A Modeller's Workbench: Experiments In Object-Oriented Simulation Programming

Dr. Wolfgang Kreutzer; Computer Science Dpim. University Of Canterbury, New Zealand
UUCP: {watmath, mcvax, munnari} ! cantuar ! wolfgang

Abstract:

The research reported in this paper is part of an ongoing effort to explore potential benefits of using new software technologies for various classes of system simulation. Queueing network scenarios have been chosen as the first area of application. Our experiences in the use of two object-oriented simulators are described, using a simple example.

Pose is a Scheme based queueing network simulator. It demonstrates the suitability of symbolic languages and exploratory programming for system simulation. Some characteristics of window-based and graphical programming environments are then briefly discussed, with reference to a Smalltalk-based simulation tool.

The final chapter suggests that object-oriented simulation languages embedded in interactive modelling environments hosted on powerful workstations may well offer major breakthroughs in terms of user acceptance. The bandwidth of user/tool interfaces should be as wide as possible, drawing on modern techniques for graphical interaction and multi-process systems supporting the 'desktop' metaphor. Use of Smalltalk permits quick and easy exploration of design alternatives through rapid prototyping. Embedding such tools in Scheme preserves their functionality while making them more accessible to a wider community. Computational efficiency, while a lesser concern to 'modelling for insight', remains unsatisfactory in simulation for quantitative predictions. This problem may hopefully be overcome through future advances in software and hardware technologies.

1. Tools & Techniques of Simulation Programming

The term **"simulation"** is used for a wide variety of methods for computational animation of a descriptive model. Any exploration of states and patterns of behaviour under a chosen experimental frame is a simulation experiment in this sense.

A "good" **model** is always the simplest one which may still be justified. Analytical models are extremely useful as long as questions are simple and models can be properly validated. There are, however, a great number of more complex situations for which simulation is the only applicable method of analysis. Often no other techniques exist to explore strongly interconnected and irregular systems. Also, direct experimentation with a real system may not be possible; i.e. it may be non-existent, unobtainable, too dangerous or costly.

One of the most important motivations of model building is rooted in a desire to predict a system's likely response, so that appropriate actions can be taken to fulfill a given objective. Depending on the required degree of accuracy this leads to the use of models; either to aid in selection between a range of a priori alternatives, or as vehicles for thought experiments. In the first case there is an emphasis on generating sufficiently precise quantitative information. This is the classical style of modelling, as use, for instance, in operations research. Its successful application is critically dependent on acceptable model validations and skillful variation of policy parameters. This will always remain an expensive tool, because each simulation run yields only a single point on a model's response surface and, if stochastic components are involved, has to be replicated a large number of times to establish statistical credibility. "Modelling for insight" as an alternative approach, strives less for numerical solutions, but for an improved understanding of complex systems. It supports a more exploratory, speculative style of model analysis and may reap the full benefits of new technological developments. Under this approach relatively little need to be known about a system. A well defined problem may not even exist. We are using modelling at a proto-theoretical stage, striving for a better intuition about a system's behaviour, essential aspects, sensitivities ... which we may gain in the process.

AI-miniworlds are simulations in this sense. Many people believe that such a computational exploration of symbol structures will soon overshadow the traditional use of simulation as an aid to decision making.

One important facet of the simulation method is the fact that it is an experimental technique, comprising aspects of model calibration and data collection as well as experimental designs and output analysis. This is often considered the "scientific" aspect of simulation methodology. Statistical methods can answer questions on how to start and stop simulation runs, and how to analyze their results ([Fishman 1978], [Bratley et. al. 1983]). A complementary aspect of computer-aided simulation revolves around the activities of model and program design, implementation and verification. At the center of this is the art of **simulation programming**. In this area a number of prototypical modelling frameworks have evolved over a period of almost fourty years (see **figure 1**), with a multitude of tools to support them [Kreutzer 1986].

The programming tools discussed in this paper are restricted to queueing network scenarios, although we believe that basic principles and conclusions hold true for other kinds of modelling activity. **Queueing network scenarios** model systems of capacity-constrained resources and the effects of different allocation and scheduling strategies on performance measures such as utilization, throughputs, and time delays. This class of models has a long history, including job shop simulations and, more recently, computer system performance analysis. Basic data types perceived in this domain include resources and transactions (often called machines and materials), data collection devices and distributions. Entities for process

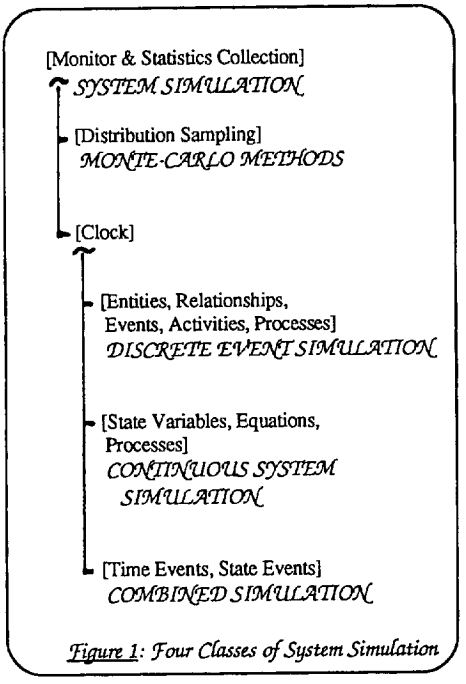

Figure 1: Four Classes of System Simulation

synchronization (i.e. queues) are also required. A great number of different programming languages have been developed to support their design and implementation. Traditionally a distinction is drawn between event, activity and process oriented world views, with the merits of process orientation becoming almost universally accepted.

Tools and notations are not independent of the model building methodologies by which they are employed; one serves to shape the perception of the other. Simulation programming's main concern has always focussed on structuring complexity. The better of theses tools strive for control of complexity through conceptual closure and "familiarity" to the task domain, in order to reduce the conceptual distance between perceptions of "relevant" parts of a system and their implementation in a program. *GPSS* and *DEMOS* are two of the most popular languages for modelling queueing network scenarios. Well designed scenario languages can graft concepts and terminology of specific application areas onto effective and user-friendly modelling frameworks, although it should be stressed that any specialized tool can also be misapplied.

GPSS (General Purpose System Simulator) [Bobillier et. al. 1976] is a good example. It is a very convenient tool for representation of material-oriented queueing networks; i.e. models in which material objects ("transactions") are the only "active" class of entities, from whose perspective all changes of state are defined. It does, however, lack the flexibility to cope with other, more complicated scenarios.

Demos (Discrete Event Modelling on Simula) [Birtwistle 1979] is a much more flexible simulator, embedded in a general purpose programming language (Simula), extending iit through specialized data types and operations carrying the main concepts of queueing network scenarios. Demos offers distribution sampling, automatic statistics collection and report generation, a process-oriented framework for modelling system dynamics and predefined templates for different classes of entities (i.e. queues and resources). When prefixed accordingly (**Demos** BEGIN ... END;) any Simula programs can use these features, which may also be modified if the need arises. Demos encourages the use of **life cycle diagrams** [Hutchinson 1975] as a graphical structuring device.

2. "The Garden Party" - a Simple Queueing Model

Suppose that we are interested in the likely degree of contention expected during a Queen's garden party. Relevant objects may therefore include a number of loyal royal subjects wishing to shake a royal hand and, of course, a Queen providing this service.

Figure 2 shows a life cycle description of a simple model of such an event. Flows between states and activities define life cycles for classes of entities. This scenario depicts an "open" (i.e.loyal subjects are temporary entities, entering through a "source" and leaving through a "sink") and material-oriented approach (i.e. temporary entites (loyal subjects) are active in seeking the Queen's favours, while she (passively !) only allows her hand to be grabbed and released). Let us further assume that the Queen's handshake be uniform with a duration of 0.5 to 1.5 minutes, while royalist's arrivals are negative exponentially distributed at a rate of 0.5 per minute. Once they have been shaken, 70 % of all subjects leave immediately, while 30% long to repeat the experience. The only relevant statistics is a tally of "time spent in garden" across all royal subjects.

Program 1 shows a corresponding Demos model, in which 3 classes of objects are defined. In the program body these are instantiated with appropriate values and bound to previously declared identifiers. A *Source* called *ArrivalProcess* is used to generate a steady stream of subjects, sampling their interarrival-time from the relevant distribution. Royal subjects themselves are created as images of the *RoyalSubject* process class definition, defining all actions of the life cycle shown in figure 2. Their last statement activates *Sink* object *ThatsAllItTook*, which performs appropriate updates on flowtime statistics. These are gathered throughy a *Tally*, a device for collecting non-weighted information about time series. *Hold* statements model the passage of model time, with *Schedule* commands used to activate or reactivate concurrent object (class instances).

Measures of queue length and server utilization are part of a standard report, shown in **figure 3** for a simulation of 100 time units (minutes). From this we deduce that we can expect a typical subject to spend about 1.5 minutes in the garden.

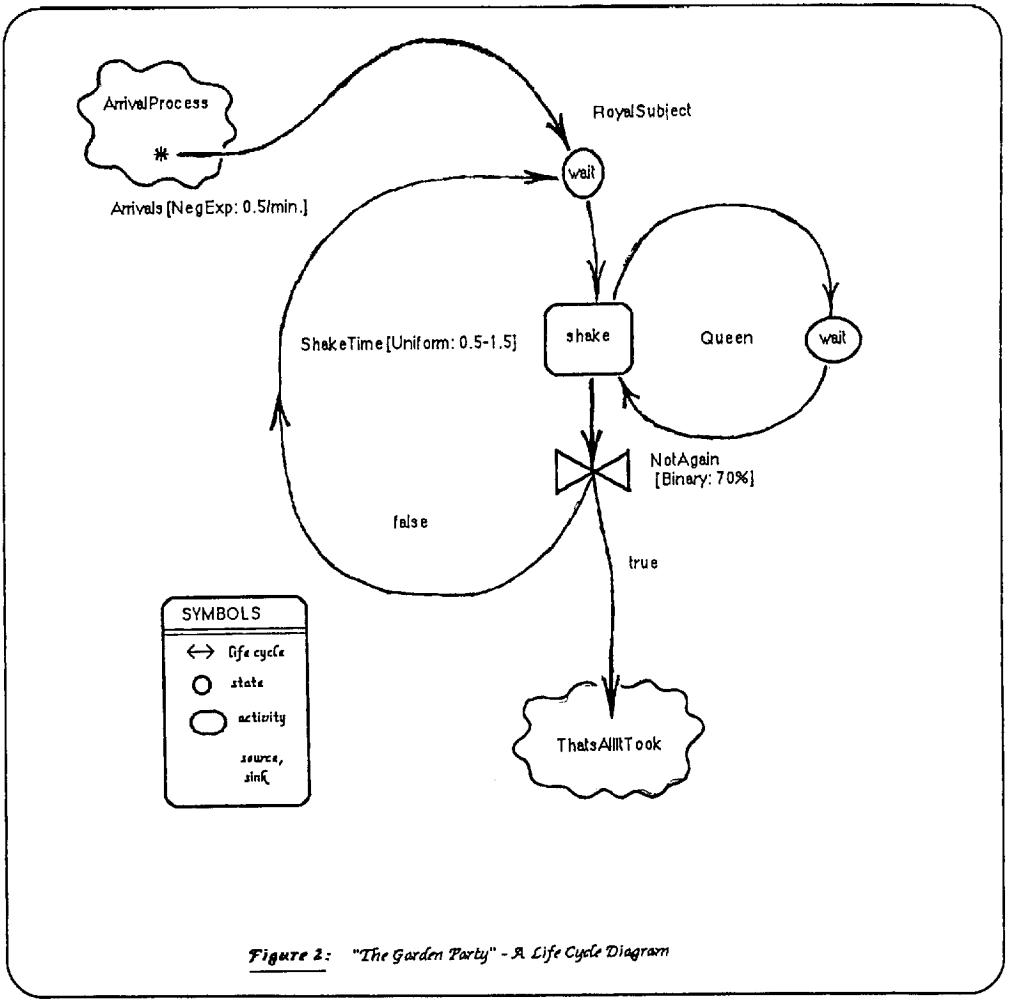

Figure 2: "The Garden Party" - A Life Cycle Diagram

3. Two Experiments in Object-Oriented System Simulation.

In the author's opinion, process oriented scenarios embedded in some flexible general purpose base language currently offer the most useful modelling tools within the framework of traditional programming methodology. Large sectors of different simulation communities have now finally accepted process orientation as the most convenient modelling world view. The benefits of closure and strong modularization, as offered by abstract data types or Simula's class concept, still have to gain sufficiently strong footholds, although the resultant ease of mapping all relevant structural and behavioural aspects of real world objects into conceptual entities can hardly be disputed.

The main drawbacks of a tool like Demos lie in the fact that it is rooted in traditional programming methodology and therefore only ill suited to more interactive and exploratory styles of systems analysis. The belief that one should always start from precise specifications, using formal deduction to arrive at a working program has long dominated "main stream" computer science. From this viewpoint programming is similar to theorem proving; "*program development and proof should go hand in hand*". The merits of this approach are convincingly demonstrated in [Wulf et. al. 1984], and Dijkstra, Hoare and Gries are some of its most vocal proponents. It works well for problems whose properties are well understood and for which rigorous specifications can be derived prior to the coding stage. In keeping with mathematical tradition it is predominantly concerned with justifying the correctness of implementations and gives little guidance to the process of discovery. "Traditional" methodology advises us to avoid applications with ill defined specifications, thereby excluding many "interesting" problems from our range. We should first gain a sufficiently precise understanding of a system before we may try to program it. But how can such understanding be obtained ? Traditional methodology is silent on this issue.

This framework has proven too restrictive for many modelling applications, which are experimental and exploratory in nature. System identification, problem specification, model implementation, verification and

```
BEGIN

EXTERNAL CLASS demos;

demos BEGIN
 REF (Source) ArrivalProcess;
 REF (Res    ) Queen;
 REF (Sink   ) ThatsAllItTook;
 REF (Rdist  ) ShakeTime;
 REF (Bdist  ) NotAgain;

 Entity CLASS Source;
 BEGIN
  REF (Rdist ) Arrivals;
 Arrivals :- NEW NegExp ("Arrivals",0.5);
 WHILE TRUE DO
  BEGIN
  NEW RoyalSubject("Hilda"). Schedule (Now );
  Hold (Arrivals.Sample );
  END while;
 END;

 Entity CLASS RoyalSubject;
 BEGIN
  BOOLEAN hooked;
  REAL    timeOfEntry;
 hooked      := TRUE;
 timeOfEntry := Time ;
 WHILE hooked DO
  BEGIN
  Queen.Acquire (1);
  Hold (ShakeTime.Sample );
  Queen.Release (1);
  hooked := NOT (NotAgain.Sample );
  END while;
 ThatsAllItTook.quitter :- THIS RoyalSubject;
 ThatsAllItTook.Schedule (Now );
 END;

 Entity CLASS Sink;
 BEGIN
  REF (Tally          ) flowTime;
  REF (RoyalSubject) quitter;
 flowTime :- NEW Tally ("ItTook");
 WHILE TRUE DO
  BEGIN
  flowTime.Update (Time - quitter.timeOfEntry);
  Cancel ;
  END while;
 END;

COMMENT: Body of MAIN program;

OutF :- NEW OUTFILE ("GardenParty.out");
OutF .OPEN (BLANKS (80));

Queen         :- NEW Res     ("Queen",1);
ShakeTime     :- NEW Uniform ("Shakes",0.5,1.0);
NotAgain      :- NEW Draw    ("NotAgain",0.7);
ArrivalProcess :- NEW Source ("Source");
ThatsAllItTook :- NEW Sink   ("Sink");
ArrivalProcess.SCHEDULE (NOW);

Hold (100); COMMENT: Simulate for 100 time units!;

END of Demos context;
END;
```

Program 1: Garden party in DEMOS

validation are closely intertwined. Initially, precise specifications are unknown. Simulation model building must therefore be an inductive, not a deductive process. It is more closely related to experimentation than to theorem proving. A different programming methodology, which views computers as tools for heuristic exploration of complex symbol systems, is therefore required. It must be possible to quickly build, test and modify prototypes and gain an improved understanding in the process. Since control of program complexity has become one of the main hurdles for many modelling applications, we need good tools to utilize the computer's potential to reduce a programmer's memory load by freeing her from any essentially "mechanical" chores. It must always be possible to change our minds without undue penalty in intellectual complexity. Recent advances in hardware technology have made machine efficiencies comparatively less relevant. The programmer needs all the help she can get; never mind the expense.

These convictions have prompted us to explore the potential of using exploratory programming tools and environments for system simulation, focussing on queueing network scenarios as a first paradigmatic class of applications {see [Astroem/Kreutzer 1986] for a parallel approach to control systems analysis}. Software tools to support this style of man/computer interaction have been built and refined by the Artificial Intelligence community since the early sixties. They are typically hosted in sophisticated interactive programming environments and emphasize ease of representation and manipulation of dynamic symbol structures. Intelligent browsers, editors, interpreters, compilers, debuggers, optimizers, tools for program instrumentation and project management aid in the manipulation of textual and graphical information. Multiple concurrent processes and some variant of the "desktop" metaphorare usually also supported.

There are various reasons why **exploratory programming** and its associated software tools are now attracting wider attention. Firstly, the advent of sufficiently powerful, self-contained workstations makes truly interactive programming styles possible and tempers much of the traditional concerns with machine efficiencies. Secondly, many of the "easy" computer applications have already been implemented. Since ambition always seems to expand to tax the limit of current technology there is a need for better support of the programming process itself, to enable us to cope with higher levels of complexity in a more reliable fashion.

Although user communities, methodologies and terminology have traditionally been almost completely independent from one another, many programming aspects of artificial intelligence and simulation model building are closely related. This is particularly obvious in some "rediscoveries"; i.e. the similarity between the activity scanning approach and production system interpreters. The importation of sophisticated tools and programming environments to the field of system simulation, grafted on an object oriented approach to program development, seems a particularly promising way to lighten the programmer's burden to remain in control of complex models' evolution.

The "*Modeller's workbench*" project at the University of Canterbury seeks to pursue this objective. In a number of experiments we will study the benefits of using

```
                    CLOCK TIME = 100.00

          ************************************
          *                                  *
          *              REPORT              *
          *                                  *
          ************************************


                        DISTRIBUTIONS
                        *************

TITLE    /   (RE)SET /  OBS  / TYPE    /  A    /  B    /    SEED
Shakes          0.000     76   UNIFORM   0.500   1.000     33427485
Notagain        0.000     75   DRAW      0.700             22276755
Arrivals        0.000     56   NEGEXP    0.500             46487980


                          TALLIES
                          *******

TITLE    /   (RE)SET /  OBS  / AVERAGE /EST.ST.DV/ MINIMUM / MAXIMUM
ItTook          1.295     53    1.507     0.926      0.500     4.561


                        RESOURCES
                        *********

TITLE    /   (RE)SET /  OBS  /LIM/MIN/NOW/ %USAGE /AV.WAIT/ QMAX
Queen           0.000     75    1   0  0    55.296   0.343     3
```

Figure 3: The Garden Party - Results

symbolic, interactive programming languages embedded in multi-window, reactive programming environments for various paradigms of system simulation [Kreutzer 1986]. A second phase will then apply these experiences to tools and techniques for model design by visual interaction, and aid in their analysis through animated symbolic scenarios.

This approach is predicated on the assumptions that object oriented programming and exploratory programming styles offer extremely appropriate metaphors for system simulation. The first achieves a close correspondence between model and reality as well as potential increases in model reliability through strong localization of information. Property inheritance, furthermore, allows the cognitive economy of "differential" system description; specifying new objects by enumerating how they differ from those which are already known. The second belief hinges on a desire to shorten the cycle of definition, testing, exploration, validation and modification of prototype models. This permits us to gain sufficiently "deep" understanding of representations and experiments appropriate for "relevant" aspects of some problem situation. The following two prototype systems were built to gather empirical evidence for this theory.

Pose [Stairmand 1987] is an object oriented queueing network simulator built on top of Scheme [Abelson et. al. 1985]; a member of the Lisp family. In the author's opinion **Scheme** is a small and compact, but also an extremely expressive and elegant language. This elegance stems from a small core of simple, orthogonal and very flexible features. Scheme supports lexical scoping, explicite declaration of all objects, closures with persistent local variables, and makes it easy to experiment with new control structures. Lambda expressions are "first class objects", which means that they can be assigned, passed or returned as parameters. Using these features, object orientation, message passing, property inheritance and coroutines can quickly be grafted onto the language kernel.

Scheme is an interactive language, supporting an exploratory programming style. Structure editors and sophisticated debugging tools are often provided as part of the programming environments in which it is embedded. Structure editors can go a long way to remove the drawbacks of its sparse syntax.They ease program design and debugging, in spite of Lisp's proverbial jungle of parentheses.

A stylistically good Scheme program should be built in levels, as demonstrated by Allen (1978) and Abelson et. al. (1985). This methodology draws on sound engineering principles and can equally easily be used to structure the

process of model identification, definition and implementation. By encapsulating and localizing information as tightly as possible, object orientation carries this idea one step further. After identifying relevant levels, all primitives of a given level (objects and operations; state variables and transformations; entities and actions) should be defined. These are then implemented through functions to construct instances, select components, modify representations, evaluate predicates and display objects.

Pose provides features for modelling queueing network scenarios which are very similar to those of Demos. Chez Scheme [Dybvig 1986] is used as a base. Although this is transparent to a naive user of the system, it is built on top of a flavours package implemented with a Smalltalk-like class system to support object oriented programming. **Program 2** shows our garden party example.

```
(DefMonitor GardenParty)

(Define-Process RoyalSubject
  ((hooked #!TRUE)) GardenParty
  (WHILE hooked
        (Queen 'Acquire 1)
        (Self 'Hold (ShakeTime 'Sample ))
        (Queen 'Release 1)
        (WHEN (NotAgain 'Sample )
              (SET! hooked #!FALSE)) )
  (ThatsAllItTook 'Swallow ) )

(DefBinary  NotAgain      0.7  12345)
(DefNegExp  Arrivals      0.5  67891)
(DefUniform ShakeTime  0.5  1.5  23457)

(DefRes     Queen         1  GardenParty)
(DefSink    ThatsAllItTook   GardenParty)
(DefSource  ArrivalProcess
  RoyalSubject Arrivals '#!TRUE GardenParty)

; simulate this scenario for 100 time units
  (GardenParty 'DURATION 100)
```

Program 2: Garden Party in POSE

The concepts of Monitor, Entity and Process form the top of a hierarchy of object classes. Distributions, Sources, Resources, Bins, WaitQs, CondQs and Sinks are currently implemented as subclasses. Statistics collection and reporting is automatic, although special data collection objects (i.e. Tally, Accumulate, Histogram) may easily be implemented.

Monitor objects are used to control simulation experiments. Normally there is only one monitor, responsible for model execution. This is, however, not a necessary restriction. The notion of simulation monitors is a recursive one and multiple layers of monitors (i.e. for controlling experiments across a number (or instatiations) of models, models containing models as subcomponents, ...) can also be accomodated. Among other relevant information each monitor owns a counter (sampler or clock), an agenda of scheduled tasks, a list of lists of model entities of various classes, and a reference to the currently active process. Instances are created by the *DefMonitor* macro. They may then respond to a number of predefined messages.

Distributions use a multiplicative congruential simulator [Knuth 1969] to obtain samples from a 0 to 1 uniform random number distribution , with appropriate transformations to other distributions delegated to subclasses. All of these respond to appropriate *Sample* messages. **Uniform, NegExp, Normal** and **Binary** distributions are currently implemented as subclasses. Note that distributions are conceived as global objects and are independent of any specific monitor.

Res is a subclass of **entity**, modelling capacity-constrained resources. Its instances respond to *acquire, release* and *report* messages. **Bins, WaitQs** and **CondQs** are also patterned after the corresponding Demos concepts. **Sinks** will report flowtime statistics of processes. For this purpose each process is automatically time-stamped at the time it is created by a source and should be swallowed by a sink just before it leaves the model.

Processes contain basic mechanisms (implemented via the Scheme function "*call/cc* ") for detaching, resuming and scheduling. They respond to *monitor, startTime, timeStamp!, hold* and *schedule* messages. *hold* is predominantly used by a process to delay its own progress (i.e. self 'hold 10), while *schedule* is used to influence other processes' evolution. A monitor is responsible for coordinating execution of all processes it is associatd with. **Sources** are a special subclass of processes. They have a predefined body of actions, which causes them to generate a new process of the appropriate type in intervals sampled from the associated distribution, until a simulation terminates or its attached condition evaluates to false.

Pose is still in an experimental stage. Preliminary explorations of various simple examples have shown it to be conceptually just as powerful as Demos, with important advantages regarding flexibility. The interactive environment permits a significant speedup in model development time. Execution efficiency remains a problem. While incremental definition and modification of models is fast, they are slow to execute. The reason for this lies in the multiple levels of interpretation imposed by the methodology of layered design and in the fact that Chez Scheme, while generally a remarkably fast and compact implementation, can currently not compete with good optimising compilers for "classical" programming languages. Currently this certainly impedes the use of Pose for quantitative decision making. Future advances in implementation techniques and hardware technology will hopefully make this a less relevant concern. As a first step towards this goal, an "intelligent" back-end compiler could be employed to automatically transform a model into a fast executable representation once its design has stabilized.

Our second experiment has focussed on the synergistic benefits of using object oriented programming styles in a more sophisticated modelling environment.

We should be free to specify, design, implement and experiment with a model, alternating between different activities at will and, if convenient, leaving processes in some intermediate stage of execution. This is typical for the style most people use when working at their desks. Several tasks may be relevant to particular projects and may need to be interleaved if the creative process is not to be inhibited. Modelling environments predicated on this idea may improve both productivity and quality of the model building process by an order of magnitude.

The *Smalltalk* system [Goldberg & Robson 1983] seemed the most appropriate vehicle to testthe suitability of this metaphor for system simulation. There are two aspects to Smalltalk, its programming paradigm and its programming environment. Smalltalk's programming paradigm centers on the concepts of classes, inheritance hierarchies and message passing, while its programming environment features a multi-process implementation of the "desktop metaphor" with windows, mouse-driven user interaction and browsers. It has sometimes been claimed that the merits of Smalltalk's programming paradigm and programming environment are two orthogonal and largely unrelated issues. Our experiences do not confirm this conjecture. It rather seems that the full benefits of object oriented systems can only be obtained in an appropriate environment containing source code inspection, debugging, analysis and project management tools. The reasons for this may be found in the fact that object oriented programming systems are large, with many predefined objects available for reuse and modification. It may be claimed that its typical programming style shifts emphasis from creation of new to skillful modification of existing pieces of code. Object oriented programs also tend to contain many small method definitions. Appropriate tools for navigation, inspection and modification of information are therefore vital; even more so than in conventional programming projects. Such tools must be tightly integrated into the editing and execution process. Smalltalk's concepts of browsers, debuggers and inspectors cater for these requirements.

We have used the simulator described in chapter 3 of the "blue book" [Goldberg & Robson 1983] as our initial point of departure. This system has been implemented in PS-Smalltalk on a Sun-3 workstation and augmented with various data collection devices [Irwin 1986]. In its current form it comes again very close to the Demos system, containing a multitude of classes of probability distributions, a simulation monitor, consumable and non-consumable resources, process synchronization and statistics collection entities.

```
Simulation subclass: #MeetTheQueen
instanceVariableNames: 'arrivals thatsAllItTook'
classVariableNames: ''
poolDictionaries: ''
category: 'Simulation Applications'

MeetTheQueen methodsFor: 'initialization'

defineArrivalSchedule
  self scheduleArrivalOf: RoyalSubject
       accordingTo: arrivals.
  self schedule: [self finishUp ] at: 100

defineResources
  self produce: 1 of: 'Queen'

initialize
  super initialize.
  arrivals ← Exponential named: 'Arrivals'
                         mean: 0.5.
  thatsAllItTook ← Tally named: 'ItTook'

MeetTheQueen methodsFor: 'recording'

recordExperience: aNumber
  thatsAllItTook update: aNumber

MeetTheQueen methodsFor: 'reporting'

printStatisticsOn: aStream
  (self provideResourceFor: 'Queen')
     printStatisticsOn: aStream.
  thatsAllItTook printStatisticsOn: aStream

SimulationObject subclass #RoyalSubject

instanceVariableNames: 'birthTime'
classVariableNames: ''
poolDictionaries: ''
category: 'Simulation Applications'

RoyalSubject methodsFor: 'accessing'

timeStamp: aTimeValue
  birthTime ← aTimeValue

birthTime
  ↑ birthTime

flowTime
  ↑ (Simulation active now ) - self birthTime

RoyalSubject methodsFor: 'simulation control'

tasks
  |shake notAgain hooked myQueen|
  self timeStamp: now .
  shake    ← Uniform named: 'Shake' min: 0.5
                          max: 1.5.
  notAgain ← Binomial named: 'NotAgain'
                          prob: 0.7.
  hooked   ← 'True'.
  hooked whileTrue:
     [ myQueen ← self acquire: 1 ofResource: 'Queen'.
       self holdFor: shake next .
       self release: myQueen.
       hooked ← (notAgain next ) not].
  (Simulation active ) recordExperience: flowTime
```

Program 3: Garden Party in SMALLTALK

Program 3 shows the Smalltalk implementation of our garden party. A monitor called *GardenParty* is created as a subclass of *Simulation*, with object arrivals and resources defined by *initialize*, *defineArrivalSchedule* and *defineResources* methods. Since we choose a material oriented approach, active processes of class *RoyalSubject* drive the simulation; implemented as subclasses of *SimulationObject*. **Figure 4** shows a typical Smalltalk screen.

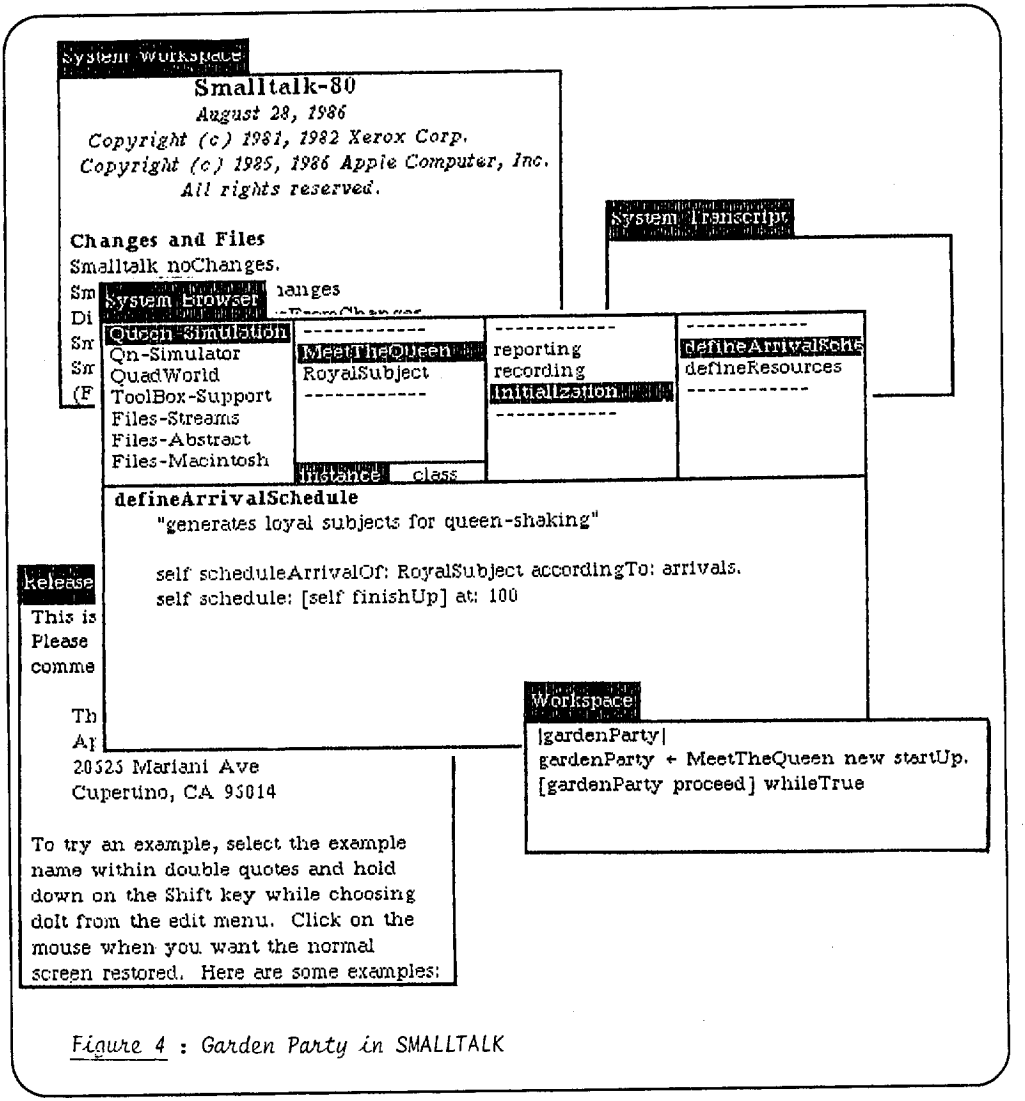

Figure 4 : *Garden Party in SMALLTALK*

It is very difficult to convey any taste of programming in Smalltalk without reference to practical experience. Browsing, construction, modification, execution and debugging of classes and methods is a highly interactive process with frequent switches among contexts. Smalltalk offers an extremely powerful environment for system developoment, but it must be stressed that it is predominantly a tool for the professional programmer.

4. Future Research & Developments

Self-contained **workstations with integrated system development environments** will make a large impact on the shape of programming in the 90s. Exploratory prototyping of complex systems, an acceptance of the inevitability of change in system specifications, and a shift in emphasis from writing new to modifying existing programs will further increase the attraction of late binding, object orientation and powerful program management tools for browsing, design, testing, modification, instrumentation and optimization of code.

Interactive programming environments for simulation have received relatively little attention in the past. Nelson (1979) showed how such a capability can be built into a simulation language (Simula) and greatly enhances its quality. The recent shift in price-performance ratios for personal workstations makes such tools conceptually attractive and economically viable. First steps in this direction have already been taken. **Stella** [Richmond 1985] offers interactive construction and analysis of graphical System Dynamics

scenarios on the Apple MacIntosh. **SimLab** is a Simscript implementation embedded in a relatively simple interactive environment for the IBM PC. Goldberg and Robson (1979) report on an experiment linking queueing network simulation and graphical animation in Smalltalk. [Birtwistle 1984], [Sheppard 1984], [Birtwistle 1984], [Vaucher 1984], [Birtwistel et al. 1982, 1984], [Unger et al. 1984], [Birtwistle & Luker 1984], [Sol 1984] and [Stanridge et al. 1984] argue convincingly for the merits of augmenting computer-based **simulation laboratories** with graphical animation. Such simulation workstations should integrate collections of general purpose and simulation specific tools, accessed through a uniform and friendly interface. In implementing such systems, methodologies and tools will be needed from the field of Artificial Intelligence. Some research in this area promises interesting results; see, for example, [SCS 1985], [Birtwistle 1985], [Lavery 1985], [O'Keefe 1985, 1986], and [Kerkhoffs & Vansteenkiste 1986].

Any notation has its particular strengths and weaknesses. None of them is equally well suited to all purposes. **Object orientation** seems a particularly appropriate metaphor for simulation model building. It has many advantages over traditional methodologies, particularly for the representation of complex systems; but most importantly it results in a change of perception. The key concepts of object-oriented programming systems are the notions of closure, message passing, concept inheritance and the idea of programming as simulation of some miniworld. The economy of differential system description and a close correspondence of formal representation and real life manifestation of segments of reality enhances our understanding and communication with application experts.

There is empirical evidence for this within the Simula community. Strongly modular object encapsulation also encourages more reliable and modifyable programs.

Negative attributes of object-oriented knowledge representations are rooted mainly in their resource requirements. The idea of message passing as the sole means of communication typically results in long chains of indirect references and many transient data structures, particularly in the case of frequently interacting objects.

Our experience with the **Pose** system has caused us to believe that object orientation is indeed an appropriate style for simulation programming. This corresponds well with experiences gained in writing Artificial Intelligence applications. Since managing complexity is at the heart of both simulation and AI programming, we should not be surprised by a convergence of concerns; even though this may be well hidden by different traditions and terminologies.

The question of **what language** would be most appropriate to host simulation laboratories is difficult to answer, since there are now a number of programming systems which claim to support both object oriented and exploratory programming. The cost of subscribing to new programming paradigms is not insignificant. New concepts and programming idioms must be learned and machine efficiencies may drop considerably. The overall advantages in shortening development cycles, increased program reliability and ease of change, however, should easily outweigh this investment.

Flexible and powerful programming environments are well established in the Lisp family of languages. It is very likely that Lisp will remain the most important language for symbolic programming for quite some time, although Prolog continues to rise in popularity as a tool for education and prototyping. This trend may accelerate as more intelligent compilers become available and hardware efficiencies become relatively less important (i.e. by exploiting parallelism). Smalltalk will probably remain primarily in the research domain, but the merits of its pioneering combination of an object oriented approach with window based programming environments are already well established. Evidence for this trend is clearly visible in many modern expert system development tools.

Good modelling environments must offer high resolution graphics and provide meaningful and extensible symbol sets. There should be some pointing device, such as a mouse, for feature selection. It should also be possible to view different processes in different stages of execution (i.e. a graphical model specification, a view of the model's state, various statistical measures, ...). The mouse may be used to select and activate processes, with more detailed control provided by menu selection. Options may include starting a simulation, resetting it to a previous state, editing models, tracing an object's states, obtaining cross references, performing statistical analyses, ...

If we ignore speed of execution as a main criterion, **Smalltalk** would be an almost ideal environment to host simulation workbenches. Our project views it as a tool for rapid prototyping in order to experiment with appropriate modelling interfaces for a number of simulation styles (i.e. queueing networks, system dynamics, monte carlo simulations, ...). This will hopefully be particularly productive during stage 2, where graphical scenarios are involved. It seems currently still unrealistic to assume availability of such systems on a large scale. The use of **Scheme** is a reasonable compromise for making some of the workbenches available on more widely accessible systems. One of our next projects will therefore, for instance, graft a simple menu-based environment onto Pose implemented on an Apple MacIntosh.

Pose will also serve as a stepping stone to explore the application of **expert systems** as intelligent advisors during different phases of the simulation modelling cycle.

There will be a gradual **convergence** or at least an interchange of experiences (tools and methods) between **artificial intelligence** and **simulation programming**, caused by a common concern with the intellectual complexity of the systems we must cope with. The technique of object orientation originated in simulation programming (Simula) and has been exported to and popularized in AI (Smalltalk, actor languages, Expert Systems). Modern expert system development tools (i.e. KEE, KnowledgeCraft, ART) already provide some simulation capability. There is also a trend towards integration of model based reasoning in knowledge based systems.

It may be hoped that the field of system simulation will now, in return, reap the benefits of a more mature technology grafted onto user friendly environments which cater for interactive and experimental styles of model design and exploration.

4. References

Abelson, H./ Sussman, J./ Sussman, J. *Structure and Interpretation of Computer Programs* Cambridge, MA 1985

Allen, J.R. *Anatomy of Lisp* New York 1978

Astroem, K.J./ Kreutzer, W. *System Representation* Proceedings of *"3rd Symposium on Computer Aided Control System Design"* IEEE 1986

Birtwistle, G.M. *Discrete Event Modelling on SIMULA* London / Basingstoke 1979

Birtwistle, G.M. *Future Directions in Simulation Software* in: [SCS 1984] 120,121

Birtwistle, G.M. (ed.) *Proceedings of 'AI, Graphics & Simulation'* SCS 1985

Birtwistle, G./Liblong, B./Unger, B./ Witten, I. *Simulation Environments* Proceedings of *"Simulation: a Research Focus"* Rutgers University 1982

Birtwistle, G./Lomov, G./Unger, B./ Luker, P. *Process Style Packages for Discrete Event Modelling: Data Structures and Packages in SIMULA* Transactions of the SCS 1(1), 61-82

Birtwistle, G./ Luker, P. *Dialogs for Simulation* in: [SCS 1984], 90 - 97

Bobillier, P.A./Kahan, P.C./ Probst, A.R. *Simulation with GPSS and GPSS V* Englewood Cliffs 1976

Bratley, P./Fox, D.C./ Schrage, L.E. *A Guide to Simulation* Heidelberg/New York 1983

Carnegie Group Inc. *Knowledge Craft - Overview* Pittsburgh 1986

Dybvig, K./Smith, B. *Chez Scheme Reference Manual* Bloomington, Indiana, Cadence Inc. 1985

Fishman, G.S. *Principles of Discrete Event Simulation* London / New York / Sydney 1978

Goldberg, A. *SMALLTALK-80 - The Interactive Programming Environment* Reading, MA 1984

Goldberg, A./ Robson, D. *A Metaphor for User Interface Desig* Proceedings of *12th Hawaii International Conference on System Sciences* 1979; 148 - 157

Goldberg, A./ Robson, D. *SMALLTALK-80 - The Language and its Implementation* Reading, MA 1983

Hutchinson, G.K. *Introduction to the Use of Activity Cycles as a Basis for Systems Decomposition and Simulation* SIMULETTER 7(1) 1975, 15 - 23

Irvin, W. *A Smalltalk Queueing Network Simulator* Honours Project, Dptm. of Computer Science, University of Canterbury 1986

IntelliCorp *The Knowledge Engineering Environment* Menlo Park 1984

Kerkhoffs, E.J.H./Vansteenkiste, G. *The Impact of Advanced Information Processing on Simulation - An Illustrative Review* SIMULATION 46(1) 1986, 17 - 26

Kreutzer, W. *System Simulation - Programming Styles and Languages* Sydney/Wokingham/Reading 1986

Lavery, R.G. (ed.) *Modelling and Simulation on Microcomputers* SCS 1985

Nelson, S.S. *CONSIM: A Study of Control Issues in Conversational Simulation* The Computer Journal 22(2) 1979, 119-126

O'Keefe, R.M. *Expert Systems and Operational Research - Mutual Benefits* Journal Opl.Res.Society 36(2) 1985, 125-129

O'Keefe, R.M. *Simulation and Expert Systems - A Taxonomy and some Examples* SIMULATION 46(1) 1986, 10-16

Richmond, B. *A Users' Guide to Stella* High Performance Systems Inc, Lyme, N.H. 1985

SCS (Simulation Council Inc.) *Proceedings of an SCS Conference on Simulation with Strongly Typed Languages* La Jolla 1984

SCS (Simulation Council Inc.) *Proceedings of an SCS Conference on Artificial Intelligence and Simulation* San Diego 1985

Sheppard, S.S. *Simulation Workstations of the Future - Panel Discussion* in: [SCS 1984], 119

Sol, H. *A SIMULA Problem Solving Environment* in: [SCS 1984], 99 - 104

Stairmand, M. *Expert Advice for Queueing Network Simulations* MSc thesis - Dptm. of Computer Science, University of Canterbury 1987

Stanridge, C.R./Pritsker, A.A.B./ O'Reilly, J. *Integrated Simualtion Support Systems. Concepts and Examples* Proceedings of the 1984 Ann. Simulation Symp.IEEE 1984, 141-151

Unger, B./Birtwistle, G./Cleary, J./ Hill, D./Lomov, G./Neale, R./ Peterson, M./ Witten, I./Wyvill, B. *JADE: a Simulation and Software Prototyping Environment* in: [SCS 1984], 77-83

Vaucher, J. *Future Directions in Simulation Software - Panel Discussion* in: [SCS 1984], 122

Wulf, W.A./Shaw, M./Hilfinger, P.N./ Flon, L. *Fundamental Structures of Computer Science* Reading 1981