to open the programming code of the pre-built block and edit it. A high level language, ModL, is provide with high-level functions can be used to define the behavior of each block. In addition to that, external XCMDs and
DLLs can be called from within ModL programming language. This feature requires a very sophisticated user, with both an understanding of modeling in general, as well as the details of the ModL language.

2.5.4 Promodel

Promodel (Harrell et al., 2003) was designed to model manufacturing systems ranging from small jobs, to large mass production. Other simulation products available from Promodel are MedModel, ServiceModel, and Promodel PI (Process Improvement). Promodel offers manufacturing oriented modeling elements and rule-based decision logic. Apart from the modeling elements for general purpose models, Promodel provides programming capabilities for special situations. The built-in language features include if-then-else logic, Boolean expressions, variables, and even access to spreadsheets.

The modeling elements of the Promodel provide building blocks for representing the physical and logical component of the system to be represented. Parts or entities refer to the items being processed in the system; these may include raw materials, assemblies, loads, WIP etc. Entities may be assigned attributes that can be tested in making decisions or used for gathering specialized statistics. Path networks are part of the modeling elements that represent the possible paths that entities and resources may travel when moving through the system. Resources defined in Promodel can be a person, tool, vehicle or other object that may be used to, transport materials between routing locations, perform an operation on material, or perform maintenance on resource that is down.

2.5.5 Event-driven Model: Sigma

Event graph models (e.g. Sigma) were introduced by Schruben (1983) and are based on the detail-level, event-driven dynamics of simulation. In event-driven simulation, events
are the fundamental actions that change the values of the variables that describe the system state and drive the scheduling of future events. System dynamics are modeled by events that change the state of the system and the logical and temporal relationships among these events (Savage et al., 2005). In an event graph model, the effect of events on the variables is described in vertices (nodes) and the relationship between events is represented by directed edges (arcs).

Figure 8: Event Graph for Simple Queuing System

Figure 8 shows an event graph with an arrival process (with interarrival times $t_a$), a service process (with service times $t_s$), initial queue $Q$ as empty, and resource $S$ with a capacity of $k$ units. Event graphs are not flow charts, and thus are quite different from process models, but are a representation of the system structure. The directed edges or arcs indicate the influence of one event on the occurrence of other events. Edges can represent event-scheduling as well as event-canceling dynamics. Event-driven models are simulated using objects such as a simulation clock and a list of future events. The simulation engine controls the relationships between the model, the clock and the events list as simulated time passes.

2.5.6 Process-driven Model: Summary

The process-driven modeling paradigm represents a higher level of abstraction than the event graph approach. The simple queuing system discussed in the earlier section can be represented in a process-driven model as shown in Figure 9. Schriber et al. (1995, 2005)
discusses generic discrete event systems. The implementation of a process model is similar, but package specific, in simulation tools like SIMAN, Arena (Pegden et al., 1995), ProModel (ProModel Corporation, 1995), GPSS/H (Crain et al., 1994), AutoMod (Phillips, 1997), SLX (Henriksen, 2000), and Extend (Krahl et al., 1997). The models that Schriber describes are process-driven, in which the basic unit is the entity, which flows through a network of resources, time delays and routing logic. The two different possible types of entities are external and internal; where external entities explicitly exist in the simulation environment and internal entities are implicit, e.g. machine failures. While a model is being simulated, entities may be in one of a number of possible states Active, Ready, Time-Delayed, Condition-Delayed, and Dormant. In Schriber’s generic simulation model, the data structures used to organize the entities in different states are the Active Entity list, Current Event list, Future Event list, Delay list, and User-Managed list. The simulation of the model takes place in two phases, namely the Entity Movement Phase (EMP) and the Clock Update Phase (CUP). The simulation time is updated in the CUP.

![Figure 9: Process Model for Simple Queuing System](image)

The corresponding terminology for the Current Events list in SIMAN (Schriber et al., 1995) is the Current Events Chain (CEC). The Ready State entities in the CEC are removed and made active in the Entity Movement Phase. When an active entity leaves the Active State and there are no Ready State entities, then the EMP checks for “wait
conditions”, and “condition delayed entities”, in order to move them to the CEC. That is followed by the Clock Update Phase. The EMP and CUP continue in alternate order until the simulation end time or condition.

The focus of this chapter is to briefly summarize the literature of DES, components, world views, and some of the popular simulation packages that are available in the market. The simulation packages that are included in the discussion are Arena, Automod, Extend, Promodel, and Sigma. The next chapter explains the concept of Integrated Entity/Event (IE²) simulation framework.
Chapter 3 INTEGRATED ENTITY/EVENT (IE²) SIMULATION FRAMEWORK

In process-driven simulation models, the system can be represented by block diagrams or system networks through which entities flow to mimic real life system objects. In event-driven models, the system can be represented by event graphs, which focus on the abstraction of the event rather than on observable physical entities. In this chapter, a simulation environment is proposed to integrate both the approaches i.e. process and event. The main purpose of this research is to mitigate complexity of the large models through process orientation, while retaining the control over the attributes, variables and the logic through event orientation.

3.1 Simulation Framework Overview

The integrated simulation framework (Figure 10) has two main components: an IE² (Integrated Entity/Event) Integrator and an IE² model. One of the main goals for the design of the framework is to preserve the elegantly simple logic to process events, even when processing of entities is taking place simultaneously.

![Figure 10: Integrated Simulation Framework Overview](image)

3.1.1 IE² Integrator

The main functions of the IE² Integrator are a) Establish an effective communication between the process-driven and event-driven model components, and b) Efficient
handling of entities and events for better coordination of the hierarchical layers of the model. The $IE^2$ Integrator has access to prioritized entity and event lists $UE^2L$ (Updated $E^2$ Lists), the processor and the simulation clock. The $UE^2L$ maintains the lists of active entities and events as they are being populated by the $IE^2$ model. The list is sorted according to the time of the next scheduled action for an entity or event, with smaller times having higher priority. Each simulation step, the $IE^2$ Integrator compares the priorities of the highest priority item on the entity and event lists, removing the one with the highest priority. This entity or event is used to update the simulation clock, and then is sent to the processor. The processor routes the entity or event to the appropriate process block or event node, respectively. After handling the entity or event, the model object makes appropriate entries in the $UE^2L$ lists.

3.1.2 $IE^2$ model

The $IE^2$ model provides hierarchical modeling capabilities with process-driven and event-driven components as the upper and lower layers, respectively. The $IE^2$ model provides an interface for the user to build process-driven models. Process-driven models created by the user are a collection of appropriate, interconnected process blocks. In order to hierarchically embed event-driven models within a process-driven model, the $IE^2$ model provides an ‘EventGraph’ block (Figure 11). The ‘EventGraph’ block behaves as a regular process block as well as a workspace for containing an event-driven model.

The event-driven model, which is created in the ‘EventGraph’ block, updates the state of the system, schedules events, and also coordinates with the process-driven model for smooth transition of the entities into and out of the block. As shown in figure 11, an entity that enters the event-driven model or ‘EventGraph’ block makes its attributes
available to the events. The entity that enters the event-driven model is not destroyed: it is tagged to any event that it triggers. The entity attributes may be updated by the events that occur within the block. The entity is held by the event-driven model until it exits the block by the triggering of an Exit event. The updated entity then returns to continue processing by the process-driven components of the simulation engine.

Figure 11: Entity-Event Interaction in an $IE^2$ Model

3.2 Simulation Model Components: Resource & Queue

An important feature of the $IE^2$ model is that it explicitly models the entity flow taking place at the event level. This feature augments the capabilities of a simulation modeler by making some aspects of process logic available in the event layer and vice versa. However, the two layers are different when they interact with resident entities like resources, queues etc. This chapter discusses the intermediate layer that handles queues and resources for the process- and event-driven models. This flexibility provided by the intermediate layer, reduces the level of modeling abstraction at the event graph level, and leads to a more seamless, $IE^2$ model that spans the two levels. However, the entities change their states depending upon the layer in which it resides. In an $IE^2$ simulation model, entities can be in one of the following three states:

**Transient:** In a typical process-driven approach, a model is simulated as an entity or entities’ going through different processes as time evolves. The state of an
entity in such a model is a transient state. As shown in figure 12, transient entities are usually associated with the process-driven layer.

**Passive:** Any entity that is either in a queue, awaiting for an idle resource, or blocked at any stage of the simulation model is called as passive entity. As shown in Figure 12, passive entities reside in a queue of the resource layer.

**Tagged:** Entities are explicitly modeled in the event-driven layer of the $IE^2$ model as a tagged entity. In this state, the entities are associated with events. Entities move around the event-driven layer, as in process logic, by ‘tagging’ themselves to a relevant event that is triggered.

The above three states of the entities and the resource layer form the basis for the communication between process- and event-driven layers in an $IE^2$ model. The remainder of this chapter presents a brief description of the resource layer and the implementation of non-preemptive and preemptive failures. This modeling of failures benefits from the availability of event graphs in defining the failure logic, and entities in managing information flow.

![Diagram](image)

Figure 12: States of Entities in different layers of $IE^2$ model
3.2.1 Resource Layer

The resource layer in the $IE^2$ simulation framework manages the resident entities like resource, queues etc. The resource layer has been designed to be a separate layer from the process- and event-driven layers. The main aim of the resource layer is to hold the process components (in this case, queues and resources) that are common to both the layers and establish a communication between the layers. The functions of the resource layer are summarized as follows:

- Control access to the resident entities i.e. resource, queue, and
- Effective management of the global and local resident entities in large simulation models.

For effective data management, a simulation model often requires defining resources that are either global or local to different parts of the model parts. In order to have a structured way of accessing resources, the ‘Resource’ and ‘Queue’ objects are instantiated and stored in the ‘Model’ class. The purpose of associating them with the ‘Model’ class is to centralize the control of the resident entity objects. Listing 1 shows the definition of the ‘Model’ class with vectors to hold the pointers of ‘Resource’ and ‘Queue’ classes.

Listing 1: Model Class Definition

```cpp
class Model
{
public:
    Model();
    .......... 
private:
    ..........
    vector<Resource*>::Res, //Vector for "Resources"
    vector<Queue*>::Que,  //Vector for "Queue"
};
```

Listing 2: Model Class Constructor

```cpp
Model::Model
{
    .......... 
    //Instantiating a new 'Queue' object
    Queue* seizeQueue = new Queue();
    //Instantiating a new 'Resource' object
    Resource* seizeRes = new Resource(resource_unit);
    .......... 
    Res.push_back(seizeRes);
    Que.push_back(seizeQue);
}
```
Listing 2 shows the constructor of the ‘Model’ class. The resident entities i.e. resources and queues, are instantiated within the constructor and stored in the corresponding vectors. The ‘Model’ class is the main class that interfaces with the modelers while building a simulation model. The modelers use these vectors of the resident entities through the user interface to associate them with appropriate parts of the model. For example, the method ‘SeizeInfo’ (listing 3) includes a Seize block in the simulation model and passes the resource and queue references to its Seize constructor. This serves the purpose of controlled access to the resident entities. While the actual objects of the resource and queue are in the Model class, the process blocks which need the resident entities are given the access through pass by reference. In addition to that, the pointer of the process block (i.e. Seize in this case) is passed to the ‘Resource’ and ‘Seize’ objects. This helps coordination of the resource layer with $IE^2$ models, by keeping track of the process blocks that are accessing the resident entities.

```cpp
void Model::SeizeInfo(EventList &event_list)
{
    Resource* R;
    Queue* Q;
    R=Res[0];
    Q=Que[0];
    Block* newblock = new Seize(R, Q);
    BlockList.push_back(newblock);
    //Blocks that use 'Resource' are Registered at the Resource layer
    R->ResourceAllottedBlocks(newblock);
    //Blocks that use 'Queue' are Registered at the Resource layer
    Q->ResourceAllottedBlocks(newblock);
}
```

Listing 3: SeizeInfo( ) Method of the Model Class

### 3.3 Enhanced Modeling Using Entities

First introduced by Schruben (1983), an event graph (Sigma) uses event procedures as the building block to update the state of the system, and schedule other events. An important
feature of standard event graphs is parameterization of the event vertices, thus combining similar model sub-graphs as a generic graph distinguished by parameter values (Schruben, 1983). The $IE^2$ framework based on an integrated entity/event approach has been further enhanced to explicitly represent entities at the event-driven level. This addition to the integrated simulation framework helps to diminish the abstraction involved in parameter passing. The solution lies in explicitly passing the entities through the event-driven model. Event parameters are replaced by entity attributes. The usage of entities in the event-driven layer serves two purpose, a) reduce the abstraction by manipulating entity objects instead of working with parameters as in a programming language, and b) gives the intuitive feel of process-driven models to modelers at the event level, which enhances the appeal of the event-driven models.

3.3.1 Event Parameters and Edge Attributes

Event parameter passing is a parameterization of the event vertices in standard event graphs. It is an important feature of the implementation of event graphs in a modeling environment (Buss, 2001; Schruben, 2006). An event graph, as shown in Figure 13, is the representation of an event-driven model. Figure 13 represents event A and event B through nodes or vertices and scheduling of event B by event A is represent by an arc or edge. For the scheduling of an event, a Boolean condition can be evaluated and this is represented by the curved line on the scheduling arc. The event graph as designed by Schruben (1983) uses a feature called “event parameters” or “parameter passing”. The concept of event parameters is to represent similar events in a simulation model by a single vertex with different parameter values.
Figure 13: Standard Event Graph Segment with Condition (i), Delay Time \( t \), and Parameter Passing

The event parameter concept is demonstrated with the example in figure 13. When event A occurs then, if the condition \( i \) is true, event B is scheduled after a delay of \( t \) time units, with the current value of \( j \) stored in the act of scheduling B. At a later simulated time, when this instance of event B occurs, its parameter variable \( k \) is set to the stored value of \( j \). Parameter passing allows a standard event graph to represent large systems in a compact model, by parameterization of similar events.

The following summarizes the benefits of the event parameters or parameter passing:

- Event parameters represent similar events in a simulation model by a single vertex or node with different parameters, and
- It allows a standard event graph to represent large systems in a compact model.

Even though the parameter passing is a powerful feature of event graphs, it falls short or needs improvement in the following aspects:

- The event parameters is based on a programming style approach and its model representation, as shown in Figure 13, is not intuitive of what it represents, and
- Though it represents large systems in a compact model, from the usability perspective, modelers need to put in extra effort to comprehend the simulation model in terms of real systems.
3.3.2 Entities at Both Levels of the $IE^2$ Model

An important feature of the $IE^2$ model is that it explicitly models the entity flow taking place at the event level. This feature augments the capabilities of a simulation modeler by making some aspects of process logic available in the event layer. One of the prominent features of simulation modeling in event graphs is the event parameters. In our framework, the role of event parameters in the event graph is replaced by entities in the event-driven model. This change reduces the level of modeling abstraction due to parameter passing at the event graph level, and leads to a more seamless, $IE^2$ model that spans the two levels.

3.3.3 $IE^2$ Model

In order to overcome the shortcomings mentioned above, the $IE^2$ environment uses entities to play the role of event parameters. An event graph segment in an $IE^2$ model is shown in Figure 14. The important elements (including the three elements usually defined in an event graph) in an $IE^2$ model are the state variables, events that update the state variables, scheduling relationships between the events, and entities that are associated with the events. As in a standard event graph, events are represented by nodes (vertices) and the scheduling relationships between events are represented by arcs (edges). In addition, in an $IE^2$ model, entities associated with arcs and nodes are represented by a black dot as shown in Figure 14. The presence or absence of the dot at the end of the scheduling arc indicates the association of an entity with the node and arc.
In Figure 14, the nodes A and B represent events. The arc connecting nodes A and B represents the scheduling relationship between the events and it has the following functionalities:

1. Schedules event B when the condition \{bool\} is true, and
2. Associates an entity with event B

The presence of a dot at the end of a scheduling arc indicates an association of an entity with the scheduling relationship. The absence of a dot in a scheduling arc indicates that no entity association is taking place in the scheduling of that event.

The following are the advantages of the $IE^2$ event graph segment with event-entity arc over the typical event graphs as proposed by Schruben (1983):

- The improved version of event graphs, i.e. $IE^2$ model, uses entities to pass the parameters instead of using event parameters and hence, entities are explicitly represented in the model representation, as shown in the Figure 14. This reduces the abstraction associated with the implementation of variables or parameters.

- Another advantage of explicitly representing the entities at the event level is that the modeler gets the intuitiveness of a typical process-driven model. The flow chart or process-driven approach of entity movement in the event-driven model makes it very easy to comprehend large systems.
3.4 $IE^2$ Simulation Framework: User Manipulated Objects

The contributions of the $IE^2$ simulation framework can only be realized through a decent interface. Past literature on the user interface of the DES packages, namely, Kuljis (1996) and Buss et al. (2002), has been considered in this research for their relevancy. Kuljis (1996) discusses essential elements of a user interface in a simulation package. These elements when considered in the research served as guidelines for constructing user interface for the $IE^2$ simulation framework. Buss et al. (2002) is an important research to discuss in this chapter for its attempt to integrate process-driven and event-driven approaches on the user interface level. Also, this chapter presents the user interface of the $IE^2$ framework and explains about the importance of the graphical elements.

Kuljis (1996) discusses simulation packages in the context of human-computer interaction (HCI). Asserting the importance of the user interface simulation environment, the author cites the following reasons:

- It can reduce the development time,
- It can support application consistency,
- It can aid the developers throughout the development cycle,
- It can support model completeness,
- It can provide checks of model validation.

Kuljis (1996) discusses the issues that influence the ‘usability’ of simulation tools. Some of the important features discussed in the paper are, system characteristics, data input/model specification, simulation experiment, simulation results, printed manuals, and on-line user assistance. The current project does not focus on all of these features.
mentioned; it primarily discusses the data input/model specification feature. As mentioned by Kuljis (1996), the data input/model specification feature is important in evaluating different simulation packages in the following aspects:

- Model logic representation,
- Graphic elements,
- Model elements,
- Element names,
- Attribute names,
- Default values provided,
- Fill-in forms design,
- Importing files supported, and
- Model verification and validation supported.

These aspects of the simulation model development are discussed in the current research for their relevancy. The most relevant and critical for any simulation environment are model logic representation, model elements, and model validation. The model logic representation is the first step towards the making the simulation engine accessible to non-specialists. Secondly, the model elements provide a highly abstracted interface to simulation layer for the non-expert users. Finally, the model validation aspect of the simulation environment is equally important in debugging and validating the simulation. The following sections of this chapter focus on these issues while developing a conceptual simulation environment for the Integrated Entity/Event ($IE^2$) models (Kesaraju et al., 2007).
3.4.1 Model Logic Representation: Integrated Entity/Event ($IE^2$) Model

As also mentioned by Odhabi et al. (1998), a key motivation of this project is to develop a simulation environment that is accessible and usable by people from outside the core simulation modeling field. As pointed by Schneiderman (1983, 1988), a variety of front-ends to computer systems generally, and simulation modeling environments specifically exist: from command-line interfaces which rely on purely textual interaction through to graphical user interfaces which allow the user to graphically drive, or directly ‘manipulate’ the interaction. Model logic representation serves the objective by providing cognizable simulation objects in the graphical user interface and supporting the novice users through the natural and intuitive process. Figure 15 shows the set of process blocks, nodes, process connectors, event arc, and event-entity arc that form the model logic elements for the $IE^2$ model. Most of the model logic elements are known to typical simulation modelers, as the graphical representation of the process blocks are similar to the process-driven simulation software Arena and event nodes & arcs are similar to the SIGMA environment. Apart from the process blocks and event nodes that are well-known, the only model logic elements that are introduced in the $IE^2$ framework are the EventGraph block and event-entity arc. The EventGraph block provides the event graph workspace, while event-entity arc enables the representation of entities in the event layer.
3.4.2 Model Logic Representation: *EventGraph Block*

Buss et al. (2002) attempted to give event graphs the look and feel of process driven model by introducing LEGO (Listener Event Graph Objects). LEGO's are fundamentally event graphs except that they are encapsulated as atomic components and these components communicate with each other through the listener pattern, a software pattern. As shown in Figure 16, the arrival of new entities is modeled using an event graph and this event graph is encapsulated as an independent atomic component. This particular component is named *Arrival Process*. Another event graph which models a *multi-server queue* is also encapsulated as an atomic component and it is called *Multi-Server Queue*. In order for these two components to be coupled to each other, to form a complete...
In this case, the Multi-Server Queue listens to any change in the value of the state variable in the Arrival Process object.

Figure 16: Arrival Process, Multi-Server Queue, and Queue Model LEGOS (Buss et al., 2002)

The LEGO approach, by Buss et al., attempts to enhance the appeal of event graphs by incorporating the outlook of process-driven models to the event driven models. The LEGO approach has the following disadvantages:

- It merely provides the appearance of process-driven models to the event graphs,
- It does not directly support a flow chart or process flow approach or any other approach that represents real life systems and hence, it not as intuitive as a process driven model,
- It does not eliminate the necessity of programming style Parameter Passing, while modeling complex logic,
- Implementation of the LEGO framework is complex and not elegant, and
- It requires the use of the listener construct: a concept that has no direct mapping to the real world and requires the modeler to have an abstracted understanding of the simulation environment.
The $IE^2$ model remedies all of these issues. It provides hierarchical modeling capabilities with process-driven and event-driven components as the upper and lower layers, respectively. The $IE^2$ model provides an interface for the user to explicitly build process-driven models without translation. Process-driven models created by the user are a collection of appropriate, interconnected process blocks. To hierarchically embed event-driven models within a process-driven model, the $IE^2$ model provides an ‘EventGraph’ block. The ‘EventGraph’ block behaves as a regular process block as well as a workspace for containing an event-driven model. As shown in Figure 17, the $IE^2$ model approach is different from the LEGO approach by explicitly defining the role of process- and event-driven models in the integrated $IE^2$ simulation framework.

![EventGraph block diagram](image)

**Figure 17: Event-Driven Model Embedded in an EventGraph Block**

The EventGraph block clearly defines the transition of entities into and out of the event-driven model. The entry and exit of an entity in the EventGraph block is managed by the Enter and Exit events as indicated in Figure 17. The model logic representation of the $IE^2$ model is a significant change over the Buss et al. (2002) LEGO approach in actually representing as well as developing a formal relationship between process- and event-driven models. When compared with the LEGO model representation (figure 16), the $IE^2$ model representation (Figure 18 & Figure 19) is more intuitive. $IE^2$ supports a direct process flow chart approach in modeling as well as represents the flow of entities through different processes and events.
The following are the advantages of the $IE^2$ simulation environment over the Buss et al. (2002) LEGOs:

- The $IE^2$ environment actually integrates the process- and event-driven models, instead of merely giving the appearance of process models to the event graphs.
- The $IE^2$ environment supports the flow chart approach in model representation. This allows modelers to build models that not only represent the model logic but also have model animation that directly maps to the real systems.
- The $IE^2$ environment eliminates the necessity of parameter passing and also the usage any programming while modeling complex logic. This would be discussed elaborately from the usability perspective in the next section.
- $IE^2$ environment is simple in implementation and elegant in design (Kesaraju et al., 2007).
3.5 Summary

The process-driven approach to simulation modeling is important in the overall context of a simulation project, especially in allowing a fast and intuitive model development environment. The event-driven approach is equally important when detailed control over non-typical system logic is required. A simulation project will often benefit from access to both levels of modeling. Elegantly and efficiently integrating the two approaches requires a carefully designed simulation engine that combines the activity of events and entities. Development of simulation engines in the past has not considered the two approaches simultaneously.
Chapter 4  EXAMPLE MODELS IN IE\textsuperscript{2} FRAMEWORK

One of the prominent features of simulation modeling in event graphs is the event parameters. In our framework, the role of event parameters in the event graph is replaced by entities in the event-driven model. This change reduces the level of modeling abstraction due to parameter passing at the event graph level, and leads to a more seamless, \( IE^2 \) model that spans the two levels. The remainder of this chapter presents an event graph construct with limited entity logic, compares specific examples in traditional event graphs vis-à-vis the same problem in an \( IE^2 \) model, and explores the level of modeling detail obtained while reducing the level of abstraction.

4.1 Example 1: Multi-Server Queue

This example illustrates both the process-driven and event-driven model components in an \( IE^2 \) model. This model is the \( IE^2 \) model version of the process model shown in Figure 9 and event graph model shown in Figure 8. The \( IE^2 \) model is shown in Figure 20.

**System Description:** Customers arrive for processing by one of the multiple servers (e.g. bank tellers) and wait in a FIFO queue. A server is selected at random from the available servers. After processing, the customers leave the bank. The purpose of the model is to track waiting time statistics for individual customers. The simulation clock time is given the name \( Time \). The service time required for a server to process a customer is \( t_s \).

The state variables describing the system are given below,

\begin{align*}
S & : \text{Number of servers available and,} \\
Q & : \text{Number of customers that are waiting for processing.}
\end{align*}
Boolean conditions on the scheduling arcs either return true or false:

\[ \text{bool}_1: \ S > 0, \ 	ext{Entity Transfer: No} \]

\[ \text{bool}_2: \ Q > 0, \ 	ext{Entity Transfer: No} \]

Attributes of the entity (customer) are as follows:

- `QueueEnterTime`: Time at which customer/entity enters the queue
- `TimeInQueue`: Waiting time in queue of the customer/entity

![IE² Model of Multiple Server Queue](image)

Figure 20: **IE²** Model of Multiple Server Queue

Using the symbol (dot) for an entity in an event graph, Figure 20 shows an **IE²** model of a simple queuing system with multiple servers. The model primarily consists of three process blocks i.e. Create, Delay, EventGraph, and Dispose. The following is a brief description of the functionalities of each block.

- **Create**: Customer (Entity) creation with inter-arrival time \( t_a \)

- **Delay**: Customer (Entity) is delayed for time \( t_d \)

- **EventGraph**: By default, EventGraph blocks consist of an Enter event and an Exit event. These blocks provide a work space for building the event-driven portion of the model.

- **Dispose**: Finished customers/entities exit the model
The event nodes in the *EventGraph* block are defined as follows:

**Enter**: *Enter* event acts as the gateway for an entity into the event graph model in the *EventGraph* block. The attributes of the entering entity are available to the event for state changes and for scheduling other events.

**Arrive**: *Arrive* events are always scheduled with an attached entity. The entity object is stored in the queue \((Q = Q + 1)\). A *Start* Event is scheduled when a server is free,

\[
\text{QueueEnterTime} = \text{Time}.
\]

**Start**: No entity is attached to a *Start* event.

An entity (Customer) is retrieved from the queue \((Q = Q - 1)\).

One unit of the resource (machine) is made busy and the entity begins the processing \((S = S - 1)\).

\[
\text{TimeInQueue} = \text{Time} - \text{QueueEnterTime}.
\]

**Finish**: End of the processing for an entity and a unit of resource is freed.

**Exit**: Acts as a gateway back into the process model for an entity.

The *Sigma* implementation of the *Multi-Server Queue* example discussed here can be implemented by defining an array variable \(Q[i]\), \(\text{QueueEnterTime}[i]\), and \(\text{TimeInQueue}[i]\). These variables allow queue times to be computed and tracked. Here ‘i’ represents the index of an entity in the model. Parameter passing is required in the model to keep track of entity indices. This “computer programming” style of variable definitions and parameters increases the abstraction required for the event-driven
simulation modelers. This problem is eliminated in the \( IE^2 \) model by defining appropriate entity attributes. The following section discusses the abstract reduction in the \( IE^2 \) model in detail.

### 4.1.1 Discussion of Waiting Times of Individual Entities/Customers

In standard event graphs, attributes of entities in a queue must be stored somewhere. The waiting times of entities could be stored in an array variable (similar to the BANK2.MOD example in the Sigma documentation, (Schruben, 2006). To store attributes, the customer ID attribute of an entity can be used as a lookup in the array. To implement this in a standard event graph, edge attributes and event parameters are necessary to implicitly model entity ownership of information. In addition, a number of updating statements to counter-type variables must be added to the model.

In the standard (BANK2.MOD) implementation of event graphs, the modeler is not required to explicitly keep track of entity flow through the events while building a simulation model. The \( IE^2 \) model approach forces a modeler to think in terms of entities and not in programmer-level abstractions such as parameter passing. At the same time the model-builder is not deprived of the event graph capabilities. On comparing the two models of the same problem in two different simulation environments, one can observe that in an \( IE^2 \) model, a modeler can reach the same goals without resorting to a “programming” type of approach. This demonstrates the attenuation of abstraction in the \( IE^2 \) model in comparison with a classical event graph.
4.2 Example 2: Rework (Sigma)

The next example demonstrates the effective use of entity attributes for modeling rework situations. Modeling of rework requires slight modifications to the model shown in Figure 21. This model demonstrates the ease of comprehension of some ideas in the event graph when enhanced with entities, as well as to show the elegance of the combined event/entity logic in an $IE^2$ model. This example is based on an example in Schruben (2006).

System Description: The parts completing processing at a machine require rework with probability $P$. When a part requires rework, it is returned to the queue and receives processing as if it were a new part. The variable $t_s$ is redefined for this model as follows:

Time required for the machine to process or rework a part: $t_s$

In addition to the events defined in the previous example, the event Rework is defined as the point in time when it is determined that rework is needed on the current part. In the rework event, the entity is returned to the queue, and a Start event is scheduled. An entity attribute could be updated at this point to track the number of times the current part has been determined to need rework.

![IE² Model of Multiple Server Queue with Rework](image)

Figure 21: $IE^2$ Model of Multiple Server Queue with Rework
The other events in the $IE^2$ model from Example 1 must be modified as follows to implement the rework example as shown in Figure 21.

**Finish**: End of the processing on the machine.

Updates the resource ($S = S + 1$) and generates a random number $R$ between 0 and 1.

**Rework**: The reworked entity is sent to the queue ($Q = Q + 1$) for processing on the machine.

Conditions on the scheduling must be added and modified to include the rework logic:

$$\{\text{bool}\}_1 : S > 0, \text{Entity Transfer: No}$$

$$\{\text{bool}\}_2 : Q > 0 \text{ and } R > P, \text{Entity Transfer: No}$$

$$\{\text{bool}\}_3 : R \leq P, \text{Entity Transfer: Yes}$$

In this example, additional simulation logic of *Rework* is implemented in the model. The same model implemented in *Sigma* requires defining an array variable for *Rework*. As in earlier discussion, this example is used to highlight the advantage of using the entity attributes in place of variables. The following discussion details the advantages of entities over the programming style variables in the event-driven models.

### 4.2.1 Discussion of $IE^2$ Model for Multiple Server and Rework

Comparing the standard event graph models for the multiple server and rework examples with the $IE^2$ model, note that the $IE^2$ model avoids the abstraction involved in defining event parameters to track individual entities. At the event level, entities are handled as
objects in a way that is analogous to their treatment in the process models. The attributes of an entity (like QueueEnterTime, TimeInQueue, Rework) are defined by the modeler, enabling the flexibility and explicit handling of entities at the event level. Instead of passing event parameters to other nodes as in a programming language, the $IE^2$ model defines them explicitly as attributes of entities that are associated with events as they are scheduled. This entity passing through the events in the event graph, gives the intuitive feel of the process-driven model to the modelers. This modeling of entity flow through the event graph enhances the appeal of event graph to modelers with a process perspective, while retaining the power and flexibility of the event logic. Next chapter of this proposal deals with an example of a supply chain system. The discussion of the modeling issues while developing an $IE^2$ model for this example helps to reveal the true potential of the $IE^2$ framework.

4.5 Failure of a Resource

Resources can be in one of the three states i.e. idle, busy, and breakdown. When a resource is not processing an entity, then the resource is said to be in the idle state. If the resource is busy with an entity, then the resource is said to be in busy state. Failures are the random events that cause the resource or servers to become unavailable. When a failure occurs then the resource state is updated to the breakdown state. There are certain activities such as parts replacement, cleaning and tool adjustments that require the resource to stop processing entities. These events or processes may not be viewed as failures but can change or update the state of the resource to breakdown.

Whenever a failure of a resource occurs and it is idle, then the state of the resource changes from idle to breakdown. If the resource is busy in processing an entity and a
failure has occurred then the resource can respond in two ways, *preemptive* and *non-preemptive*. In the *preemptive* case, the resource preempts the repair process by terminating the processing of the working entity. The state of resource is then changed or updated to the *breakdown* state. The remaining processing of the entity is continued once the machine returns to its working state. Alternatively, in the *non-preemptive* case, the resource starts the repair or maintenance only after finishing the process of the entity. The state of the resource is then changed or updated to *breakdown* state. The approach towards failures in the IE$^2$ model is different from that of any process-driven DES software. The following section discusses failures in *Arena*. This discussion is helpful in making direct comparisons of the handling of failures in IE$^2$ models and *Arena* respectively.

### 4.5.1 Failures in Process-Driven Software (*Arena*)

Generally, the *Process* module (Figure 22) in *Arena* is used to handle ‘Multi-Server Queue’ models. The *Process* module represents a resource and its queue, and supports the tracking of entity delay times. The dialog box that allows a user to modify the properties of the *Process* module, as shown in Figure 22, has options for different failure logic. These correspond to combinations of more primitive process blocks. For example, ‘*Seize Delay Release*’ action would allow the entity to ‘*Seize*’ the resource for the entity, ‘*Delay*’ the entity for the time specified as service time, and finally, ‘*Release*’ the resource from the entity. The information related to resources can be specified by the users in this dialog box. Notice that this does not specify any failure logic.
In addition, information about the *Process* modules can be entered in the model through the spreadsheet view of the simulation objects. Figure 23 shows the spreadsheet view of the resource used in this Process module and the various failure options available. The failures options available in *Arena* are, *Wait*, *Ignore* and *Preempt*. From the description provided by Kelton et al. (2004), *Wait* and *Preempt* options are similar to the *non-preemptive* and *preemptive* failures. The *Ignore* option is similar to the *Wait* option, but the failure time or down time is shortened by a duration that is equivalent to the duration the resource was busy with the entity, during the occurrence of the failure.
The functionality of the failure options is not apparent from the dialog box or spreadsheet view of the Arena. The Arena users had to be well-informed about the failure options. The Arena documentation explains the behavior of Wait, Ignore and Preempt options and mentions general rules of thumb to choose a specific option for a simulation model. However, Arena does not provide any option at the modeling level to add any additional details to the failure options or implement any non-standard failures. The user has to be a VBA programmer to access lower level logic of the model. The IE² model, on the other hand, exposes three different layers Process, Event and Resource. The shortcomings of the Arena discussed in this paragraph are resolved in the IE² simulation framework. The following section describes the modeling of the failures in an IE² Model.

4.5.2 Failures in the IE² Model

In the IE² simulation framework, preemptive and non-preemptive failures are modeled using different blocks at the process level. These blocks implement different logic at the event graph level. As shown in the Figure 24 and Figure 26, two pre-built EventGraph blocks are included in the framework to model the preemptive and non-preemptive failures. The pre-built IE² model has three main events to model the failures i.e. MachineFail, RepairStart, and RepairFinish. As the name suggests, the MachineFail event changes the status of the resource to breakdown, RepairStart event starts the repair, while RepairFinish represents the end of the machine repair.

INIT: Initializes the machine failure event with a time delay of $t_{fa}$

MachineFail: Update the failure variable, $F=F+1$

Schedules a RepairStart event if the resource is not busy, $(S>0)$
RepairStart: Start of the resource/machine repair

Change the status of the resource to breakdown, i.e. $S = -1$

Schedule a RepairFinish event unconditionally, with a time delay of ‘$t_R$’

RepairFinish: End of the resource/machine repair,

Change the status of the resource to idle, i.e. $S = 1$

Schedule a MachineFail event with time delay of ‘$t_{fa}$’

4.5.3 Non-Preemptive Failures in $IE^2$ Model

The functionality of the pre-built non-preemptive process block is to model a non-preemptive failure along with the typical operations of a process block i.e. Seize, Delay, and Release. The major difference between the typical process block and the non-preemptive process is that the Seize, Delay, and Release processes are modeled using $IE^2$ approach.

![Figure 24: Non-Preemptive Failures in an $IE^2$ Model](image)

The first machine failure (MachineFail) event is scheduled by the Enter event of the non-preemptive process block. This will trigger the non-preemptive failure process. As mentioned above, the MachineFail event changes the state of the resource to breakdown
(S=-1). The following points (Figure 24 & Figure 25) explain the functioning of non-preemptive failure,

- The Finish event schedules the RepairStart event if the condition \(\text{bool}3\) is true. The \(\text{bool}3\) checks if the state of the resource \((\text{bool}3: \text{if } (S<1))\) is breakdown.

- RepairFinish event checks \((\text{bool}4: \text{if } (Q>0))\) for any entity waiting for service, if true then the entity is removed from the queue and inserted in the event list for the Start event.

Figure 25 shows a different version of the non-preemptive failure model using the entity-event node. The entity-event node is used in the \(IE^2\) model to handle entities at the event level. The example shown in Figure 25 uses the entity-event node to handle failures as entities at the event level that is analogous to the failure entities at the process level.

![Figure 25: Non-Preemptive Failures with an Event-Entity Arc in \(IE^2\) Model](image)

4.5.4 Preemptive Failures in \(IE^2\) Model

The preemptive failure process block is similar to the non-preemptive process block, except for a few major changes. The important change is a canceling event that is
scheduled by the *MachineFail* event. This *canceling* of the *Finish* event characterizes the *preemptive* failure.

![Figure 26: Preemptive Failures in IE^2 model](image)

As discussed earlier, the first *MachineFail* event is scheduled by the *INIT* event (Figure 26). This will act as a trigger to the *preemptive* failure process. The following are the major events scheduled and changes in state variables,

- When the *MachineFail* event occurs, it updates the failure variable, \( F = F + 1 \). The *MachineFail* event schedules the *RepairStart*, if the resource is available (\( S > 0 \)).

- The *RepairStart* event preemptively schedules the *Finish* event. The *Finish* event is canceled by the *RepairStart* event through *canceling* edge. The *RepairStart* event updates the resource variable to *breakdown* (\( S = -1 \)), before scheduling *RepairFinish* event (with time delay of \( t_f \)).

- *RepairFinish* event checks for any entity waiting for service (\( \{ \text{bool} \}_4\): if \( Q > 0 \)), if true then the entity is removed from the queue (\( Q = Q - 1 \) and inserted in the event list for the *Finish* event.
4.5.5 Benefits

A major benefit of the $IE^2$ model is that failure can be discussed seamlessly at both the event and process levels. Compare this to the handling of the failures in typical DES software, e.g. Arena, where failures are options provided to the modelers. For training of new simulation modelers, the discussion of both of these views of failures would typically require introducing a formal programming language. In the $IE^2$ framework, these two models of failures can be contrasted easily. In addition, for more sophisticated users, these models can be made more elaborate without knowledge of detailed programming logic. For example, although in the ARENA model adding customized logic is impossible through the dialog box, in an $IE^2$ model, the user could easily modify the event graph to meet a specific modeling situation. And in the $IE^2$ model these modifications could be made using the event graph formalism, rather than having to access logic and code specific to the language. Thus, the integrated framework creates a platform for all levels of users to interact by building components that utilize a knowledge base (event scheduling) that is fundamental to simulation modeling.

4.6 Enhanced Modeling in an $IE^2$ Framework: Example

In order to illustrate both the process- and event-driven model components in an $IE^2$ model, a supply chain system was studied as an example. Evaluating the Impact of Retailer Gaming and Supplier Capacity Allocation on Supply Chain Costs (Vutukuru, 2006), focuses on a supplier-retailer supply chain, consisting of a single supplier and three retailers. The model considers how partial information sharing has an impact on the supply chain costs with different allocation mechanisms on the supplier side vis-à-vis gaming behaviors on the retailer side. The model was originally implemented in Arena.
4.6.1 Problem Definition

The highlights of the problem definition that are mentioned in the research can be summarized as follows:

- To study the activity between a supplier and three retailers in a supply chain with partial information sharing under varying supplier capacity allocation policies, retailer policies and retailer backorder to inventory holding cost ratios.

- Apply simulation to generate an output in terms of costs for a given set of input parameters i.e. in terms of retailer order up to levels, retailer mean demand, retailer standard deviation and supplier order up to level.

- Developing an efficient cost framework that identifies the optimum scenario in which retailers, suppliers as well as the entire system benefits.

4.6.2 System Definition and Model Formulation

There are certain assumptions and restrictions considered by Vutukuru (2006) on the simulated system.

Assumptions

- The time period (i.e. variable indicating what day it is) in the system gets updated every day at 20:30 hours

- The retailer demand occurs daily at 21:00 hours and their inventory get reduced by the amount equal to the demand

- After the retailer demand occurs, retailer inventory is updated; costs are assigned, and retailers submit their orders to the supplier at 22:00 hours.
After the supplier receives all the orders, inventory is checked to see if there is sufficient stock to cover all orders. After deciding upon the order fulfillment, the supplier places an order with his manufacturer at 23:00 hours.

All these assumptions are summarized in Figure 27:

First, the retailer checks his inventory and submits his orders.

Next, the submitted orders are received by the supplier, who checks inventory and decides how to allocate the inventory.

Next, retailers receive their orders after a specified supplier lead time.

![Diagram of Supplier-Retailer Order Cycle](image)

**Figure 27: Supplier-Retailer Order Cycle (Vutukuru, 2006)**

### 4.6.3 Model Formulation

While building the model, the modelers identified the vital components of the model and considered the assumptions made earlier. Supplier and retailer activities are in the form of blocks and arrows represent the flow of orders and information between them. The flow in the logical model starts with retailers submitting their orders and ends with supplier delivering completed orders to the retailers.
Retailer order entities are created each time a retailer reviews its inventory. Retailer order quantity is defined as the size of order that a retailer submits to the supplier. If the retailer order quantity is equal to zero, then there is no need for an order and the entity gets disposed. For a positive set of retailer order quantities arriving to the supplier, the supplier first checks his inventory for any shortage. If the available inventory is greater than the received orders, the supplier fills all the orders completely. Otherwise the retailers’ orders are filled partially. In partial fills, an allocation mechanism is employed by the supplier to distribute the available inventory. The allocation mechanisms used were proportional allocation and ranking based on order size allocation.

Implementing allocation mechanisms in the Arena model is a complex task and hence a more detailed language was used to develop the allocation logic and link it to Arena. VBA was employed to handle the allocation mechanism computations as VBA has an excellent interface with Arena and is inbuilt in Arena. Allocation mechanisms were coded in VBA and linked to the Arena model. VBA-based blocks are an integral part of Arena. The logical model shown below, Figure 28, depicts the capacity allocation component, where the retailers’ orders are allocated stock when there is a shortage of supplier stock.

The unfilled retailer orders are held by the supplier until there is enough stock. The supplier cannot fill any orders when he is out of stock. After the supplier decides to fill complete or partial orders, the orders get delivered to the retailer inventory with a lead-time of two days. The interesting phase in this model is the allocation of the available supplier inventory to the retailers.
Figure 28: Formulated Model with Capacity Allocation Logic (VBA) (Vutukuru, 2006)

4.6.4 Components of the Arena Model

The Arena model represents the supplier retailer interaction with the help of order entities, which are created by the retailers, interact with the supplier, get processed and exit the system. The following are the components of the model under consideration:
Entities

The entities considered are the retailer orders, supplier orders, retailer daily demand entities and a daily clock entity. These four entities are interlinked with each other and allow the order processing logic to proceed in a realistic way.

These four entities arrive in the system in the order given below:

- Daily clock entity is created and updates the day counter at 20:30 hours everyday,
- Retailer daily demand is created each day at 21:00 hours,
- Retailers order once a week and this entity is created weekly at 22:00 hours,
- A supplier order occurs every day at 23:00 hours.

There are a total of three retailers in the system. All retailer variables in the model have three rows associated with their data, representing the data of the retailers. The holding and backorder cost variables represent the inventory holding and shortage penalty costs for both the retailers and suppliers. The total retailer cost variable is calculated by adding the holding and backorder costs for each retailer and the average retailer cost is calculated by dividing the total cost by number of periods. Retailer and supplier costs are the key performance variables as the system’s performance is evaluated based on these costs. The pipeline variable for both supplier and retailers represent the outstanding orders that have been placed, but not yet filled. Figure 29 shows the retailer sub-models that generates retailer orders and demands for three different retailers.
4.6.5 Comparison of Arena based Process-Driven Model with IE² Model

The sub-models of the retailers for the demand and orders are compared with their counterpart IE² model. This comparison highlights the use of entities at the event level, allowing similar model sub-graphs of the process-driven model to be combined together as a generic sub-graph distinguished by the attributes that entities carry into the event graph block. Figure 30 shows the process-driven version of the single retailer demand, inventory and cost model.

Figure 30: Details of the Retailer demand and cost sub model in Arena (Vutukuru, 2006)

System Description: Figure 31 shows the comprehensive IE² model for retailer demand and cost sub-models. The retailer demand entities are created in the create blocks, where the number of Create blocks depend upon the number of retailers in the model. The
entities enter the *EventGraph* block through the *Enter* event and schedules events, update attributes, and check conditions as follows:

**State Variables:**
- RetailerInventory\[i\],
- TotalRetailerCost\[i\],
- AvgRetailerCost\[i\],
- HRetailer\[i\],

where ‘\(i\)’ represents entity parameter and

**NumPeriods**, represents number of time periods

Boolean conditions, represented by \{bool\}, on the scheduling arcs either return true or false:

\[
\{\text{bool}\}_1 : \quad \text{RetailerInventory}[\text{Entity.RetailerNum}] > 0, \quad \text{Entity Transfer: Yes}
\]

\[
\{\text{bool}\}_2 : \quad \text{RetailerInventory}[\text{Entity.RetailerNum}] \leq 0, \quad \text{Entity Transfer: Yes}
\]

Attributes of the entity (customer) are as follows:

**RetailerNum:** Time at which customer/entity enters the queue

**Figure 31:** Comprehensive *IE*^2* Model for Retailer Demand & Cost Sub-Models
Using the symbol (dot) for an entity in an event graph, Figure 31 shows an \( IE^2 \) model of the retailer demand and cost. The model primarily consists of three process blocks i.e. \( Create, \ EventGraph, \) and \( Dispose. \) The following is a brief description of the functionalities of each block.

\( Create: \) Retailer daily demand (Entity is created) occurs and arrives at 21:00 hours

\( Entity.RetailerNum \) is updated as 1, 2, 3......etc accordingly, depending upon the \( create \) block in which it is created, to represent the different retailers.

\( EventGraph: \) By default, \( EventGraph \) blocks consist of an \( Enter \) event and an \( Exit \) event.

These blocks provide a work space for building the event-driven portion of the model.

\( Dispose: \) Finished retailer demand/entities exit the model

The event nodes in the \( EventGraph \) block are defined as follows:

\( Enter: \) Entities use \( Enter \) event as a gateway to event graph model, in the \( EventGraph \) block. The attributes of the entering entity are available to the event for state changes and for scheduling other events, in this case, an event called \( DemandOccurs \) is scheduled.

\( DemandOccurs: \) This event generates the demand through a known distribution \( Dist(x,y) \) to update the variable \( Demand. \) Two different events are scheduled by checking the conditions \( \{bool\}_1 \) and \( \{bool\}_2 \).
CalculateHoldingCost:

Holding cost of the retailer is calculated in this event as follows,

\[
\]

Exit event is scheduled unconditionally.

CalculateBackOrderCost:

Backorder cost of the retailer is calculated in this event as follows,

\[
\]

Exit event is scheduled unconditionally.

4.6.6 Discussion of Comprehensive IE2 Model

Comparing the standard Arena based process-driven model with the IE2 model, we can note that the IE2 model uses the concept of event parameterization through the entity attributes. The advantage of using the entity attributes in the IE2 model is that the similar model sub-graphs can be combined together as a generic sub-graph distinguished by entity attribute values (e.g. RetailerNum). At the event level, entities are handled as objects in a way that is analogous to their treatment in the process models. The attributes of an entity (like RetailerNum) are defined by the modeler, enabling the flexibility and explicit handling of entities at the event level. Instead of passing information as event parameters to other nodes as in a programming language, the IE2 model defines them
explicitly as attributes of entities that are associated with events as they are scheduled. This entity passing through the events in the event graph, gives the intuitive feel of the process-driven model to the modelers. This modeling of entity flow through the event graph enhances the appeal of event graphs to modelers with a process perspective, while retaining the power and flexibility of the event logic. At the process level, the modelers’ ability is enhanced to model complex logic without resorting to programming languages in a simulation model. One of the major objectives of the $IE^2$ model is to diminish the gap between real world processes and their representation in the simulation environment and not limiting itself to the graphical representation.
CHAPTER 5 PROOF OF CONCEPT: SOFTWARE ARCHITECTURE

The $IE^2$ simulation framework is a proof of concept that demonstrates and establishes the viability of integrating two different approaches to Discrete Event Simulation (DES). This research was developed with an emphasis on an industrial engineers’ and model builders’ perspective rather than as a software development project. This is reflected in the design of the framework.

5.1 Simulation Engine

A simulation engine that supports process-driven models was developed during the inception phase of the research. The choice of programming language, C++, was based on its imperative, object-oriented and generic paradigms. The behavior/algorithm of the DES should be captured in a software architecture that is elegant in representing the $IE^2$ structure and efficient as a simulation engine. At this stage, different software design patterns were considered, as they provide a more tested and proven development paradigm that can speed up the process of software development process. The mediator pattern was the most appropriate software pattern which can express the required functionality.

A simulation framework consists of a number of objects (blocks/classes) and, logic and computation is distributed among different objects (blocks/classes). As the simulation model increases in size and complexity, communication and interaction between different objects increases. This can lead into a complex maze of references between the objects (blocks) as shown in Figure 32. As mentioned by Kuchana (2004) a high degree of referencing affects the maintainability of the application and highly coupled objects greatly reduces the scope for reuse.
The mediator pattern eliminates the complexity arising from objects referring to each other. It forms the basis for the controlled, coordinated communication model for the group of objects. As shown in Figure 33, all the object (blocks) interaction details are abstracted into a separate *Simulation* (Mediator) class. Every block is still responsible for the intended functionality, but they do not interact with each directly. The interaction between any two different blocks (such as, *Process* block sending an entity to *Dispose* block) is routed through the *Simulation* (Mediator) class. All the blocks or nodes send their entities or events respectively to the *Simulation* (Mediator) class. The *Simulation* (Mediator) class the assigns the appropriate entity or event to the corresponding block or node respectively, as per the simulation model requirement.

---

**Figure 32: Complexity in Communication and Interaction between Process Blocks**

**Figure 33: Mediator Pattern Applied to the Simulation Framework**
The resulting software design has the following advantages:

- As the entire block interaction behavior is moved to the Simulation (Mediator) class, it will be easier to alter the behavior of the block interrelationships.
- Centralizing the inter block dependencies to the Simulation class results in enhanced object reusability.
- As blocks are less coupled with other blocks, their behavior/functionality can be easily modified and tested.

5.2 Design Approach

To implement this simulation framework, the classes shown in Figure 34 were created with the following functionalities:

- A Simulation class (listing 1) that handles both entities and events, updates the simulation clock, and determines the order of processing.

```cpp
void Simulation::ManageSimulation(EntityList &entity_list, EventList &event_list, Entity *current_entity,
                                    Event *current_event) {
  if(entity_list.IsEntityListOccupied()) {
    EntityTime = current_entity->time;  // current time of the Entity
  }...
  ...

  if(event_list.IsEventListOccupied()) {
    EventTime = current_event->time;    // current time of the Event
  }...
  ...

  if(EntityTime > EventTime) {
    // Compare the current time of Entity & Event
    current_event = Get_Event(event_list, current_event); // Retrieve the Event from the list
    UpdateSimClock(current_event, sim_time);              // Updating the Simulation Clock
    /*Process the Event at the relevant node*/
    current_event->CurrentNode->process_event(event_list, entity_list, current_event);
  }
  else {
    current_entity = Get_Entity(entity_list, current_entity);    // Retrieve the Entity from the list
    UpdateSimClock(current_entity, sim_time);                    // Updating the Simulation Clock
    /*Process the Entity at the relevant block*/
    current_entity->NextBlock->process_entity(entity_list, event_list, current_entity, SimEndTime);
  }
}
```

Listing 4: Implementation of the Simulation Class
• A **Block** class that implements commonly used process-driven components, with methods that update entity attributes and state variables appropriately (listing 2). The event graph capability is available hierarchically within a special type of block.

```cpp
void Block::process_entity(EntityList &entity_list, EventList &event_list, Entity* current_entity, double SimEndTime)
{
    ...;
    current_entity->NextBlock = current_entity->NextBlock->NextList[0];  //Update the next block variable for the entity */
    entity_list.InsertInList(current_entity);  //Reinserting the updated Entity into the entity list */
}

Listing 5: Implementation of *process_entity*() method in a **Block** class
```

• A **EventGraph** class is defined as an sub-class of the **Block** class (listing 3). The entities are processed in this class through a method (listing 4) which tags the entities to an appropriate event.

```cpp
class EventGraph: Block{  //EventGraph class is defined as a sub-class of the Block class */
public:
    BlockID = NumberOfEventGraphBlocks;
    BlockName = "EventGraph";
};
```

Listing 6: Definition of **EventGraph** Class

```cpp
void EventGraph::process_entity(EntityList &entity_list, EventList &event_list, Entity* current_entity, double SimEndTime)
{
    EventGraphEntry(current_entity,event_list);  //Entry of Entity into a Event Graph Model */
}
```

Listing 7: Entity processing in an **EventGraph** class
- A **Model** class (listing 5) whose methods build and store process-driven models from predefined **Block** objects and which provides hierarchical model building capability for event graphs in blocks with unique capabilities.

```c
void Model::read(EntityList &entity_list, EventList &event_list) {
    /* Reads the model from the user */
}
```

Listing 8: **Model** class method **read**() acts as an interface for the user to build models

- A **Node** class (listing 6) supports the event graph structure. This class enables the event/entity communication with appropriate methods for entry and exit of entities.

```c
void Node::process_event(EventList &event_list, EntityList &entity_list, Event* &current_event) {
    UpdateState(current_event, current_entity, event_list); /* Updating State Variables */
    ScheduleEvent(current_event, event_list, entity_list, current_entity); /* Schedule Future Event */
}
```

Listing 9: **Node** class implements methods that update state variables and schedule events

![Class Association to Implement the Simulation Framework](image)

Figure 34: Class Association to Implement the Simulation Framework
5.2.1 The EntityList Class & EventList Class

The EntityList and EventList classes, though not discussed as a part of the framework in Figure 34, are necessary for storing and retrieving entities and events. These classes are not directly accessed by users, but their functionality is available through entity movement between blocks in the process-driven model and through the event scheduling activity in the event graph blocks.

5.2.2 The Entity Class & Event Class

The Entity class has two important variables, one for the time of the next process action, and another for a reference to the next process block the entity will visit in its model, NextBlock. The functions of the two variables are as follows:

- The ‘time’ variable is used to update the simulation clock as the simulation evolves, and to choose the next action in the $E^2$ Integrator.

- A process-driven model involves entities passing through a series of process blocks: the variable ‘NextBlock’ indicates where in the model an entity must go after the current action.

As in the Entity class, the Event class has a ‘time’ variable to indicate when the event is scheduled to occur, and a reference to the node in the event graph model to which it corresponds. In order to facilitate entity/event communication each event is tagged to an entity and to the next block or process:

- The variable ‘TaggedEntity’ holds the reference to the entity that entered the event graph block.
The ‘ExitBlock’ variable is a reference for variable of type Block, which holds the reference to the next process block. This information is used by the ‘TaggedEntity’ when it exits the event graph block and returns to the process-driven model.

5.3 Implementation

Figure 35 shows the message flows among the Simulation, Model, Block, and Node classes. The read( ) method of the Model object, that is associated with the Simulation class is invoked to read the model from the user. The Model class instantiates the Block and Node classes. The instantiated classes are used to create the process-driven and event-driven models. After reading the model, the ManageSimulation( ) method of the Simulation class executes a loop, to process the entities and events, until the end of simulation. In this method, the GetEntity( ) and GetEvent( ) methods read an entity and an event (respectively) from the appropriate list. The class association diagrams are included in the Appendix.
The ‘time’ associated with an entity and an event is compared and the element with the earliest time is removed from the corresponding list. The entity or event that is removed from the list is passed as a reference to an appropriate method, i.e. `process_entity()` or `process_event()`. These methods will transfer control of the entity or event to the process-driven (Block) or event-driven (Node) component where it is handled accordingly.

### 5.4 ‘Resource’ and ‘Queue’ Classes

The resources in a simulation model can represent machines, operators, or any other finite capacity objects. The entities in the model compete for these Resources. When the resource is occupied or busy with other entities, the newly arriving entities wait in the Queue. This inherent relationship between the Resource and Queue and their interactions with other parts of the model is very important to a simulation modeler. These critical

---

**Figure 35: Simulation Framework: Message Flow**

The diagram illustrates the message flow in the simulation framework, showing how entities and events are managed through various methods such as `read()`, `ManageSimulation()`, `GetEntity()`, `GetEvent()`, `UpdateSimClock()`, `process_entity()`, and `process_event()`. The flow is directed through the list management and resource allocation processes, highlighting the dynamic nature of the simulation model.
factors are carefully considered while building the framework for $IE^2$ models. The class definitions for Resource and Queue are shown in the listing 10 and 11. Resource class has methods for releasing the resource or updating the status of the resource after a busy period, assigning a resource for an entity, returns a Boolean value to inform the status of the resource etc. The Queue class inserts an entity into the queue, removes an entity from the queue, checks if the queue is occupied and returns a Boolean value etc.

```cpp
class Resource
{
public:
    Resource();
    // Constructor for Resource class, input parameters: # of resource units
    Resource(int resource);
private:
    void ReleaseResourceUnit(); // S++
    void AssignResourceUnit(); // S--
    bool IsResourceAvailable(); // (S>0) Condition
    // Blocks that are Allotted 'Resource' are stored
    void ResourceAllottedBlocks(Block*);
    vector<Block*>* AllottedBlocks;
    int availability;
    int maintenance;
};

Listing 10: Resource Class Definition

class Queue
{
public:
    Queue();
private:
    bool CheckQueueOccupied();
    void InsertQueue(Entity* current_entity);
    void Insert(Entity* current_entity);
    Entity* RemoveQueue();
    ..............
    // Blocks that are Allotted 'Resource' are stored
    void ResourceAllottedBlocks(Block*);
    vector<Block*>* AllottedBlocks;
    ..............
};

Listing 11: Queue Class Definition
5.5 Process Components in $IE^2$ Framework

The $IE^2$ framework provides a set of basic process blocks for building process-driven models. In addition to that, a sub-model block is also provided for building multiple layers or hierarchical simulation models. These process components were built on the existing simulation framework namely, queues, resources, blocks, etc. The purpose of building the process-driven model components, in an integrated framework, is to augment the modeling capabilities of a modeler. However, an elegant and accurate model can only be realized by the modelers’ acumen in using the event- and process-driven models appropriately. In this chapter, the process components like, Create, Seize, Delay, Release, Dispose, Decision, Batch, Separate, and Sub-Model are explained in detail. Typical options available in each of these blocks are also incorporated in these process-driven components and their construct is explained in this chapter.

The process components are defined as a generic class ‘Block’ (listing 12). As shown in Figure 36, all the other process components are defined as sub-classes of the generic ‘Block’ class. This is helpful in dynamically binding the common methods in the sub-classes. Therefore the blocks with special functionalities are simply inherited or defined as sub-classes.
Figure 36: Block Inheritance

```cpp
class Block {
    public:
        Block();
        virtual void process_entity(EntityList &entity_list, EventList &event_list, Entity* current_entity);
        ... ...
        vector<Block*>* NextList;
        ... ...
    }
```

Listing 12: Generic Block class

The `process_entity( )` method is defined as `virtual` in the base class i.e. Block class. The purpose of the defining it as `virtual` is that, when the method is invoked then the appropriate version of the method in the sub-classes is executed. A variable ‘NextList’ is defined as a vector of Block pointers. The variable ‘NextList’ is used to store the memory address of the blocks that are connected next to the present block when the model is built.

The next sections discuss the different process blocks that are available in the integrated simulation framework. All the process blocks are defined as the sub-classes of the generic Block class.
5.5.1 Create Process Block

A Create process block is used in a process-driven model to introduce new entities into a process with a specified inter arrival time. The Create class is defined as a sub-class of the generic Block class.

```java
class Create : Block
{
    int EntityCounter; //Number of Entities Created
    double EntityTime; //The time at which Entity is created
    int CreateBlockID; //Create Block Identifier
    double inter; //Variable for InterArrival times

    public:

    Create(EntityList &entity_list, int NumberofCreateBlocks);
    //Instantiate a New Entity
    void CreateEntity(EntityList &entity_list, double &ReplaceTime);
    //Process Entity in a 'Create' Process Module
    void process_entity(EntityList &entity_list, EventList &event_list,
                        Entity* current_entity);
    //Variables to keep track of 'Entity' and 'Create' Block Identifiers
    int EntityCounter;
    double EntityTime;
    int CreateBlockID;
    double inter;
};
```

Listing 13: Create class

The constructor of the Create (listing 13) class instantiates an entity and inserts it into the entity list. This will trigger the simulation of the model and schedules it for the Create process. When the entity is processed through the `process_entity()` method of the Create block, another entity is instantiated and inserted into the entity list. In this way, the entities are created in an iterative fashion until the termination condition is encountered. Each creation of an entity causes the creation of another entity in the future.

5.5.2 Seize Process Block

The main function of the Seize process block is to seize a resource, if available, or else insert the entity into the appropriate queue. As shown in the definition of the Seize class (listing 14), variables for Resource and Queue are included. These variables hold the memory addresses that refer to objects of the Resource and Queue in the resource layer.
Listing 14: Seize class

The method `process_entity()` (listing 15) checks the availability of the ‘Resource’ through its reference pointer `SeizeRes`. If there is availability of ‘Resource’, then the status of ‘Resource’ is reset to ‘Busy’ else, the entity is inserted into a ‘Queue’.

```
void Seize::process_entity(EntityList &entity_list, EventList event_list, Entity* current_entity) {
    ... ... ...
    if(SeizeRes->availability<1) {
        SeizeQue->InsertQueue(current_entity);  // Entity waits in the ‘Queue’
    } else {
        SeizeRes->AssignResourceUnit();       // Resets the status of the ‘Resource’ to ‘Busy’
    }
    ... ... ...
}
```

Listing 15: `process_entity()` method of Seize Class

5.5.3 Delay Process Block

The Delay process block is the simplest among the process blocks in the process-driven components of the integrated $IE^2$ framework. The constructor of the Delay class (listing 16) updates the variable ‘DelayTime’ for the delay time. While the `process_entity()` (listing 17) updates the ‘time’ variable of the entity through its delay time variable ‘DelayTime’ and inserts the entity into the entity list with the corresponding time controlling when it completes its delay.
Listing 16: Delay class

```cpp
class Delay::Block
{
public:
    Delay(double exp):
        //Variable for Delay times
double DelayTime;
}

Listing 17: process_entity( ) method of Delay Class

5.5.4 Release Process Block

Similar to the definition of the Seize class, the Release class (listing 18) also declares variables for Resource and Queue. These variables hold the memory addresses that refer to objects of the Resource and Queue in the resource layer. The functionalities of the Release process block are as follows:

- Release or update the resource from busy to available
- Check if any entities waiting for the resource
- If any entity is waiting in the queue, then schedule the entity for the Seize process.

Listing 18: Release class
As shown in listing 19, the `process_entity()` method of the `Release` class updates the state of the `Resource` class through the pointer reference `SeizeRes`. The `ScheduleNextProcess()` method is called to check if any entity is waiting for the resource, and if true, then schedule the entity for the `Seize` process.

```cpp
void Release::process_entity(EntityList &entity_list, EventList event_list, Entity* current_entity)
{
    ............
    SeizeRes->ReleaseResourceUnit();
    ScheduleNextProcess(entity_list, current_entity);
}
```

Listing 19: `process_entity()` method of the `Release` class

### 5.5.5 Dispose Process Block

The main purpose of `Dispose` block is to act as the end of a process for an entity in the model. In order to model the entities leaving a system, `Dispose` blocks are used in the process-driven model. Listing 20 and 21 show the constructor and `process_entity()` method of the `Dispose` class. In the `process_entity()` method of the `Dispose` object, the pointer to the `NextBlock` variable is `NULL`, and the object pointed to by `current_entity` is deleted.

```cpp
class Dispose:Block
{
public:
    Dispose();
    ............
}
```

Listing 20: `Delay` class
void Dispose::process_entity(EntityList &entity_list,EventList &event_list, Entity *current_entity)
{
    ... ... ...
    current_entity->NextBlock = NULL;
    delete current_entity;
}

Listing 21: process_entity( ) method of the Release class

5.5.6 Decision Process Block

Decision process can have a single or multiple input connections. A Decision block can have more than one output connections. A Decision block is used in a process-driven model to implement one or more of the following functionalities:

- Entities are to be distributed proportionally to different points of the model, e.g. 10% to point1, 30% to point2...etc,

- Entities are to be distributed to different points of the model depending upon the true or false evaluation of a Boolean.

Due to the limitations in the proof of concept interface for the integrated $IE^2$ environment, only proportional allocation of entities in the Decision process block (Figure 37) is implemented. A generic Block object can be connected to multiple blocks, both at the input and output terminals. Since, a Decision block (listing 22) is inherited from the generic Block class; it can be also connected to multiple block objects at both input and output terminals.

Figure 37: Decision Process Block

Listing 22: Decision class
When an entity is processed at a block by the `process_entity()` method. The `NextBlock` information is updated as shown in listing 23. The variable `NextList[i]` is an array of pointers to the next block in the model. Entities use these variables to move around the simulation model as intended by the modelers.

```c++
current_entity->NextBlock = current_entity->NextBlock->NextList[i];  // Update the NextBlock variable
```

Listing 23: Entity with reference to next block

When the `Decision` process block is instantiated, the number of outlets and the probabilities associated with each outlet is provided by the modeler as shown in the listing 24.

```c++
for(int j=0;j<NumOutlets;j++) {
    cout<< "Enter the probability of the outlet: "<<j<<"\n";
    cin>>ProbChance;
    NWayChance.push_back(ProbChance);
}
```

Listing 24: Input for the Probabilities of the `Decision` Block Outlets

As in any block class or its sub-class, entities entering a `Decision` block call the method `process_entity()`. As discussed earlier in the section, one of the functionalities of the `Decision` block is to distribute the entities proportionally to different points in the model. This functionality is implemented using the logic shown in listing 25.
The Batch process block is a frequently used block in a process-driven model. The main functionality of the Batch process block is to batch a group of entities into one entity, either temporarily or permanently. If the entities are batched temporarily, then the entities can be separated elsewhere in the model by the Separate process block. The modelers opting for permanent batching of the entities are required to specify the attributes of the newly formed entity. Available options for the entity attributes are,

- Select the attributes of the first or last entity in the batch, or
- Specifying new sets of attributes.

A Batch process block can be very useful in modeling certain real world situations. The following are some of instances where a Batch process block can be used appropriately,

- In a semi conductor processing system, wafers are batched in some activities,
- In Automobile assembly, parts are assembled to make a vehicle
- In inventory systems, orders from different sources may be batched
- In office management, appointments of a particular day may be batched

The Batch process block is also defined as the sub-class of the generic Block class (listing 26). As the functionality of Batch is to group a set of entities from different points of a simulation model into a single entity, a Batch block has multiple input terminals and a single output terminal (Figure 38).

```
class Batch:public Block
{
   public:
      Batch();
   ...
};
```

Figure 38: Batch Process Block   Listing 26: Batch Class

The process_entity( ) method (listing 27) of the Batch process block stores the entities in a container, which is a Queue object. Every time an entity is stored or batched in the container, the size of the container is compared with batch size specified by the modeler. If the container size is less than the specified batch size, then the Batch process block waits for additional entities. Once the batch size is attained, a new entity is instantiated and its attributes are updated as per the specified option. Before inserting the entity into the entity list, the method checks for the batch type i.e. temporary or permanent. If the batch type is permanent, then the new entity is inserted into the entity list. Otherwise, the entities in the container are transferred to the new entity. As shown in listing 28, the entities in the container are transferred to the new entity using a method TransferBatch( ).
5.5.8 Separate Process Block

The Separate process block can be used in conjunction with Batch process block. The main functionalities of the Separate process block are as follows:

- Separate the temporarily batched entities, or
- Duplicate an entity into multiple entities

The Separate process block has a single input terminal and multiple output terminals as shown in Figure 39. As in previous cases, the Separate block is also defined as a sub-class of a generic Block class (listing 29).
The `process_entity()` method of the `Separate` process block updates the `NextBlock` variable of the entity and calls the method for separating or duplicating entities as specified by the modeler (listing 30).

```
void Separate::process_entity(EntityList &entity_list, EventList &event_list, Entity* current_entity, double SimEndTime)
{
    current_entity->NextBlock = current_entity->NextBlock->NextList[0]; /*Update the NextBlock Variable*/
    if(!Duplicate) /*Condition for checking options i.e. Duplicate or Separate*/
        SeparateEntity(current_entity, entity_list);
    else
        DuplicateEntity(current_entity, entity_list); /*Method to Duplicate the Entity*/
}
```

Listing 30: Implementation of the `Separate` Process block

The `SeparateEntity()` method separates the batched entities into individual entities. As shown in listing 31, the entities from the container in the batched entity are removed. ‘time’, and ‘Nextblock’ variables are updated and the entities are inserted into the entity list for further processing at subsequent blocks.

The `DuplicateEntity()` method creates multiple copies of the entity. As shown in listing 32, new entities are instantiated and its attributes are updated using the attributes of the original entity. The newly created entities are inserted into the entity list for further processing at subsequent blocks.
The **SubModel** process block provides hierarchical modeling capabilities to the modelers. The **SubModel** class is defined as a sub class of a generic **Block** class (listing 33). The main advantage of defining it as a sub class of **Block** class is to effectively utilize the existing framework and ease of implementation. This will enable the modelers to place a **SubModel** process block anywhere in the process-driven model. In the class definition of **SubModel**, a new object ‘**SecondModel**’ of **Model** class is instantiated. This object ‘**SecondModel**’ is used for creating a new sub-model.

```cpp
class SubModel : public Block
{
  public:
    SubModel();
    .......
    Model SecondModel;
  .......
};
```

Listing 33: **SubModel** Class
The ‘SecondModel’ object calls the read( ) method of the Model class in the constructor of the SubModel (listing 34). The read( ) method takes the input of the modeler for the sub model as he/she interacts with the user interface. The constructor keeps track of the pointers, and size of the sub-model. In the process_entity( ) method, the entities entering the SubModel block are updated with sub-model information and inserted into the entity list (listing 35).

```c
SubModel::SubModel(EntityList &entity_list, EventList &event_list)
{
    SecondModel.read(entity_list.event_list);                /*Reading the Sub-Model*/
    FirstBlock = SecondModel.BlockList[0];                   /*Updating 'FirstBlock with initial pointer of the Sub-Model*/
    ModelSize = SecondModel.BlockList.size()-1;              /* Upating 'ModelSize */
}
```

Listing 34: SubModel Constructor

```c
void SubModel::process_entity(EntityList &entity_list, EventList &event_list, Entity *current_entity, double SimEndTime)
{
    current_entity->NextBlock = FirstBlock;                /*Updationg the Entity with First Block of the Sub-Model*/
    entity_list.InsertInList(current_entity);               /*Reinserting the Entity into the Entity List*/
    LastBlock = this->NextList[0];                        /* Updating ‘LastBlock’ with the Last Block of the Sub-Model*/
    for(int i = 0; i < ModelSize; i++)
    {
        SecondModel.BlockList[i]->NextList[0] = LastBlock;  /*Connecting the Sub-Model with the Main Model*/
    }
}
```

Listing 35: process_entity Method

The main focus of this chapter was the description of the architecture of the \(IE^2\) framework and various process-driven blocks available in the \(IE^2\) simulation framework. The \(IE^2\) framework provides a set of basic process blocks namely, Create, Seize, Delay, Release, Dispose, Decision, Batch, Separate for building process-driven models. In addition to that, a sub-model block is also provided for building multiple layers or hierarchical simulation models. These process components were built on the existing
simulation framework namely, queues, resources, blocks, etc. The purpose of building the process-driven model components in an integrated framework is to augment the modeling capabilities of a modeler. In this chapter, the architecture of the simulation framework and process components is explained in detail through class diagrams and code snippets. The next chapter explains the interface and testing of the $IE^2$ simulation framework.
CHAPTER 6  PROOF OF CONCEPT: USER INTERFACE AND TESTING

In order to demonstrate the concept of the $IE^2$ framework, and refine its ideas, a software implementation of the simulation engine and model components was necessary. Therefore, skills in software and user interface development also supported this research. This chapter describes some of the technical and software-oriented details that must be considered beyond the conceptual structure of the framework. The philosophy of the framework and software development effort has been to avoid compromises in model-building power because of barriers encountered on the software development effort. We believe that some of the limitations in current simulation languages come from accepting changes in the model-building framework to facilitate a quicker software development cycle. We have closely scrutinized each decision in the development of the object-oriented framework in the software to preserve the essential structure of the $IE^2$ philosophy and modeling framework. Major priorities in this dissertation research can be summarized as follows:

- Simulation engine that supports both process- and event-driven models,
- Integrating the entity and the events in a model,
- Interface that realizes/demonstrates the $IE^2$ framework,

These challenges were addressed by making careful choices based on profound understanding of DES systems and how modelers use them. Some of the design challenges were intertwined with certain technical hurdles. Table 1 summarizes the technical hurdles involved in this dissertation research.
### Challenges

<table>
<thead>
<tr>
<th>a)</th>
<th>Simulation engine that supports both the process- and event –driven models</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Programming languages, design patterns</td>
</tr>
<tr>
<td>b)</td>
<td>Integrating entity and the event</td>
</tr>
<tr>
<td></td>
<td>Object Oriented Design</td>
</tr>
<tr>
<td>c)</td>
<td>Interface that realizes/demonstrates the $IE^2$ framework</td>
</tr>
<tr>
<td></td>
<td>Integrated Development Environment (IDE) that supports interface and database</td>
</tr>
</tbody>
</table>

Table 1: Technical Hurdles in the Dissertation Research

#### 6.1 Integrated Development Environment (IDE)

The $IE^2$ simulation framework required an integrated development environment (IDE) that can support its simulation engine, interface and the database. Microsoft .NET was considered to develop the $IE^2$ simulation software. Microsoft provides a software technology called .NET framework (www.microsoft.com/net) that provides a solution to common programming problems, and a set of tools for configuring and building applications. Programs written in the .NET framework use the Virtual Execution Machine (VEM), Common Language Runtime (CLR), to execute and manage the runtime requirements (Figure 40). The Base Class Library (BCL) of the .NET framework forms a set of pre-coded solutions for the large range of programming needs that include: interface, data access, database connectivity, web applications etc. Other important services are security, memory management, and exception handling. The CLR and the BCL together form the .NET framework (Mayo, 2001).
6.2 Visual Studio

Visual studio is the Integrated Development Environment (IDE) from Microsoft. It can be used to develop wide range of applications like, consoles, windows forms, web sites, web applications etc. Visual studio includes a code editor, designer, and debugger with a support for languages such as C++, C# and Visual Basic. Figure 41 shows the snapshot of the visual studio interface with windows for code editor, designer editor, solution explorer, tool box, properties etc. The simulation engine, which was developed in C++, was converted into the C# programming language. The primary reason to opt for C# is its widespread use in the developer community. Practically speaking, the popularity of C# among bloggers and discussion forums makes it easy for the developers to look for answers to technical questions.
6.3 Windows Forms (WinForms)

Windows forms are the Application Programming Interface (API), provided by the Microsoft .NET framework, for developing Graphical User Interface (GUI) applications. The interface for both process- and event-driven layers were developed using windows forms as shown in Figure 42. The basic functionality of the interface for both layers is based on the drag and drop of the simulation objects. The modelers should be able to drag and drop blocks and nodes from the menu panel and connect them together to form a simulation model.

![Figure 42: WinForms Interface for Process- and Event-driven layers](image)

DES software development has to deal with the drag and drop of objects when developing the interface. The drag and drop operation has three important mouse button event associated with it, DragEnter, DragOver, and DragDrop (.NET framework developer center).

**DragEnter** – The DragEnter event occurs when the mouse cursor is dragged over the control during the drag and drop operation.

**DragOver** – The DragOver event is raised when the mouse cursor moves within the control bounds during the drag and drop operation.
**DragDrop** – The *DragDrop* event occurs when the drag and drop operation is completed.

### 6.4 ADO .NET (ActiveX Data Objects)

ADO .NET (.NET framework developer center) is a software component provided by Microsoft in the .NET framework. It enables programmers to access data and data services. In the \(IE^2\) simulation software, ADO .NET was used to access and store the data in the relational database (SQL server). ADO .NET primarily consists of two parts, namely data providers, and data sets.

**Data Providers** – Data provider objects provide access to the data source, in this case Microsoft SQL Server. A common set of utility classes available in the data provider are: *Connection, Command, Parameter, DataAdapter*, and *DataReader*.

**Data Sets** – Data set objects represent a group of classes that describe the memory resident data. A data set object provides a consistent relational programming model of the data irrespective of the source. A common set of utility classes available in the data set are: *DataTable, DataRelation*, and *Constraint*.

The ADO .NET technology was used to interact with the data source for the \(IE^2\) simulation framework. The interface of the framework is wired with the database through the ADO .NET APIs. Table 4 shows the different tables that support the database needs of the framework. These tables are populated with data from the modelers through the interface.
Table 2: Tables in the SQL Server Database

<table>
<thead>
<tr>
<th>Database</th>
<th>EntityEvent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AssignBlock</td>
</tr>
<tr>
<td></td>
<td>CreateBlock</td>
</tr>
<tr>
<td></td>
<td>DelayBlock</td>
</tr>
<tr>
<td></td>
<td>SeizeBlock</td>
</tr>
<tr>
<td></td>
<td>ReleaseBlock</td>
</tr>
<tr>
<td></td>
<td>DisposeBlock</td>
</tr>
<tr>
<td></td>
<td>EventGraphBlock</td>
</tr>
<tr>
<td></td>
<td>EventNode</td>
</tr>
<tr>
<td></td>
<td>QueueEntities</td>
</tr>
<tr>
<td></td>
<td>QueueTable</td>
</tr>
<tr>
<td></td>
<td>ResourceTable</td>
</tr>
<tr>
<td></td>
<td>ScheduleArcs</td>
</tr>
<tr>
<td></td>
<td>EntityResults</td>
</tr>
<tr>
<td></td>
<td>EventResults</td>
</tr>
</tbody>
</table>

6.5 Interface

Figure 43 shows the interface of the working proof of concept of the $IE^2$ simulation framework. The important components of the interface are a process block menu panel, work space, menu strip, animation, and variables display. The process block menu panel consists of commonly used process block i.e. Create, Delay, Seize, Release, Dispose, and Assign. It also contains the new block, EventGraph. These blocks can be dragged and dropped onto the work space, just like any other commercially available DES software. The menu strip provides tools for the modelers to 1) connect any two process blocks, 2) disconnect any two process blocks, 3) provide information about the resources and queues, 4) enter data for the process blocks, 5) run the simulation, and 6) return to default cursor. The animation part of the interface displays the animation of the machine or server and its interaction with the entities. The variables display portion of the interface displays the values of the important variables as the simulation is running.
Figure 43: User Interface for the $IE^2$ Simulation Framework

The interface for the event-driven model is displayed when the modeler clicks the $EventGraph$ for data entry. Figure 44 shows the interface for the event-driven layer in the $IE^2$ simulation framework.

Figure 44: Interface for the Event-Driven layer of the $IE^2$ Simulation Framework
The important portions of the interface to the \( IE^2 \) event-driven layer are the event node panel, menu strip, \( \text{Enter} \), and \( \text{Exit} \) nodes (Figure 44). The event node panel consists of three types of nodes: regular node, initialization node, and initialization node with entity tagged to it. The menu strip has buttons to 1) provide information about resources and queues, 2) enter data into the event nodes, 3) create a scheduling arc between two nodes, 4) create a scheduling arc with a tagged entity between two nodes, 5) return to the default cursor, and 6) hide or minimize the \textit{EventGraph} window. The following are the menu strip buttons used in the interface and their functionalities:

<table>
<thead>
<tr>
<th>Button</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Connecting two process blocks. Mouse arrow will turn into a cross hair ‘+’ shape.</td>
<td></td>
</tr>
<tr>
<td>- Disconnects two process blocks.</td>
<td></td>
</tr>
<tr>
<td>- Return the mouse cursor to default icon.</td>
<td></td>
</tr>
<tr>
<td>- \textit{Schedule Arc}, to connect two event nodes</td>
<td></td>
</tr>
<tr>
<td>- \textit{Schedule Arc with tagged entity}, to connect two event nodes with an entity attached to it.</td>
<td></td>
</tr>
<tr>
<td>- \textit{Resource &amp; Queue} button. Enter information about the resources and queues in the simulation model.</td>
<td></td>
</tr>
<tr>
<td>- (\textit{Data Entry Mode}) Enter information for different blocks of the simulation model. Clicking on the \textit{EventGraph} block in the data entry mode will open up the event-driven layer.</td>
<td></td>
</tr>
<tr>
<td>- Run the simulation with a dialog prompt for the end time of the simulation.</td>
<td></td>
</tr>
</tbody>
</table>
6.6 $IE^2$ Simulation Framework: Testing

The $IE^2$ simulation framework has to be tested against current modeling frameworks, to measure the benefits of the framework as standard simulation software. An experiment was conducted to test the features of the $IE^2$ software vis-à-vis pure process-driven models. The most relevant DES software features considered in the experiment are drawn from Banks (1996). The objectives set forth for the current research are considered while forming the hypothesis of this experiment. The subjects for the experiment are students with DES background. The test results showed improved average performance using the $IE^2$ simulation framework versus the pure process-driven models.

Banks (1996), Table 3 & Table 4, discusses the features that are relevant when selecting simulation software. In summary Banks stresses that modelers should not focus on a single issue, such as ease of use. Various factors ranging from accuracy and level of detail obtainable to vendor support and documentation should be considered. The factors mentioned by Banks (1996) to select simulation software are summarized in the following Table 3 and Table 4. The present research focuses only on certain critical aspects that are unique to the $IE^2$ model and reflect its potential improvements over existing modeling frameworks.
<table>
<thead>
<tr>
<th>Model-building Features</th>
<th>Runtime Environment</th>
<th>Animation and Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling Worldview</td>
<td>Execution Speed</td>
<td>Type of Animation</td>
</tr>
<tr>
<td>Input-data Analysis Capability</td>
<td>Model Size</td>
<td>Import Drawing and Object Files</td>
</tr>
<tr>
<td>Graphical Model-Building</td>
<td>Interactive Debugger</td>
<td>Dimension</td>
</tr>
<tr>
<td>Conditional Routing</td>
<td>Model Status and Statistics</td>
<td>Movement i.e. Motion of Entities or Status</td>
</tr>
<tr>
<td>Simulation Programming</td>
<td>Runtime License</td>
<td>Quality of Motion</td>
</tr>
<tr>
<td>Syntax</td>
<td></td>
<td>Libraries of Common Objects</td>
</tr>
<tr>
<td>Input Flexibility</td>
<td></td>
<td>Navigation</td>
</tr>
<tr>
<td>Modeling Conciseness</td>
<td></td>
<td>Views</td>
</tr>
<tr>
<td>Randomness</td>
<td></td>
<td>Display Step</td>
</tr>
<tr>
<td>Specialized Components and templates</td>
<td></td>
<td>Selectable Objects</td>
</tr>
<tr>
<td>User-built Custom Objects</td>
<td></td>
<td>Hardware Requirements</td>
</tr>
<tr>
<td>Continuous Flow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interface with General-Programming Language</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Simulation Software Features I (Banks, 1996)

<table>
<thead>
<tr>
<th>Output Features</th>
<th>Vendor Support and Product Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario Manager</td>
<td>Training</td>
</tr>
<tr>
<td>Run Manager</td>
<td>Documentation</td>
</tr>
<tr>
<td>Warm up capability</td>
<td>Help System</td>
</tr>
<tr>
<td>Independent Replications</td>
<td>Tutorials</td>
</tr>
<tr>
<td>Optimization</td>
<td>Support</td>
</tr>
<tr>
<td>Standardized Reports</td>
<td>Upgrades, Maintenance</td>
</tr>
<tr>
<td>Customized Reports</td>
<td>Track Record</td>
</tr>
<tr>
<td>Statistical Analysis</td>
<td></td>
</tr>
<tr>
<td>Business Graphics</td>
<td></td>
</tr>
<tr>
<td>Costing Module</td>
<td></td>
</tr>
<tr>
<td>File Export</td>
<td></td>
</tr>
<tr>
<td>Database Maintenance</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Simulation Software Features II (Banks, 1996)
6.6.1 Experiment

An experiment was conducted to assess the achievement of the objectives set forth at the onset of the research. The main objectives of this research, as mentioned earlier, are to

- Mitigate the complexity of large models through a process orientation, while retaining the control over the attributes, variables and the logic through event orientation.
- Develop a simulation tool that effectively preserves both the process and event levels that will support the discrete event simulation (DES) education perspective.

In order to test the $IE^2$ simulation environment, we posed a formal hypothesis. Investigation of this hypothesis helped gauge the effectiveness of the proof of concept of the $IE^2$ simulation environment. The formal hypothesis for testing the proof of concept simulation environment for effectiveness:

**Hypothesis:** $IE^2$ model provides more effective support for the modeler over either of the conventional process-driven and event-driven models.

6.6.1.1 Subjects

The experiment was conducted using students with some discrete event simulation (DES) background. Students who have taken courses such as, HFE 671 (Systems Performance Modeling) and/or HFE 735 (Advanced Systems Modeling) were eligible for the experiment.

6.6.1.2 Apparatus

HP dv2416us, Pentium IV processor with 14.1” monitor along with a keyboard and a mouse was used for the experiment.
6.6.1.3 Procedure

Each subject was briefed on the background and details of the experiment. The briefing included the written description of the simulation problem (Appendix A10) to be solved. The subject was then allowed half-an-hour to build the simulation model using a pure process-driven approach and then another half hour to build the simulation model using the IE\(^2\) simulation environment. The subjects were advised to utilize the time allocated to them by appropriately planning the simulation model and also to keep in view the components available to them in the two different simulation environments. A post-test questionnaire (Appendix A11) was given to the subjects to assess the simulation framework and the user experience during the experiment. There was no randomization involved in the experiment.

6.6.1.4 Results Summary

A total of seven subjects volunteered for the experiment. Five of the seven subjects have experience in both the process- and event-driven models, one subject only in event-driven models and one subject only in process-driven models. The following questions about their simulation expertise were posed to the subjects:

- How do you rate your expertise in the area of discrete event simulation (DES)?
  
  (Novice) 0  1  2  3  4  5 (Expert)

- Do you have experience in process-driven models (i.e. Arena, Flexsim, Extend etc.)?
  
  (Novice) 0  1  2  3  4  5 (Expert)

- Do you have experience in event-driven models (Event Graphs)?
  
  (Novice) 0  1  2  3  4  5 (Expert)
Most of the subjects, as we expected, identified themselves as experts in the field of DES (Average = 3.57, Chart 1). Six of the seven subjects have experience in process-driven models (Average = 3.43, Chart 1). As shown in Chart 1, subject 6 has no prior experience with process-driven models. It would be interesting to note the responses of this subject given that the event-driven models are an integral part of the modeling process in the $IE^2$ simulation framework. Subject 7 has little or no experience with event-driven models. In this case, Subject 7 was given a brief overview of event-driven models.

The next set of questions in the questionnaire focused on the subjects’ comprehension of the simulation problem and the entity-event interaction in the $IE^2$ environment. The simulation problem in the experiment asks the subjects to model non-preemptive failures using first only process-driven components and then the full $IE^2$ simulation environment. The need to model resource failures should be familiar to experienced simulation modelers. To build the process-driven model, subjects were made aware of the process components available (i.e. create, delay, seize, assign, etc.) in the $IE^2$ simulation

Chart 1: Expertise of the subjects in DES, Process- and Event-driven models

![Chart showing expertise levels of subjects in DES, Process- and Event-driven models](chart1.png)
framework. The principal investigator worked with the subjects to polish up the solutions of the problem in order to finish the experiment within the time constraint.

The most common approach used by the subjects to model non-preemptive failures, in the process-driven environment was to consider the failures as entities. Most of the subjects were successful in modeling the non-preemptive failures and their solution had a basic structure similar to the process-driven model shown in Figure 45. The process-driven model shown in Figure 45 has two Create blocks, one to generate jobs/customers and one to generate “failure entities”. The priority of the job/customer entity is set to normal and the failure entity to highest. Priority determines the ranking within the queue. Jobs/customers and failures have different inter arrival time distributions. They are assigned processing and repair times respectively in the Assign blocks. The job/customer and failure entities are then routed through Seize, Delay, Release, and Dispose blocks. Since the problem is defined as a single server in the problem description, the number of resource units available should be ‘1’. The Seize block seizes the resource for the entity (Job/Customer or failure) or queues the entity if the resource is busy. The Delay block delays the entity before scheduling the entity to arrive at the Release block. The length of the delay depends upon the processing or repair times of the entities. The Release block

Figure 45: Non-Preemptive Failures in the Process-Driven Environment
releases the resource, allowing the Seize block to assign a queued entity to the resource, taking into account the priorities of all queued entities. Because the priority of a queued failure entity is set to highest, if a failure entity is present in the queue, the Seize block assigns the failure entity to the resource. The details of the operation of Seize and Release blocks were described in Chapter 5.

The following questions were posed to the subject after the exercise to get information about the comprehension of the simulation problem given to them:

- Do you understand the problem given to you for modeling in two different simulation environments?

  (Not much) 0  1  2  3  4  5 (Very well)
The subjects were given orientation on the $IE^2$ simulation framework, especially on the event-driven components available within the process-driven environment. Most of the subjects (6 out of 7) chose to model the job/customer as entities at the process-level (Figure 46(a)). Single server and failure aspects of the simulation model were transferred to the event-driven level (Figure 46(b)). The subjects, who modeled jobs/customers as entities had to keep track of the entity flow at the event level through the entity-event arc (represented by red lines, Figure 46(b)). The following question was used to gather information about the subjects’ comprehension of the entity-event interaction.

- In the current experiment, how well do you understand the entity-event interaction in the $IE^2$ simulation framework?

  (Not much) 0  1  2  3  4  5 (Very well)

Chart 2 shows the comprehension of the subjects with respect to simulation problem and entity-event interaction. Almost all the subjects reported that they understood the simulation problem (non-preemptive failures) given in the experiment (Average = 4.29, Chart 2) and the subjects also reported that the concept of entity-event interaction was well understood with an average rating of 3.86 (Chart 2).

![Chart 2: Subjects’ Comprehension of the Simulation Problem and Entity-Event Interaction](image)
The next set of questions was related to the user experience of the subjects. These questions were open ended and intended to get general feedback about the usability of the proof of concept of the \( IE^2 \) simulation framework. On average, the user experience of the subjects for both the pure process-driven and \( IE^2 \) models scored 4.14 (Chart 3). The following questions are related to the user experience and chart 3 summarizes the responses of the subjects.

- How would you rate your experience while developing the process-driven model?
  
<table>
<thead>
<tr>
<th>(Bad) 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5 (Good)</th>
</tr>
</thead>
</table>

- How would you rate your experience while developing the \( IE^2 \) model?

<table>
<thead>
<tr>
<th>(Bad) 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5 (Good)</th>
</tr>
</thead>
</table>

![Chart 3: User/Development Experience of the Subjects](chart3.png)

The questionnaire also focused on questions for the subjects to extrapolate the level of effectiveness/elegance in the models (Process/\( IE^2 \)) with increasing complexity. These questions were aimed to identify whether the subjects realize the potential of event-driven models within a process-driven simulation environment. The experiment, however, considered the case of ambiguity in the following questions. In that case, subjects are
allowed to express their perspective on the importance or essence of the $IE^2$ simulation framework through comments.

- How would rate the level of effectiveness/elegance in the model, if asked to add some more complexity to the process-driven model?
  (Low) 0 1 2 3 4 5 (High)
  Why?_________________________

- How would rate the level of effectiveness/elegance in the model, If asked to add some more complexity to the $IE^2$ model?
  (Low) 0 1 2 3 4 5 (High)
  Why?_________________________

Chart 4: Effectiveness/Elegance of the Process and $IE^2$ Model with Increasing Complexity

Chart 4 summarizes the responses of the subjects on the questions of the effective/elegance of the process/$IE^2$ models. The average performance of the $IE^2$ models is better than the process-driven models (Process-driven models, Average = 3.43, $IE^2$ models, Average = 3.86). The results shown in chart 4 are discussed in detail in the discussion section. The last question of the questionnaire asked the subjects whether the
process-driven or $IE^2$ model is useful in verifying the simulation logic. As described in detail earlier, the $IE^2$ simulation framework supports the representation of entity flow at the event level using the entity-event arc (Red Lines) (Figure 46(b)). The following question from the questionnaire was intended to verify whether the entity-event arc increases the intuitiveness of the $IE^2$ models. As shown in chart 5, most of the subjects responded in favor of the $IE^2$ models (6 out of 7 subjects).

- Which simulation framework would you think will be help in verifying the simulation logic?

  __Process-driven Model__

  __$IE^2$ model__

![Chart 5: Verification of the Simulation Logic, Process Vs $IE^2$ Models](image)

6.7 Discussion

Six of the seven subjects have considerable experience in process-driven or event-driven models. As shown in Chart 1, Subject 6 had no prior experience with process-driven models and Subject 7 had little or no experience with event-driven models. As noted earlier, the responses from these two subjects contributed to some of the interesting observations in the overall data collected. Non-preemptive failures seem to be an
appropriate problem for a limited experiment to test the prowess of $IE^2$ models vis-à-vis process-driven models. Through the use of a simple but interesting problem for the experiment we wanted to ensure that the subjects understood the simulation problem in the experiment. Almost all the subjects understood the simulation problem with an average rating of 4.29 (Chart 2). The next important aspect for a rationale experiment is the comprehension of the entity-event interaction (Average = 3.86, Chart 2) in an $IE^2$ simulation framework. Subjects (6 & 7), having only process or event-driven model experience (but not both), found little relevance for the entity-event interaction in their understanding of the DES models.

The experiment did not attempt an elaborate usability test of the $IE^2$ framework interface. The subjects’ responses on the user experience, on average, were equal for both the process and $IE^2$ models (Average = 4.14, Chart 3). However, the users’ comments at the end of the questionnaire were quite useful for further improvement of the framework and software. The following are some of the comments provided by the subjects regarding the usability of the $IE^2$ models.

“I would try to make more visible elements so user can easily see changes in variables and delay times. If too much is hidden then its starts to look purely process driven aka flowchartish” – Subject 3

“The $IE^2$ model provides easier visualization of logic, whereas models in Arena and Extend require knowledge of how each block operates and how the block settings change what block does behind the visual interface” - Subject 5

“The user interface needs a lot of improvement” – Subject 6
Subject 3’s comment to have more visual elements seems more reasonable i.e. to display the change in variables and delay times. The explicit representation of variable changes and the entity flow at the event level will definitely make the \( IE^2 \) models more intuitive. Subject 5 observed that the \( IE^2 \) model provides easier visualization of logic over the traditional process-driven models like Arena and Extend. These views echoed the objectives and accomplishments of the current research. However, Subject 6 suggested that the user interface needs “a lot of improvement”. Subject 6 had no prior experience in process-driven models (Chart 1) and observed entity-event interaction less relevant (Chart 2) to the DES models.

The \( IE^2 \) simulation proof of concept requires significant improvement to make it more user-friendly, including a significant effort to design a complete user interface. The current software interface served the purpose of a proof of concept, rather than a fully functional proof of concept.

As discussed earlier, the average rating of the \( IE^2 \) models is better than the process-driven models (Process-driven models - Average = 3.43, \( IE^2 \) models - Average = 3.86) in modeling complex simulation problems. From the charts 1-4, it can be observed that Subject 7 has given the lowest rating. However, the remarks made by the Subject 7 captured some of the important features of the \( IE^2 \) models.

“In some cases Arena does a good job and sometimes it doesn’t. When we need to manipulate some variables inside the model, \( IE^2 \) would be a good help because you have control on the variables” – Subject 7

131
Some of the remarks made by the other subjects, mentioned below, reflect the objectives of the current research and its relevance in the context of their simulation models. The hypothesis of this experiment was that the $IE^2$ model provides more effectiveness over either of the conventional process-driven and event driven models. In the current context of the experiment, we have observed some limited evidence that $IE^2$ models provide more effectiveness over the process- and event-driven models.

“Process-driven is easy if person has less of a simulation background. $IE^2$ is suggestible only for experts in the area. Not for everyday simulation, but only for research” – Subject 1

“Combination of process driven and event driven model is very useful in understanding the simulation logic. It is also very helpful for users who prefer to use different types of models when modeling different scenarios. The interaction between the process and event driven model seemed very smooth and interacting in both seemed very easy to understand” – Subject 4

6.5 Summary

The development of the $IE^2$ simulation framework required considerable software engineering details. This chapter described the software techniques and technologies to solve the technical challenges involved in the development process. The important technologies that are described in this chapter are, software patterns (Mediator), Visual Studio (Integrated Development Environment, IDE), C# (Programming Language), Windows Forms (User Interface), ADO.NET (Database Support). The $IE^2$ simulation framework is proof of concept DES software. It will take another development cycle to develop the $IE^2$ proof of concept into commercial software.
This chapter also discussed the testing of the $IE^2$ simulation framework. An experiment was conducted to establish the efficacy of the $IE^2$ simulation framework proof of concept. The experiment setup examined the performance of the framework with respect to the user experience, effectiveness, elegance, and verification of the simulation logic of the $IE^2$ models. The results indicate that the average performance of the $IE^2$ models is better than that of pure process-driven models. The feedback obtained from the subjects was helpful to gain insights into the understanding of the $IE^2$ models. The final chapter summarizes the contributions of the research, conclusion, and future research.
CHAPTER 7 CONCLUSIONS, CONTRIBUTIONS AND FUTURE RESEARCH

Based on a simulation framework that combines event-driven and process driven approaches, we have defined a model building environment that embodies a natural and effective interface between entities and events. The explicit availability of entities and their attributes in an event-driven model helps reduces the abstraction required for simulation users to build event-driven models. The effective usage of process-driven model components on top of a consistent event-driven mindset can further enhance the capabilities of an event-driven model and vice versa. The process driven approach to simulation modeling is important in the overall context of a simulation project, especially in allowing a fast and intuitive model development environment. The event-driven approach is equally important when detailed control over non-typical system logic is required.

7.1 Conclusion and Contributions

In this research, we proposed an integrated simulation framework that combines the process- and event-driven models. These two different models i.e. process- and event-driven, are integrated in the simulation framework as hierarchical layers. The simulation framework is designed to handle the processing of entities and events. A formal relationship among process-driven models, event-driven models and resident entities, like resources and queues, has been established. This formalism enables the DES models in the integrated simulation framework to be more accurate and elegant by using both process- and event-driven components in a logically consistent way. In an effort to build models that accurately represent real-world structure, this ability is critical.
An important feature of the $IE^2$ model is that it explicitly models the entity flow taking place at the event level. This feature augments the capabilities of a simulation modeler by making some aspects of process logic available in the event layer and vice versa. However, the two layers are different when they interact with resident entities like resources, queues etc. The $IE^2$ framework handles queues and resources for the process- and event-driven models through an intermediate layer. This flexibility provided by the intermediate layer, reduces the level of modeling abstraction at the event graph level, and leads to a more seamless, $IE^2$ model that spans the two levels. However, the entities change their states depending upon the layer in which it resides.

The $IE^2$ framework provides a set of basic process blocks namely, Create, Seize, Delay, Release, Dispose, Decision, Batch, Separate for building process-driven models. In addition to that, a sub-model block is also provided for building multiple layers or hierarchical simulation models. These process components were built on the existing simulation framework namely, queues, resources, blocks, etc. The purpose of building the process-driven model components, in an integrated framework, is to augment the modeling capabilities of a modeler.

In order to illustrate both the process- and event-driven model components in an $IE^2$ model, a supply chain management system has been studied as an example. *Evaluating the Impact of Retailer Gaming and Supplier Capacity Allocation on Supply Chain Costs* (Vutukuru, 2006), focuses on a supplier-retailer supply chain, consisting of a single supplier and three retailers. The model considers how partial information sharing has an impact on the supply chain costs with different allocation mechanisms on the supplier
side vis-à-vis gaming behaviors on the retailer side. The model was originally implemented in *Arena*.

A comprehensive $IE^2$ model for retailer demand and cost sub-models is presented in Chapter 4, to allow direct comparisons between a pure process model with VBA programming and an integrated entity/event model. The example demonstrates how the logic of allocation and entity flow can be more elegantly represented with the $IE^2$ model. It also demonstrates the well structured interface defined for entities that interact with the process layer, transitioning to the event layer, and then returning to the process layer.

Comparing the standard *Arena* based process-driven model with an $IE^2$ model, the $IE^2$ model embodies the function of event parameterization through the entity attributes. Entity attributes are a natural construct for modelers familiar with process-based logic. The advantage of using the entity attributes in the $IE^2$ model is that similar model sub-graphs can be combined together as a generic sub-graph distinguished by the attribute values of entities flowing through them. At the event level, entities are handled as objects in a way that is analogous to their treatment in the process models. The attributes of an entity are defined by the modeler, enabling the flexibility and explicit handling of entities at the event level. Instead of passing information as event parameters to other nodes as in a programming language, the $IE^2$ model defines them explicitly as attributes of entities that are associated with events as they are scheduled. This entity passing through the events in the event graph, gives the intuitive feel of the process-driven model to the modelers. This modeling of entity flow through the event graph enhances the appeal of event graphs to modelers with a process perspective, while retaining the power and flexibility of the event logic. At the process level, the modelers’ ability to model
complex logic is enhanced without resorting to programming languages in a simulation model. One of the major objectives of the $IE^2$ model is to diminish the gap between real world processes and their representation in the simulation environment while not limiting itself to the graphical representation as in most commercially available process-driven simulation tools.

The user interface for this integrated simulation environment was designed with appropriate attention to both modeling ease as well as effective access to the process- and event-driven capabilities of the simulation engine. A rich, proof-of-concept graphical user interface was developed for the integrated framework which allows the user to graphically navigate between the different levels of the model. The user interface has graphic elements which represent the model logic and structure. This model logic representation serves the objective by providing cognizable simulation objects in the graphical user interface and supporting both novice and sophisticated users through a natural and intuitive model hierarchy. To hierarchically embed event-driven models within a process-driven model, the $IE^2$ model provides an ‘EventGraph’ block. The ‘EventGraph’ block behaves as a regular process block as well as a workspace for containing an event-driven model. The design of the EventGraph block embodies the $IE^2$ approach integrating the event-driven and process driven modeling layers.

Important earlier works, in the area of integrating process- and event-driven models, include the LEGO (Listener Event Graph Objects) by Buss et al. (2002). Buss et al. attempted to give event graphs an outlook of process driven model by introducing LEGO. LEGO are fundamentally event graphs except that they are encapsulated as atomic components and these components communicate with each other through a listener
pattern. The following are the advantages of the $IE^2$ simulation environment over the Buss et al. (2002) LEGOs:

- $IE^2$ environment actually integrates the process- and event-driven models, instead of merely giving the appearance of process models to the event graphs.
- $IE^2$ environment supports the flow chart approach in the model representation. This would allow the modelers to build models that not only represent the model logic but also have model animation that directly maps to the real systems.
- $IE^2$ environment eliminates the necessity of parameter passing and also the usage any programming while modeling complex logic.
- $IE^2$ environment is simple in implementation and elegant in design (Kesaraju et al., 2007) reflecting a consistent application of a model building philosophy through its use of a Mediator software architecture.

As a part of the dissertation research, a limited experiment was conducted to assess the accomplishment of the objectives set forth at the onset of the research. Subjects in the experiment had experience in both the process- and event-driven models. The responses from the subjects contributed to some of the interesting observations in the overall data collected. The subjects with only process or only event-driven model experience found little relevance for the entity-event interaction within their understanding of the DES models. Overall, in the limited context, the average performance of the $IE^2$ models is better than the process-driven models.
7.2 Future Research

For future research, the $IE^2$ simulation framework can be tested against some of the important modeling issues that appear in discrete event modeling. In this research, $IE^2$ was used to model the failures (preemptive and non preemptive) template. *Blocking* would be another interesting modeling issue to be examined. *Blocking* in a tightly coupled system is a scenario in which the entities have to be allocated resources downstream before they can move on. Tightly coupled systems are systems with a limited space for parts buffering between workstations (Kelton et al., 2004). Figure 47 illustrates the *blocking* of the workstations.

```
Machine Blocked
Queue Capacity: 6
```

Figure 47: Illustration of the *Blocking* Scenario

“Overlapping the resources” is a popular technique to resolve the blocking issue. Figure 48 illustrates the overlapping resources in a process-driven model. It would be an interesting problem to be solved in the $IE^2$ simulation framework, with the help of the event-driven components. An elegant, unexpected solution to this modeling problem (and other) may emerge from the additional flexibility created by the layered, $IE^2$ approach. There are other classic DES modeling issues that can be investigated in this framework. Solutions to these problems can then be presented in detail rather than in the “black box” approach found in most of the commercially available software. This could aid in the education process for developing simulation modeling experts.
Another interesting direction would be to add object-oriented modeling approach as the upper level of the hierarchical $IE^2$ simulation framework. Object-oriented approach has important properties like inheritance, polymorphism, and encapsulation. As shown in Figure 49, the 3-tier $IE^2$ simulation framework would have event-driven, process-driven, and object-oriented layers. The $IE^2$ framework with the object-oriented feature would give the simulation modelers much broader scope to model the DES models.

**Figure 48: Overlapping Resources to Solve the *Blocking* Issue**

**Figure 49: 3-Tier $IE^2$ Simulation Framework**
8 REFERENCE


D. M. Morrice, D. T. Brunner, and J. J. Swain, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, 564-568.


Eds., S. E. Chick, P. J. Sanchez, D. M. Ferrin, D. J. Morrice, Lousiana, Dec. 7-10, pp. 201-209


9 APPENDICES

A 1: Node Class and Sub Classes INITNode, INITEntNode
A 2: Block Class and Sub Classes: Create, Delay, EventGraph, Dispose, Seize, and Release
A 3: Class Associations: Block Class
A 4: Class Associations: \textit{Entity} and \textit{EntityList} Classes

A 5: Class Associations: \textit{Event} and \textit{EventList} Classes

A 6: Class Associations: \textit{ScheduleArc} Class
A 7: Class Associations: Simulation Class
A 8: Class Associations: Model Class
### A9: Class Associations: Node Class

#### ScheduleArc
- StartNode : Node
- EndNode : Node
- BlockReference : Block
- EntityManagement : bool
- DelayTime : double
- ConditionOption : int
- ScheduleDelay()
- CheckCondition() : bool
- ResourceAvailability() : bool
- IsQueueOccupied() : bool
- IsFailureScheduled() : bool
- EntityManager()

#### NodeList
- NodeName : string
- BlockReference : Block
- CurrentEntity : Entity
- NodePB : PictureBox
- NodeLB
- NodePtX1 : int
- NodePtX2 : int
- NodePtY1 : int
- NodePtY2 : int
- QueueChoice : int
- ResourceChoice : int
- FailureChoice : int
- EntityManager()
- EntityEntry()
- QueueHandler()
- ResourceHandler()
- Failure.Handler()
- IncrementFailure()
- DecrementFailure()
- IncrementQueue()
- DecrementQueue()
- IncrementResource()
- DecrementResource()
- ResourceDown()
- ResourceUp()
- UpdateState()
- ScheduleEvent()
- ProcessEvent()
- PopulateNodeData()

#### Block
- BlockName : string
- BlockID : int
- SelRes : Resource
- SelQue : Queue
- NextList : Block
- ProcessEntity()
- PopulateBlockData()

#### Event
- Time : double
- NextBlock : Block
- CurrentNode
- TaggedEntity : Entity

#### EntityList
- CurrentEntity
- EntityList : Entity
- IsEntityListOccupied() : bool
- InsertInList()
- RemoveFromList() : Entity

#### Entity
- Time : double
- EntityID : int
- CreateBlockID : int
- NextBlock : Block
A 10: DESCRIPTION OF THE SIMULATION PROBLEM

**Failure of a Resource**

Resources can be in one of the three states i.e. *idle*, *busy*, and *breakdown*. When a resource is not processing an entity, then the resource is said to be in the *idle* state. On the other hand, if the resource is busy with an entity, then the resource is said to be in the *busy* state. Failures are the random events that cause the resource or servers to become unavailable. When a failure occurs then the resource state will be updated to the *breakdown* state. There are certain activities such as parts replacement, cleaning and tool adjustments that require the resource to stop processing entities. These events or processes may not be viewed as failures but can change or update the state of the resource to *breakdown*.

Whenever a failure of a resource occurs and it is *idle*, then the state of the resource changes from *idle* to *breakdown*. If the resource is *busy* in processing an entity and a failure has occurred then the resource can respond in two ways, *preemptive* and *non-preemptive*. In the *preemptive* case, the resource preempts the repair process by terminating the processing of the working entity and immediately moving forward with a repair process. The state of resource is then changed or updated to the *breakdown* state. The remaining processing of the entity is started once the machine returns to its working state. Alternatively, in the *non-preemptive* case, the resource starts the repair or maintenance only after finishing the processing of the current entity. Then the state of the resource is changed to the *breakdown* state.

In this experiment, the subjects are required to model *preemptive failures* in both the process-driven and *IE²* simulation environment. The subjects are allowed half-an-hour for
each task to develop the model in the two different simulation environments. The subject are allowed to refer to any study material i.e. books, papers, etc. and clarify any questions regarding user interface of the $IE^2$ simulation software.
A 11: POST-TEST QUESTIONNAIRE

- How do you rate your expertise in the area of discrete event simulation?
  (Novice) 0  1  2  3  4  5 (Expert)

- Do you have experience in process-driven models (i.e. Arena, Flexsim, Extend etc.)?
  (Novice) 0  1  2  3  4  5 (Expert)

- Do you have experience in event-driven models (Event Graphs)?
  (Novice) 0  1  2  3  4  5 (Expert)

- Do you understand the problem given to you for modeling in two different simulation environments?
  (Not much) 0  1  2  3  4  5 (Very well)

- In the current experiment, how well do you understand the entity-event interaction in the \( IE^2 \) simulation framework?
  (Not much) 0  1  2  3  4  5 (Very well)

- How would you rate your experience while developing the model a process-driven model?
  (Bad) 0  1  2  3  4  5 (Good)

- How would you rate your experience while developing the model an \( IE^2 \) model?
  (Bad) 0  1  2  3  4  5 (Good)

- How would rate the level of effectiveness/elegance in the model, if asked to add some more complexity to the process-driven model?
  (Low) 0  1  2  3  4  5 (High)

Why?______________________________
• How would rate the level of effectiveness/elegance in the model, If asked to add some more complexity to the $IE^2$ model?

(Low) 0  1  2  3  4  5  (High)

Why?__________________________

• Which simulation framework would you think will be help in verifying the simulation logic?

__Process-driven Model __ $IE^2$ model
## A 12: DATABASE TABLES

1. **AssignBlock**

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ExpressionName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 Expression</td>
<td>real</td>
</tr>
<tr>
<td>3 BlockID</td>
<td>real</td>
</tr>
<tr>
<td>4 BlockName</td>
<td>varchar</td>
</tr>
</tbody>
</table>

2. **CreateBlock**

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BlockName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 InterArrivalTime</td>
<td>real</td>
</tr>
<tr>
<td>3 FirstEntityTime</td>
<td>real</td>
</tr>
<tr>
<td>4 BlockID</td>
<td>real</td>
</tr>
<tr>
<td>5 EntityPriority</td>
<td>real</td>
</tr>
</tbody>
</table>

3. **DelayBlock**

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 DelayName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 DelayTime</td>
<td>real</td>
</tr>
<tr>
<td>3 BlockID</td>
<td>real</td>
</tr>
</tbody>
</table>

4. **DisposeBlock**

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 DisposeName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 BlockID</td>
<td>real</td>
</tr>
</tbody>
</table>

5. **EntityResults**

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CreateBlockID</td>
<td>real</td>
</tr>
<tr>
<td>2 EntityID</td>
<td>real</td>
</tr>
<tr>
<td>3 EntityBlockName</td>
<td>varchar</td>
</tr>
<tr>
<td></td>
<td>EntityBlockTimeIn</td>
</tr>
<tr>
<td>---</td>
<td>------------------</td>
</tr>
<tr>
<td>5</td>
<td>EntityBlockTimeOut</td>
</tr>
</tbody>
</table>

6. **EventGraphBlock**

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 EventGraphName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 ResourceName</td>
<td>varchar</td>
</tr>
<tr>
<td>3 Availability</td>
<td>real</td>
</tr>
<tr>
<td>4 QueueName</td>
<td>varchar</td>
</tr>
<tr>
<td>5 QueueID</td>
<td>real</td>
</tr>
<tr>
<td>6 ResourceID</td>
<td>real</td>
</tr>
<tr>
<td>7 BlockID</td>
<td>real</td>
</tr>
</tbody>
</table>

7. **EventNode**

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 NodeName</td>
<td>varchar</td>
</tr>
</tbody>
</table>

8. **EventResults**

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 NodeName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 NodeTime</td>
<td>real</td>
</tr>
</tbody>
</table>

9. **Model**

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BlockName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 BlockNum</td>
<td>real</td>
</tr>
</tbody>
</table>

10. **QueueEntities**
<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 EntityID</td>
<td>real</td>
</tr>
<tr>
<td>2 EntityBlockTimeIn</td>
<td>real</td>
</tr>
<tr>
<td>3 CreateBlockID</td>
<td>real</td>
</tr>
<tr>
<td>4 EntityBlockName</td>
<td>varchar</td>
</tr>
</tbody>
</table>

11. *QueueTable*

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 QueueName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 QueueID</td>
<td>real</td>
</tr>
</tbody>
</table>

12. *ReleaseBlock*

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ReleaseName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 ResourceName</td>
<td>varchar</td>
</tr>
<tr>
<td>3 Availability</td>
<td>real</td>
</tr>
<tr>
<td>4 QueueName</td>
<td>varchar</td>
</tr>
<tr>
<td>5 QueueID</td>
<td>real</td>
</tr>
<tr>
<td>6 ResourceID</td>
<td>real</td>
</tr>
<tr>
<td>7 BlockID</td>
<td>real</td>
</tr>
</tbody>
</table>

13. *ResourceTable*

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ResourceName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 Availability</td>
<td>real</td>
</tr>
<tr>
<td>3 ResourceID</td>
<td>real</td>
</tr>
</tbody>
</table>
14. ScheduleArcs

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 StartNode</td>
<td>varchar</td>
</tr>
<tr>
<td>2 EndNode</td>
<td>varchar</td>
</tr>
<tr>
<td>3 EventTime</td>
<td>real</td>
</tr>
<tr>
<td>4 EntityID</td>
<td>real</td>
</tr>
<tr>
<td>5 EntityCreateID</td>
<td>real</td>
</tr>
</tbody>
</table>

15. SeizeBlock

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SeizeName</td>
<td>varchar</td>
</tr>
<tr>
<td>2 ResourceName</td>
<td>varchar</td>
</tr>
<tr>
<td>3 Availability</td>
<td>real</td>
</tr>
<tr>
<td>4 QueueName</td>
<td>varchar</td>
</tr>
<tr>
<td>5 QueueID</td>
<td>real</td>
</tr>
<tr>
<td>6 ResourceID</td>
<td>real</td>
</tr>
<tr>
<td>7 BlockID</td>
<td>real</td>
</tr>
</tbody>
</table>