National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

vPARASOL
**A Visual Distributed/Parallel
Computer System Simulator**

by

Bo Xiao

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

July 8, 1993

© Copyright

1993, Bo Xiao

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-89814-3

Canada

Name

Bo Xiao

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

0984

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0291
Literature 0401
General 0294
Classical 0295
Comparative 0297
Medieval 0298
Modern 0316
African 0591
American 0305
Asian 0352
Canadian (English) 0355
Canadian (French) 0593
English 0311
Germanic 0312
Latin American 0315
Middle Eastern 0313
Romance 0314
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0770
Accounting 0454
Banking 0338
Management 0385
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce-Business 0510
Finance 0511
History 0358
Labor 0366
Theory 0351
Folklore 0578
Geography 0366
Gerontology 0351
History 0578
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Low 0398
Political Science 0615
General 0616
International Law and Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
General 0938
Criminology and Penology 0631
Demography 0628
Ethnic and Racial Studies 0629
Individual and Family Studies 0630
Industrial and Labor Relations 0700
Public and Social Welfare 0344
Social Structure and Development 0709
Theory and Methods 0999
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology 0306
Biology 0287
General 0308
Anatomy 0309
Biostatistics 0379
Botany 0329
Cell 0353
Ecology 0369
Entomology 0793
Genetics 0410
Limnology 0793
Microbiology 0307
Molecular 0317
Neuroscience 0416
Oceanography 0433
Physiology 0821
Radiation 0778
Veterinary Science 0472
Zoology 0786
Biophysics 0760
General 0425
Medical 0996

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palaurology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and Therapy 0381
Ophthalmology 0571
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0738
Inorganic 0490
Nuclear 0491
Organic 0494
Pharmaceutical 0495
Physical 0754
Polymer 0405
Radiation 0605
Mathematics 0986
Physics 0606
General 0608
Acoustics 0748
Astronomy and Astrophysics 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal 0790
System Science 0428
Geotechnology 0796
Operations Research 0795
Plastics Technology 0994
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,

vPARASOL

**A Visual Distributed/Parallel
Computer System Simulator**

submitted by

Bo Xiao

in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

August 4, 1993

Abstract

vPARASOL, or visual PARASOL, offers an enhanced version of the computer system simulator PARASOL, which aims at efficient modelling. It provides a graphical interface for parallel/distributed system model building through graphical representations of PARASOL abstractions such as nodes, tasks, ports, buses, links and messages. vPARASOL improves modelling productivity by providing three tools: a graphical user interface(GUI) for fast model construction, a code generator for the generation of the model framework code, and an execution monitor for interactive debugging and model verification.

vPARASOL is developed on a Sun UNIX platform. X11 window system is the basis of its graphical display. Its graphical interface makes intensive use of XView 3, an X11 programming toolkit and Developer's Guide 3, an XView interface builder, both from Sun Microsystems.

Acknowledgements

I would like to present my most sincere thanks to my supervisor, Professor John Neilson. I owe him a lot. Without his patience, his countless hours and his most invaluable advices, the vPARASOL project and this thesis itself would hardly be possible. Also his courses on distributed operating systems and computer system modelling surely benefited the work. Finally, his simulator PARASOL is acknowledged. It is the root of vPARASOL.

Thanks also go to Peter Choynowski for his providing of a reliable, comfortable working environment on which vPARASOL was developed. His contribution of the idea of invisible bus was proven to be simple and effective in the implementation.

Finally, I acknowledge the much appreciated financial support received for this work from the Software Performance Analysis With Concurrent Entities (SPACE) project headed by Professor C. M. Woodside of the Department of Systems and Computer Engineering.

Contents

Abstract	iii
Acknowledgements	iv
List Of Tables	ix
List Of Figures	x
1 Introduction	1
1.1 Background	1
1.2 Thesis Objectives	10
1.3 Thesis Organization	11
2 PARASOL Overview	13
2.1 Introduction	13
2.2 Model Building	13
2.2.1 Execution Environment	14
2.2.2 Tasks	15
2.2.3 Other Entities	17
2.3 Model Execution	18
2.3.1 Simulated Time and Event Handling	19
2.3.2 System Calls	20

2.3.3	Task States and State Transitions	21
2.3.4	Debugging	23
2.4	From PARASOL to vPARASOL	24
3	Design of Graphical User Interface	26
3.1	Overview of Graphical User Interface	26
3.1.1	Attributes	26
3.1.2	Advantages and Disadvantages	28
3.2	vPARASOL Graphical User Interface	30
3.2.1	Model Definition	30
3.2.2	Model Animation	31
3.2.3	Platform and Tools	32
3.3	Design and Implementation Details	33
3.3.1	Three Level View	33
3.3.2	Definition Interfaces	39
3.3.3	Animation Interfaces	42
3.3.4	Other Related Matters	47
4	Code Generation and Code Interface	50
4.1	Overview	50
4.2	The Header File	51
4.3	Genesis	53
4.4	User Task Stubs	58

5	Monitoring Model Execution	60
5.1	Overview	60
5.2	IPC Between PARASOL and vPARASOL	61
5.3	Extracting and Presenting Status and Statistics	64
5.3.1	Extracting Information from PARASOL Models	65
5.3.2	Presenting System Status	65
5.3.3	Presenting System Statistics	68
6	vPARASOL Walkthrough	70
6.1	An Email System Model	70
6.2	Building Model with vPARASOL	73
6.3	Supplying User Code	77
6.3.1	User Class Code	78
6.3.2	Mailer Class Code	80
6.3.3	Mail Server Class Code	80
6.3.4	Other Codes	80
6.4	Debugging and Executing	83
6.5	Analyzing Results	85
7	Conclusions and Future Directions	88
7.1	A Final Look at vPARASOL	88
7.2	Future Work	89
A	Example Code Generated by vPARASOL	91
A.1	Header	91
A.2	Genesis	94
A.3	Tasks	97
B	Glossory of Symbols in vPARASOL	101

List of Tables

4.1	Example Summary Section of a vPARASOL Header	52
4.2	Example Node Section of a vPARASOL Header	53
4.3	The Default Task Stub	59
4.4	A Simple Task	59
6.1	Simulation Results of the Email System Model	85
6.2	Effects of Varying CPU Speed	85
6.3	Results of the Modified Email System Model	86

List of Figures

1.1	Design Process and Simulation Process	2
1.2	Combining Design with Modelling	3
1.3	A Queueing Network and Its 2-Customer Markov State Diagram	4
1.4	A Marked Petri Net and Its Markov State Diagram	6
2.1	PARASOL Entities	14
2.2	PARASOL Task with Ports	16
2.3	A Simplified PARASOL Task State Transition Chart	22
2.4	Modelling Cycle in vPARASOL	25
3.1	vPARASOL Animations	32
3.2	Three Level View	33
3.3	vPARASOL Graphical User Interface	34
3.4	The System View of a Model	36
3.5	Inside a Node: Tasks, Task Status and Ready-to-run Queue	37
3.6	Inside a Task: Ports and Queued Messages	39
3.7	Default Values in a Dialogue Window	40
3.8	vPARASOL Help	41
3.9	The Debug Window	43
3.10	Device States and State History	45
3.11	Task States	46
3.12	Five Turns of Links	49

4.1	Flowchart of Genesis	58
5.1	Interprocess Communication Using UNIX Pipes	62
5.2	Model Execution	66
5.3	Values of Statistics	69
6.1	An Email System Model	71
6.2	The Send Mail Phase and the Receive Mail Phase	72
6.3	Starting a New Model	73
6.4	Building Nodes	74
6.5	Constructing Buses	75
6.6	Adding Tasks and Task Classes	76
6.7	Adding Statistics	77
6.8	User Class Code	79
6.9	Mailer Class Code	81
6.10	Mail Server Class Code	82
6.11	Messages Queued in <i>MailServer</i> and <i>Client4</i>	84

Chapter 1

Introduction

1.1 Background

Distributed and parallel computer systems are having a growing importance in today's computing community. Advanced electronic techniques have made more powerful processors available at lower costs, which gives the possibility of building large, cost effective distributed and/or parallel systems. A well-defined distributed/parallel architecture, especially large scale ones, yields more computing power, more flexibility, better accessibility and fault tolerance. All these features are most desirable characteristics of a computer system.

However, the design of such a system may be expensive and time consuming. System goals have to be defined carefully. Usually, next to functional correctness, system performance is the most important design goal. Unfortunately, it is not uncommon for such systems to have an unsatisfactory performance because system performance was not considered seriously until late stages of system evolution. Changing the system architecture to remedy performance defects at the time a system is completed could be very expensive, if possible at all. Thus, system performance evaluation is important during the system development process.

Since evaluations should be carried out throughout the design and implementa-

Figure 1.1: Design Process and Simulation Process

tion phases, traditional measurement approaches to performance evaluation, such as measuring and benchmarking, are not suitable since they rely on existing systems. Modelling, on the other hand, can provide performance predictions which would serve to guide the design and implementation. A model captures the most important, or most interesting, features of a system. These features largely determine the performance of the system.

In many aspects, the design process and modelling process are comparable, although there are also remarkable differences. Figure 1.1 depicts typical views of the design process and the modelling process([ZEIG79]). The *design process* starts with specifications of the desired system performance, and a description of available technologies. Its initial output, or intermediate product, is a blueprint. Following the prescription of the blueprint, a system can be built with the available technology and satisfying the given performance specifications. After the blueprint is implemented, the result is tested to verify whether it meets the performance spec-

Figure 1.2: Combining Design with Modelling

ifications or not. The *modelling process* takes in specifications of objectives, and constraints of available knowledge. A model is its intermediate product this time. The model must be interpreted in some concrete forms, such as simulation or analysis. The interpretation is then checked against the objective specifications. In performance modelling, the design process is apparently in a form of iteration, as illustrated in Figure 1.2. At an early design stage, a model with the basic system features can be built. Modelling results, either simulation or analytic, can be used as feedback to the design stage, and lead to refinements of design. Such refining procedures continue throughout the design and implementation phases. When the system construction is completed, in addition to verifying the design specifications, the actual system performance should be compared with the model results to see how well they agree, to help develop and refine the modelling process.

There are two general approaches to performance modelling. One is known as the analytic approach, based on mathematical theory. The other one is based on discrete simulation. These two approaches are complementary. They have their respective strengths and limitations. Analytic models typically require more restrictive assumptions or higher level abstractions so that they can be solved, and hence tend to be less accurate. On the other hand, they can, at times, give closed-form

results which are more general and much easier to use than the experimental results from a simulation. Even when closed-form solutions are not available by analytical models, they generally yield usable results with much less computational cost than do their simulation counterparts. The major advantage for simulation models is their virtually universal applicability.

Markov process modelling is a potential analytical approach involving the enumeration of all possible system states and the computation of their steady-state probabilities to determine system performance metrics such as utilization, through-

Figure 1.3: A Queueing Network and Its 2-Customer Markov State Diagram

put, etc.. The calculation involves the solution of often very large systems of linear equations. These equations are based on knowledge of state transition probabilities

which, in turn, depend on memoryless state duration distributions. The Markov property defines a stochastic process for which the system state in the future depends only on the present system state, not on its history. Although there are some basic results available, directly applying Markov models to complex computer systems may be particularly difficult because it is necessary to identify all the system states and the transition probabilities from one state to another. The combinatorial explosion of the number of states can make the modelling process difficult and error prone. For a system as simple as Figure 1.3 with 2 identical customers, there are as many as 10 states. In a fully connected queueing network, enumerating the number of system states becomes a classic combinatorial problem, with repetitions. If there are n queues and m customers in the system, the formula

$$C_{n+m-1}^m = \frac{(n+m-1)!}{m!(n-1)!}$$

gives the total number of different system states. For $n = 4$ and $m = 4$, it yields 35. When n and m grow up to 10 and 20, respectively, the number of states quickly goes up to 10,015,005. If customers are distinguished, this number will be even bigger. In addition, the magnitude of the resulting system state can make numerical solution intractable.

A *queueing network model* is a group of queues connected together. Each queue contains a server, a waiting area (the queue), and probability distribution functions describing customer arrival time and service time. Customers (transactions) arrive at a server, wait in the queue and receive the service, then leave. After departure, customers either join another queue or leave the system. Apart from the problem of solution, these models do not easily handle the problem of simultaneous resource (server) possession, since the inherent model treats resources serially. For example, in computer systems, a process may require a processor, a segment of memory, some disk space as its prerequisite for running. Such simultaneous resource possession requires a multi-level, decomposable queueing network model. Queueing network evaluation methodologies, i.e., product form solutions and mean value analysis (MVA) ([KLEI], [SAUE]), are limited with respect to scope. For example, exact solutions are available for FCFS (First Come First Served) if service time

is exponentially distributed. Although more general service time distributions are allowed, queueing disciplines must be PS(Processor Sharing), LCFSPR(Last Come First Served with Preemptive Resume), or IS(Infinite Servers) for models to have a closed-form solution.

Petri net(PN) was first proposed by C. A. Petri in 1962([PETR]). It was soon to be found an attractive modelling tool for computer system performance evaluation, especially for multiprocessor systems due to its natural representation of parallelism and synchronization. A standard PN is a 3-tuple (P, T, A) , where P is a set of places, T is a set of transitions, and A , set of directed arcs. Arcs connect places

Figure 1.4: A Marked Petri Net and Its Markov State Diagram

to transitions and transitions to places. When containing tokens, a PN becomes a *marked* PN which is used to define states of a system. *Execution*, or *firing*, of a marked PN is then introduced, and used to describe state transformation of a system. Figure 1.4 is an example of a marked PN and corresponding Markov state diagram. Many characteristics are parallel between a Petri net and the system it describes. The constant number of tokens in a Petri net is comparable to the configuration in a computer system, which is not likely to change, while the liveness of a PN indicates a dead lock free environment. One important result is that a certain kind of Petri nets(inhibitor, not k -bound Petri nets) are proven to be equivalent to the Turing Machine([PETE]). Extensions have been added to the basic Petri net models. *Timed* Petri net models, obtained by introducing the time notion(firing delays) to the basic Petri net models, can be used for the quantitative system performance

analysis. When using random variables to specify the time behavior of a model, timed Petri nets are called *stochastic Petri nets*, or SPN. Although the difficulty of system description as a Markov model is largely eased with SPN, its state-based nature remains the same. This can become a big drawback of SPN modelling since the number of states in a system tends to be so large that even the most powerful computers can not solve it. For this reason, stochastic Petri net modelling is limited to relatively small systems.

Thus, while viable for some systems, the analytical approach is not universal since many such models become intractable as the system grows in complexity or size. The more universal modelling approach, although admittedly not as elegant, is simulation. Simulation involves generating equivalent behaviors to those occurring in real systems. The problem of a simulation often lies in specifying the model or implementing algorithms. As well, validation of the model, given its complexity or size, is rarely easy.

Simulation modelling usually involves an iterative process of refinement of structure and detail. To the uninitiated, simulation appears somewhat unstructured with a high degree of complexity. Whicher([WHIC]) has described simulation as involving the following five elements:

1. Assumptions

The reality has infinite characteristics, which can not be detailed or enumerated completely. For this reason, assumptions underlie all simulations, either explicitly or implicitly. For example, while simulating a computer system, it is easy to assume that all users in the system are identical, CPU service times are independent and memoryless and possibly obey exponential probability distribution, disks are working independently, etc..

2. Parameters

Parameters are similar to control variables in empirical research. Although they could vary, they are fixed at particular values during a simulation. Parameters have some potential impact on the model simulated. Holding pa-

parameters constant clarifies the issue of causality, which is a vital advantage in simulation.

3. Inputs

Input variables and values are independent variables. In each instance they are constructed as factors that affect the primary phenomenon of interest, the dependent variables. Independent variables are manipulated by the researcher. An example is assigning different cpu speeds in computer systems simulations.

4. Algorithms

Algorithms are decision rules. They conduct the actual work of the simulation by converting inputs to outputs according to the specified rules. Algorithms must be compatible with the assumptions, but more specific.

5. Outputs

The dependent variables are both the primary phenomenon of interest in computer simulation and the main output of the simulation model. Dependent variables are identified and used in the simulation and guided by theoretical concerns and real world problems towards the outputs.

Simulation models can be classified in several ways. For example, system state may involve discrete, or continuous variables or both. Models may be deterministic or have stochastic components, e.g., the load of a computer system. Simulation tools, on the other hand, are available in a variety of forms, such as languages, libraries, etc. These tools are means of communicating simulation models to computers. Some of them are general purpose, e.g., SLAM, while others are specialized for particular applications, e.g., PARASOL. Since computer systems models usually involve stochastic models of discrete state systems, the following survey emphasizes tools appropriate for such studies.

GPSS, or General Purpose Systems Simulator, is a process oriented type of discrete event simulation language([GORD], [GORD78], [ADAM]). It describes a model mainly through an active type of entity(transaction) moving in time through

a set of passive system entities (facilities and/or storages). Storages are multiserver stations, or shared resources. Queues can develop at certain points along the transaction's path. Block diagrams are used to represent GPSS model structures. Different numbers of block types are implemented in different versions, or favorites, of GPSS. The modelling power of GPSS is determined, and consequently limited, by the defined types of blocks. Processes are defined in terms of blocks as well, which is lacking the flexibility of user definable processes.

SIMSCRIPT is event oriented ([MARK], [KIVI], [ADAM]). It was designed originally for discrete simulations, although it had been extended to handle continuous and combined models as well. Systems to be simulated by SIMSCRIPT are described in terms of entities (either temporary or permanent), attributes of entities and sets (logical groups of entities).

SIMULA is a process oriented, discrete event simulation language. A simulated system is a set of processes under SIMULA.

SLAM, or a Simulation Language for Alternative Modelling, attempts to be a universal language ([PRIT]). As its name suggests, SLAM provides a unified modelling framework for alternate modelling world views of a system. A discrete system can be modelled within an event orientation, process orientation or both. Continuous change systems can be modelled by either differential or difference equations. Combining different views within the same simulation model is allowed. Assistance is offered for describing a process in a process orientation model, managing event calendar in a event orientation model, or integrating equations in a continuous model.

PARASOL is a task oriented, discrete event simulator, specially designed for simulating distributed/parallel computer systems ([NEIL]). Tasks, distinct from the process concept used by other simulation languages, are user definable and execute in a simulated distributed computer system. They are schedulable, and run independently in a multitask operating environment similar to UNIX. Their definitions are actual functions written in C or C++. This kind of task approach gives the maximal flexibility, as well as the similarity to the real world system. System topology in PARASOL is described with computing nodes, communication buses and

links. A default node and an originator task take the responsibility of constructing the system topology and launching a set of initial tasks. Thereafter, other parts of system can be dynamically constructed by the active tasks if necessary. Users supply code to tasks.

Presently PARASOL offers only a non-graphical user interface to the model builder. The value of a graphical user interface(GUI) is well proven in many applications including simulation. Hence, the objective of this thesis is the development of a suitable GUI to facilitate PARASOL model construction and interpretation of modelling results. Ideally, this interface should enhance PARASOL, not through adding new functionalities, but by making PARASOL modelling easier by attending to many of its rather tedious details and by providing new ways to visualize the workings of PARASOL models.

1.2 Thesis Objectives

The current trends in modelling tools have moved from simple modelling tools to integrated modelling environments([BALB]). These tools range from a simple model solver(either analytic or simulation) to a complex modelling environment offering a variety of both solution techniques, assistance in model building, workload generating and formalisms for expressing models. Despite the large difference in terms of capability, graphical interfaces are commonly available among most newly developed modelling tools.

The most important features of a visual simulation tool, in addition to its model solving ability, should include at least three components: graphic representation of system abstractions, animating of model execution and automatic code generation. All of the three topics are treated in the vPARASOL project.

The primary objective of this thesis is to develop a graphical environment for speeding up PARASOL modelling. Under this banner, more specific goals are listed as follow:

- Ease construction of modelled execution environments

PARASOL modelled execution environments are easy to build by dragging in the PARASOL nodes, buses and links. The system topology is clearly depicted with the iconic representation of system entities.

- Speed up task initialization

Creating the initial set of tasks is simplified. Connections between tasks and task code classes can be made and changed through the GUI as well. Clonable tasks(tasks with multiple instances) are possible.

- Generate initialization code

Codes are generated in a manageable manner. Specified initialization system objects are created by the *genesis*. Stubs for user defined task classes are provided. Configuration report in the header file helps identifying objects and cross referencing.

- Provide monitoring facilities for prototyping and debugging

Debugging is much improved. Execution is illustrated graphically and updated dynamically.

- Minimize errors

Errors in constructing modelling environment and specifying entity properties are eliminated by automatic generated code and system supplied default values.

The vPARASOL project is partially funded by a large performance engineering project, SPACE(System Performance Analysis for Concurrent Entities), going on at Carleton University.

1.3 Thesis Organization

Chapter 2 of this thesis is a brief introduction to PARASOL, the predecessor of vPARASOL. A very high level overview of PARASOL, a simple description of its

system components and their relationships, its task scheduling and interprocess communication are given here. More details of PARASOL are found in [NEIL] and [NEIL91].

Chapter 3 describes vPARASOL graphical user interface design, including representation of abstractions, data structures and some details of implementation, along with available operations.

Chapter 4 is dedicated to code generation. Since PARASOL assures efficiency and compatibility mostly, and vPARASOL assists debugging, codes are produced in such a way that they can be launched either under vPARASOL or under the operating system. Use of vPARASOL requires the X11-Window environment.

Chapter 5 discusses executing and debugging issues. Extracting and presenting states and statistics of the running model are also presented.

In Chapter 6, a complete email system model is examined. This example shows the use of vPARASOL, and reveals its limitations, on the other hand. For users unfamiliar with PARASOL, it serves as an introduction as well. Comparing handwritten code and machine-generated code is interesting. For those concerning the ability and usage of PARASOL, [NEIL] and [NEIL91] are excellent references.

Finally, Chapter 7 reviews experience gained during the course of design and implementation of vPARASOL, as well as discusses some possible future directions.

Appendix A contains source code of an email model discussed in chapter 6, produced by vPARASOL. Studying examples is a fast way to get started with both PARASOL and vPARASOL.

Appendix B is a glossary of graphical symbols used in vPARASOL.

Chapter 2

PARASOL Overview

2.1 Introduction

PARASOL is a task oriented discrete event simulation system. It is specially designed as "a medium-to-fine-grained system simulator/emulator for distributed and/or parallel computer systems" ([NEIL91]). Among others, the most remarkable feature of PARASOL is its task concept and usage. Tasks are actual functions in a programming language such as C, and schedulable in a simulated PARASOL execution environment. With PARASOL, users are able to define system components and control their behaviors, in both hardware and software. It can be used as a performance prediction tool in an early stage of a system design, as well as a testing tool for parallel and distributed algorithms. However, PARASOL itself runs in a traditional single processor environment.

2.2 Model Building

PARASOL model building involves the construction of a simulated execution environment and the creation of a set of tasks that reside and execute on that environment. A specially node, *node 0*, is automatically built for every PARASOL model with a special task, **genesis**, executing on it. This is the universal starting point. Clearly, coding or defining the actions of the various tasks is an essential part of

Figure 2.1: PARASOL Entities

model building process. This begins with **genesis** since it is the place where the modeller can create other elements of the execution environment and launch other tasks to simulate, for example, software of the system under study.

2.2.1 Execution Environment

An execution environment represents the hardware aspect of a computer system. Typically, a distributed system consists of a group of computers connected together for the purpose of resource sharing or communication. In PARASOL modelling world, computers are abstracted to nodes, while connections among these computers can be either buses or links. Figure 2.1 shows the basic PARASOL entities and their relationships.

PARASOL *nodes* are computational engines. They contain a pool of one or more homogeneous hosts, or processors. Within a node, all hosts share a common ready-to-run task queue and the queueing discipline. A node has a name. Its speed factor indicates the relative speed of processors inside the node. A specified quantum is the time slice for round-robin(RR) scheduling. Queueing discipline for

the ready-to-run task queue can be either FCFS(First Come First Sever), NP(Non-preemptive Priority) or PR(Preemptive Resume). With or without round-robin, they give a total of six different queueing disciplines, representing the most common used strategies. A flag is set on the creation of a node, indicating whether or not the utilization statistics for this node should be collected. All attributes are assigned at the time the node is built, and immutable thereafter.

PARASOL *bus* communication facility provides multiway communication among a set of nodes using a shared medium. Data flows over buses may be FCFS or random, similar to the CSMA/CD protocol of the Ethernet in the latter case. A name, the number of nodes, the nodes, a packet size, a transmission rate, its arbitration or queueing discipline, and again, a statistic flag must be specified when a bus is being created.

PARASOL *links* analog asynchronous, one-way communication lines. Each link connects a pair of nodes. Pairs of links can be used when two-way data flows are required. Data sent over a link may be packetized in various sizes, and the queueing is strictly FCFS. Different links may have different transmission rates.

Multihop communication is not directly supported by PARASOL, although it can be modelled explicitly by using available primitives. Multihop communication can be built on buses, or links, or both.

2.2.2 Tasks

PARASOL *tasks* usually represent the software world of the modelled system. When created, tasks are assigned to execute on specific nodes. A task migration mechanism is available to permit such node-task assignments to be changed. Usually, tasks are permitted to execute on any of the processors within their assigned node. However, modellers can tie a task to a specific cpu within the assigned node, when necessary, so that the task can only execute on the specified cpu. Creation attributes of a task include its name, its assigned node and host(if desired), a task class or function entry, and its execution priority. The task function has neither entry parameters nor

Figure 2.2: PARASOL Task with Ports

return values. The same function may describe the execution behavior of several tasks and thus it defines a task class.

PARASOL uses tasks to model everything but nodes, buses and links. Various kind of servers, resources such as disks and memories, and of course, user defined tasks are all good examples of PARASOL tasks. At the extreme, a task can be used to model a node, a bus, or a link in greater detail. In all cases, users must write the necessary code for tasks, which, as stated above, can be shared by a class of tasks. Using such a task-based approach allows maximal flexibility in simulation.

Tasks are the most active entities in a system. They can be created, resumed, suspended or killed, on one hand. On the other hand, their execution may bring up new tasks, affect other tasks by means of message passing and so on. Clearly, there is a predecessor-descendent relation between tasks and all the tasks form a task tree. This relationship is important in a few ways. Some PARASOL operations have effects on tasks other than the calling task itself, or on more than one task, for example, killing or terminating a task implies killing all its descendants. On the other hand, in order to carry out some operations, some relationship between the operation issuer and the operand has to be held. Killing a task is an example again. A task can only kill itself or its descendants.

Task **genesis** is the first task to execute in a PARASOL model. Because PARASOL simulations usually involve the execution of many user defined tasks in a user defined execution environment, system initialization is complex. In order to simplify the user interface, PARASOL launches **genesis** which is then responsible for

the remainder of system initialization including the creation, either directly or indirectly, of all other user tasks. Typically, **genesis** constructs the simulated execution environment, i.e., building nodes, links and buses, on which the user defined tasks execute. **Genesis** is the predecessor of all other PARASOL tasks and has access to all tasks. Thus its termination will cause all other tasks to terminate, which ends the simulation. Therefore, **genesis** should be suspended when it finishes the initialization process.

2.2.3 Other Entities

Other important entities include *ports* and *messages* for interprocess communication, *spinlocks* for process synchronization, and statistic structures for gathering statistics.

PARASOL interprocess communication is defined and implemented through a port mechanism, similar to the port concept in operating system Mach([YOUN]). To send a message, a task creates a message and directs it to a port belonging to the target task.

A standard *port* is assigned to each task at the time the task is created. In addition to the standard port, a task may acquire as many ports as it needs. Each port provides a FCFS queue for messages directed to it. Similar to that in the Mach operating system, messages arriving at a port are accessible only by the task which owns that port. At any given time a port is owned by precisely one task so that simultaneous port sharing is impossible, although a mechanism exists to transfer port ownership(and implicitly, messages held in the port). Thus, in order to send a message to a task, the sending task must know the port id of at least one port owned by the target task.

PARASOL *messages* have four components, a user defined type code, a system supplied timestamp, a user supplied reply port and a user specified pointer to an arbitrary string.

Spinlocks provide synchronization addition to message passing. They can be used directly for fine grain mutual exclusion synchronization or as building blocks for other more elaborate synchronization mechanisms.

Simulation statistics based on observing the behavior of the system under study are important since they represent the output of the study. PARASOL provides three mechanisms for the purpose of statistic gathering. Simple device utilization statistics are collected by specifying the statistic flag to be true on the creation of hardware entities, i.e., nodes, buses and links. Further statistics must be collected explicitly by user written code in PARASOL tasks. Two other types of statistics are considered important in PARASOL. First, there is the type used to capture device utilization statistics. It is referred to as "TIME" statistics in PARASOL, since it is mainly used for tracking system state variables over time and measuring their mean values, e.g., queue length, cpu utilization, etc.. The second type, i.e., "POP" statistics, are concerned with population statistics. They record when populations are samples during the course of the simulation and report sample mean values, e.g., mean response time, mean waiting time, etc..

PARASOL provides a built-in means to estimate confidence intervals on the observed mean values from a single, usually long, simulation run. It does so by blocking the run into a small number of segments of equal duration and calculating "sample" means for each segment or block. Assuming these block means are uncorrelated, a student t test analysis may be performed to estimate the 95% and 99% confidence intervals. Alternatively, users may prefer to conduct independent replication runs using distinct random number generator seeds. Although the latter approach may be theoretically more sound, it requires the user to perform the analysis rather than PARASOL.

2.3 Model Execution

After **genesis** finishes constructing the execution environment for the model and creating the initial set of tasks, tasks start running. PARASOL uses a real valued simulated time. It is advanced by appropriate events. An event manager, or calendar manager, handles all the events as they happen. System services are provided for different uses such instructing a task to consume a certain amount of cpu ticks, or to

send a message to another task. Frequently, PARASOL's built-in debugger may be invoked for debugging the model.

2.3.1 Simulated Time and Event Handling

PARASOL performs its event management as most other discrete event simulators. It maintains a future event list, or event calendar. PARASOL driver handles events one by one. During the processing of an event, the simulated time is frozen. When the driver finds user defined task code to execute, it dequeues the next event from the calendar and advances the simulated time according to the event time. It then identifies the type of the event and calls the appropriate event handler. That handler is responsible for altering the system state, perhaps scheduling one or more future events as it does so, and for invoking the PARASOL execution scheduler as appropriate.

PARASOL event handlers include the following:

- **END.BLOCK**

This is the handler for ending a blocked state. Typically this execution state is used to hold a processor until the task can become hot, or active.

- **END.COMPUTE**

This routine handles compute time-out event. The task will end its compute state and either reenter the hot state or enter the ready state. A task enters the compute state by making a `compute()` call;

- **END.QUANTUM**

When a task runs out its quantum in a round-robin scheduling environment, this handler is called. The task will usually be moved to the ready state from the execution state.

- **END.RECEIVE**

This handler sends a receive time-out message to a receive blocked task. This action moves the task to the ready state.

- **END_SLEEP**

The sleep time-out handler awakens a sleeping task back and inserts it into the ready-to-run queue.

- **END_SYNC**

Similar to the END_COMPUTE handler, this routine handles a sync time-out event by allowing the task in question to reenter the hot state or to be readied.

- **END_TRANS**

This handler deals with the arrival of a message at its destination node.

2.3.2 System Calls

PARASOL is supplied as a C library. Its system services are in a form of system call. Parameters are passed to calls and either a status or a result is returned. System calls fall in the following categories:

1. **Hardware Configuration**

Calls in this category are for defining simulated system hardware or execution environment. They are usually called by the *genesis* task. There are functions to build nodes, buses or links.

2. **Task Management**

Besides creating tasks, calls are also available to let tasks consume simulated processor ticks, sleep, adjust priorities, and migrate over to other nodes. Tasks begin life in a suspended state and must be resumed to become active. They may be suspended or killed(terminated) subsequently.

3. **Intertask Communication**

Intertask communication is initiated by a task sending or broadcasting a message, and completed by another task receiving on the destination port. According to the medium or the receptors, send or broadcast may happen in different forms. Receives, however, are always applied to a port owned by the caller task. Auxiliary functions are available for allocating, releasing, and passing the ownership of a port.

4. Task Synchronization

Message passing offers basic task synchronization mechanism. Locks in PARASOL provide finer grain mutual exclusion synchronization, e.g., in shared memory multiprocessor systems. `Lock()` and `unlock()` are the only two calls in this category.

5. Statistics Collection

Mechanisms are provided to create structures for capturing two type() of statistical data. Type `TIME` tracks system state variables over time and computes mean values. Type `POP` counts population observations during a simulation and also estimates the mean. A statistic structure can be allocated or reset, and more important, instructed to record data through system calls.

6. Random Number Generation

Random numbers are used everywhere in a discrete simulation. PARASOL functions produce random variates of a number of distributions including the exponential and Erlang distributions.

2.3.3 Task States and State Transitions

PARASOL task scheduling is similar to process scheduling in operating systems. However, one should be aware that PARASOL is simulating a multi-threaded, multiprocessor environment in a single threaded system. Tasks may all be created by genesis at the beginning, or by some other tasks during the execution of the simulation. Task scheduling depends upon task states, priorities and node queueing

Figure 2.3: A Simplified PARASOL Task State Transition Chart

disciplines. The latter are defined at node creation time and are immutable. In any case, newly created tasks enter the *suspended* state, which holds all the inactive tasks. A task in suspended state will not be considered as a scheduler candidate, and does not compete for any system resources. A **resume** command takes a suspended task to the *ready* state where it can be considered by the scheduler when a processor on which it can execute is available. Since PARASOL itself is single threaded, no more than one task can be actually executing at any given time. Thus, concurrency of a multiprocessor environment has to be simulated. For this reason, there are several pseudo execution states within PARASOL. When a task is actually executing in the host machine, it is in the *hot* state. A task in the hot state, although executing code, can not move the simulated time forward. In order to do so, a hot task executes a **compute** system call and enters the *computing* state, where it consumes processor time while allowing other tasks to become hot. While in this state, it may be preempted if scheduling permits, and transferred to the ready state. Alternatively, it may cease computing voluntarily after consuming a specified amount of

processor time, and return to the ready state. A third execution state, the *blocked* state, is occupied when a ready task is assigned a processor by the scheduler before it can be made hot. This permits efficient "hands off" scheduling to take place when appropriate. Three similar mechanisms, i.e., *lock*, *hold*, and *sync*, are available to allow a modeller to specify the consumption of processor time in different scenarios. For example, a *hold* command causes a task to enter the *blocked* state for a specified duration, where it consumes processor time on the target machine. In other words, it moves virtual time ahead. This state is appropriate for systems where processor/task assignments are static and one-to-one, for example, in most highly parallel systems. There are also other execution states such as the *spinning* and the *sync* states. They provide finer control over tasks. In fact, PARASOL supports a total of seven execution states including a *spinning* state where a task performs a busy wait on a spinlock. Refer to [NEIL] for details.

A hot task may voluntarily give up the processor by executing a *sleep* call, after which it enters the *sleeping* state. When a sleeping task times out or is awakened by another task, it re-enters the ready state. If a task does a *receive* call on a port but no message is being held in the port, it blocks itself by entering the *receiving* state. An optional time out or the arrival of an incoming message causes the task to enter the ready state again, where it waits to be scheduled a processor.

Most of the cases, a *suspend* system call sends a task to the suspended state from wherever it was and a *kill* command removes tasks from the system.

2.3.4 Debugging

PARASOL kernel offers a small built in debugger. The debugger can report status of various queues, trace and single step through a simulation, and set breakpoints. It is rather primitive but very useful in debugging a model.

When starting a simulation with the trace flag on, PARASOL debugger will report events as they happen. Event reports will state who, where, when and how of events. A sample script of running the model *ntp*(Distributed Transition Processing, see [NEIL]) may look like this:

```
Time: 0; Node: 1; Task 1 (Down Relay) created (suspended).
Time: 0; Node: 1; Task 1 (Down Relay) executing.
Time: 0; Node: 1; Task 2 (Up Relay) created (suspended).
Time: 0; Node: 1; Task 2 (Up Relay) ready.
Time: 0; Node: 2; Task 3 (DB Dispatcher) created (suspended).
Time: 0; Node: 2; Task 3 (DB Dispatcher) executing.
Time: 0; Node: 2; Task 4 created (suspended).
Time: 0; Node: 2; Task 4 executing.
Time: 0; Node: 2; Task 5 created (suspended).
Time: 0; Node: 2; Task 5 executing.
(0)->
```

At the PARASOL debug prompt, current simulation time is shown in parentheses, followed by an arrow. Interactions can be accepted for displaying queues, showing bus/link status, setting breakpoints, single stepping, etc..

2.4 From PARASOL to vPARASOL

vPARASOL is a graphical user interface built on top of PARASOL kernel. The primary goal of designing such an interface is to speed up modelling under PARASOL. vPARASOL offers a graphical interface for model building, a code generator for producing code interface for models built via the graphical front end, and an execution monitor for interactive debugging.

Modelling activities under vPARASOL are cyclic. Defining the model is the first step, which includes establishing an execution environment which usually involves building all initial computing nodes and communication links, and assigning initial tasks. Statistic structures for collecting statistic data are also defined in this phase, although they are not part of the modelled system. This results in automatic generating of a portion of the PARASOL simulator code. During the second phase of the cycle, the details of task functionality must be provided by vPARASOL users, by supplying code to each task. When the coding is completed, the model is ready to run. An important feature of the vPARASOL is its monitoring of model execution. Because distributed system models often have many components, interpretation of results can be complex. The graphical interface associated with model

Figure 2.4: Modelling Cycle in vPARASOL

definition then becomes a useful organizational reference for output display. This is useful particularly for the third phase of the cycle, i.e., for debugging models and for prototyping systems. Should this phase result in model modifications, the cycle is repeated. Refer to Figure 2.4.

Chapter 3

Design of Graphical User Interface

3.1 Overview of Graphical User Interface

A user interface represents the users' view of a computer system. Its functions usually include accepting inputs, displaying outputs, showing status and prompting end users. The traditional solution is the command language interface, in which users interact with computers through commands. As the popularity of computers grows, particularly in the field of office automation applications, more and more inexperienced and infrequent users are involved. Command language interfaces appear not well adapted as they assume users possess more than minimal computer knowledge.

Graphical user interfaces(GUIs), on the other hand, can be equipped with windows, pointing devices and a full array of widgets that make operations easy and self explanatory. Moreover, graphics used in GUIs often map applications closer to the real world so that users are more likely to derive correct actions based on their everyday experience. Hence, graphical user interfaces have been recognized as one of the most effective means to accomplish human-computer communication.

3.1.1 Attributes

GUI is an active field of research in man-machine communication. It has developed beyond the early days' WIMP(Windows, Icons, Mice and Pulldown menus).

Following is a summary of some known attributes of GUIs, or techniques used in implementing GUIs.

Windows

A window is a specified area on the physical screen, through which a user views a particular aspect of his interaction with a particular task. A window has a frame so that it can be separated from other windows. Other accessories are available for controlling the window such as its size, position, and status. GUIs usually use multiple windows so that various kinds of interaction take place in different windows, protecting users from making mistakes. Since space on a screen is limited, it is important that windows can be resized, relocated, overlapped, and closed. Popup windows, or dialogue boxes are examples of windows, in addition to the application main window. They appear only at the time they are needed, and hold a dialogue with the user to gain information and complete a task. Some of the popup windows can lock the application or even the entire system, to force users to respond to a particular application requirement.

Icons

Icons provide a visual representation of data objects in a system. Icons should be designed realistically so that users can recognize the object or command being represented. Since icons themselves are also GUI objects, graphical operations may be applied to them, such as move, open, and close. Furthermore, logical connections between the icons and their associated application level objects or commands must be made. In this way icons provide a simple intuitive interface to the application, e.g, through mouse clicks to select or open an object.

Mice

Mice are the most popular pointing devices which allow users to take the full advantage of GUIs. With a mouse, a user can easily move around(change focus) between different windows of one or more applications, choose objects, respond to prompts

and access operations. Most of the time, a mouse appears to be an arrow cursor on the screen. To facilitate mouse operations, the shape of the cursor should vary according to applications or actions taking place.

Widgets

A set of predefined window widgets provide easy, standard means to communicate with the underlying applications. Windows widgets include menus, buttons, text fields, gauges, choices, lists, messages, and many others. Widgets do not exist by themselves. They must be associated with windows. Menus, buttons and text fields are those most widely used. Menu items or buttons are associated with a handler which is activated by choosing the item or button with the mouse. Text fields are usually used for acquiring, or editing, text data.

Drag and Drop

A common feature in windows systems is the so-called "drag and drop" operation. It is used for communication between two different applications or different parts of a common one. Most of the time, a drag and drop involves selecting(drag) an object and then relocating(drop) it elsewhere. To permit such operations, windows-based applications must first identify those objects to which these operations may be applied, i.e., the sources of drag and drop. It is also necessary for the destinations or targets to be defined since they can alter the meaning of the operation. Other information may also be required to complete the semantic definitions of such operations and to provide visual feedback to users, e.g., change the shape of the cursor during the drag and drop operation.

3.1.2 Advantages and Disadvantages

The most significant benefit of a GUI is its user friendliness. More specifically, GUIs

- Reduce the need for computer knowledge

As mentioned above, GUIs enable users with limited computer experience to operate applications using their intuition. This intuition may arise from the parallelism of the graphical representations of objects and reality, from specific contexts of applications, or from the enumeration of choices.

- **Feedback results immediately**

GUIs display the result of a user action immediately providing visual feedback to the user so that the context of future actions is understood. WYSIWYG(What You See Is What You Get) style of editing is a typical example. In some cases, the feedback may be a warning message indicating an error has occurred and an alternative action is appropriate. In other cases, users may correct their error by undoing or cancelling the last action.

- **Maximize the use of precious display space**

A display screen is always limited. GUIs allow more information to be displayed by using multiple, overlapped windows, or by popping up windows only when they are needed. Users should also be able to organize their windows as needed.

Certain disadvantages are also evident with GUIs, in both hardware and software. GUIs often

- **Require advanced display equipment**

GUIs require displays have at least minimal graphical capability. For acceptable graphical quality, advanced monitors, such as bit-mapped, high resolution ones, are desired. Combining the considerations of speed, color, etc., cost of graphical displays could easily grow to an order of magnitude higher than simple text only ones.

- **Require sophisticated software support**

Although some early GUI-based applications did exist without windows support from operating systems, GUIs run much smoother in a windows system.

The nature of GUIs requires multitasking and event management so that calls are made to event handlers in GUIs upon appropriate events. Window event management is usually complicated.

- Introduce large performance overhead

Performance impact is noticeable in most GUIs since managing events and sending updates to the display need extra cpu cycles. However, GUIs need not be used except when interactive use is contemplated. In such cases, the performance overhead is less important since user response time dominates the system speed.

3.2 vPARASOL Graphical User Interface

There are three areas in which a graphical user interface would be useful within the PARASOL modelling paradigm. First there is the issue of model definition. PARASOL models require the user to define the execution environment, i.e., the "hardware" network of computing nodes and communication facilities. A graphical interface is much more user friendly than a procedural one for fulfilling this requirement. A second potential use for the GUI is in interactive exploratory simulation where a user may wish to alter parameters and re-run a model in a spreadsheet "what if" fashion. When large scale models are involved, it is frequently easier to focus on the model through a well designed GUI. Finally, perhaps more important than either of the above, there is the potential usefulness of such a tool for debugging where system animation can provide greatly enhanced representations of system states and their changes.

3.2.1 Model Definition

Speeding up model building was the initial motivation for designing a GUI for PARASOL. Ideally, the interface should be easy to use and require only a short

learning period. It should allow incremental construction so that users can build their models by adding system components piece by piece.

The most essential editing operations include adding a new object, moving and removing an object, assigning and modifying properties of an object, etc.. Since objects in PARASOL logically belong to different levels, the GUI should reflect such a classification accordingly and allow users to work in different levels. All operations are expected to perform graphically and the results should be output promptly.

Various available techniques are employed in designing and implementing vPARASOL. For example, adding a computational node is done by dragging a node from the node component pool and dropping it at a desired place in the workspace. Popup windows allow users to see the properties of system objects and make modifications. Scroll bars permit a change of viewpoint within a large field that can not be shown in its entirety.

3.2.2 Model Animation

Model animation exposes events occurring in model execution. Graphical representations of system states change as events happen. It can answer questions such as "what is happening in the system" and perhaps, to a degree, why. For example, if the model appears not to be running as expected, animation may show the reason as insufficient processing power in a node, which could manifest itself in the form of a large ready-to-run task queue.

The third, and perhaps the most significant, value of a PARASOL GUI is its potential to assist in debugging (as opposed to understanding) PARASOL models. By their nature, PARASOL models are complex, often involving tens or hundreds of concurrent entities. A GUI which permits users to isolate problem areas and provides important, detailed information on system behavior would be of great benefit to PARASOL model developers. Such an interface should ideally permit user interaction to "step" the system forward in time while examining the system state. System states are presented and updated graphically as they change. For example, if

a task sends a message to a port of another task and waits for a reply, it should leave the execution state and enter the receiving state. On the other hand, the message should appear at the destination port and the port owner should accept the message and enter the ready state from the receiving state, assuming it was receiving on the port. Figure 3.1 shows the states before and after the message passing.

Figure 3.1: vPARASOL Animations

3.2.3 Platform and Tools

The platform for vPARASOL is a SunSPARC workstation, with the X11 Window system. XView 3, an X11 graphical programming toolkit from Sun Microsystems, was chosen as the principal development tool for vPARASOL since it offers a high level, integrated programming interface above the primitive X11 Programming Library, as well as some essential widgets like drag and drop([HELL]). Consequently, OpenWindows Developer's Guide 3.0, also from Sun Microsystems, was chosen as the auxiliary tool. It is an excellent graphical tool for fast construction of graphical interfaces for XView applications, although certain restrictions apply([DevGde]). Most of the restrictions arise since the graphical interface can not enumerate all the options XView has provided. Some are due to the omission of some less frequently used XView objects, such as full screen. An example is that applications can only use scroll bars, but do not have full control on them, if they have been built with the Developer's Guide. All scroll bar events are processed by default. Users must

define their own scroll bars if they want full control, so that events can be passed to the application and desired actions can be performed accordingly.

3.3 Design and Implementation Details

3.3.1 Three Level View

vPARASOL describes a PARASOL model in three levels(or layers), namely the *system level*, the *node level*, and the *task level*. The system level is also referred as the *model level*. Figure 3.2 depicts the relationship among these three views. Logically, these view levels correspond to a model, a node and a task in PARASOL.

Figure 3.2: Three Level View

The main workspace of vPARASOL is a large, windowed canvas. At any given time, only a portion of the big canvas is visible in the window. Tools and building blocks are around it(Figure 3.3). This is a standard XView window. On the top, a *title bar* shows the name of the model in edit. Underneath it, a set of *dropdown menus* provides all the operations from opening a model file to generating code for the model to executing the model. Below the menus, a set of *tablets* are available for speeding up most frequently used operations, such as opening a file or replicating an object. The largest part of the window is the scrollable *workspace pane*, which

Figure 3.3: vPARASOL Graphical User Interface

is used to display model views. For example, in the system level view, a portion of the system is visible in the workspace pane. Under the workspace pane, to the left, a *road map* gives the global picture of the model, showing the current workspace pane as a small rectangle within. To the right of the road map, a *recycle box* collects all discarded objects. Next are the *component pools* for each of the building blocks, i.e., nodes, tasks, ports, buses, links and statistics. A *navigator* at the end of the row provides fast access to adjacent panes in the workspace. Finally, at the bottom, informative messages and warning messages are displayed in the *left footer*, while the *right footer* always shows the current view level and object name. A number of popup windows appear only when they are needed. Mostly they ask for an input.

Movement within the model can be accomplished by one of three methods.

Within the road map, the small rectangle representing the current workspace may be dragged and dropped to a new location, which is a fast way to move over long distances, if the destination is known. Using the scroll bars attached to the working space scrolls the window slowly, in either direction. Finally, a press on the navigating pedal switches the display window to one of its eight adjacent neighboring panes.

Switching between different views happens by either moving to a lower level view(into a particular object) or moving out to a higher level. In order to move into an object, first an object must be selected, then one activates a zoom-in command. For moving out of an object to a higher level view, it can be directly done by a zoom-out command. Refer to Figure 3.2 for the three level views. Only one level view is displayed at any given time.

Model configurations are saved in vPARASOL model files with *.mdl* extension, along with the necessary working environment.

The System View

The *system view* is the highest view level of a PARASOL model. It shows the system hardware topology, i.e., the relationships between nodes, buses and links. Figure 3.4 is the system view of a six node model. PARASOL nodes are depicted as boxes in vPARASOL, with features displayed. The so-called PARASOL node, *node 0*, is always built by default and located at the top left of the working space. User defined nodes can be laid out as desired. Nodes fall in a designated grid so that they do not overlap each other. PARASOL buses and links also appear in this level. To avoid the visual complexity and distractions of complicated bus structures, buses are shown simply by labelling nodes as connected to buses. For example, the bus id labels at the bottom of node blocks in Figure 3.4 show two buses, with all nodes but the PARASOL node connected by bus 1 and the servers connected by an extra bus, bus 0. Links are depicted as directed lines, pointing from source nodes to destination nodes. There is no need for labelling a link since there are only two nodes involved(source and destination) and they are clearly marked by the

Figure 3.4: The System View of a Model

starting and ending points of the link. To simplify and order the resulting node-link picture, a link always leaves its source from the right, and enters the destination from left. In the example shown in Figure 3.4, node *Client1* can talk to *Client2* via the link connecting them. PARASOL statistics are another type of objects to be manipulated in the system level. Since statistics are often used by more than one task and last to the end of simulation, it is appropriate to create them during the model building process.

Finally, a highlighted object is the current *selected object*. Many vPARASOL operations apply to the selected object. Operations available in the system level are mainly for constructing the model execution environment, such as building nodes, buses and links. Once an object exists, its position can be changed, and its properties

Figure 3.5: Inside a Node: Tasks, Task Status and Ready-to-run Queue

can be edited. Objects can also be deleted. Although tasks can be created at the system level, they are the major interest of the next view level, the node view.

The Node View

The *node view* is the inside view of a node. PARASOL tasks residing in nodes are accessible in this level. Tasks are depicted as parallelograms and may be added or removed from a node using appropriate drag and drop operations together with the task pool or the recycle box. Every task has a status bar associated with it. The status bar dynamically mirrors the state of a task during the model execution. A task can be in one of six vPARASOL states, namely *ready*, *executing*, *sleeping*, *receiving*, *suspended* and *spinning*. Note that the six PARASOL execution states,

i.e., *hot*, *computing*, *blocked*, *sync*, *suspended* and *spinning*, are mapped into a single vPARASOL executing state to simplify user understanding. In Figure 3.3.1, task *Client2Task2* is currently executing and the rest are all in the ready state. Ready tasks are queued in the ready-to-run task queue, also shown in the figure. Tasks in the queue are represented by their names. Since the space for display items in the queue is limited, if more tasks than the queue can display are being held in a queue, a number appears simply to indicate the number of tasks in the queue. Actual task names are not shown. Task and queue status is refreshed dynamically during execution.

There are two types of tasks in vPARASOL. Although they all are translated to PARASOL tasks when the model executes, they are treated differently in vPARASOL. The first type is a *static task* and the second is a *dynamic task*. The main difference between them is that a static task corresponds to precisely one PARASOL task at run time while a dynamic task, or a *clonable task*, can have a number of instances. The exact number of instances of each clonable task is determined by the user at run time. The reason of clonable tasks is that it gives the possibility of running models with different problem sizes and working loads without re-editing and re-compiling the model code. The other way a static task differs from a dynamic task is that the static task is named, i.e., it can be referred via a variable name, while instances of a dynamic task are not named. When considering a set of similar tasks, if only their existence is important, not individuals, a clonable task is a possible choice.

The Task View

The *task view* allows one to view the internals of a task, i.e., its communication ports and the messages held in ports. A port is depicted as a rectangle showing any message queued within. Each task has at least one port, its standard port. Tasks may have other ports as well. Again, a task can be edited with respect to ports through the drag and drop paradigm. vPARASOL does not allow a task to release its standard port, although that can be done in PARASOL. During execution,

Figure 3.6: Inside a Task: Ports and Queued Messages

message flow is depicted at this level. Messages held in a port are represented by the source task name of the message. In case of message overflow, a number will be shown to indicate the number of messages in the port, similar to the ready-to-run task queue in a node(See Figure 3.6).

3.3.2 Definition Interfaces

Default Values

Default values eliminate the need of entering values for every parameter. A set of reasonable default parameter values meets users' requirements in most cases and therefore simplifies the object property assignment. Besides making model definition

Figure 3.7: Default Values in a Dialogue Window

easier, default values may assist novice users by suggesting appropriate parameter values. Default values can easily be overridden by redefinition.

Object Replications

Large computer systems often consists of hundreds of similar computational nodes with communication connections, or a number of smaller, structured subsystems such as bus or token ring local area networks(LANs), or common sets of processes running in a number of different machines. Replications in large systems could require a considerable amount of time. To cope with this problem, vPARASOL has a set of replicating operations to be used in various types of replications. For example, a push of a button gives a simple copy of the current node without copying any of its bus connections, while another button will additionally copy all of these connections. Similarly, structures like bus LANs, token ring LANs, and meshes with

Figure 3.8: vPARASOL Help

different number of nodes can be easily defined with vPARASOL menus.

Layout is the major problem in automatic object replications. Since it is not easily overcome, alternate strategies are used in different situations. For example, the closest available neighbor space is a suitable place for a simple node copy. In a mass duplication, holding all newly created objects in a minimal square is the strategy used by vPARASOL. Manual rearrangements of replicated objects may be required at times to ensure objects are located in the most appropriate way.

Help Facilities

A good help facility is essential in any successful applications. It serves as an on-line reference for experienced users and a getting-start guide for new users. In addition to a clear text, context-sensitivity is most desired for a help system. It is convenient because it is fast and, more importantly, accurate.

vPARASOL offers two types of help. Similar to other help facilities, the *vPARASOL help* has a hierarchical text structure. Users usually start from the root of the hierarchy and retrieve to the points of interest. Figure 3.8 shows the root of vPARA-

SOL help text hierarchy. This is a very simple help system. Users select a topic by simply choosing it from the topic list. Text of the selected topic will then appear in the text windows to the left. The *XView help* is a feature integrated in the XView graphical package. It provides help on visible XView objects. Activated by placing the cursor on the object of interest and pressing the **help** key on a Sun Workstation keyboard, XView help is truly context-sensitive. However, this type of help views help text as a set of unrelated pieces of information without a hierarchical structure. Hence, users can not move through an organized information structure, since such a structure does not exist. The XView help is a subset of the vPARASOL help.

3.3.3 Animation Interfaces

Debug Window

The debug window is designed for presenting system states and controlling model execution. Although vPARASOL continues using the model main window when debugging, most interactions will take place in the debug window. The debug window is the interface through which users interact with the underlying running model. Debug command set has been enhanced, compared with the original set provided by PARASOL, so that vPARASOL can extract more information out of the model and perform more operations than those offered by PARASOL.

Figure 3.9 shows the debug window on the completion of an execution. The window includes a *state chart*, an *object list*, execution parameters, command buttons, and a text output window. States of selected tasks or devices are presented in the state chart, while a summary of the simulation run is found in the output window. Debug command buttons are placed in the middle of the debug window.

From top left, the state chart shows the state history of the selected objects. Objects can be either PARASOL tasks or hardware devices, i.e., nodes, buses, or links. For devices, users have a choice of showing states history or utilization history. Utilization is intended to show the working load curve of a device over a certain period of time(a window) prior to 'now'. Tasks do not have utilization history.

Figure 3.9: The Debug Window

Ideally, the chart windows the simulation in a specified time period so it will not be restricted by the length of simulations, and the window's physical width. To the right of the state chart, an object list enables users to choose objects of interest by clicking the them in the list. Selections are exclusive for devices, i.e., only one device at a time. In case of a multiprocessor node, all cpus in the node are shown in the state chart. However, selections of tasks are non-exclusive. All defined tasks are presented in the object list, including dynamic ones and those created by user tasks. They can be selected freely by toggling each item on or off. Under the list, the random number seed and the duration of this simulation run is shown. Note that the duration is a double precise number in PARASOL but simplified to an integer in vPARASOL. These numbers can be changed so they will take effect in the next run. The time scale underneath always shows the elapsed simulation time. Reset on each run. Next, two switches let users switch among different types of objects, as well as between status and utilization in case of a device selection. Under the state chart, a message always states the time range currently shown in the chart. vPARASOL assigns a default time duration of 64 to each simulation run, since 64 is a reasonable length of time and it can be easily halved. Therefore, the default time range is [0, 64]. The range changes as the simulation time exceeds the specified range or the user changes the time range. Buttons to the right of the message are designed for manipulating the time scale. Buttons with single angle brackets(< or >) will double or half the time range currently showing in the state chart. These with double angle brackets(<< or >>) extend or shrink the time range in an order of four. The time width of the state chart display is upper bounded by the simulation duration and lower bounded by one. Whenever a change is made, display in state chart will be reset and start again from 'now'. Finally, the lower part of the debug window is a large scrollable text window which simulates a traditional terminal from where users interact with PARASOL. This window retains the similarity of PARASOL debugging.

Debug command buttons are located in the middle of the debug window(Figure 3.9). Each command button corresponds to a command in PARASOL, except ReStart

and **Help**. **ReStart** is available for starting the simulation again before it reaches its end. Meanly, it is invoked after detecting bugs or changing environment such the random number seed or the time scale. The **Help** button is a synonym of PARASOL help command except the text has been trimmed to better fit the graphical environment and it is displayed by vPARASOL. Commands that need extra information such as time for a breakpoint will bring up a popup window. Users can then enter values to the window.

State Indicators

During execution, the states of cpus, buses, links and tasks change. Various state indicators are used in vPARASOL to present such state changes. As well, indicating messages queued in task ports and tasks in the ready-to-run queues are important. All of these elements which collectively represent system state contribute to system animation.

Figure 3.10: Device States and State History

States of hardware devices such cpus, buses and links can be either *idle* or *busy* or *down*. vPARASOL depicts device states in bars with different patterns, shown in the state chart within the debug window. Each state bar shows the current state of a selected hardware device, as well as its history. Legends of device states are given in Figure 3.10, as well as an example history of a device. This history shows the device was first idle, then busy, then down and finally up and busy again. Its current state is busy. Note that state durations may be very short so they appear in the window

Figure 3.11: Task States

as a pair of, or even a single, vertical lines indicating state switchings, instead of a clearly visible state bar. Although there is mechanism to amplify time over the history bar, there are still cases in which two event times remain indistinguishable.

Task states are more complicated. A PARASOL task can be in one of eleven possible states. As mentioned in the pervious chapter, PARASOL has six execution states. They are *hot*, *computing*, *blocked*, *sync*, *sync_free* and *sync_suspend*. All PARASOL execution states are considered to be equivalent in vPARASOL and mapped into a common vPARASOL executing state. Other PARASOL states have their correspondent states in vPARASOL, namely, *ready*, *suspended*, *receiving*, *sleeping* and *spinning*. Therefore, the number of task states in vPARASOL is reduced to six. Figure 3.11 are the legends of all six vPARASOL task states. vPARASOL reports task states in two different windows. In the state chart of the debug window, a state bar is assigned to each selected task. This state bar is dynamically updated as the model runs in a manner similar to device state bar described above, except that tasks have more states. The other place vPARASOL shows the task states is the main work space. If the work space currently focuses in a node, the status bars of the shown tasks will be updated to reflect their current states.

Message passing is an important aspect of a distributed system. Since all messages in PARASOL are directed to a target port, not a target task, vPARASOL does not animate a message passes through a path. Instead, it displays the messages held in ports. Messages are only visible at the task view level since that is where ports are

manipulated. Messages are usually displayed in the destination port by the name of its source task. When the display space is overflowed, messages exceeding the 'port capacity' will simply be presented by their total.

Ready-to-run task queues are handled similarly to the ports. They are presented in node level alone with tasks. Normally, name of the ready tasks are displayed. In case of an overflowing, a count is displayed for all exceeding tasks. Since state of tasks are also shown in the same window by means of task status bar, it is easy verify if a task is in the ready state.

3.3.4 Other Related Matters

A few other topics may not directly connect with the graphical interface, but they are important in the design and implementation of the vPARASOL graphical user interface.

Naming Conventions

The meaning of naming varies by context. Two external naming schemes are involved in vPARASOL. They are used for representing model components in the model building phase, and identifying variables in source code generated by vPARASOL code generator.

Object names represent objects in a model. As they are used in PARASOL, object names are merely strings meaningful to users. They do not need to be unique. vPARASOL supplies default names to objects according to their types and creation order. Thus a node will have a default name like `node5`. Moreover, a task in that node can have a name of `node5task2`. Users can override default names with more meaningful names, like "The Undersea Relay in Mid-Atlantic". In practice, lengths of names are often limited, especially their display lengths.

Variable names are used by vPARASOL code generator for identifying objects in source code, at the time of generating source code for the model. Needless to say, they must be unique. The strategy for insuring the uniqueness is keeping an increasing name id for each type of object. This id goes up strictly on the creation of

every new object, even on deletion of objects. Thus a variable name for a PARASOL object is strictly dependent upon the object type and its creation order, and not accessible via the GUI. vPARASOL does not use the object names in any way to create corresponding variable names.

In addition to the external naming schemes, vPARASOL uses another naming scheme internally, mainly for model debugging. Internal names map objects between vPARASOL front end and the background PARASOL model so that communication can be set up. This is vital to vPARASOL itself but has less meaning to the users.

Task Classes

Although they are not part of the modelled system like other PARASOL entities are, task classes are important in PARASOL modelling. A task class defines behavior of one or more tasks. In vPARASOL, task classes can not exist by themselves. They have to be associated and defined with tasks. They are usually sharable so that a number of tasks can be in a common task class. Most user defined task classes are examples of sharable classes. In a few rare cases, a task class is non-sharable, such as class *genesis*. A class exists only if there is at least one task of that class exists. When the last task of a class is deleted, the class is removed consequently. vPARASOL creates a code stub for each task class defined.

Geometric Properties

Since PARASOL does not deal with graphical objects, it has no representation of geometric properties of an object, such as its position, shape and color. Clearly, when implementing a graphical user interface, these properties are essential. vPARASOL has added a set of data structures to bridge the gap.

For example, vPARASOL uses the following data structure to record attributes of a link. Since a link has a source node and a destination node, its start point and end point are determined implicitly by such two nodes. A typical link has five turns (Figure 3.12). When a link is presented in the workspace, it must be adjusted at every turn so that it does not conflict with other links. In the link structure,

Figure 3.12: Five Turns of Links

fields `turn1` through `turn5` denote such adjustments. They are integers indicating how far the actual line segment should be away from its original positions, i.e., the position when there is no interference from other links.

```
typedef struct vpara_link {          /* vPARASOL link struct */
    int      p_id;                   /* PARASOL id */
    int      v_id;                   /* vPARASOL id */
    int      n_id;                   /* name id */
    char      name[vPARAMAX_NAME];
    int      snode, dnode;           /* source and destination */
    int      psize;                  /* packet size */
    double    trate;                 /* Xfer rate */
    int      stat;                   /* statistics flag */
    int      turn1;                  /* order of the first turn */
    int      turn2;                  /* the second turn */
    int      turn3;                  /* the third turn */
    int      turn4;                  /* the forth turn */
    int      turn5;                  /* the last turn */
    Dnd      dnd;                   /* drag and drop */
    int      selected;               /* currently selected? */
} vpara_link;
```


Chapter 4

Code Generation and Code Interface

4.1 Overview

Two primary goals are considered in designing vPARASOL automatic code generator. The first one is to liberate modellers from the tedious details of PARASOL programming, a further step towards efficient modelling. The other one is to set up the connection between vPARASOL front end and the underlying model, a necessary preparation for execution monitoring. The code generator creates the 'framework' code, or the code interface, for a PARASOL model built with the graphical interface. Such a framework is the ground on which users add details.

vPARASOL code generator is a specification-to-language generator. It takes in systems specifications recorded in model files(suffixed by *.mdl*) by the graphical front end during the course of model building and converts the systems configurations to corresponding C function calls to the PARASOL library. Since vPARASOL presently lacks provision to define the fine-grained internal details of tasks, i.e., execution cycles in terms of operations on data structures, etc., vPARASOL code generation is limited to more coarse-grained concerns such as the creation of the execution environment including nodes, buses, links, ports, statistics. Since, however, tasks are also included in the graphical user interface, they too should be involved in some appropriate way. A suitable "generated code framework" consists of three code files: a C header file, *_cfg.h*, which specifies the PARASOL configuration in

terms of global variables, a C source file, *_genesis.c*, which contains the code for the PARASOL genesis task to create the defined execution environment, etc., and a task stub C source file, *_tasks.c*, which contains the framework code for the user defined tasks plus a little more. Normally, the stub code generated in the last file must be completed by the user to fill in details of task behavior, etc.. On the other hand, *_genesis.c* file does not normally require editing since it fully automates the creation of all the initial simulation entities. The header or configuration file is a vital part of the code interface since it provides the definitions and declarations of all data types and global instances thereof. It also serves as documentation to guide users in mapping user names to PARASOL names or variables, e.g., *DBServer* to *node2*, etc.. Much of the vPARASOL header is thus comments collectively called the configuration report which is very much meant to be read and understood by the user when filling in stub code details. The forth file is the *Makefile* used in building the model target. It includes references to the necessary header files and libraries.

4.2 The Header File

The header file is a combination of configuration report and table of C global variable definitions. It is divided into several sections. The first section summarizes the numbers of each kind of system entity used. Also given in this section are the maximum numbers of various kinds of entity that can be used. These limitations are defined in PARASOL kernel, i.e., system limits. *Class* is a new entity type to vPARASOL and its limitation is therefore specified by vPARASOL. All other upper boundaries are defined by PARASOL. The summary section of the example in Figure 3.4 is shown in Table 4.2.

Next are detailed information on nodes, buses, links, tasks, ports, statistics, and task classes. One section for each category. The node section of the same example is given in Table 4.2. This section shows properties of all defined nodes. Most of the information in the table is used to generate code for node construction. Attention

Item	#Used	#Max

Node	6	100
Bus	2	100
Link	0	1000
Task	10	1000
Port	10	1000
Statistic	2	----
Class	6	100

Table 4.1: Example Summary Section of a vPARASOL Header

should be drawn to the correspondence between fields *Name* and *ID*. Names are strings which are defined by users. They are the object names defined in 3.3.4 and not necessarily unique. IDs, on the other hand, are assigned by vPARASOL, and are unique. They are actually the global variables used by PARASOL, i.e., these variables must be used within PARASOL code segments to refer to nodes. These are actually variable names defined in Section 3.3.4. *#CPU* is the number of processors in the node. Recall that every PARASOL node can hold one or more homogeneous processors. *Speed* is the relative speed of processors. *Slice* is the time slice or quantum used for round robin timesharing task scheduling in the node. Field *Queue* is the queueing discipline for the ready-to-run task queue. The statistic flag is specified in *Stats*, should be either *Yes* or *No*. Finally, *#Task* gives the number of tasks in the node. Since it is not possible to count instances of a clonable task until run time, vPARASOL simply reports it as if it were a single static task. This field is the only one not used in code generation. It is informative since PARASOL limits the number of tasks used.

Sections for other entity types are similar to the node section. Refer to Appendix B for a complete example of header file.

Identifiers for entities(field ID in above table) are chosen in such a way that the numbers within them will only increase, not be reused. More precisely, if the node represented by variable, say, *node1* is removed, identifier *node1* will never be used

Name	ID	#CPU	Speed	Slice	Queue	Stats	#Task
PARASOL	node0	1	1.00	1.00	FCFS	No	1
NFServer	node1	1	2.00	1.00	FCFS	Yes	1
DBServer	node2	4	2.00	1.00	FCFS	Yes	1
Client1	node3	1	1.00	1.00	FCFS	Yes	3
Client2	node4	1	1.00	1.00	FCFS	Yes	2
Client3	node5	1	1.00	1.00	FCFS	Yes	2

```
extern int    node0, node1, node2, node3, node4, node5;
```

Table 4.2: Example Node Section of a vPARASOL Header

again in the same model. Therefore, every reference to discarded `node1` in task code will cause a compilation error. In that way, all such orphan references can be detected and fixed.

4.3 Genesis

Code for `genesis`, the root of PARASOL task creation tree, is located in file `.genesis.c`. According to the PARASOL manual([NEIL]), `genesis` can do many kinds of system administration but its principal use is to initialize the simulation by constructing the model execution environment and by starting the initial set of tasks. Accordingly, vPARASOL has made `genesis` in such a way that after it finishes the initialization, it makes a call to function `v_monitor`, in which users can assign other jobs that should be performed at initialization time by `genesis`. Thus, `genesis` itself is invariant as far as users are concerned. Eventually, control should return to `genesis`. Since `genesis` can not exit before the completion of a simulation, it enters an endless loop upon finishing the initialization. In this loop, it acts as a "grim reaper" killing all tasks who request suicide assistance. This technique is used in preference to a `kill(SELF)` call which results in a PARASOL memory leak.

In addition to `genesis`, there are other related functions defined in the `.genesis.c`. Typically, a suicide function and a block statistic function appear. `Suicide` is

designed to inform PARASOL that a task is about to terminate and the system resource occupied by the task, such as internal table entries, can be reclaimed. It does so by first sending up the id of the calling task to grim reaper, which is part of the **genesis**, the predecessor of all other tasks, and then suspending the caller itself. Grim reaper takes further actions to kill the task with the received id and to reclaim resources.

For each recorded node in a model, vPARASOL generates a C function call. For example, to create node **NFServer**, the following code is generated.

```
node1 = build_node("NFServer", 1, 2.00, 1.00, FCFS, TRUE);
```

Note that all the parameters are taken from the model file.

Buses and links are also mapped to appropriate calls. Since both of them connect nodes, the nodes involved have to be built first. In building buses, an array must be declared and assigned to hold all applicable node ids. Thus, building a bus will take more than one C statement. For example, a three node bus named "BackBone" and referred to as **bus0** is created as follows:

```
bus_nodes[0] = node1;  
bus_nodes[1] = node2;  
bus_nodes[2] = node3;  
bus0 = build_bus("BackBone", 3, bus_nodes, 1500, 1.25e6, FCFS, TRUE);
```

Since PARASOL tasks execute in nodes, they must not be created before their hosting nodes are built. After the node is built, initializing a task includes two steps. First, the task has to be created. And second, it must be resumed so that it becomes ready to run. Although these steps can be done in a single C statement, vPARASOL uses two statements to ensure a correct object mapping between vPARASOL and PARASOL. vPARASOL does so by delaying the second step until all the tasks are created, since any task may create new tasks which could invalidate the mapping. The following lines create and resume tasks named **ServerT1** and **ServerT2**, referred as **task1** and **task2**, respectively. Note that these tasks are in different classes.

```

task1 = create("ServerT1", node1, ANY_HOST, ServerC1, 10);
task2 = create("ServerT2", node1, ANY_HOST, ServerC2, 10);
:
resume(task1);
resume(task2);

```

Clonable tasks, or dynamic tasks, are treated differently. The reason for using a clonable task is that there are cases users need a certain number of task instances running a common code class, but do not concern any of them individually. Thus, members in clonable tasks are not identifiable, i.e., they can not be referred through variable names as static tasks can. Since they are resumed right after creation, they must wait until all other objects are created. Numbers of instances of each clonable task are supplied by the `main` function in global variable `clone1` at run time. Following is an example use of clonable tasks.

```

for (i = 0; i < clone1; i++) {
    resume(create("CTask1", node1, ANY_HOST, CTaskC1, 10));
}

```

Putting all the pieces together, a very simple `genesis` may look like this:

```

#include    <parasol.h>                /* include PARASOL head file */

int        node0 = 0;                  /* node id of PARASOL node */
int        node1, node2;
int        bus0;
int        link0;
int        task0 = 0;                  /* task id of genesis */
int        task1, task2;
int        task3, task4;
int        bus_nodes[3];              /* for bus building */
int        reaper_port;               /* grim reaper */

void
genesis()
{
    int     p_id, port;                /* for message passing */

```

```
double  ts;
char    *tp;
void    ClientC1();
void    ServerC1();
void    ServerC2();

reaper_port = MY_STD_PORT;      /* make globally accessible */

/* build nodes */

node1 = build_node("Server", 2, 2.00, FCFS, TRUE);
node2 = build_node("Client", 1, 1.00, FCFS, TRUE);

/* build buses */

bus_node[0] = node1;
bus_node[1] = node2;
bus0 = build_bus("BackBone", 2, bus_nodes, 1500, 1.25e6, FCFS, TRUE);

/* create tasks */

task1 = create("ServerT1", node1, ANY_HOST, ServerC1, 10);
task2 = create("ServerT2", node1, ANY_HOST, ServerC2, 10);
task3 = create("ClientT1", node2, ANY_HOST, ClientC1, 10);
task4 = create("ClientT2", node2, ANY_HOST, ClientC1, 10);

/* resume tasks */

resume(task1);
resume(task2);
resume(task3);
resume(task4);

/* call v_monitor() -- execute user defined code */

v_monitor();

/* inform vPARASOL genesis() is done */

genesis_done();

while (TRUE) {
```

```
        receive(MY_STD_PORT, NEVER, &p_id, &ts, &tp, &port);
        kill(p_id);
    }

}

/* function suicide() sends the caller's id(as type) to the grim */
/* reaper, and the latter will kill the caller and clean up */

void
suicide()
{
    send(reaper_port, SELF, "", NULL);
    suspend(SELF);
}
```

With cross referencing to the PARASOL manual([NEIL]), this sample **genesis** should be straightforward. After including the necessary header file, it declares global variables. These global variables are defined as external variables in the corresponding *_cfg.h* so that other code segments can refer to them by simply including this header file. All of the defined task classes are declared in **genesis**. Note that they all have a type of **void**. The creation of system components follows, in the order of nodes, buses, links, static tasks, ports and statistics. Function **genesis_done()** informs vPARASOL that **genesis** has finished constructing the execution environment and is about to enter model loop so that vPARASOL can enable and disable appropriate commands and setup the debug environment, if the model is started in the debug mode.

A very important feature of vPARASOL is its ability to provide graphical debugging. To be able to carry out the debugging, the mapping between PARASOL entities and vPARASOL entities has to be maintained. vPARASOL currently uses the order of object creation to ensure such a mapping. Therefore, all the static entities, such as nodes, buses, links, statistics, some of tasks and ports, must be created before any dynamic entities. Furthermore, no user task should be running before finishing such creation, since tasks are potential entity creators.

To summarize, the flowchart of task **genesis** is given in Figure 4.1.

Figure 4.1: Flowchart of Genesis

4.4 User Task Stubs

Function **main** and stubs of all user defined task classes are in file *_tasks.c*, as well as the mentioned **v_monitor**. This is the only file that users should modify, among the ones generated by vPARASOL.

The **main** function supplied by vPARASOL basically does two things. One is to acquire parameter values and the other is to start the simulation. Users need to supply various information to a model before it can run. **Main** first scans the command line for simulation parameters such as the random number seed and the duration of the simulation. If it does not find them, it acquires them from the user. Then it will ask for model oriented inputs such as numbers of instances of clonable tasks. Since the **main** function is typically the only place where user inputs are accepted as all PARASOL tasks are running in background, users should add code for their inputs and save them in global variables for later reference. vPARASOL provides six built in variables, *para1* to *para6*, for this purpose. These parameters can be integers, strings or real numbers and must be specified during the model building phase. Note that all user add-ons to **main** must be placed in a section located right after all system supplied parameters or they will be wiped out if code is regenerated. After saving all necessary inputs in appropriate globally declared variables, a call to **run_parasol** turns the control over to PARASOL.

Since file *_genesis.c* is not intended to be modified by users, function **v_monitor** was introduced as a substitute mechanism. It is called by **genesis** near the end of its execution. Note that **v_monitor** is a function, not a PARASOL task. Eventually,

```
void
DefaultStub()
{
    /* inform grim reaper to kill self */

    suicide();
}
```

Table 4.3: The Default Task Stub

`v_monitor` will exit and return the control to `genesis`. Leave this function blank if there is no need to augment `genesis`.

Although PARASOL tasks appear to be parameterless functions in C language, they are fundamentally different. The difference lays on that a PARASOL task is an independent entity schedulable in a model while a regular function is not.

Stub user tasks do little at this stage. By default, they simply call a suicide function, which sends up a message to the grim reaper. The grim reaper kills the calling task. Users should augment such stubs this with more productive code segments. The default task stub is shown in Table 4.3, followed by the a simple task with user defined code in Table 4.4.

```
void
SimpleTask()
{
    receive();
    compute();
    send();

    /* inform grim reaper to kill self */

    suicide();
}
```

Table 4.4: A Simple Task

Chapter 5

Monitoring Model Execution

5.1 Overview

vPARASOL execution monitor is an extension of PARASOL built-in debugger. It provides users better control over model execution and better representation of system states for models built with vPARASOL. Such models can execute in one of two different environments: under the UNIX operating system or under vPARASOL. Although running a model directly under operating system gives the best efficiency, built-in diagnostic facilities are primitive. It is expected that lengthy, production simulations would be run in this manner. However, running a model under vPARASOL allows users to take the full advantage of vPARASOL graphical debugger. When a model is run within vPARASOL, vPARASOL first creates a child process to execute the model. The vPARASOL execution monitor then extracts model status by communicating with the child process. As well, it presents information in graphical formats, such as bars and curves, in appropriate windows. Such information can be extremely helpful in debugging and model verification.

Clearly, there are two processes involved and they exchange messages with each other. Means of interprocess communication must be established. Section 5.2 discusses interprocess communication issues between PARASOL and vPARASOL. And the last section, section 5.3 deals with the extraction and presentation of system status.

5.2 IPC Between PARASOL and vPARASOL

One of the vPARASOL design goals is to make models built by vPARASOL independent of vPARASOL, which means they can execute as standalone processes, without vPARASOL's presence. Actually, even when a model is started by vPARASOL, it is still an independent process. The difference is that it is now controlled by vPARASOL, instead of the operating system. The model, therefore, responds to vPARASOL debug commands and reports system states to vPARASOL in a format specified by vPARASOL. To serve this propose, vPARASOL adds a new flag, the *vPARASOL flag*, to PARASOL. This flag implies both PARASOL's tracing flag and single stepping flag. When a model is invoked though vPARASOL front end, this flag is set to true and the model acts accordingly.

A reliable communication channel between the vPARASOL front end and the PARASOL model under control plays a key role in execution monitoring. Although there are other possibilities, UNIX pipes seem to be a good candidate as they provide simple, reliable interprocess communication between processes having a parent-child relationship. When vPARASOL receives the command for running a model, it creates pipes before it forks a child to execute the model. Communications in both directions are facilitated. vPARASOL uses three pipes to emulate the standard streams of the model process, i.e., *stdin*, *stdout* and *stderr*. After setting up the pipes, a child process is forked. The child process first redirects its *stdin*, *stdout* and *stderr* to an appropriate pipe through the judicious use of *dup()* system call. Meanwhile, vPARASOL, the parent process, sets up handlers for these pipes on the other end. Of the three streams, *stderr* is dedicated for communication required by vPARASOL itself, such as passing state data. Message flows over *stderr* are interpreted by vPARASOL so that all user output should go to *stdout*. Other than this, there is no restrictions on the user side. Figure 5.1 depicts the parent-child relationship and the communication channels between vPARASOL front end and the model itself.

Data passing through *stderr* is mainly reporting events to vPARASOL. It has a

Figure 5.1: Interprocess Communication Using UNIX Pipes

fixed length and a fixed format. Since it usually causes vPARASOL to take some action, its format is defined in data structure `action_t`.

```

typedef struct action_t {
    double time;           /* message format */
    int node;              /* event time */
    int task;              /* PARASOL node ID */
    int action;            /* PARASOL task ID */
    int status;            /* action */
    int dest;              /* new status */
    int dest1;             /* destination */
    int dest2;             /* aux destination 1 */
    int dest3;             /* aux destination 2 */
} action_t;
  
```

Event time is passed in field *time*. In most cases, event issuer is a task in a node. They are what fields *node* and *task* stand for. *Action* is the event actually taking place. For example, creating a task or sending a message. *Status* is the state in which the involved task will enter. An example is that after creation, a task enters the suspended state. Fields *dest*, *dest1* and *dest2* are objects of the action. They are presented only when needed. A complete event may read: at time 10(*time*), task 5(*task*) in node 2(*node*) sends(*action*) a message to task 6(*dest*) via port 2(*dest1*)

and bus 0(*dest2*). No *status* is necessary since the task continues in the executing state. Messages will be displayed in the tty window(see Figure 5.2), at the lower part of the debug window, with names of involved nodes, tasks, etc., if possible.

Since vPARASOL uses X11 window system and the X protocols prevent a child process from talking to the X server through the connection set up by its parent process, the model process can not send system updates to the display by itself. All such messages must go to vPARASOL first. And the later will bring the display up to date. This restriction introduce more complexity to vPARASOL since it has to handle all the messages from the running model and send them to a proper window, on one side. On the other, it keeps the model(and PARASOL kernel) simple as there is no need for it to deal with screen outputs, and unified as all updates are handled by vPARASOL. By carefully defining the use of pipes, users will not even notice any difference from using PARASOL. This complies with another goal of vPARASOL: minimizing modification to PARASOL kernel.

Another difficulty with this approach is that vPARASOL must respond to messages flowing through the pipes instantly while still be able to handle window events initialized by users. The first part can be accomplished by polling the communication pipes constantly. However, doing so will not only impede the process from carrying out any other functions, but also unnecessarily consume all the cpu ticks. Thus, it is not feasible. Splitting another process is the ready answer for processing simultaneously but here it will be too expensive in terms of system resources and have to face the same problem of communication. Fortunately, XView solves this problem by providing a service of registering streams and associating them with their handlers. Whenever an event occurs to a registered stream, the corresponding handler gets called and appropriate actions can be performed, just as XView treats any other events. For example, to associate an input event handler to a pipe, or more general, a file, function call

```
notify_set_input_func(ClientID, input_func, fd);
```

is made, where *fd* is the file descriptor, *input_func* is the handler, and *ClientID* is an identifier distinct from any other client. After registration, the handler will

be called when the file descriptor has data ready to be read(it was registered as an input event handler). General events can be opening or closing of the stream, data arriving or buffer emptied, etc..

5.3 Extracting and Presenting Status and Statistics

In order to display model information in vPARASOL graphical user interface, vPARASOL must be aware of values of all necessary system variables. Generally, this can be done in one of two ways:

- **Making PARASOL report variables**

After small changes in code, PARASOL can report state to vPARASOL. By adding a few more commands, PARASOL sends up only necessary state data. The major plus of this approach is its simplicity in implementation and unification in variable values.

- **Maintaining all variables in vPARASOL**

By keeping track of all events reported by PARASOL and knowing important system disciplines such as queueing in each node, it is possible for vPARASOL to maintain its own copy all system state variables. A big advantages is that changes to PARASOL are avoided. Its mean drawback is the extra complexity introduced to store and calculate each affected value upon each event reported.

vPARASOL takes the first approach but with modifications. Code has been added to send up state data on two occasions. When simulated time changes, all data necessary for updating the state chart in the debug window is reported automatically. Data for updating queues and statistics in workspace is reported on request, since only a small set of variables are involved, and the set changes as users move in the model from one part to another.

5.3.1 Extracting Information from PARASOL Models

Modification to PARASOL is largely reflected in this section. Word PARASOL in this section refers to the version after all changes are introduced. First, if a model is running with the vPARASOL flag on, all trace report will refer to vPARASOL's trace reporter. Events are all in the mentioned `action.t` format.

Next, when simulation time advances in PARASOL's driver loop, a small routine will be called by the driver to send up new values of all state variables needed for bringing vPARASOL state chart up to date. Choice of state variables is made at the time users chose their objects in the object list of the debug window, and stored by PARASOL. vPARASOL maintains a list of most recent values of such variables. When it receives updates, it compares new values with the ones in the list. The state of objects persists until an event time and if there is the state change for some object, vPARASOL will draw a vertical line in the corresponding state bar. After vPARASOL outputs the states to the state chart, it stores the new set of values.

For state variables in the main work space, such as queues and statistics, PARASOL will wait for vPARASOL's request. vPARASOL will pass the node id for the ready-to-run task queue if it is currently in a node, or the task id for the message queues if in a task. If vPARASOL is staying in the model level, it also passes this message to instruct PARASOL report all user defined statistics. Device statistics are presented in their property popup windows and updated only if the windows appear.

5.3.2 Presenting System Status

The phenomenon of a PARASOL model is largely determined by ready-to-run task queues in nodes, messages queues(ports) of tasks, status of devices and tasks. One of the objectives of the vPARASOL execution monitor is to expose system state changes promptly so that modellers can diagnose the model based on the observations.

vPARASOL takes the same approach in presenting system information as it did

Figure 5.2: Model Execution

in constructing models, i.e., it views a model as a tree structure with the views of, from root down, the model, tasks in a node and ports in a task. The configuration report file, *_cfg.h*, produced by vPARASOL code generator, lists all the system components in the model and their relationships. Moving between layers or objects at the same layer is similar to that described in the graphical user interface chapter except that no change to the model is allowed during a debug session.

Statistics and queues are considered belonging to different layers in the model structure tree. In order to trace these system variables graphically, users must select the appropriate layer. Statistic data structures are usually used for specific statistics and referred by tasks in one or more nodes. They are in the model layer. Hardware device statistics are also in this layer since devices themselves are visible there. Ready-to-run task queues belong to nodes. They are displayed in the node layer. Naturally, message ports are in the task layer as they are associated with tasks.

Message Queues

Message passing is the basic means for tasks to exchange information. In PARASOL, messages are sent by tasks and directed to ports owned by other tasks. A port is actually a FCFS queue which holds all messages directed to it. Contents and length of a queue contribute to the system state. They may, as well, indicate bugs or bottlenecks of a model.

Message queues are examined at task level. Since all the queues are updated automatically as the simulation unfolds, all users do is to select the right task. Queues in a task can expose different aspects of their owner task. If a task is waiting on its port but the port is always empty, there will be a good chance that some other task is not functioning properly. Messages may be sent to a wrong address or not sent at all. Another possibility is that a task attempts to send a message via a non-existing bus or link. On the other hand, a port may be flooded with messages indicating insufficient processing power(a possible bottleneck) in the node or that the owner task is not listening to the port.

Read-To-Run Task Queues

There is one ready-to-run task queue in each node. The queue is displayed if the work window is currently in a node. In a uniprocessor node, at most one task can be in executing state at any given time (this is easy to verify since state of all tasks are displayed at the same level). In a multiprocessor node, more than one task may be executing. In most cases, if there are tasks in the queue, all processors in the node are usually busy (exceptions are that in a multiprocessor node, tasks may be tied to a certain cpu so they can not execute even there are other cpus idle in the node). Once again, a large task queue indicates the node is overloaded.

5.3.3 Presenting System Statistics

At the end of each simulation run, PARASOL automatically reports statistics for devices with their statistic flags on, and all statistic structures defined in the model. Following is an example final output of a simulation. Of which, *Receive* and *Send* are user defined and both of type POP.

Simulation statistics for time = 100000.

Name	Type	Mean	Obs(# interval)
BackBone Utilization	TIME	7.7328E-05	100000
Client5 (cpu 0) Utilization	TIME	0	100000
Client4 (cpu 0) Utilization	TIME	0.0501988	100000
Client3 (cpu 0) Utilization	TIME	0.0497703	100000
Client2 (cpu 0) Utilization	TIME	0.0465437	100000
Client1 (cpu 0) Utilization	TIME	0.0473212	100000
MailServer (cpu 0) Utilization	TIME	0.20166	100000
Receive	POP	151.363	2401
Send	POP	2.97489	1638

However, it is desirable to access values of statistic variables during the course of a simulation. vPARASOL provides means to accomplish this purpose. Figure 5.3 shows two defined statistics displayed in the model view level, within the statistics icons. Type and name of a statistic are shown. Its mean value (number 5.15993

Figure 5.3: Values of Statistics

and 38.7783 in figure) and sample count(number 3 and 1 in figure) are display next. They are two most important aspects of a statistic. Note that for type TIME, sample count is replaced by time duration. For device statistics, user must explicitly open the device property window. An item named *Statistics* in the window shows the current device utilization. Presently, only statistics of one node, one bus or one link can be displayed at a time.

Chapter 6

vPARASOL Walkthrough

This chapter presents an example of using vPARASOL in modelling a simplified email system. The example attempts to clarify concepts introduced in previous chapters and to demonstrate the phases of modelling with vPARASOL. These phases include model construction, debugging, and and finally production of results. Source code of the example is found in Appendix A.

6.1 An Email System Model

An email system is a typical distributed system. Servers and users reside in different sites and communicate with each other via various media connecting them. Although a mail system can not be complete without support from name servers, file servers, transfer protocols, etc., the example system examined here will be simplified. It contains a mail server node, **MailServer**, running a mail server task providing mail services and five client nodes, **Client1** to **Client5**, running mailer tasks. Nodes are connected with a single bus, named **Backbone**, to form a bus LAN. Although the mail server could be easily modified to communicate with the outside world, to simulate subtle details of disk access, etc., this example system is a local, closed system and all disk activities are simulated. Operational failures are not considered either. The system topology is shown in Figure 6.1.

From the user's point of view, a complete mail operation consists of two phases, i.e., the *send phase* and the *receive phase*. Phases are independent. When a user

Figure 6.1: An Email System Model

sends a mail, a send phase starts and it lasts until an acknowledgement of successful transmission is received. The receive phase starts with a request and ends with the arrival of a message or a negative acknowledgement. Although they look similar, the send phase is acknowledged by the mailer (the mailer will forward the mail to the mail server but that is beyond the mail user's concern), while the receive phase expands the acknowledgement loop to include the mail server. See Figure 6.2.

There are three different task classes in the system. One is the user task which is used to simulate users at a terminal. In this system, all users do is think, send mail and receive mail. Since the primary interest of the model is centered on mail service, simple user tasks are located in the PARASOL node. A user generates a message or decides to read mail according to some pre-defined time pattern and sends the message or request to mailer at its "local" client node. The mailer acts as a bridge as it forwards mails and requests from users to the mail server, and relays any response from the server to the users. It will also acknowledge users' send mail actions. The mail server task simulates the storage and retrieval of messages from its data base. For demonstration purposes, this task is kept very simple by simply consuming cpu to represent any processing that is required and using sleep delays to represent disk latencies. Notice, there is no attempt to model actual mail messages although this could be done at the expense of greater model complexity.

Note that, with this model, users synchronize through rendezvous with the local

a) The Send Phase

b) The Receive Phase

Figure 6.2: The Send Mail Phase and the Receive Mail Phase

mailer task for both phases. Similarly, mailers synchronize with the mail server, in this case over the **BackBone** bus. However, the synchronization between mailers and the mail server only occurs and is embedded in the user/mailer interaction when receiving mail. This behavior recognizes the need for the user to wait for mail from the server when receiving mail while users need not wait on the server (only the local mailer) when sending mail.

Statistics are collected during simulation. Device utilization statistic flags are all turned on at the time the device was created so that device utilizations are recorded automatically. Response time statistics are captured separately for both the send phase and the receive phase. For the send phase, the elapsed time between a user sending a mail and receiving an acknowledgement from the mailer is the "send" response time. In the receive phase, the "receive" response time is elapsed time between the request and the reception of mail. Since the receive phase extends to the mail server, a longer response time is expected. These statistics also count every send and receive, and report them. The average mail traffic rate can be derived

Figure 6.3: Starting a New Model

from the counts and the simulation duration.

6.2 Building Model with vPARASOL

When the model system topology and task classes are determined, the next phase is to use vPARASOL to build the system. Since all PARASOL models contain the PARASOL node and the genesis task, the *new* operation under menu *file* gives a new model with the PARASOL node and the genesis task. Refer to Figure 6.3. This is the standard starting point of all vPARASOL models. Other system components are introduced thereafter.

Next one sets up the server node and client nodes. At the system view level, after dragging a node from the node pool and dropping it at a suitable place in the working space (Figure 6.4 (a)), a popup window appears to acquire node properties (Figure 6.4 (b)). Default values are displayed in the popup window. The only thing likely to be changed is the name of the node. In the example, one replaces the default node name to **MailServer** and presses button *OK*. Since display space for names are limited, it

Figure 6.4: Building Nodes

is a good practice to keep names within a reasonable length. Client nodes are made similarly, or one can use the replicating feature of vPARASOL. Should changes be needed, selecting a node by clicking on it and choosing item *properties* under menu *edit* will bring the property window up again. Figure 6.4 (c) shows the result after creating all nodes.

Communication is accomplished by a single bus. After constructing all the nodes, buses can be added to connect them. Two cases can occur when creating a new bus. If a node is currently selected, a click on the bus pool below the workspace will ask for a second node on the bus. Clicking on another node will give a bus connecting these two nodes (Figure 6.5 (a) and (b)). If no node is selected, vPARASOL will ask for two nodes. Adding a node to an existing bus requires one to select the bus first, then click on the bus pool and then the node. The email example needs a bus,

Figure 6.5: Constructing Buses

named **BackBus**, to connect all but the PARASOL node. Figure 6.5 (c) shows the result.

Tasks can be created in either the system level or the node level. The difference is that at the system level, tasks must be dropped in a node, and that node can be any node visible in the workspace. When at the node level, tasks can only be added to the focused node. Task property window pops up in both cases. Task classes are specified in the window. vPARASOL creates a new class for each new task by default. Choosing one of the existing classes listed will associate that class with the newly created task. In this example, a server task is dropped in the server node and one mailer task is dropped in each client node, sharing a common task class. A clonable task class is used for user tasks. Since users communicate to five different mailers(simulating users in five different locations), a random number is used to

a) Editing Task Properties

b) All Tasks Are Added

Figure 6.6: Adding Tasks and Task Classes

determine which mailer is called. Instances of the clonable task are located in the PARASOL node with the actual numbers of instances of these tasks determined at run time.

Mailer tasks use extra ports since they communicate with both user tasks and the server. Although there are other possibilities, this approach shows how extra ports are used for different purposes. Ports are objects inside tasks. They are not visible or operatable at the system view level. In order to add ports to a task, one should move into the node level or task level. After mailer tasks are created, ports can be allocated to them. Adding extra ports is similar to adding tasks and the figure is omitted.

The last step is to add necessary statistics. According to the model specifications in the previous section, two statistics are needed. One is for calculating the average "send" response times and the other for "receive" response times. They are created

Figure 6.7: Adding Statistics

by dragging from the statistic pool and dropping in the workspace, as Figure 6.7 (a) and (b). The completed model is shown in Figure 6.7 (c).

When the model is saved in a model file, *email.mdl*, choosing item *code* under menu *file*, or pressing the code tablet on the tablet bar will generate the framework code for the model. All code files will have the prefix *email*, such as *email_genesis.c*.

6.3 Supplying User Code

This section discusses, in detail, task class code supplied by users to the framework produced by vPARASOL. Task stubs are in file *email_tasks.c*. Other than task classes, there are also a few small additions, such as the one in the post-genesis function, *v_monitor()*. Users should use the stub file as a place holder, i.e., to place function

calls, instead of a great amount of code in it. Code is easy to maintain that way.

Three task classes define the basic scenario of the mail system. They are user class, mailer class and mail server class. Tasks in class `user` send requests to tasks in class `mailer`, randomly or following a specified time scheme. Upon reception of such requests, `mailers` either confirm a send mail request and forward the mail to the task in class `mail server`, or relay a receive mail request to the server and wait for the mail. The mail server simulates computing and disk operations and if the request is a receive, it acknowledges the sender to complete the receive mail operation. Statistic structures are defined and initialized in function `genesis()`. Code for collecting statistics is incorporated in user tasks. The following sections examine each of the three task classes in detail.

Prior to coding ‘ ‘ task classes, it is important to read the configuration file, *email.cfg.h*. This file summarizes the model structure and gives identifier assignments for global variables, which is critical for programming. For example, the server task is assigned to variable `task2` in the email example, the five mailers to `task3` through `task7` and the bus to `bus0`.

6.3.1 User Class Code

See the code in Figure 6.8.

Users execute a simple “life cycle” loop involving first a `sleep()` call to simulate the think phase. The embedded `exponential()` call makes the sleep time exponentially distributed. Next a random choice, using the specified send probability, is made between sending and receiving a message and the request time is recorded as `start.time`. The request is then made by the `send()` call to the mailer and the user awaits a response by the `receive()` call. Finally, the response time is calculated as `now - start.time` and captured by the `record_pop()` call.

```
#define CLIENT_NODE      5
#define MEAN_THINK      8.0
#define SEND_PROB      0.4

int    mailers[5];

void
user_class()
{
    double  start_time;
    double  ts;                                /* receive time stamp */
    int     send_flag;                          /* send or receive */
    int     type;                              /* message type */
    int     rp;                                /* reply port */
    int     mp;                                /* mailer port */
    char    *tp;                              /* message pointer */

    mp = std_port(mailers[choice(CLIENT_NODE)]);
    while (TRUE) {
        sleep(exponential(MEAN_THINK)); /* simulate user thinking */
        if (random < SEND_PROB)
            send_flag = TRUE;
        else
            send_flag = FALSE;
        start_time = now;                    /* record current time */
        send(mp, send_flag, "", MY_STD_PORT);
        receive(MY_STD_PORT, NEVER, &type, &ts, &tp, &rp);
        if (send_flag)
            record_pop(ptr_sp0, now - start_time);
        else
            record_pop(ptr_sp1, now - start_time);
    }
}
```

Figure 6.8: User Class Code

6.3.2 Mailer Class Code

Since mailers communicate to more than one task class, messages from different sources should be separated. In general, that can be done by either using different ports for each source or by encapsulating the message source in the message. The example uses the former approach. The function `my_ports()` return the total number of ports owned by the caller, as well as all the port ids. Mailers use the "extra" ports to receive messages from the mail server. As the mail server is remotely located, a bus send is used instead of a simple send.

The mailer execution loop is only slightly more complicated(see Figure 6.9). It begins with the reception of a user request with the `receive()` call. Here `MY_STD_PORT` refers to the original or standard port associated the mailer task. Next, to simulate processing, a `compute()` call is made. Then according to the request type, a send phase or a receive phase is initiated. The former involves sending an acknowledgement to the user and then forwarding the message to the mail server via the `bus_send()` call. The latter requires the mailer to synchronize with the mail server to retrieve a message. This is accomplished by `bus_send()` and `receive()` calls. This is followed by forwarding the simulated message via the `send()` to the user. Note that the mailer listens for the mail server's response on its "extra" port.

6.3.3 Mail Server Class Code

The mail server maintains a mail database. When it receives a message, it stores it in the database. On the other hand, it searches the database for read requests. The `compute()` function call simulates processing the request while `sleep()` call simulates the database latency. The mail server does not acknowledge a send but it does confirm a read request by sending back the required mail. Actual code is shown in Figure 6.10.

6.3.4 Other Codes

Since user tasks simulate users at different nodes, they send their mail requests only to their "local" mailers and let the mailers talk to the server. A small random

```

#define MAIL_PROC      3.0
#define MAIL_SIZE      40;

void
mailer_class()
{
    double  ts;                /* receive time stamp */
    int     type;              /* message type */
    int     rp, rp1;           /* reply port */
    int     pcount;            /* port count */
    int     ports[5];          /* ports */
    char    *tp;               /* message pointer */

    my_ports(&pcount, ports);
    if (ports[0] == MY_STD_PORT);
        pcount = ports[1];
    else
        pcount = ports[0];
    while (TRUE) {
        receive(MY_STD_PORT, NEVER, &type, &ts, &tp, &rp);
        compute(exponential(MAIL_PROC));
        if (type) {             /* send phase */
            send(rp, type, tp, MY_STD_PORT);
            bus_send(bus0, std_port(task2), type, MAIL_SIZE, tp, MY_STD_PORT);
        }
        else {                  /* receive phase */
            bus_send(bus0, std_port(task2), type, MAIL_SIZE, tp, pcount);
            receive(pcount, NEVER, &type, &ts, &tp, &rp1);
            send(rp, type, tp, MY_STD_PORT);
        }
    }
}

```

Figure 6.9: Mailer Class Code


```

#define MEAN_PROC      5.0
#define DISK_ACCESS    20.0

void
server_class()
{
    double  ts;                /* receive time stamp */
    int     type;              /* message type */
    int     rp;                /* reply port */
    char    *tp;               /* message pointer */

    while (TRUE) {
        receive(MY_STD_PORT, NEVER, &type, &ts, &tp, &rp);
        compute(exponential(MAIL_PROC));
        sleep(exponential(DISK_ACCESS));
        if (!type)              /* receive? */
            bus_send(bus0, rp, type, MAIL_SIZE, tp, MY_STD_PORT);
    }
}

```

Figure 6.10: Mail Server Class Code

integer is used to decide which mailer is local and consequently, to route the request to that mailer. Mailers' task identifiers must be placed in an array so that they can be indexed by the random number. The best place, and the best time to do this is in the post-genesis function: `v_monitor()`. Note that `v_monitor()` is a C function, not a PARASOL task.

```

int mailers[5];

void
v_monitor()
{
    mailers[0] = task3;
    mailers[1] = task4;
    mailers[2] = task5;
    mailers[3] = task6;
    mailers[4] = task7;
}

```

Array `mailers[]` is globally declared in file `email_tasks.c` and task identifiers are external variables defined in `email_genesis.c` and included in `email_cfg.h`.

6.4 Debugging and Executing

Since code quoted here works properly, the debugging section only shows how debugging information is presented and where to find it. Also it shows the format of typical simulation results.

After supplying all the necessary code, the model can be compiled. vPARASOL also produces a `Makefile` which tells how the model target should be made. If additional files are needed, the `Makefile` should be changed to reflect the additions. The compiling result is an executable file which runs directly under the operating system. This process is just like the way working with any other PARASOL models. Following is the result of a simulation run.

Simulation statistics for time = 1E+06.

Name	Type	Mean	Obs(# interval)
BackBone Utilization	TIME	7.7292E-05	1E+06
Client5 (cpu 0) Utilization	TIME	0.0249364	1E+06
Client4 (cpu 0) Utilization	TIME	0.0232918	1E+06
Client3 (cpu 0) Utilization	TIME	0.0244473	1E+06
Client2 (cpu 0) Utilization	TIME	0.0229654	1E+06
Client1 (cpu 0) Utilization	TIME	0.0248908	1E+06
MailServer (cpu 0) Utilization	TIME	0.201251	1E+06
Receive	POP	318.379	24221
Send	POP	123.212	15963

However, running the model under vPARASOL requires loading it in vPARASOL first. Pressing the run button will make the model target(if necessary) and start running the model in the vPARASOL debug window(Figure 5.2). The model first asks for the number of mail users in the tty window, since the user task is a clonable task. Users can debug a model in the tty window as if they were working with PARASOL, except that the `<cr>` is required for every command. Then it

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

Figure 6.11: Messages Queued in *MailServer* and *Client4*

starts reporting events as they happen. Among other events, all instances of the clonable task are created by `genesis()` at the beginning and their icons appear in the PARASOL node.

Status and status history of selected device or tasks are displayed and updated automatically in the state chart of the debug window. For tracing various queues and running device statistics, users must turn to the object of interest. Device statistics are presented in their property popup windows. They are updated at the time they appear. Values of user defined statistic structures are shown in their icons at the model level. These values are changed as the simulation goes. Task queues and message queues are private to nodes and tasks, respectively. As vPARASOL always presents the model at its highest level, one needs to zoom in appropriate nodes or tasks for queue state. Queue state also changes as the simulated time goes. Figure 6.11 shows instances of the message queues of the mail server task and a mailer task. Messages displayed in the queues are represented by names of their sender tasks. From the queues, users can tell that messages are piled up at the mail server. And if there are enough mail users, a message queue can also be found at the "local" mailers. Since all other nodes contain only a single task, the only place where ready tasks can possibly be queued is in the PARASOL node. However, they are actually not queued due to their neglectable processing time.

Mail Users	1	2	10	40	100	200
Send Time	2.99	2.99	123.21	869.28	2381.12	4840.79
Send Count	12449	15622	15963	16097	15943	16005
Receive Time	38.07	69.98	318.38	1068.87	2579.95	5066.22
Receive Count	18747	23476	24221	24099	23877	24031

Table 6.1: Simulation Results of the Email System Model

6.5 Analyzing Results

Although this is a relatively simple model, it does exhibit some interesting behavior. For example, Table 6.1 suggests strong bottleneck somewhere in the system as both response times grow to large values as the number of users increases. This behavior for mail receive request is not unexpected due to the lengthy processing delays involved. However, the send response times are not materially better despite the fact that processing delays should be so small.

Use of vPARASOL as an aid to understanding system behavior is well illustrated by this example. Without such an aid, one might be tempted to remedy the problem

Server Speed	1.00	2.00	4.00	1.00	2.00	4.00
Mail Users	1	1	1	10	10	10
Send Time	2.99	2.97	2.98	123.21	110.24	104.84
Send Count	12449	13281	13904	15963	17648	18752
Receive Time	38.07	34.66	32.85	318.38	287.88	269.10
Receive Count	18747	20051	20758	24221	26734	28456

Table 6.2: Effects of Varying CPU Speed

by improving the processing speed of the mail server. While this helps, the saturation characteristic of send time remains (Table 6.2). In fact, by using vPARASOL's

Mail Users	1	2	10	40	100	200
Send Time	3.01	2.98	3.09	3.09	3.31	3.50
Send Count	11709	15525	15724	16000	15978	16082
Receive Time	41.34	71.14	397.91	1649.65	4130.00	8266.66
Receive Count	17692	23119	24198	24006	24084	24053

Table 6.3: Results of the Modified Email System Model

visualization features, one can see that the utilization of all processors as well as the bus is quite small. On the other hand, the standard port queues of each of the mailers tend to grow very quick with the number of users. This suggests a software bottleneck at mailer tasks. Furthermore, it explains the long send mail response times since most of the time is spent waiting for send mail requests to be received by mailers. Consequently a better approach to improving send mail performance is to redesign the mailer task. In the code shown below, a mailer does not delay user request message reception by waiting to receive mail from the mail server. Instead it uses a single port for both types of messages and distinguishes them by the type parameter. This approach is slightly complicated by the need to associate the mail user's reply port to the message when retrieving mail. This information is encoded in the "text" field of a request when passed to the mail server, along with the simulated mail, and used to address the latter to the appropriate user. Tables 6.1 and 6.3 show the dramatic difference in performance in send mail response times.

```
#define MAIL_PROC      3.0
#define MAIL_SIZE      40;

void
mailer_class()
{
    double  ts;                /* receive time stamp */
    int     type;              /* message type */
    int     rp;                /* reply port */
    char    *tp;               /* message pointer */
}
```

```
while (TRUE) {
    receive(MY_STD_PORT, NEVER, &type, &ts, &tp, &rp);
    compute(exponential(MAIL_PROC));
    switch (type) {
        case 0: /* receive */
            tp = (char *)malloc(MAIL_SIZE);
            sprintf(text, "%d", rp);
            bus_send(bus0, std_port(task2), type, MAIL_SIZE, tp, MY_STD_PORT);
            break;
        case 1: /* send */
            send(rp, type, tp, MY_STD_PORT);
            bus_send(bus0, std_port(task2), type, MAIL_SIZE, tp, MY_STD_PORT);
            break;
        case 2: /* reply from server */
            sscanf(tp, "%d", &rp);
            free(tp);
            send(rp, 0, tp, MY_STD_PORT);
            break;
    }
}
```

Chapter 7

Conclusions and Future Directions

7.1 A Final Look at vPARASOL

vPARASOL has achieved its design goals stated at the beginning of the project. All three of its functional components are designed and implemented. The graphical user interface provides a structured, visual representation of models and means to construct models graphically. Its automatic code generator produces framework code for models, which hides most tedious programming details from users. And the graphical execution monitor is a very useful aid in model debugging. Modelling is both more convenient and faster with the aid of vPARASOL.

Independence is ensured for models built with vPARASOL graphical user interface. Such models can run under vPARASOL or under the operating system without incurring any change. Models running within vPARASOL can take the full advantage of the graphical execution monitor while those running under the operating system will have the best efficiency for production simulation. They are complementary.

In terms of modifying PARASOL, it is impossible to leave the standard PARASOL kernel intact. However, an attempt has been made to hold changes in minimum. To summarize, two substantial changes have been made.

- A new flag, the vPARASOL flag, is introduced, and conditional branches are added so PARASOL senses this flag and acts accordingly. Most of the changes

were required in PARASOL trace reporting as the standard text output is not sufficiently structured.

- A source file, *para_vpara.c*, is added to the standard PARASOL package. This file isolates all changes but except the ones mentioned above. A rewritten debug driver is located in this file to accommodate new commands. And all new state reporters such as task trace reporter, link state reporter are also in this file. Also a header file, *para_vpara.h*, contains necessary definition of data structures and constants.

Since these modifications to PARASOL are comparatively few in number and well isolated for the most part, it is feasible to incorporate all of them into the "official" release of PARASOL without sacrificing performance. At present, there is a minor inconvenience to the vPARASOL user, i.e., the need of reforming the vPARASOL version of the library when the "standard" version is updated. Currently, the vPARASOL library is based on PARASOL Version 1.4, the latest "official" release.

7.2 Future Work

vPARASOL, like any other system, will undoubtedly evolve as it sees use. Most improvements and enhancements are likely to be concentrated in the vPARASOL execution monitor, as experience suggests worthwhile features. Few are expected in its graphical user interface.

Model source code reuse can save time in both coding and debugging. Among others, one possibility is to build task class libraries. The difficult part is that there is no easy way to make tasks refer to external entities since models are dynamically built but a library is rather stable. Editing task code within the GUI is attractive. But one should take into account that it is rare to write the correct code at the first attempt, and it is common for one piece of code need to refer to either variables or functions defined elsewhere.

Backward compatibility, or reverse engineering, is only partially solved with the current implementation. vPARASOL does have the capability to execute and control a PARASOL model not built with the vPARASOL graphical user interface, through its pipe interface. It does this by exchanging messages with the model at run time. What is lacking is that vPARASOL can not build an equivalent vPARASOL model for an existed PARASOL model. A possible solution is to parse the *genesis* code.

A more secure object mapping between vPARASOL and PARASOL can be built dynamically by catching each creation event including hardware devices and statistic structures, as it does for tasks now. This can not be done without first making PARASOL report these events.

Multiwindow debugging is also a desirable feature. vPARASOL uses a debug window and its main window when debugging. There are situations when entities in different levels or locations interact, in which several concurrent windows would be desired.

Event logging is another possible addition to the vPARASOL execution monitor. By logging all events in a model execution, simulation traces can be reproduced exactly for the purposes of debugging and analysis, without incurring the time and expense of rerunning a simulation. However, since most debugging runs are comparatively short, as compared with production simulation runs, computing time is not usually a major consideration for vPARASOL. Simulations are easy to be reproduced by rerunning.

Appendix A

Example Code Generated by vPARASOL

This appendix gives the framework code for the email example discussed in text, generated by vPARASOL. For completed task class code in file *email_tasks.c*, users should refer to chapter 6.

A.1 Header

```
/* **** */
/*
/*  email.cfg.h                                */
/*
/*  The .cfg.h file generated by vPARASOL V1.4      */
/*  for PARASOL simulation. It is a combination of  */
/*  header file and system configuration report.     */
/*
/*          DON'T EDIT THIS FILE BY HAND!          */
/*
/* **** */
/*
/* [Summary Section]
/*
/* Item          #Used          #Max
/* -----
/* Node          7              100
/* Bus           1              100
```

```

/* Link      0      1000      */
/* Task      8      1000      */
/* Port      8      1000      */
/* Statistic  2      ---      */
/* Class     4      100      */
/*
/* [Node Section]
/*
/* Name      ID      #CPU    Speed  Slice  Queue  Stats  #Task  */
/* -----
/* PARASOL   node0  1      1.00   1.00   FCFS   No     2      */
/* MailServer node1  1      1.00   1.00   FCFS   Yes    1      */
/* Client1   node2  1      1.00   1.00   FCFS   Yes    1      */
/* Client2   node3  1      1.00   1.00   FCFS   Yes    1      */
/* Client3   node4  1      1.00   1.00   FCFS   Yes    1      */
/* Client4   node5  1      1.00   1.00   FCFS   Yes    1      */
/* Client5   node6  1      1.00   1.00   FCFS   Yes    1      */

extern int    node0, node1, node2, node3, node4, node5;
extern int    node6;

/* [Bus Section]
/*
/* Name      ID      #Node  PSize  TRate  Queue  Stats
/* -----
/* BackBone  bus0    6      1500   1.25   FCFS   Yes

extern int    bus0;

/* [Link Section]
/*
/* Name      ID      SNode  DNode  PSize  TRate  Stats
/* -----

/* [Task Section]
/*
/* Name      ID      Node   CPU    Prior  Class      Clone
/* -----
/* genesis   task0   node0  ANY    999     genesis    No
/* UserTask   task1N  node0  ANY    10      UserTaskC  Yes
/* MailServer task2   node1  ANY    10      MailServerC No
/* Mailer1    task3   node2  ANY    10      MailerC    No

```

```

/* Mailer2      task4 node3 ANY 10 MailerC No */
/* Mailer3      task5 node4 ANY 10 MailerC No */
/* Mailer4      task6 node5 ANY 10 MailerC No */
/* Mailer5      task7 node6 ANY 10 MailerC No */

extern int      task0, task1N;
extern int      task2;
extern int      task3;
extern int      task4;
extern int      task5;
extern int      task6;
extern int      task7;

/* [Port Section] */
/* */
/* Name          ID          Task          Task-ID          */
/* ----- */
/* Mailer1P0     port0       Mailer1       task3             */
/* Mailer2P0     port1       Mailer2       task4             */
/* Mailer3P0     port2       Mailer3       task5             */
/* Mailer4P0     port3       Mailer4       task6             */
/* Mailer5P0     port4       Mailer5       task7             */

extern int      port0;
extern int      port1;
extern int      port2;
extern int      port3;
extern int      port4;

/* [Statistic Section] */
/* */
/* Type          ID          Name          */
/* ----- */
/* POP           ptr_sp0     Send          */
/* POP           ptr_sp1     Receive         */

extern stat_t   *ptr_sp0;
extern stat_t   *ptr_sp1;

/* [Miscellaneous Section] */

extern int      reaper_port;          /* std port of grim reaper */

```

A.2 Genesis

```

/*****
/*
/*  email_genesis.c
/*
/*  The _genesis.c file generated by vPARASOL V1.4
/*  for PARASOL simulation.
/*
/*
/*      DON'T EDIT THIS FILE BY HAND!
/*
/*
*****/

#include    <parasol.h>        /* <stdio.h> is included */

int         node0 = 0;         /* PARASOL node 0 */
int         node1, node2, node3, node4, node5, node6;
int         bus0;
int         task0, task1N = 1;
int         task2;
int         task3;
int         task4;
int         task5;
int         task6;
int         task7;
int         port0;
int         port1;
int         port2;
int         port3;
int         port4;
int         bus_nodes[7];     /* For bus building */
int         reaper_port;      /* For grim reaper */
stat_t      *time_sp0;         /* For TIME statistic Send */
stat_t      *time_sp1;         /* For TIME statistic Receive */

void
genesis()
{

```

```
int p_id, port;
double ts;
char *tp;
void MailServerC();
void MailerC();
void UserTaskC();
int i;

reaper_port = MY_STD_PORT;

/* Set up global Statistics */

open_stat(TIME, "Send", &time_sp0);
open_stat(TIME, "Receive", &time_sp1);

/* build nodes */

node1 = build_node("MailServer", 1, 1.00, 1.00, FCFS, TRUE);
node2 = build_node("Client1", 1, 1.00, 1.00, FCFS, TRUE);
node3 = build_node("Client2", 1, 1.00, 1.00, FCFS, TRUE);
node4 = build_node("Client3", 1, 1.00, 1.00, FCFS, TRUE);
node5 = build_node("Client4", 1, 1.00, 1.00, FCFS, TRUE);
node6 = build_node("Client5", 1, 1.00, 1.00, FCFS, TRUE);

/* build buses */

bus_nodes[0] = node1;
bus_nodes[1] = node2;
bus_nodes[2] = node3;
bus_nodes[3] = node4;
bus_nodes[4] = node5;
bus_nodes[5] = node6;
bus0 = build_bus("BackBone", 6, bus_nodes, 1500, 1.25e6, FCFS, TRUE);

/* build links */

/* Create tasks. Only STATIC tasks are distincted. */

task0 = 0; /* task genesis */

task2 = create("MailServer", node1, ANY_HOST, MailServerC, 10);
```

```
task3 = create("Mailer1", node2, ANY_HOST, MailerC, 10);
task4 = create("Mailer2", node3, ANY_HOST, MailerC, 10);
task5 = create("Mailer3", node4, ANY_HOST, MailerC, 10);
task6 = create("Mailer4", node5, ANY_HOST, MailerC, 10);
task7 = create("Mailer5", node6, ANY_HOST, MailerC, 10);

/* Allocate ports other than std-ports of tasks */

port0 = allocate("Mailer1P0", task3);
port1 = allocate("Mailer2P0", task4);
port2 = allocate("Mailer3P0", task5);
port3 = allocate("Mailer4P0", task6);
port4 = allocate("Mailer5P0", task7);

/* Resume tasks */

resume(task2);
resume(task3);
resume(task4);
resume(task5);
resume(task6);
resume(task7);

/* Create and resume DYNAMIC tasks(if any) */

for (i = 0; i < task1N; i++)
    resume(create("UserTask", node6, ANY_HOST, UserTaskC, 10));

/* Call v_monitor.  execute user defined task.  */

v_monitor();

/* Inform vPARASOL that genesis() is done.  */

genesis_done();

/* grim reaper */

while (TRUE) {
    receive(MY_STD_PORT, NEVER, &p_id, &ts, &tp, &port);
```



```

        kill(p_id);
    }

}

/* suicide function */

void
suicide()
{
    send(reaper_port, SELF, "", NULL);
    suspend(SELF);
}

```

A.3 Tasks

```

/***** *****/
/*                                                    */
/*  email_tasks.c                                    */
/*                                                    */
/*  The _tasks.c file generated by vPARASOL V1.4      */
/*  for PARASOL simulation.                          */
/*                                                    */
/*  No lines will be taken from this file after its  */
/*  creation, but some may be added.                */
/*                                                    */
/***** *****/

#include <parasol.h>
#include "email_cfg.h"

#define CLIENT_NODE      5
#define MEAN_THINK       8.0
#define SEND_PROB       0.4
#define MAIL_PROC       3.0
#define MAIL_SIZE       40
#define MEAN_PROC       5.0
#define DISK_ACCESS     20.0

```

```
main(argc, argv)
int argc;
char **argv;
{
    double run_duration;
    long run_seed;
    int run_flags;

    /* These two function calls insure proper piping. */

    fflush(stdout);
    setbuf(stdout, NULL);

    /* Acquire simulation parameters */

    switch (argc) {
        case 2:
            if (strcmp(argv[1], "-t") == 0)
                run_flags = FLAG_TRACE;
            else if (strcmp(argv[1], "-s") == 0)
                run_flags = FLAG_TRACE | FLAG_STEP;
            else if (strcmp(argv[1], "-v") == 0)
                run_flags = FLAG_TRACE | FLAG_STEP | FLAG_VPARASOL;
            else
                abort("Invalid PARASOL run-time flag");
        case 1:
            printf("Enter DURATION of Simulation run(double): ");
            scanf("%lf", &run_duration);
            printf("Enter Random Number SEED of Simulation run(long): ");
            scanf("%ld", &run_seed);
            break;
        case 4:
            if (strcmp(argv[1], "-t") == 0)
                run_flags = FLAG_TRACE;
            else if (strcmp(argv[1], "-s") == 0)
                run_flags = FLAG_TRACE | FLAG_STEP;
            else if (strcmp(argv[1], "-v") == 0)
                run_flags = FLAG_TRACE | FLAG_STEP | FLAG_VPARASOL;
            else
                abort("Invalid PARASOL run-time flag");
        case 3:
            sscanf(argv[--argc], "%G", &run_duration);
```

```
        if (run_duration <= 0.0)
            abort("Non-positive simulation duration");
        sscanf(argv[--argc], "%ld", &run_seed);
        break;
    default:
        abort("Invalid PARASOL run-time flag");
}

/* Model dependant code.  DON'T EDIT -- Start.  */

/* Ask for numbers of instances in each clonable task */

printf("Number of instances in UserTask, class code UserTaskC(int): ");
scanf("%d", &task1N);

/* ask for value of each parameter */

/* Model dependant code.  DON'T EDIT -- End.  */

/* Start simulation */

run_parasol(run_duration, run_seed, run_flags);
}

/* v_monitor is called by genesis.  It is designed */
/* for users to access genesis if required.  */

void
v_monitor()
{
}

/* All other task classes go here */

void
MailServerC()
{
    /* inform grim reaper to kill self */

    suicide();
}
```

```
void
MailerC()
{
    /* inform grim reaper to kill self */

    suicide();
}

void
UserTaskC()
{
    /* inform grim reaper to kill self */

    suicide();
}
```

Appendix B

Glossory of Symbols in vPARASOL

This appendix gives a summary of graphical symbols used in vPARASOL graphical user interface. Mainly, symbols explained here are used in the main window, since those in the debugging window are relatively straight forward.

The main window contain a title bar, a group of menus, a group of tablets, a few paint windows, system component pools and two footers(Figure 3.3). The title bar shows the vPARASOL version number and the name of the model under editing. Menus contain all the command available in vPARASOL. The tablets speed up the most frequently used commands(a subset of the menus). The large working window shows a portion of the working place. A small paint window depicts the relationship between the working place and the portion shown in the working window. The recycle bin is for discarded system components. To its right, there are various pools for each system component. The left footer displays informative messages and the right footer shows which part of the model is displayed in the working window.

Since the menus and the working place are self explanatory, only graphical symbols are explained below.

This button creates a new model. By default, all models start with the PARASOL node(*node 0*) built in and the genesis task created in the node. Properties

such as the number of cpus in node 0 and the priority of the genesis task are set to default values. If a model is currently loaded, vPARASOL will prompt user to save it before creating a new model, when necessary.

- Load Model

This button loads a model from a model file. All models are saved in model files(with the extension *.mdl*). Both model system components and environment are saved. A popup windows appears to allow retrieving the directory tree. If a model is currently loaded, vPARASOL will prompt user to save it before loading another model, when necessary.

- Save Model

This button saves a model to a model file. If it is a new, a window pops up to acquire a model file name.

- Zoom In

This button focuses the work place to the next lower level, i.e., from the model level to the node level or from node level to task level. A node or task must be selected first.

- Lock Working Level

This button locks the work space at the current level. Some commands behave differently under lock mode, such as double clicks.

- Zoom Out

This button moves the work space one level up.

- Code

This button generates framework code for the model under editing. It saves the model before it produces any code.

- Genesis

This button brings up a window through which users can change the behaviour of task *genesis*, such as creating a grim reaper or block statistic routine. Parameters can also be defined here so that they can be entered interactively during the model execution, as a model may need a few inputs.

- Run

This button executes the model in the debugging window. The debugging window will appear and take over to control the model execution.

- Replicate Object

There are a group of object replicating buttons. Not all of them work at all levels. Mostly, they are designed for duplicating nodes. Before a replicating operation, an object must be selected as the source.

- Base Replicate

Base replicating copies the node only. A new node with the same properties as the source node is created. Tasks inside the node are not copied.

Node property window is opened for editing the new node. Base replicate is the basis of all the rest replicating operations.

- Full Replicate

Full replicating does the same as base replicating, plus it will duplicate the tasks inside the node.

- Bus Replicate

Bus replicating connects the new node to the selected bus, in addition to the base replicate. It acts as a binary operator: select a node, press the button, and select a bus.

- Ring Replicate

Ring Replicate is the most complicated. It is a binary operator as well. A ring replicate inserts a new node to a ring. The connection medium is determined by the existing medium. In case of a link, the direction agrees with the original link, i.e., if the original link is pointing out of the source node, the new link will point out of the new node as well.

- Link Object

Link replicate is similar to the bus replicate. It will build a new link pointing from the source node to the newly created node.

- Star Replicate

Similar to the link replicate, but this time a new bus is created.

- Refresh Working Place

After moving or removing links, the working may result not as clean as it should be. The refresh button redraws the working window.

- Roadmap

In the roadmap window, the big rectangle represents the entire work space, while the small rectangle inside shows the position of current work window. Black blocks in the roadmap are nodes. It is possible to reposition the work window through moving the small rectangle.

- Recycle Bin

As the name suggests, the recycle bin collects all discarded objects. An object can be removed by dragging it from the working window and dropping it in the recycle bin.

- Node Pool

At the model level, nodes are built by dragging them from this node pool and dropping them at the appropriate places in the working window. Node property window pops up after dropping a node. Nodes can be moved but can not be overlapped.

- Task Pool

Tasks are created in one of two ways. At the model level, a task can be dragging from the task pool and dropping in a node. At the node level, it can be dropped anywhere in the work window. Tasks have default positions

according to the order in which they are created. Users can not change the position of a task(except by deleting and adding).

- Port Pool

The relationship between a port and a task is just the same as the one between a task and a node. Operations are similar as well.

- Bus Pool

Bus pool is more like a binary operator. When building a new bus, one selects a node, then clicks on the bus pool and finally selects a second node. If the operation is to add a node to an existing bus, one should select the bus first, instead of a node.

- Link Pool

Link pool functions more or less the same as the bus pool. Select the source node first, click the link pool and select the destination node. It is not possible to add nodes to an existing link since a link connects only a pair of nodes.

- Statistic Pool

Statistics are treated as nodes. They are independent objects and can not be overlapped.

- Navigator

The navigator provides fast access to the adjacent model portions, since the scroll bar operation is time consuming and the roadmap is less accurate. Pressing an arrow makes the display window shift to the corresponding neighbor.

References

- [ADAM] N. R. Adam, A. Dogramaci(ed), *Current Issues in Computer Simulation*, Academic Press, 1979
- [BALB] G. Balbo, G. Serazzi(ed), *Computer Performance Evaluation - Modelling Techniques and Tools*, Elsevier, 1992
- [DevGde] *OpenWindows Developer's Guide 3.0: Users' Guide*, Sun Microsystems, Inc., 1991
- [GORD] G. Gordon, *The Application of GPSS V to Discrete System Simulation*, Englewood Cliffs, 1975
- [GORD78] G. Gordon, *The Development of the general purpose simulation system(GPSS)*, ACM SIGPLAN Notice, 13(8), pp 183 -198, 1978
- [HELL] Dan Heller, *XView Programming Manual, Third Edition*, The Definitive Guides to the X Window System, Volume 7, O'Reilly & Associates, Inc., 1992
- [KHEI] Naim A. Kheir(ed), *System Modelling and Computer Simulation*, Marcel Dekker, 1988
- [KING] Peter J. B. King, *Computer and Communication Systems Performance Modelling*, Prentice Hall, 1990
- [KLEI] L. Kleinrock, *Queueing Systems Volume I: Theory*, Wiley, 1975

- [KIVI] P. J. Kiviat, R. Villanueva, H. M. Markowitz, *SIMSCRIPT II.5 Programming Language*, CACI Inc., Los Angeles, 1975
- [MACD] Myron H. MacDougall, *Simulating Computer Systems: Techniques and Tools*, The MIT Press, 1987
- [MARK] H. M. Markowitz, B. Hausner, H. W. Karr, *SIMSCRIPT: A Simulation Programming Language*, Prentice-Hall, Englewood Cliffs, 1963
- [MARS] M. Ajmone Marsan, G. Balbo, G. Conte, *Performance Models of Multiprocessor systems*, MIT, 1986
- [MARS90] M. Ajmone Marsan, *Stochastic Petri nets: an elementary introduction*, Advances in Petri Nets 1989, pp 1 - 29, Springer-Verlag 1990
- [NEIL] John E. Neilson, *PARASOL Users' Manual*.
- [NEIL91] John E. Neilson, *PARASOL: A Simulator for Distributed and/or Parallel Systems*, School of Computer Science, Carleton University, Technique report SCS-TR-192, May 1991.
- [PETE] J. L. Peterson, *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981
- [PETR] C. A. Petri, *Communication with Automata*, Ph. D. Thesis, Technical Report RADC-TR-65-377, New York, 1966
- [POOL] R. J. Pooley, *The Integrated Modelling Support Environment - A New Generation of Performance Modelling Tools*, Proceedings of the Fifth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, pp 1 - 15, February, 1992
- [PRIT] A. B. Pritsker, C. D. Pegden, *Introduction to Simulation and SLAM*, John Wiley & Sons, 1979

- [SAUE] Charles H. Sauer, K. Mani Chandy, *Computer Systems Performance Modelling*, Prentice Hall, 1981
- [WHIC] M. L. Whicker, L. Sigelman, *Computer Simulation Applications*, Sage, 1991
- [YOUN] Michael Young, Avadis Tevanian, Richard Rashid, etc. *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System*, System Proceedings of the 11th Symposium on Operating System Principles, November 1987.
- [ZEIG] Bernard P. Zeigler, *Theory of Modelling and Simulation*, John Wiley & Sons, 1976.
- [ZEIG79] B. P. Zeigler(ed), *Methodology in Systems Modelling and Simulation*, North-Holland 1979

Appendix B

Glossory of Symbols in vPARASOL

This appendix gives a summary of graphical symbols used in vPARASOL graphical user interface. Mainly, symbols explained here are used in the main window, since those in the debugging window are relatively straight forward.

The main window contain a title bar, a group of menus, a group of tablets, a few paint windows, system component pools and two footers(Figure 3.3). The title bar shows the vPARASOL version number and the name of the model under editing. Menus contain all the command available in vPARASOL. The tablets speed up the most frequent used commands(a subset of the menus). The large working window shows a portion of the working place. A small paint window depicts the relationship between the working place and the portion shown in the working window. The recycle bin is for discarded system components. Follows are various pools for each system component. The left footer displays informative messages and the right footer shows which part of the model is displayed in the working window.

Since the menus and the working place are self explained, only graphical symbols are explained.

- New Model

This button creates a new model. By default, all models start with the PARASOL node(*node 0*) built in and the genesis task created in the node. Properties

such as the number of cpus in node 0 and the priority of the genesis task are set to default values. If a model is currently loaded, vPARASOL will prompt user to save it before creating a new model, when necessary.

- Load Model

This button loads a model from a model file. All models are saved in model files(with the extension *.mdl*). Both model system components and environment are saved. A popup windows appears to allow retrieving the directory tree. If a model is currently loaded, vPARASOL will prompt user to save it before loading another model, when necessary.

- Save Model

This button saves a model to a model file. If it is a new, a window pops up to acquire a model file name.

- Zoom In

This button focuses the work place to the next lower level, i.e., from the model level to the node level or from node level to task level. A node or task must be selected first.

- Lock Working Level

This button locks the work space at the current level. Some commands behaves differently under lock mode, such as double click.

- Zoom Out

This button moves the work space one level up.

- Code

This button generates framework code for the model under editing. It saves the model before it produces any code.

- Genesis

This button brings up a window through which users can change the behavior of task *genesis*, such as creating a grim reaper or block statistic routine. A few parameters can also be defined here so that they can be entered interactively during the model execution, as a model may need a few inputs.

- Run

This button executes the model in the debugging window. The debugging window will appear and take over to control the model execution.

- Replicate Object

There are a group of object replicating buttons. Not all of them work at all levels. Mostly, they are designed for duplicating nodes. Before a replicating operation, an object must be selected as the source.

- Base Replicate

Base replicating copies the node only. A new node with the same properties as the source node is created. Tasks inside the node are not copied. Node property window is opened for editing the new node. Base replicate is the basis of all the rest replicating operations.

- Full Replicate

Full replicating does the same as base replicating, plus it will duplicate the tasks inside the node.

- Bus Replicate

Bus replicating connects the new node to the selected bus, in addition to the base replicate. It acts as a binary operator: select a node, press the button, and select a bus.

- Ring Replicate

Ring Replicate is the most complicated. It is a binary operator as well. A ring replicate inserts a new node to a ring. The connection medium is determined by the existing medium. In case of a link, the direction agrees with the original link, i.e., if the original link is pointing out of the source node, the new link will point out of the new node as well.

- Link Object

Link replicate is similar to the bus replicate. It will build a new link pointing from the source node to the newly created node.

- Star Replicate

Similar to the link replicate, but this time a new bus is created.

- Refresh Working Place

After moving or removing links, the working may result not as clean as it should. The refresh button redraws the working window.

- Roadmap

In the roadmap window, the big rectangle represents the entire work space, while the small rectangle inside shows the position of current work window. Black blocks in the roadmap are nodes. It is possible to reposition the work window through moving the small rectangle.

- Recycle Bin

As the name suggests, the recycle bin collects all discarded objects. An object can be removed by dragging it from the working window and dropping it in the recycle bin.

- Node Pool

At the model level, nodes are built by dragging them from this node pool and dropping them at the appropriate places in the working window. Node property window pops up after dropping a node. Nodes can be moved but can not be overlapped.

- Task Pool

Tasks are created in one of two ways. At the model level, a task can be dragging from the task pool and dropping in a node. At the node level, it can be dropped anywhere in the work window. Tasks have default positions according to the order in which they are created. Users can not change the position of a task(except by deleting and adding).

- Port Pool

The relationship between a port and a task is just the same as the one between a task and a node. Operations are similar as well.

- Bus Pool

Bus pool is more like a binary operator. When building a new bus, one selects a node, then clicks on the bus pool and finally selects a second node. If the operation is to add a node to an existing bus, one should select a bus first, instead of a node.

- Link Pool

Link pool functions more or less the same as the bus pool. Select the source node first, click the link pool and select the destination node. It is not possible to add nodes to an existing link since a link connects only a pair of nodes.

- Statistic Pool

Statistics are treated as nodes. They are independent objects and can not be overlapped.

- Navigator

The navigator provides fast access to the adjacent model portions, since the scroll bar operation is time consuming and the roadmap is less accurate. Pressing a arrow makes the display window shift to the corresponding neighbor.

References

- [ADAM] N. R. Adam, A. Dogramaci(ed), *Current Issues in Computer Simulation*, Academic Press, 1979
- [BALB] G. Balbo, G. Serazzi(ed), *Computer Performance Evaluation - Modelling Techniques and Tools*, Elsevier, 1992
- [DevGde] *OpenWindows Developer's Guide 3.0: Users' Guide*, Sun Microsystems, Inc., 1991
- [GORD] G. Gordon, *The Application of GPSS V to Discrete System Simulation*, Englewood Cliffs, 1975
- [GORD78] G. Gordon, *The Development of the general purpose simulation system(GPSS)*, ACM SIGPLAN Notice, 13(8), pp 183 -198, 1978
- [HELL] Dan Heller, *XView Programming Manual, Third Edition*, The Definitive Guides to the X Window System, Volume 7, O'Reilly & Associates, Inc., 1992
- [KHEI] Naim A. Kheir(ed), *System Modelling and Computer Simulation*, Marcel Dekker, 1988
- [KING] Peter J. B. King, *Computer and Communication Systems Performance Modelling*, Prentice Hall, 1990
- [KLEI] L. Kleinrock, *Queueing Systems Volume I: Theory*, Wiley, 1975

- [KIVI] P. J. Kiviat, R. Villanueva, H. M. Markowitz, *SIMSCRIPT II.5 Programming Language*, CACI Inc., Los Angeles, 1975
- [MACD] Myron H. MacDougall, *Simulating Computer Systems: Techniques and Tools*, The MIT Press, 1987
- [MARK] H. M. Markowitz, B. Hausner, H. W. Karr, *SIMSCRIPT: A Simulation Programming Language*, Prentice-Hall, Englewood Cliffs, 1963
- [MARS] M. Ajmone Marsan, G. Balbo, G. Conte, *Performance Models of Multiprocessor systems*, MIT, 1986
- [MARS90] M. Ajmone Marsan, *Stochastic Petri nets: an elementary introduction*, Advances in Petri Nets 1989, pp 1 - 29, Springer-Verlag 1990
- [NEIL] John E. Neilson, *PARASOL Users' Manual*.
- [NEIL91] John E. Neilson, *PARASOL: A Simulator for Distributed and/or Parallel Systems*, School of Computer Science, Carleton University, Technique report SCS-TR-192, May 1991.
- [PETE] J. L. Peterson, *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981
- [PETR] C. A. Petri, *Communication with Automata*, Ph. D. Thesis, Technical Report RADC-TR-65-377, New York, 1966
- [POOL] R. J. Pooley, *The Integrated Modelling Support Environment - A New Generation of Performance Modelling Tools*, Proceedings of the Fifth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, pp 1 - 15, February, 1992
- [PRIT] A. B. Pritsker, C. D. Pegden, *Introduction to Simulation and SLAM*, John Wiley & Sons, 1979

- [SAUE] Charles H. Sauer, K. Mani Chandy, *Computer Systems Performance Modelling*, Prentice Hall, 1981
- [WHIC] M. L. Whicker, L. Sigelman, *Computer Simulation Applications*, Sage, 1991
- [YOUN] Michael Young, Avadis Tevanian, Richard Rashid, etc. *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System*, System Proceedings of the 11th Symposium on Operating System Principles, November 1987.
- [ZEIG] Bernard P. Zeigler, *Theory of Modelling and Simulation*, John Wiley & Sons, 1976.
- [ZEIG79] B. P. Zeigler(ed), *Methodology in Systems Modelling and Simulation*, North-Holland 1979

END

1 | 9-07-94

FIN