Some Patterns for Building Discrete Event Models & Simulators

Wolfgang Kreutzer, Department of Computer Science

University of Canterbury, New Zealand

ABSTRACT:

This paper describes the foundations for a family of patterns for simulation software design. After some introductory comments on the nature and purpose of simulation a number of different frameworks are briefly classified and discussed based on their dominant abstractions. The main part of the paper then presents a more detailed survey of patterns in the simulation modelling domain taking examples from discrete event simulator and model construction. The paper concludes with a summary of benefits of pattern languages for simulation modelling and their potential impact on component reuse. Some problems which need to be overcome to make this vision a reality are also commented on.

1. INTRODUCTION

The complexity of the **models** we use to make sense of experience has always been constrained by our limited mental capacities. Computer-based modelling tools allow us to relax these constraints, and **computer simulation** has therefore become a particularly powerful means for studying complex systems. By animating and observing a model's behaviour it allows us to answer "what if?" types of questions and recent technological advances combined with a desire to study ever more complex scenarios have made it one of the fastest growing areas of computer technology, whose applications range from teaching and training to support of commercial planning decisions and the provision of testbeds for scientific theories and engineering designs.

Figure 1 shows a much simplified model of the model development cycle, separating the phases of system identification (i.e. applying abstraction and aggregation to map a perceived system into a conceptual model), model design (i.e. deriving a formal model based on a suitable method and notation), model implementation (i.e. coding a formal model into a program), and experimentation (i.e. observing the program's execution in order to derive conclusions about the original systems's behaviour).

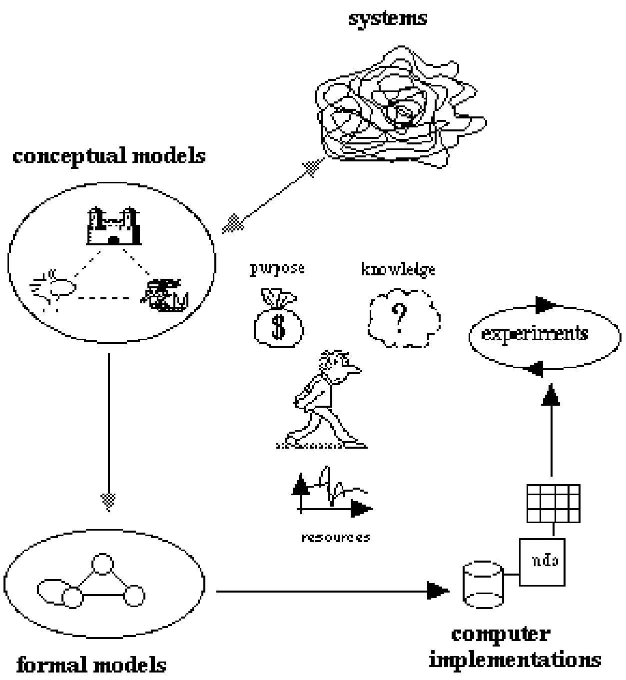

Figure 1: A Model of the Modelling Process

Within this context it is important to remember that any such linear sequence of activities must always be artificial and that building a model requires abstraction, a process which is strongly determined by both knowledge (i.e. the "patterns" used) and goals of the modeller. As an additional constraint we must also never loose sight of the need for validation (i.e. to show that our models correspond to reality in a meaningful way) and verification (i.e. to demonstrate that our programs are correct representations of our models). Many different types of simulation lead to different types of conceptual models, which all have their specific methodologies, tools, traditions, user communities and typical applications [see Kreutzer (1986)]. A wide variety of textual and graphical notations as well as different programming tools (i.e. so called simulation programming languages) all have their place in a simulator's toolbox, together with a rich repertoire of methodological knowledge about their safe application. Finding "the right" framework for a particular task is a difficult problem and skilled selection of tools needs a careful blending of theory and practice. For this reason simulation modelling offers a rich target for mining for patterns.&141; Many stable and well established idioms, frameworks and paradigms can readily be identified at a whole range of levels, whose documentation can aid us in both our teaching and practice.

2. SIMULATION METAPHORS & FRAMEWORKS

Simulation models and the programs we write to support them are often large and complex pieces of software, which require abstraction and layering to bridge a wide gap between "the system" and its coded representation (i.e. "a program"). Both **classification** and **composition** can help us in this undertaking, revealing patterns at progressively more detailed levels of description. Bridging the conceptual gap between models and programs starts with structures and processes prescribed by a modelling paradigm or "top level" scenario based on highly specific patterns; e.g. a queuing network using a client/server pattern to model computer networks or flexible manufacturing systems, or a system dynamics model using a predator/prey pattern to model the ecology of a lake or market economy. At lower levels we can identify ever more general patterns of increasingly abstract concepts (e.g. patterns involving transactions, resources and queues, or levels and rates) until this decomposition "bottoms out" in some general purpose programming language (e.g. Simula, Smalltalk or C++), whose vocabulary typically reflects "low level" computer operations, control sequences and memory organization. This strategy of progressive abstraction and decomposition is often labelled reductionism and has had a long and successful history of in western style Science. Although it is not suitable for all systems and purposes such decomposition of systems into hierarchies of abstraction is still the best strategy for reducing complexity we have so far been able to discover. Figure 2 summarizes the principles behind this idea.

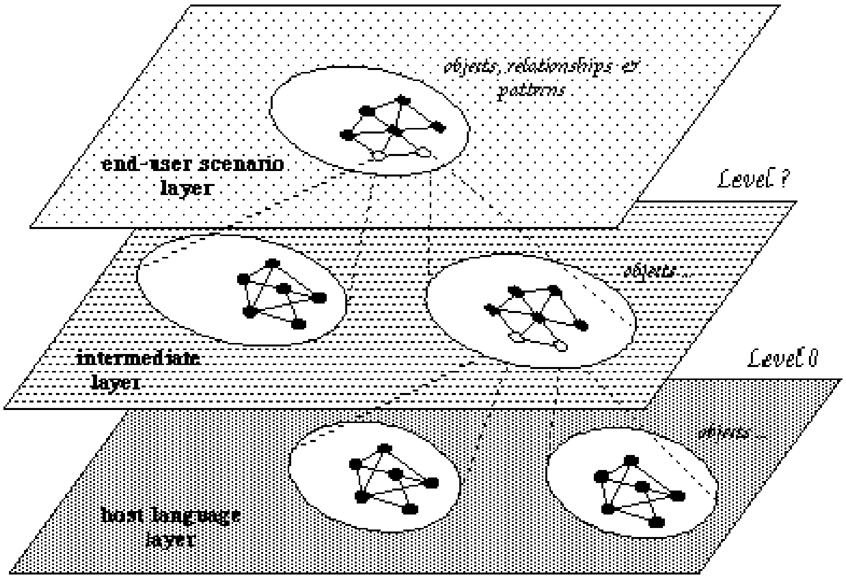

Figure 2: A Layered Architecture for Model and Program Design

By focussing on their dominant abstractions frameworks for simulation modelling can also be classified in this fashion. At the "top level" every simulation is classed as an experimental technique, for which facilities for data collection and presentation will always be needed. This is the most basic aspect which all types of simulation have in common. Beyond this each modelling method prescribes its own more specific patterns. **Monte Carlo techniques**, for example, need to support distribution sampling, but no time management is required. Such models can help us find answers to questions regarding a system described in statistical terms; e.g. we can compute distributions to summarize the likelihood of events, estimate the duration of projects, or predict the expected quality of a product from statistical profiles of its ingredients. Dynamic types of simulation such as continuous or discrete event models, on the other hand, need to include a time dimension. They differ, however, in the way in which behaviour will be described. In case of "**continuous simulations**" we express system dynamics by writing equations which characterize multi-dimensional fluctuations of state (i.e. as so called "distributed parameter systems") or regular patterns of change in the values of variables (i.e. as "lumped parameter systems"). Questions of interest may concern the shape of a feedback system's behaviour, e.g. whether it is stable or explosive, how steady state can be reached and how such systems will react to disruptions (i.e. are they robust or chaotic). Often we will be content if at least some qualitative predictions can be made. Such models have proved to be useful in engineering, biology, economics or sociology, and the **system dynamics** method in particular has become widely practiced. In contrast to this approach we can also describe discrete changes in state through procedures, which offer more flexibility for modelling non-regularities. This approach is known as **discrete event simulation** and has become popular in computer science and operations research. Here we describe models in terms of structured collections of objects bound into webs of relationships and transformations across space and time. A wide diversity of systems can profitably be studied from this perspective and queuing network scenarios, in which we seek to explore the effects of capacity limitations and routing strategies on the flow of some classes of items, are among the most common of these. In this context the objects of interest are different kinds of resources, queues and transactions, while relevant questions relate to quantitative measures of throughput, response time, utilizations and bottlenecks.

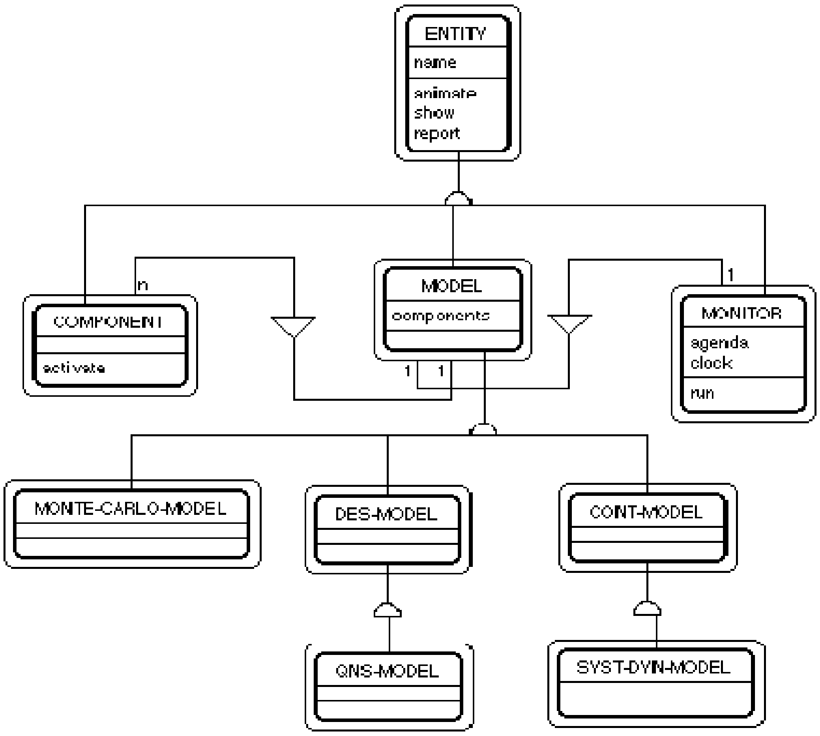

Figure 3: Structure and Relationships of Models of Different Types

Figure 3 uses a Coad/Yourdon diagram [Coad (1991)] to summarize this discussion. It clearly stresses the fact that some kind of **monitor** entity will always be needed to control a model's execution. Note that while monitors are shown as a model component here, they may also be used to control series of experiments at a yet "higher level".

In terms of implementation technology object-oriented styles of software development and simulation are closely linked. Although Smalltalk established much of the terminology of the field it was a simulation language (i.e. Simula) which invented most of object-oriented programming's most central ideas; such as classes, inheritance, polymorphism. Originally designed as a simulation programming tool in the mid 1960s Simula still offers a more comprehensive object model than most contemporary object oriented languages; i.e. each Simula object can own a lifecyle describing its history and behaviour. Additional to inheritance Simula also allows class hierarchies to be nested, so that we can have composition relationships at class levels as well as among objects. This design is sometimes referred to as the "Scandinavian school", [see Madsen (1995)] since it also underlies the more modern Beta language [see Madsen (1993)] which unifies Simula's ideas on the basis of patterns. Simula was also the first language which offered the scaffolding needed for reusable frameworks. For example, class "Simulation" which is part of the core Simula system, offers a reusable architecture for discrete event simulation design. While similar frameworks have been developed for other domains (e.g. for continuous simulation, CAD and graphics) the best demonstration of the power of this idea was given by Graham Birtwhistle's DEMOS system [see Birtwhistle (1979)] in the late 1970s. Unfortunately Demos did not achieve the popularity it deserved, but by inventing or describing many reusable patterns in the queuing domain [see Birtwhistle (1984)] it has laid a foundation on which most modern queuing simulators are still being built.

3. TOWARDS A PATTERN LANGUAGE FOR SIMULATION

Patterns capture recurring relationships among objects and may put customizable solutions in context [see Gabriel (1994)]. They therefore find their best application when reflecting a discipline's standard practices. In this context they play useful roles in documenting and prescribing vocabularies for talking about designs and can serve as platforms for experimenting with alternative compositions of structures and processes. Related patterns can be grouped into clusters called "pattern languages", whose development is typically the result of a lengthy evolutionary process. Within the simulation domain the author has experimented with patterns over a period of 25 years, as part of research aimed at designing extensible toolkits and visual environments for different styles of simulation modelling [see Kreutzer (1987, 1994)]. Pattern languages aid us in building effective models on purpose, in that they document and communicate common design knowledge and help to constrain the mental complexity of assembling large pieces of software.

To use patterns effectively model builders must understand the architectural concepts on which they are based. This can be difficult if there are many patterns with many meaningful combinations. Although object-based encapsulation and helpful examples can aid us in making sense, we must also make good use of the reductionist "divide & conquer" pattern we referred to above. A direct consequence of this insight is the recognition that patterns can also be applied to generalize other patterns; i.e. a good pattern language must span a whole hierarchy of different levels of abstraction - from general rules for system design to language specific idioms. At the top we would expect to find very concrete concepts (e.g. For modelling computer systems, VLSI designs, ecological models or traffic logistics) while abstract idioms of a specific programming tool (e.g. GPSS, Simula or C) will typically be found at the bottom layer. While higher levels will tend to prescribe architectural designs (i.e. describing how to fit components together) lower levels will need to give specific examples of to construct them in program code. Mining such patterns is similar to object-oriented analysis and design (OOA&D;), for which many guidelines have recently been given. A thorough understanding of the task domain guided by metrics such as coupling and cohesion is needed to identify the "right" layers, objects and patterns.

LAYER:	OBJECTS:	PATTERNS:
instrumentation layer	tally, histogram ..	observer, compression, presenter, animater ..
sampling layer	uniform, normal ..	sampling, transformation ..
monitor layer	clock, agenda ..	loop, time slicing, next event, integration ..
process layer	semaphore, actor ..	shared queues, message passing, client/server ..
scenario layer		
system dynamics layer	level, rate ..	pos. feedback, neg. feedback, predator/prey ..
queueing layer	server, queue ..	workload generator, machine-or., material-or., process-interaction ..

Figure 4: Some Patterns for Simulation Model Design

Figure 4 sketches some of the layers which are commonly found in simulation software. Adding more detail leads to a pattern language for simulation and simulator design, subsets of which will be relevant for different types of models (for example system dynamics and queuing network scenarios). Each of the layers shown in the figure is composed of many objects and patterns - which are composed of more general objects and patterns at the layer below and used as building blocks for more specific objects and patterns at the layer above. Note that we distinguish between objects and patterns here; i.e. a pattern describes usage of and interactions between objects. For example, while a tally object serves as a repository for a time series' state and behaviour, an observer pattern describes how this state is updated (e.g. by tracing all of a variable's changes or by sampling its value in specified intervals). Note that not all of a layer's objects and patterns are needed for all model types. For example, a Monte Carlo toolbox may contain counter, tally and histogram objects together with observer, compression, presentation and animator patterns from the instrumentation layer, as well as a collection of distribution objects and random generators together with suitable patterns from the sampling layer. Since no time dimension is involved here, a simple loop pattern from the monitor layer would suffice to drive such a model's execution. A system dynamics toolbox, on the other hand, may select time series and scatter graph objects and relevant patterns from the instrumentation layer, a clock object and Runge-Kutta integration pattern for time and change management, and level and rate objects with feedback control patterns from the scenario layers. Since stochastic components are not normally part of this framework no selections from the sampling layer need to be made. While these are relatively simple cases, a suitable toolbox for modelling queuing scenarios would require many more patterns. Counter, tally, accumulate, histogram, chronological graph, correlation and regression objects and suitable patterns from the instrumentation layer may all be of use. Many random generators and sampling patterns from the sampling layer may need to be included, as well as clocks and agendas together with process style monitor patterns from the monitor layer, and client/server and condition queue patterns from the synchronization layer. Additionally such a framework would add more specific transaction, server and queuing objects, driven by suitable patterns with many variations (e.g. trace and distribution-driven workload creation, machine and material oriented process synchronization, client/server cooperation and resource contention patterns through containers and servers, ...). Note that composition/decomposition hierarchies may run orthogonal to this classification. While a queuing scenario is composed of instrumentation patterns, sampling patterns, a monitor pattern, synchronization patterns and resource patterns, a sampling pattern may in turn be composed of a random generator, transformation and instrumentation patterns. In the same way a process style monitor pattern may consist of a next-event time advance pattern, an agenda pattern, and a coroutine-based process pattern.

Figures 5 to 8 offer more detailed views of some relevant objects, together with relationships, properties and behaviours, for the data collection, distribution, monitor and queuing network layers shown in figure 4.

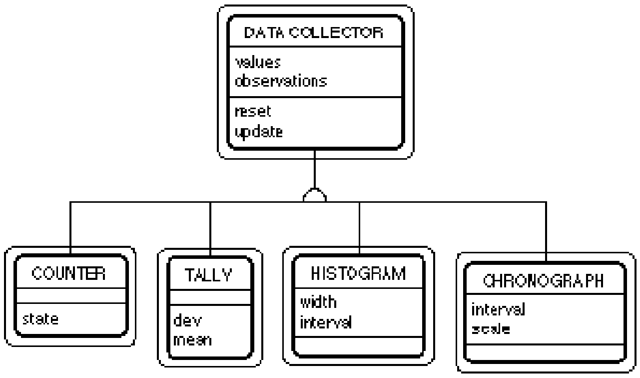

Figure 5: A Coad-Yourdon Diagram of the Data Collection Layer

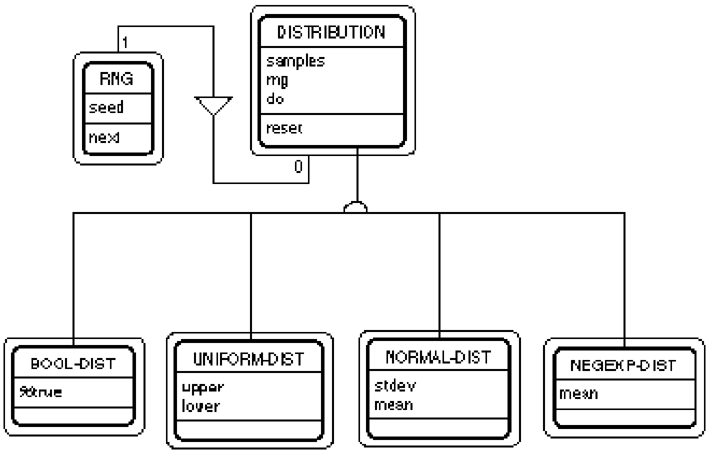

Figure 6: A Coad-Yourdon Diagram of the Distribution Sampling Layer

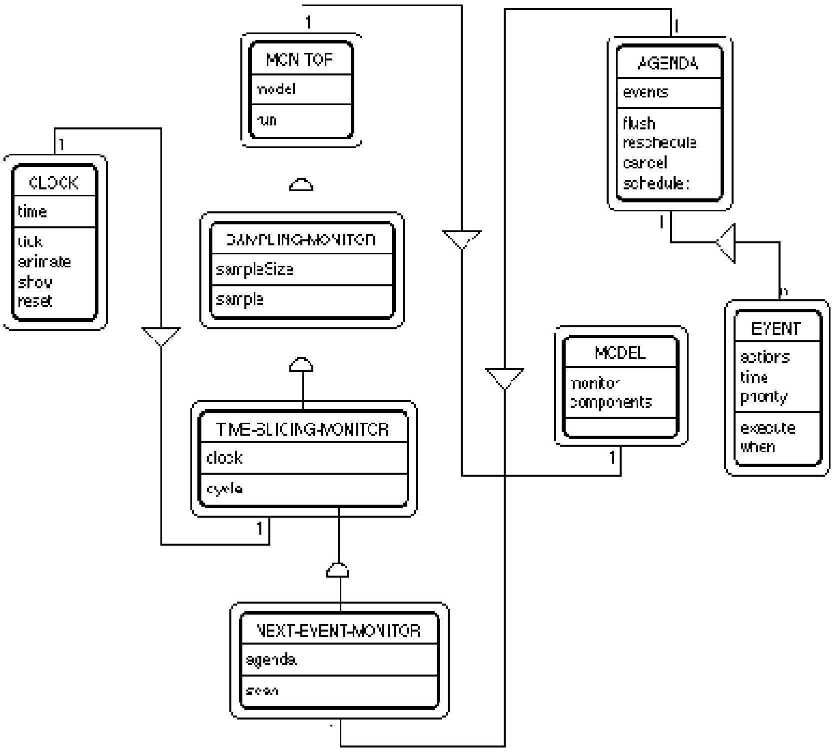

Figure 7: A Coad-Yourdon Diagram of the Process Synchronization Layer

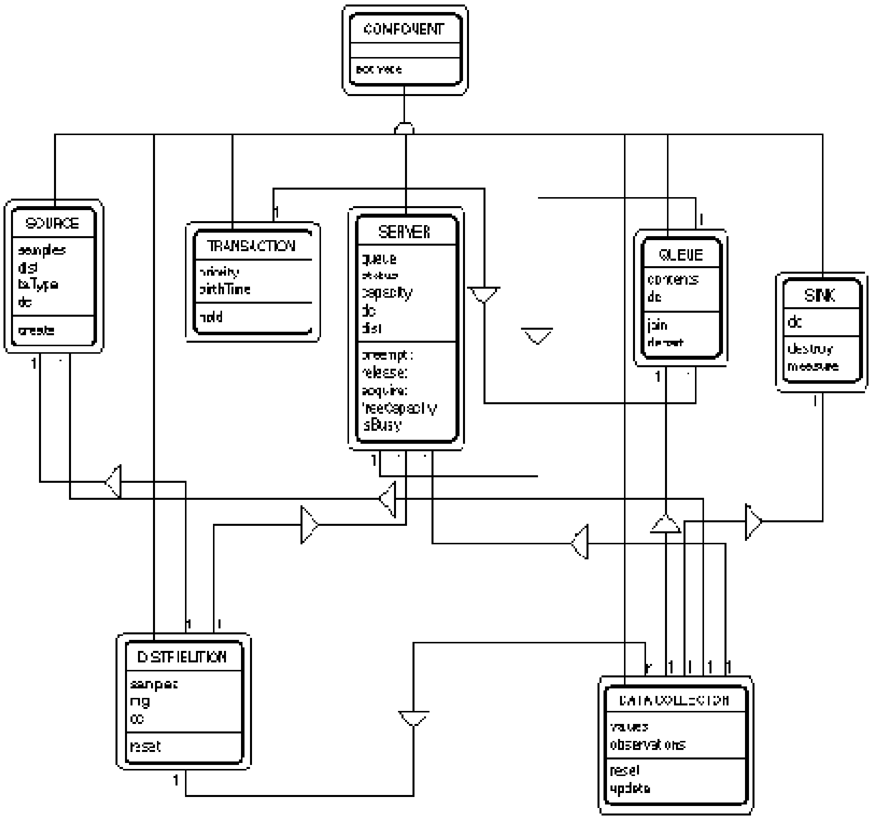

Figure 8: A Coad-Yourdon Diagram of the Queuing Scenario Layer

Note that many further specializations and extensions can still be made. For example, a "FifoQueue with balking" is a specialization of a the generic queue pattern, while a conveyor facility (i.e. which takes workload items and "moves" them through steps in a manufacturing process) may be a suitable specialization of the server pattern in a flexible manufacturing toolbox. Either top-down (i.e. from specific applications to an abstract base) or bottom-up (i.e. from an abstract base to specific applications) strategies may be employed to discover suitable patterns, but experience shows that one needs to use both strategies in combination. One of the most positive experiences during the design of toolboxes for a Modeller's Workbench [see Kreutzer (1986, 1994)] was the ease with which complex models and simulators can be built with the aid of a pattern language such as the one we describe; i.e. by following a "lego-style" approach to assembling building blocks of ever increasing complexity. By using classes of objects to encapsulate information in manageable and separately testable chunks we prevent mental complexity from increasing exponentially with model and program size [see Simon (1969)].

Although some work aimed at systematic classification and description of modelling patterns has been done in the past, such activities have typically only occurred in the context of abstract specification languages [e.g. Nance (1979, 1981), Frankowski (1980), Franta (1977), Zeigler (1976, 1984), Oren (1979)] and much effort is still needed to convert their results into pattern form. The next section presents some examples of flavour and contents of how such a pattern language could look like. A fully developed framework is currently under development, together with implementations in different base languages (e.g. Simula, Beta, Scheme, Smalltalk, Oberon and Java).

4. Some Example Patterns

Pattern languages can be used at both model and simulator construction levels. When building a model we need patterns to help us in choosing a suitable design, based on our knowledge and goals as well as the tools available to us. One of these tools will be a simulator to execute models, and it is this interaction between models and interpreters which can be observed at all levels of description and reflects the layering strategy we discussed before. More "high level" languages will more directly support a domain by offering more specific patterns already built into them, while "lower level" tools are based on more general patterns and provide the foundations for the levels above. The end user level which forms the top of this hierarchy must therefore take the domain and type of model into account, while patterns at the programming language level may ultimately reflect common idioms of a particular paradigm (e.g. Imperative, functional, logic-based or object-oriented) or even a particular language's programming style (e.g. Smalltalk, Simula, C++).

In this section we draw again on the often made distinction between models and their execution. While a modeller needs to invent suitable descriptions of relevant parts of reality, a computerised simulator must map these into a computation (see figure 9). In order for any such communication to work there must be a shared context, e.g. the linguistic abstractions of a particular programming tool. Both partners also need patterns to map these abstractions into their world models (i.e. "the problem" and "the hardware components"), and we use this distinction to separate modelling patterns (which describe how models are built) from pattern for simulator design (which describe how we can execute them).

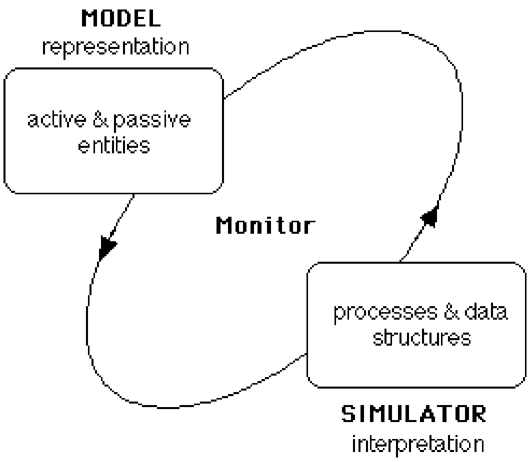

Figure 9: A Process View of Modelling & Simulation

4.1 Some Patterns for Model Construction

Within the modelling context we will look at a number of patterns for building discrete event models; from data collection and workload creation to the synchronization of conceptually parallel processes (i.e. sets of modelling entities with separate life cycles). Note that these patterns are nested, i.e. more specific ones (e.g. sources, master-slave synchronization ...) may be "plugged into" more general ones (e.g. Machine or material orientation). All pattern descriptions are given in the form used by Gamma and others (1984).

4.1.1 Machine & material orientation - two alternative viewpoints of model design

Our first patterns describe two strategies which are often used to code the behaviour of discrete event models with multiple processes. Note that the example we give for its application is based on a scenario we will also draw on to explain other patterns. We therefore give a brief summary below.

Consider a garden party held in honour of a visiting monarch, where local dignitaries are given a chance to mingle and shake the queen's hand. From a suitably abstract perspective we may well wish to model this scenario as a queuing network, assuming typical arrival patterns for dignitaries and severe capacity limitations on the hand on the monarch (i.e. she shakes only with one). Let us further assume that the shaking activity has random duration and that after a shake some of the dignitaries may wish to go back for another experience (while others will hurriedly leave). Interestingly enough, even this simple scenario can be described from a number of different perspectives.

Pattern Name: *Machine-oriented synchronization in discrete event models.*

Intent: This pattern describes a strategy for modelling systems with multiple classes of objects from a machine-oriented perspective. The name relates to the simulation of priority rules in so called job shop models which are composed of machines processing materials; the domain in which it was first applied. In this context "machines" are permanent and active parts of the model; i.e. they own a life cycle composed time consuming and state changing activities. "Materials", on the other hand, are transient and passive; i.e. they may act as repositories of properties only, and are just operated upon by the machines.

Motivation: We want to build a discrete event model of a system in which multiple processes are of interest. The relevant entities we identify can be classed as permanent versus transient and active (i.e. properties and behaviour) versus passive (i.e. properties only). Although both types of entities will need to be represented in some fashion, the materials' life cycle is irrelevant to our purposes and all interesting information about them can adequately be modelled through attributes. In extreme cases we may be interested only in a material object's existence (i.e. it can then be modelled as a token or their number may be stored in a counter).

Applicability: This pattern is applicable to any discrete event simulation involving more than a single type of object.

Participants: The system is modelled from the viewpoint of active "machine" entities whose life cycles are described by dynamic processes, and which operate on passive "material" entities whose attributes' values they change. Figure 9 shows such a scenario for our benchmark model (i.e. the garden party). Note that depending on what information about "material" items we wish to store they can be represented either by boolean flags, counters or passive objects without their own lifecycle.

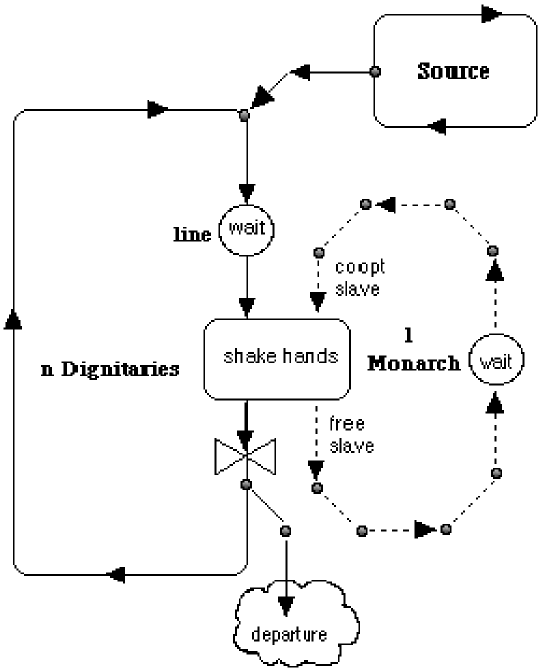

Figure 10: A Machine-Oriented Pattern applied to a Garden Party

Collaborations: In contrast to a process-interaction pattern processes representing "machines" do not interact directly with one another. All collaborations occur by entering "material" entities into or taking them out of lists. Note that machines may be blocked if no suitable material is available and that material may need to wait patiently for a machine's attention.

Consequences: This pattern is not as popular as its inverse: the material-oriented pattern. This has mainly historical reasons since some early simulation programming languages such as GPSS favoured the implementation of material-orientation viewpoints (i.e. they made easy to build). Machine-orientation is a suitable modelling perspective to take if there are many different items of different types, which are dynamically created and consumed while the model is running but whose interactions with other objects are not complex enough to warrant description as process (i.e. They have things done to them rather than doing anything themselves).

Implementation: This pattern can easily be implemented in any programming language, although it is convenient to have some support for dynamic memory management and pseudo-concurrent processes (i.e. coroutines).

Sample Code and Usage:

Here is a machine-oriented model of a queen's gardenparty [written in Demos\Simula - see Birtwhistle (1979)]. Note that we use a master/slave pattern (called a "WaitQ") for synchronization here (with monarchs as masters of course), and that the machine entity (i.e. the Monarch) will need to decide if a material entity (i.e. a Dignitary) should leave or will want to experience another shake. Capitalized terms are language keywords (i.e. built-in abstractions) and Simula uses the ":-" symbol to indicate reference assignment. Note also that we get standard statistics about length and delays in the queue, and record and compute the mean time a dignitary will spend at the party (i.e. Using a TALLY). However, no statistics are being collected on server utilization.

```
DEMOS BEGIN

REF (TALLY ) howLongWeStayed;
REF (NEGEXP) arrivals;
REF (NEGEXP) shakes;
REF (DRAW ) anotherGo;
REF (Monarch) queen;
REF (Source ) gate;
REF (WAITQ ) line;

ENTITY CLASS Source (d); REF (DIST) d;
BEGIN
  REF (Dignitary) fred;

WHILE TRUE DO
  BEGIN
  fred :- NEW Dignitary("fred");
  fred.entryTime := TIME;
  fred.WAIT(line);
  HOLD (d.SAMPLE)
  END of Source's life cycle;
END of Source;

ENTITY CLASS Monarch (d); REF (DIST) d;
BEGIN
  REF (Dignitary) fred;

WHILE TRUE DO
  BEGIN
  fred:- line.COOPT;
  HOLD (d.SAMPLE);
  IF anotherGO.SAMPLE
     THEN fred.WAIT(line)
     ELSE
      BEGIN
      howLongWeStayed.UPDATE (TIME - fred.entryTime);
      fred :- NONE;
      COMMENT: get rid of the guy (garbage collect him);
      END of else


  END of Monarch's life cycle
END of Monarch;

ENTITY CLASS Dignitary;
BEGIN
  REAL entryTime;
END of Dignitary (has no life of her own !);

COMMENT: The main program instantiates and initializes all objects;

howLongWeStayed :- NEW TALLY ("flow time statistics");
arrivals  :- NEW NEGEXP with suitable name and parameters;
shakes    :- NEW NEGEXP with suitable name and parameters;
anotherGo :- NEW DRAW with suitable name and parameters;
line      :- NEW WAITQ ("line of waiting dignitaries");
queen     :- NEW Monarch ("queen", shakes);
queen.SCHEDULE (NOW);
gate      :- NEW Source ("gate", arrivals);
```

```
gate.SCHEDULE (NOW);

COMMENT: Now start a simulation for a specified (i.e. read in) time period;
HOLD (ININT);
END of model;
```

Known Usage: While machine-oriented patterns were often used in connection with early activity oriented discrete event simulators, material-oriented and process-interaction based patterns have proved more popular since

Related Patterns: material-orientation, process-interaction.

In contrast to the machine-oriented pattern a material-oriented pattern reflects model dynamics from the viewpoint of transient and dynamic "material" entities, which grab and release passive "machine" resources. Note that for our benchmark model this pattern will lead to a somewhat simpler program.

Pattern Name: *Material-oriented synchronization in discrete event models.*

Intent: This pattern describes a strategy for modelling systems with multiple objects from a material oriented perspective. The name stems from the simulation of priority rules in so called job shop models, which are composed of machines processing materials; the domain in which this pattern was first applied. In this context "machines" are permanent and active (i.e they have a life cycle), while "materials" are transient and passive (i.e. they store only properties and are operated upon by machines).

Motivation: We want to build a discrete event model of a system in which multiple processes are of interest. The relevant entities we identify can be classed as permanent versus transient and active (i.e. properties and behaviour) versus passive (i.e. properties only). Although both types of entities will need to be represented the machines' life cycle is irrelevant to our purposes and all interesting information about them can adequately be modelled through servers entities with suitable attributes. In extreme cases we may even only be interested in a machine object's existence (i.e. it can then be modelled as a semaphore or boolean flag).

Applicability: This pattern is applicable to any discrete event simulation involving more than a single type of object.

Participants: The system is modelled from the viewpoint of active "material" entities whose life cycles are described by dynamic processes, and which acquire and release passive "machine" entities. Materials' attribute values may change during processing activities. Figure 10 shows a scenario for our benchmark model (i.e. the garden party). Note that depending on the information about "machine" items we wish to store "machines" can be represented either by tokens, semaphores or boolean flags, but they won't have their own lifecycles.

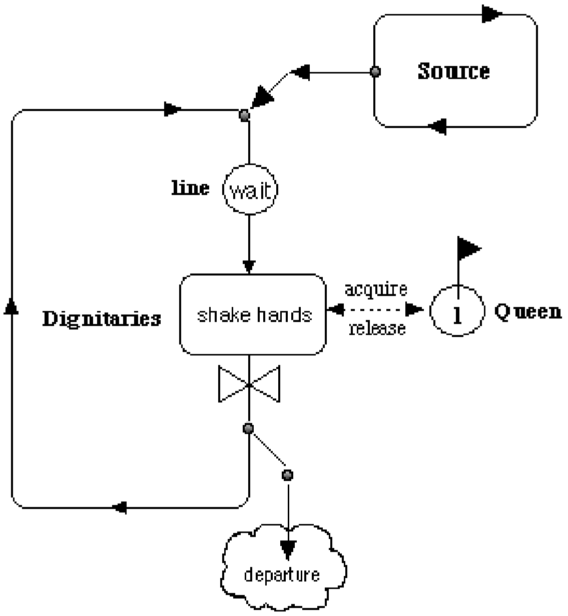

Figure 11: A Material-Oriented Pattern applied to a Garden Party

Collaborations: In contrast to a process-interaction pattern processes representing "materials" do not interact directly with one another. All collaborations occur by acquiring or releasing "machine" entities. Note that material processes may be blocked if no suitable machines are available and that machines may need to wait patiently until acquired by a material entity.

Consequences: This pattern is more popular than its inverse: the machine-oriented pattern. This has mainly historical reasons since early simulation languages such as GPSS favoured system description from a material-orientated point of view (i.e. they could be coded more directly). Material-orientation is a suitable modelling style to use whenever an entity's operation is not complex enough to warrant its own life cycle.

Implementation: This pattern can easily be implemented in any programming language, although it is convenient to have some support for dynamic memory management and pseudo-concurrent processes (i.e. coroutines).

Sample Code and Usage:

Here is a material-oriented model of the queen's gardenparty written in Demos\Simula [see Birtwhistle (1979)]. Note that dignitaries are now modelled as active entities, grabbing, holding and releasing their monarch's resources as needed. Dignitaries now also decide for themselves if they wish to leave or want to indulge themselves with another shake. Capitalized terms represent language keywords (i.e. built-in abstractions). Note also that Demos provides standard statistics on length and delays in the queue as well as the queen's utilization. Additionally we record and compute the mean time a dignitary will spend at the party by explicitly using a data collector.

```
DEMOS BEGIN

REF (Source) gate;
REF (RES) Queen;
```

```
REF (NEGEXP ) arrivals;
REF (NEGEXP) shake;
REF (DRAW) anotherGo;
REF (TALLY) howLongWeStayed;

ENTITY CLASS Source (d); REF (NEGEXP) d;
BEGIN
  REF (Dignitary) fred;

WHILE TRUE DO
  BEGIN
  fred :- NEW Dignitary ("fred");
  fred.entryTime := TIME;
  fred.SCHEDULE(NOW);
  HOLD (d.SAMPLE);
  END of Source's life cycle
END of source;

ENTITY CLASS Dignitary;
BEGIN
  BOOLEAN isHooked;
  REAL entryTime;

isHooked := TRUE;
timeOfEntry := TIME;

WHILE isHooked DO
  BEGIN
  queen.ACQUIRE (1);
  HOLD (shake.SAMPLE);
  queen.RELEASE (1);
  isHooked := anotherGo.SAMPLE;
  END loop;

howLongWeStayed.UPDATE (TIME - entryTime);
END of Dignitary;

COMMENT: The main program instantiates and initializes all objects;
howLongWeStayed :- NEW TALLY ("flow time statistics);
shake           :- NEW NEGEXP with suitable name and parameters;
arrivals        :- NEW NEGEXP with suitable name and parameters;
anotherGo       :- NEW DRAW with suitable name and parameters;
queen           :- NEW RES ("The Queen", 1); COMMENT: shakes only one-handed;
gate            :- NEW Source ("gate", arrivals);
gate.SCHEDULE (NOW);

COMMENT: Now start a simulation for a specified (i.e. read in) time period;
HOLD (ININT);
END of model;
```

Known Usage: Most open models of queuing scenarios are written from a material-oriented perspective.

Related Patterns:machine-orientation, process-interaction.

You should note that these pattern descriptions make use of many subsidiary patterns, defined at different layers, e.g. for data collection, sampling, process synchronization, resource allocation ... . Note also that we refer to a few patterns for which no further descriptions are given here. For example, a trace-driven simulation creates workload items by reading relevant information (e.g. time of arrival, entity type, relevant properties ..) from files, where this data relates to some "real world" measurements. While this style of workload modelling used to be popular in computer performance modelling, it has not played a similar role in other simulation domains, where samples from distributions are typically used to "drive" models (e.g. see the examples we give for the "source" and "generate next arrival" patterns).

4.1.2 A simple pattern for data collection in open scenarios

Pattern Name: *Observe and record flowtimes of transient workload items*

Intent: This pattern describes a technique for measuring and recording the flowtimes of transient items in open models (i.e. models in which workload items arrive and depart throughout a simulation's duration).

Motivation: We want to build a discrete event model in which the progress of transient workload items is measured in terms of the model time which passes between arrival and departure. This is a useful statistics in many scenarios (e.g. for response time in computer system simulations).

Applicability: This pattern is applicable to any scenario which models arrival patterns of transient entities.

Participants: Arriving items may be created by either a source or a successor pattern. At the end of a life cycle statistics may be collected by the departing item or by a separate sink entity.

Collaborations: Sources and sink objects may cooperate in gathering and recording relevant statistics.

Consequences: This is one of many patterns for data collection in discrete event simulation models.

Implementation: This pattern can easily be implemented in any simulation language.

Sample Code and Usage:

The following code sketches an implementation in Demos/Simula. We assume that the data collection is performed by the workload item itself - not a source or a sink.

```
REF(DataCollector) flowTimes;

ENTITY CLASS workloadItem;
BEGIN
  REAL arrivalTime;

arrivalTime := TIME;
COMMENT: item step through its lifecycle;
flowTimes.UPDATE(TIME - arrivalTime);
END of class workloadItem;
```

Known Usage: This is a common pattern in all open discrete event simulation models.

Related Patterns: trace-driven simulation, stochastic simulation, machine orientation, process-interaction.

4.1.3 Two patterns for workload creation in open scenarios

Pattern Name: *source-driven workload creation*

Intent: This pattern describes a technique for simulating arrival of workload items in open systems (i.e. models in which workload items arrive and depart throughout a simulation's duration) by using a separate source entity to create them.

Motivation: We want to build a discrete event model in which transient workload items arrive, are being processed and leave according to patterns prescribed by a trace file or a probability distribution.

Applicability: This pattern is applicable to any scenario which models arrival patterns of transient entities.

Participants: Arrivals are generated using an active source process, which cycles through a sequence of workload creations. Workload items can be passive (in which case they are deposited in a collection entity) or active (in which case they are scheduled to start on their life cycle). Generating new items continues until the simulation terminates or some other condition is met.

Collaborations: Other than activating a newly made item's lifecycle, sources and workload items to not collaborate directly. Sources may, however, signal other entities of a new workload item's arrival.

Consequences: This pattern is somewhat more complex than a "create successor" pattern, which may be used for the same purpose, but it provides cleaner better model structures and allows us to factor collection of arrival statistics into a self-contained module.

Implementation: This pattern is best implemented in a process-based simulation language.

Sample Code and Usage:

The following code gives an implementation in Demos/Simula. Note that we assume a sample-driven arrival process, the distribution for which is passed as an argument.

```
ENTITY CLASS workloadItem ... ;

ENTITY CLASS Source (d); REF(DIST) d;
BEGIN
  REF (COUNT) counter;
  REF (class of workload item) fred;
  COMMENT: ... And other statistics collectors as needed;

counter :- NEW COUNT ("arrivals");
WHILE NOT condition DO
  BEGIN
  fred :- NEW class of workload item (arguments to workload item);
  counter.UPDATE(1);
  COMMENT: and updates of other statistics collectors as needed;
  COMMENT: enter fred into a collection or SCHEDULE him for execution;
  HOLD (d.SAMPLE);
  END of life cycle;
```

```
END of class Source;
```

Note that both the class of workload item and the condition which causes new workload creations to stop are "hard-coded" into the model. Although it would be better to pass these as arguments to class Source, this is not easily possible unless the implementation language allows us to pass arguments in unevaluated form (which is not supported by Simula). Note also that we assume that if the condition is never met the model's termination will be ensured from elsewhere.

Known Usage: This is a common pattern in all open discrete event simulation models.

Related Patterns: trace-driven simulation, stochastic simulation, material-orientation, machine orientation, process-interaction, create successor.

Pattern Name: *Each workload item creates its own successor.*

Intent: This pattern describes a technique for simulating arrivals of workload items in open systems.

Motivation: We want to build a discrete event model in which transient workload items arrive, are being processed and leave according to specified patterns.

Applicability: This pattern is applicable to any model which requires streams of arrivals of transient entities.

Participants: Arrivals are generated by letting each workload item generate its own successor immediately after arrival, scheduled for execution after a suitable model time delay. By using recorded arrival times the same pattern can also be applied to trace-driven simulations.

Collaborations: Workload objects collaborate with their own successors, which they create immediately after arrival, but suspend until it is time for the next item to appear. This interval can determined deterministically (i.e. from traces) or stochastically.

Consequences: This is the simplest and most popular means for generating "traffic" in open models.

Implementation: This pattern can be easily implemented in any simulation language.

Sample Code and Usage:

The following code gives an implementation in Demos/Simula.

```
ENTITY CLASS WorkloadItem;
BEGIN
  REF (WorkloadItem) fred;

COMMENT: declare any other properties of workloaditems;
fred :- WorkloadItem NEW (any arguments);
fred. SCHEDULE (some delay);
COMMENT: start this workload item's lifecycle;
...
```

```
END of class WorkloadItem;
```

Known Usage: This is the most common pattern for generating traffic in all open discrete event simulation models.

Related Patterns: trace-driven simulation, distribution-driven simulation, material-orientation, machine orientation, process-interaction, source-driven workload creation.

4.1.4 Three patterns for synchronization of processes in discrete event models

Pattern Name: *producer-consumer pattern for process synchronization.*

Intent: This pattern describes a technique synchronizing two active processes, in which one is charged with production and the other with consumption of workload items.

Motivation: We want to build a discrete event model in which processes of production and consumption or requests and fulfilments occur.

Applicability: This pattern is applicable to any scenario with multiple cooperating processes.

Participants: At least two and possibly more processes are involved in this patter, classed as wither "producers" or "consumers".

Collaborations: "Producers" deposit items which are to be processed by "consumers". While producers will never be delayed consumers must wait for producers if there are no items to consume. Note that the same process may well be both producer and consumer in different contexts.

Consequences: This is a common pattern in models of manufacturing systems or computer system simulations (e.g. for modelling memory usage or access to files). The "buffers" in which items will be delivered and from which they are taken can either be represented as counters (in which case we have a "semaphore") or as containers in which the identity of workload items may be preserved (see also the "master/slave" pattern).

Implementation: This pattern can easily be implemented in any simulation language.

Sample Code and Usage:

The following code gives an implementation in Demos/Simula. Here we assume a simple semaphore pattern, i.e. Workload items are not represented explicitly but are only modelled through counters (so called BINs).

```
REF (BIN) Buffer;

ENTITY CLASS Producer;
  BEGIN
  WHILE condition DO
    BEGIN
    COMMENT: ... Whatever producers do;
    HOLD (production time);
    Buffer.GIVE (1);
```

```
    END of production;

END of class Producer;

ENTITY CLASS Consumer;
BEGIN
  WHILE condition DO
    BEGIN
    Buffer.TAKE (1);
    HOLD (consumption time);
    COMMENT: ... Whatever consumers do;
    END of consumption;

END of class Consumer;

NEW Producer("tom").SCHEDULE (NOW);
NEW Consumer("dick").SCHEDULE (NOW);
...
```

Known Usage: This pattern is common in manufacturing and computer simulation models.

Related Patterns: trace-driven simulation, stochastic simulation, material-orientation, machine orientation, process-interaction, customer/resources, master/slave synchronization, condition queues.

Pattern Name: *master/slave synchronization of processes*

Intent: This pattern describes a technique for coordinating the cooperation of multiple processes in discrete event simulation models.

Motivation: We want to build a discrete event model in which multiple processes participate in shared activities.

Applicability: This pattern is applicable to any scenario which contains multiple cooperating processes.

Participants: We partition all processes in two categories: masters and slaves. Masters control slaves while their lifecycles are joined (i.e. They remain active while slaves become passive). Note that masters may act as slaves and slaves as masters in different contexts within the same model, and that until taken slaves may wait for more than one class of master and masters may well coopt different types of slaves.

Collaborations: All joint activities are described from the masters' perspective; slaves only have done things to them. It is the masters' responsibility to "free" slaves once joint activities have been completed. Both masters and slaves will then again follow life cycles of their own. Note that we need two queues, for waiting slaves and masters respectively. Slaves need to wait until masters coopt them and masters must wait until slaves are available.

Consequences: This is a useful pattern in any model in which cooperation among multiple processes occur.

Implementation: This pattern can be implemented in any process-oriented simulation language, but needs some infrastructure to make it convenient to use.

Sample Code and Usage:

The following code sketches an implementation in Demos/Simula. Note that master/slave synchronizations are well supported via the WAITQ abstraction.

```
REF(WAITQ) idlingSlaves;

ENTITY CLASS Slave;
BEGIN
  COMMENT: ... Whatever slaves do;
  idlingSlaves.WAIT;
  COMMENT: ... Whatever slaves do after they are freed;
END of class Slave;

ENTITY CLASS Master;
BEGIN
  REF (Slave) fred;

COMMENT: ... Whatever masters do;
fred :- idlingSlaves.COOPT;
COMMENT: whatever masters & slaves do together;
fred.SCHEDULE (NOW); COMMENT: set fred free;
COMMENT: whatever masters do after they've freed a slave;
END of class Master;
...
```

Known Usage: This is useful pattern in any model in which activities need the participation of multiple processes.

Related Patterns: trace-driven simulation, stochastic simulation, material-orientation, machine orientation, process-interaction, producer/consumer, condition queues, customer/resources.

Pattern Name: *using condition queues for process synchronization*

Intent: This pattern describes a technique for synchronizing the interactions of multiple processes.

Motivation: We want to build a discrete event model in which multiple processes are engaged in activities.

Applicability: This pattern is applicable to any scenario in which more than one process is involved. It can even be usefully applied if there is only one class of process with many instances.

Participants: Processes wait in a queue, which will either be scanned in fixed intervals (or whenever the state of the model changes) by the monitor [called "active waiting"] or whenever signalled by processes which have been involved in a relevant change [called "passive waiting"].

Collaborations: Many different processes may be involved in this pattern, as well as the monitor and one or more queues.

Consequences: If active waiting is used execution speed can slow to a crawl (e.g. As in some versions of GPSS). Passive waiting increases the programmer's burden, since she must make sure that she gets all the signals right. In both cases it will depend on the composition of the queues if only the first waiting item (i.e. Sufficient if all wait for the same condition) or all items need to be scanned (i.e. If waits for different conditions are involved).

Implementation: This pattern can be implemented in any simulation language. It is "lower level" than other synchronization patterns, so that all of them can be defined in term of condition queues.

Sample Code and Usage:

The following code sketches an implementation in Demos/Simula. We assume passive waiting and test only the block on the first process waiting in line.

```
REF(WAITQ) q;

ENTITY CLASS Process;
BEGIN
  COMMENT: whatever processes do;
  q.WAITUNTIL (some complex condition);
  COMMENT: whatever processes do after;
END of class process;

COMMENT: ... from somewhere in the model when something has changed;
q.SIGNAL;
...
```

Known Usage: This is a fundamental pattern for all types of process synchronizations.

Related Patterns: trace-driven simulation, stochastic simulation, material-orientation, machine orientation, process-interaction, master/slave, producer/consumer, customer/resources.

Pattern Name: *customer/resource synchronization of processes*

Intent: This pattern describes a technique for coordinating multiple processes competing for capacity constrained resources.

Motivation: We want to build a discrete event model in which multiple processes compete for resources.

Applicability: This pattern is applicable to any queuing scenario.

Participants: We coordinate the interaction of processes by guarding their access to activities through resources, which they must acquire, hold and release.

Collaborations: There may be any number and types of processes and any number and types of resources.

Consequences: This is a useful pattern in any model in where access to limited resources is involved.

Implementation: This pattern can easily be implemented in any simulation language. Depending on the modelling perspective and type of information we wish to obtain resources can be represented by boolean flags, counters, passive entities with multiple attributes, or active processes with a capacity property.

Sample Code and Usage:

The following code sketches an implementation in Demos/Simula. In this case we have used the built-in RES abstraction, which models a passive resource entity with automatic statistics collection (e.g. Utilization, queuing length and times ..).

```
REF(RES) someObjectWeMayDesire;

ENTITY CLASS Customer;
BEGIN
COMMENT: ... Whatever customers do;
  someObjectWeMayDesire. ACQUIRE (1);
COMMENT: ... Whatever we may want to do;
  HOLD (howeverLongItMayTake);
  someObjectWeMayDesire. RELEASE (1);
COMMENT: ... Whatever customers do afterwards;
END of class Customer;

someObjectWeMayDesire :- NEW RES ("Me", 1); COMMENT: a single server;
```

Known Usage: A very common pattern in all queuing models

Related Patterns: trace-driven simulation, stochastic simulation, material-orientation, machine orientation, process-interaction, master/slave, producer/consumer, condition queues.

4.2 A Pattern for Simulator Design

All patterns discussed in the previous section are applicable at the modelling construction level (see figure 9). Mapping them into "lower level" implementations so that the models in which they occur can be simulated requires a simulator, i.e. an execution monitor whose responsibility it is to "move" model entities through simulated time. As part of this process it must step through the life cycles of conceptually active entities and trigger all relevant changes in state. To accomplish this task a discrete event monitor uses a number of components, such as a clock and agenda, and needs a suitable strategy for time and event management (see figure7). Although simple sampling or time slicing monitors, where the value of some sequencing parameter is always stepped up by the same increment and the whole model is evaluated after each of these steps (e.g. the "game of life" or a spreadsheet), will suffice for highly aggregate models or ones with a simple structure, a next event monitor pattern is needed to support more complex designs. A number of different approaches have been applied to this task (e.g. event, activity and process orientation), but we here we will only describe the simplest of these. So called event-oriented monitors are a common design pattern for home-spun discrete event simulators, largely because they are easy to implement, offer efficient execution, and do not require any unusual language support. Unfortunately they require their users to provide and remember many details in order to communicate a model’s design, which is the reason why process orientation is now generally used in commercial systems.

Pattern Name: *Event-Oriented Time Management in Discrete Event Simulations.*

Intent: This pattern serves as a blueprint for building monitors which control model time advance and execution sequences of event routines; where event routines describe changes of state in a discrete event simulation model.

Motivation: We want to build a time-dynamic discrete event simulation to trace how causally connected changes in a system of interest occur over a period of time; for example in order to study the process of queuing to obtain service from a set of capacity constrained resources.

Applicability: This pattern is applicable to any discrete event simulation model.

Participants: Figure 12 shows all participants in this pattern: clock, agenda, event notices, event routines and monitor objects. The clock serves to keep track of model time. Event notices store information about events in terms of their time of occurrence and references to a procedure describing how changes of state are performed (there can be many such "events" for each class of event). Event routines (i.e. one for each class of events) describe the nature of any such changes, and an agenda collects event notices (for any number of any number of types of events) together - sorted on time of occurrence. Note that executing event routines will cause changes in model state and may also invoke scheduling statements which cause new event notices to be made and put on the agenda. The monitor's task is to coordinate interactions between all these objects.

Collaborations: All other objects' life cycles are coordinated by the monitor, which cycles through the following three activities:

- get next event notice (i.e. the one on top of the agenda),
- update the clock to this event notice's time aspect (i.e. the moment at which the next relevant event is expected to happen),
- transfer control to the relevant procedure (event routine)

until the modeller asks it to stop (e.g. because some time or event threshold is reached) or the agenda becomes empty. Executing event routines may result in new event notices being made and placed in their proper position on the agenda; i.e. after all event notices with a time aspect smaller or equal to its own. Variations of this strategy may attach different priorities to types of events.

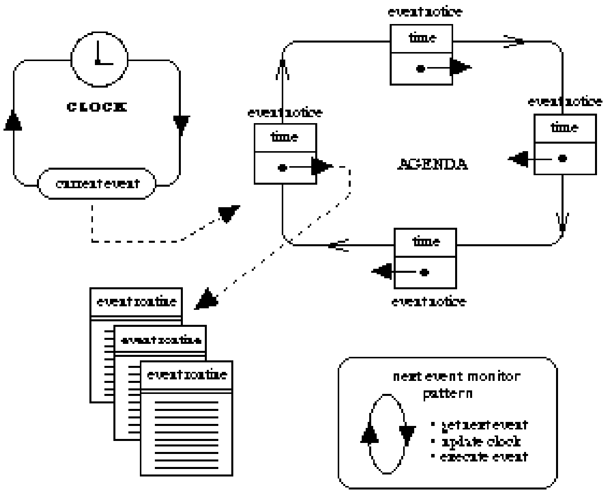

Figure 12: Components of an Event-Oriented Simulation Monitor

Consequences: This pattern offers a common compromise for building next-event simulation monitors. While it has relatively low time and space overhead, simple time-slicing patterns are more efficient at the execution level while activity- and process-oriented monitor patterns offer lower mental complexity at the model construction level.

Implementation: The popularity of this pattern is due to the fact that it can easily be implemented with any general purpose programming tool. Although languages with good list handling facilities and provision for dynamic creation of event notices at model execution time make it easier to apply, monitors based on this pattern have even been built in languages such as Fortran (where lists of event notices can be simulated by fixed-size arrays).

Sample Code and Usage: The following code fragment is written in Simula (augmented with simulation facilities), based on the architecture shown in figure 12. Language keywords are capitalized.

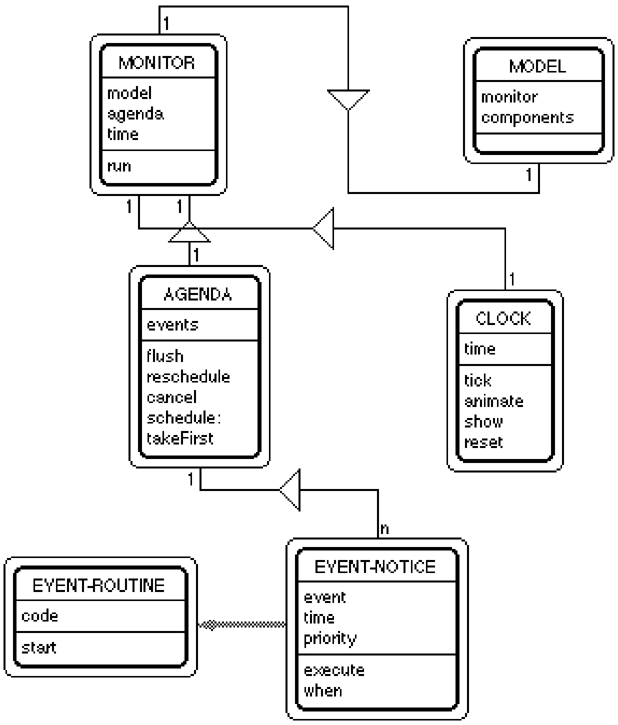

Figure 13: Classification and Composition Relationships of a DES Monitor

```
CLASS Monitor
BEGIN
  REAL simtime;
  REF (Model) m;
  REF (Clock) c;
  REF (Agenda) a;

WHILE (clock.time) < simtime DO
  BEGIN
    REF (EventNotice) e;

  e :- a.takeFirst;
  c := e.time;
  e.event.start;
  END of monitor cycle;

END of class Monitor;
```

Note that we assume the existence of classes Model, Clock, Agenda, EventNotice and Event - with suitably instantiated objects, properties and behaviours (see figures 12 and 13).

Known Usage: This pattern is often applied or reinvented whenever simulators are built from scratch, and there are many hand-crafted and even commercial systems which use it. Since it pushes an inappropriately high mental burden onto the modeller (who will need to keep track of all interactions among the many different events into which relevant processes have been decomposed), with often negative consequences for model correctness, modern discrete event monitors use activity or process-oriented patterns instead.

Related Patterns: time slicing, next-event time advance, array-structured agenda, list-structured agenda, tree-structured agenda, activity-oriented monitor, process-oriented monitor.

5. CONCLUSIONS

Patterns provide abstractions which codify knowledge so that it can be better understood, taught and reused. In this paper we have argued that many well established techniques in the simulation domain can be summarized in this fashion, and we have also begun to sketch the structure and contents of pattern languages for model and simulator design; a framework which we intend to develop more fully in future. To give some examples we looked at both the model design and simulator construction domains and described a number of patterns which modellers frequently use at each of these levels. When looking at the code fragments we gave as examples it should have become obvious that these patterns are nested, i.e. more specific ones (e.g. using different data collectors, or choosing a "source" or a "generate next arrival" pattern for workload creation) can be "plugged" into more general architectural frameworks. In our experience this is how any non-trivial pattern language should be designed; i.e. In terms of a collection of patterns at different layers where each of the patterns has slots whose instantiation with "lower level" patterns can be easily changed - depending on purposes and resources.

While such pattern languages offer many benefits to both students and practitioners of simulation modelling it may be worth stressing that all patterns must be treated as guidelines not dogma. In the same way that all models are abstractions which can only be labelled as good or bad if we know the purposes which they are meant to serve, patterns are intimately tied to the cultural commitments of their authors and the communities which they are expected to serve. One of the greatest benefits of a pattern language is that it makes such shared contexts explicit. Since reality is far from orderly or neat we can rarely use patterns in their pure form. Rather they must be adapted to the task at hand, which becomes very much easier if they are based on an object oriented design.

Implementing a pattern will inevitably be influenced by the strengths and weaknesses of whatever tools we have chosen. In our examples we have used Simula, but other languages may resolve issues differently. Generally we should strive for the simplest and cleanest design we can find, but simplicity may of course be very much in the eye of the beholder (i.e. which structures are pleasing and familiar to us). Different perspectives offer different insights and must be allowed to prosper. For patterns to become successful, however, they must be known and used widely, and a number of conditions must be met to make this more likely. Firstly, patterns must be easy to understand by their target audience. They are abstractions which are only useful if they make sense of experience. This means that they must be explained in familiar terms and must not depend on context which may not be shared. Understandability also requires good and readable documentation with well chosen examples. The problem is that one should not begin to with those patterns which are the most general and abstract, but must first present some more concrete ones, and find examples to motivate and situate them. If relevant experience is lacking, abstractions often do not make much sense. Patterns should therefore serve to establish a vocabulary so that they can then act as a shorthand for people who are already knowledgeable in a subject, or they should be aimed at teaching good rules of design to beginners. In the second case good examples are particularly crucial. This requirement also stresses the need for environments which foster experimentation - using partially instantiated models which can be derived from and help us to understand the patterns on which they are based. The importance of this aspect of pattern languages has become abundantly clear to us during our own exploration of programming idioms [e.g. see Kreutzer (1990) and Kreutzer (1996)].

Another important aspect of patterns relates to the trust we can put in their correctness. This is an issue whose importance may still be underestimated, particularly where complex algorithms may be involved (e.g. for generating random numbers or integrating differential equations). Trust can be grounded in formal or informal authority of a developer, or it can be established through independent validations. No such infrastructure is currently in place in the modelling area.

Finally, there is the question of "elegance" or style. This aspect of pattern languages is often referred to as "the quality without name" [see Alexander (1977, 1979)]. Although it can not be easily measured it is a very important aspect of any good model. Unfortunately we know of no patterns which can ensure that we reach this ideal; rather all our designs must be viewed as experiments which involve many trade-offs and need time to mature. This means that no pattern should be regarded as static, since it may change, sometimes substantially, as our experience and insight keeps growing. The problem is that computing is such a rapidly changing technology and that market pressures often force premature product deliveries and de facto standardisation. In the simulation domain, however, very few genuinely new patterns have emerged over the last 20 years, with the exception of areas such as interface design and parallelism (which we intentionally have not commented on in this paper). Although tinged with personal preferences and tastes the modelling patterns we have identified have been derived from long experience with building such systems and are now quite stable.

Regarding the patterns' implementation the lack of a coroutine feature in widely used object-oriented languages remains the most serious obstacle for good simulator designs. Simula or Beta are therefore two obvious choices for hosting the implementation of object-oriented modelling frameworks with a process perspective, although this can also be done elegantly in languages which allows us to manipulate continuations directly [e.g. Scheme, see Kreutzer (1990a)] or ones with access to processes and semaphores [e.g. Smalltalk, see Kreutzer (1987)]. Among others Extend [see Krahl (1995)] and ModSim [see Herring (1990)] offer object-oriented language and process-based simulator support, as well as asynchronous methods and synchronization primitives. They would therefore also be excellent alternative choices for hosting the modelling patterns we have identified.

REFERENCES

Alexander, C. Ishikawa, S., Luker, P. (1977) A Pattern Language. Oxford
Alexander, C. (1979) The Timeless Way of Building. Oxford
Birtwistle, G.M. (1979) Discrete Event Modelling on SIMULA. London/Basingstoke
Birtwistle, G.M., Lomov, G., Unger, B., Luker, P. (1984) "An Approach to Discrete Event Modelling" (parts 1 to 4). Transactions of the SCS 1(1): 61 - 82
Coad, P. And Yourdon. E. (1991) Object-Oriented Analysis, Englewood Cliffs
Frankowski, E. and Franta, W.R. (1980) "A Process Oriented Simulation Model Specification and Documentation Language" SOFTWARE Practice & Experience 10: 721 - 742
Franta, W.R. (1977) The Process View of Simulation. New York
Gabriel, R.P. (1994) Pattern Languages. Journal of Object-Oriented Programming. January 1994, 72-75
Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994) Design Patterns: Elements of Reusable Object-Oriented Software, Reading(MA), Addison Wesley
Herring, C. (1990). ModSim: A new object-oriented simulation language. In: Ege, R.K. (Ed.) "Object-Oriented Simulation", SCS
Krahl, D. (1995). Building End-User Applications with Extend. In: Alexopoulos et al. (Eds.) Proceedings 1995 Winter Simulation Conference, 423-419

Kreutzer, W. (1986). System Simulation - Programming Styles & Languages. Reading(MA), Addison Wesley.
Kreutzer, W. (1987). "A Modeller’s Workbench: Experiments in Object-Oriented Simulation Programming." ECOOP’87 - European Conference on Object- Oriented Programming, Springer: Lecture Notes in Computer Science, 203-212
Kreutzer, W. and B. McKenzie (1990). Programming for Artificial Intelligence - Methods, Tools and Applications. Reading(MA), Addison Wesley.
Kreutzer, W. (1990a). "C-Flavours: A Scheme-Based Flavour System with Coroutines and its Application to the Design of Object-Oriented Simulation Software" International Journal of Computer Languages 15(4): 225-249
Kreutzer, W. (1994) "Object Oriented Modelling & Simulation: Towards Reusable Frameworks for Simulator Design" - Invited Keynote Speech - Proceedings ESM94: Object Oriented Modelling and Simulation, Barcelona, SCS
Kreutzer, W., Krishnan, P., McKenzie, B, (1996) An Introduction to Computer Science Using a Functional Programming Language. Draft, Department of Computer Science, University of Canterbury, Christchurch, New Zealand
Madsen, O. L., Moller-Pedersen, B., Nygaard, K. (1993) Object-Oriented Programming in the Beta Programming Language. Reading(MA), Addison Wesley
Madsen, O. L. (1995) "Open Issues in Object-Oriented Programming - A Scandinavian Perspective" Software - Practice & Experience. 25(4), 3-43
Nance, R.E. (1979) "Model representation in discrete event simulation: The Conical Methodology" Report CS81003, Virginia Tech, Blacksburg(VA)
Nance, R.E. (1981) "The Time and State Relationships in Simulation Modelling" Communications of the ACM 24(4): 173-179
Oren, T.I. and Zeigler, B.P. (1979) "Concepts for advanced simulation methodologies" Simulation 32(3): 69-82
Overstreet, C.M. and Nance, R.E. (1983) "A Specification Language to assist in Analysis of Discrete Event Simulation Models" Technical report 83026-R, Virginia Polytechnic and State University, Blacksburg (VA)
Simon, H.A.(1969) The Sciences of the Artificial Cambridge (MA) 1969
Zeigler, B.P. (1976) Theory of Modelling and Simulation London/N.Y./Toronto
Zeigler, B.P. (1984) Multifacetted Modelling and Discrete Event Simulation New York