

SimJAVA - A FRAMEWORK FOR MODELING QUEUEING NETWORKS IN JAVA

Wolfgang Kreutzer
Jane Hopkins
Marcel van Mierlo

Computer Science Department
University of Canterbury, Christchurch, NEW ZEALAND

ABSTRACT

The paper present a layered design for a discrete event simulation framework based on the Java programming language. A description of this project's goals and motivation is followed by a some brief comments on Java's suitability for work in this area. The main body illustrates and discusses the design of the simJAVA framework in the context of a simple queuing scenario. A summary of strengths and limitations of object orientation and Java for this class of application concludes the paper.

1 INTRODUCTION

The importance of conceptual and notational frameworks for guiding and structuring a modeling enterprise can hardly be overstated and a wide variety of programming tools together with a rich repertoire of knowledge about their safe application has long been employed in the design and construction of simulation models. Since the early 1960s we have seen the emergence of many so called simulation programming languages, such as GPSS, CSMP, GASP, Simscript, Simula, CSSL, ModSim and others [see Kreutzer 1986]. Since the rapid pace of technological change has left little time for their design to mature, commercial simulation tools all too often have unfortunately not been based on solid foundations. Instead bits and pieces of new technologies have been added without giving much thought to their interaction with already existing features. Consequently many popular simulation programming systems today are riddled with non-intuitive interfaces and inconsistent and unnecessary restrictions. Better and cleaner designs must be explored, taking advantage of the latest research in both model and software design [e.g. see Winograd 1996]. The simJAVA framework is part of such a project, based on a pattern language for simulation

software [see Kreutzer 1996]. Beyond this it will also serve as a testbed for ideas targeted at providing better modeling tools for the world-wide-web (WWW). In order to tune and refine this design while learning through experimentation, alternative implementations are also developed for other domains (e.g. environmental models) and in other base languages (e.g. Scheme, Smalltalk, and Beta). **Java** [see e.g. Arnold & Gosling 1996] is a programming tool for writing portable web applications which can be embedded in html, a mark-up language used to describe documents on the WWW. While its syntax is similar to C++ it is much more modern in its design, trading many of C's low level features (e.g. casts and pointer arithmetic) and some run time efficiency for better expressiveness and conceptual simplicity (e.g. automatic garbage collection). As a result Java offers a purely object-oriented framework based on the best features of Objective C, Smalltalk, C++ and other object-oriented programming languages. Goals for its design include easy cross-platform portability as well as a strong focus on software security, reliability and simplicity. Java's core functionality can be augmented via libraries called "packages", whose number and quality is rapidly growing. One of Java's most interesting features is its support for concurrency through parallel threads of execution, which is particularly useful for hosting process oriented simulation software. Java's close association with the WWW makes it also a very attractive delivery vehicle for teaching.

2 THE simJAVA FRAMEWORK

There are two ways to "cope" with complexity in models and programs, i.e. simplification and delegation of tasks. While delegation lets us automate all of the many "mechanical" tasks faced by a programmer, simplification is based on abstraction, i.e. the removal or aggregation of detail, components and/or relationships. To decrease mental complexity in this fashion we must partition a system into layers, where we can then treat concepts as

primitives, while decomposing them into sub systems at the layers below. Object oriented descriptions offer an elegant and powerful metaphor for organizing such layers of knowledge through locality of description; i.e. by restricting the scope of the context a programmer needs to consider at any point in time. *Encapsulation* ensures that objects can be insulated from a surrounding context by allowing access to internal representations (ie data and methods) only through well defined interfaces; the so called message protocols. Since message names can be disambiguated by context, i.e. their "meaning" depends on the class of receiver, we can make use of *polymorphism*, where generic names can be employed to denote different actions of similar type; linked to different implementations for different kinds of objects (e.g. "update" for different types of data collection objects). Finally, *inheritance* of structure and behaviour along class/subclass links as well as chunking components into higher-level composites make it easy to extend systems by adding new layers. This strategy enables us to bridge large semantic gaps between linguistic support for a concrete domain (e.g. queuing networks) and some general purpose but abstract implementation language (e.g. Java) via a sequence of short intermediate steps (e.g. utility packages) This strategy offers both an elegant means for successive composition from "lower level" concepts towards ever more specific, application related features (a "bottom up" approach) and a pattern for successive decomposition of "higher level" functionality into ever more basic building blocks (a "top down" approach). Using these features skilfully can result in better structured and more flexible models and programs, which are potentially more reliable, easier to understand, change and maintain.

simJAVA uses this strategy to compose increasingly more specialised components (e.g. for data collection, distribution sampling...), which may serve as building blocks for "higher level" components (e.g. a discrete event monitor or a resource in a queuing network). Throughout the design of the patterns on which this framework is based we have tried to identify layers and concepts with which the description of given classes of applications becomes simple, while providing extensibility through recursive composition in terms of these underlying structures. If such abstractions are carefully chosen one may start with a small base of general and flexible ideas, which are elaborated through a number of layers until they support convenient frameworks for specific applications (e.g. queuing scenarios). while ensuring that no irrelevant details (e.g. how a server is implemented) will "leak" (i.e. no unnecessary knowledge of implementation is revealed) to a higher layer and that all abstractions are "safe" (i.e. they can not be easily misapplied). This approach offers a good compromise between the often conflicting needs for generality and abstraction on the one hand, and familiarity and concreteness on the other. Users can now choose to "operate" at the level they feel most

comfortable with, based on their needs, preferences and experience. In this way we hope to come closer to the elusive goal of crafting tools which have "low thresholds and high ceilings" [Fischer 1989].

Figure 1 summarizes simJAVA's features from this perspective. Here the lowest layer is derived from the *host language's linguistic abstractions*. At the next layer a range of *model instrumentation* tools can be used for statistics collection and their display. In addition to these most basic features a *monitor* object will also be needed; e.g. to advance model time and start, control and terminate all processes' execution. Based on this core, different styles of model construction have their own special needs. So called stochastic simulations, for example, use probability distributions to reflect the effects of all those aspects which are not considered important enough to warrant more detailed causal descriptions. For such models appropriate frequency or probability distributions (i.e. a *sampling layer*) must be supplied. All layers define classes in terms of a protocol of messages to which they react (e.g. `sample` for all distributions). Implementation of this functionality can conveniently draw on the features provided at levels below (e.g. on the data collection layer for recording and summarising sampling statistics). Process-oriented discrete event simulations extend this perspective by viewing a model as structured collections of active and passive entities, bound into webs of relationships which define transformations and patterns of interaction across both space and time (i.e. a *synchronisation layer*).

LAYER:	OBJECTS:	PATTERNS:
●●●	●●●	●●●
scenario layer	e.g. resources, transactions, schedules, priorities ..	open & closed qns, event- & process-orientation ..
synchronization layer	conditions, interrupts, server groups ..	rendez-vous, master-slave ..
sampling layer	draw, uniform, exponential ..	sampling, transformation ..
monitor layer	entity, clock, agenda, processqueue ..	time slicing, nest event ..
instrumentation layer	tally, histogram ..	observation, presentation ..
Java layer	class, interface, thread ..	data abstractions, control abstractions ..

Figure 1: Layers, Objects & Patterns in simJAVA

Queuing scenarios are a well known example of a subclass of discrete event scenarios in which specialised

abstractions (e.g. transactions, queues and resources) describe the effects of capacity limitations and routing strategies on the interactions among workload items and server objects. Note that this layer can easily be further extended to cater more directly for more specialised scenarios, such as the analysis of communication networks, the modeling of material flows in a factory, or the simulation of office procedures. In such cases suitable class definitions and patterns of combination as extensions and compositions of the existing ones would need to be provided. Note that although classes and patterns at lower layers seem to be stable simJAVA's design has not reached "the end of the road" yet, since a substantial amount of experimentation with concepts over some period of time is needed for "good" reusable components to evolve (often trading "local convenience" and efficiency against increased generality and reusability). This is also the only way to design "habitable" interfaces and "revealing" graphical representations, work on which we have started more recently.

To demonstrate JavaSIM's modeling style we will now show how it can be used to represent a simple queuing scenario. 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. Although this will most likely be a colourful affair with many intriguing aspects, from a suitably abstract perspective we may only be interested in predicting the party's success in terms of a few abstract measures; such as the Queen's utilization and the average duration of the guests' stay. Assuming some empirical basis for summarising the dignitaries' arrivals and the duration of handshakes through distributions, we can then reflect "the essence" of this event in a queuing model - with the Queen's hand as a server of capacity one. Let us further assume that after a shake some of the dignitaries may wish to repeat this experience, while others will hurriedly leave. Figure 2 shows a life cycle diagram for this scenario. The meaning of symbols should be clear; i.e. circles denote states and boxes activities, which may bind resources at the start and release them once they are finished.

Before we cast these events into a JavaSIM model you should note that even this simple scenario could be described from a number of different perspectives. The one we have chosen is often referred to as *material orientation*. The roots of this term reach back to 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 (ie they have a life cycle and must be modelled as processes), while "materials" are transient and passive (i.e. they store only properties, are operated upon by machines, and can

be modelled as data). From a material-oriented perspective a dignitary's life cycle is a recurring sequence of queuing for, grabbing, holding, and releasing the monarch's hand; until it is time to leave. simJAVA supports both material and machine-oriented viewpoints.

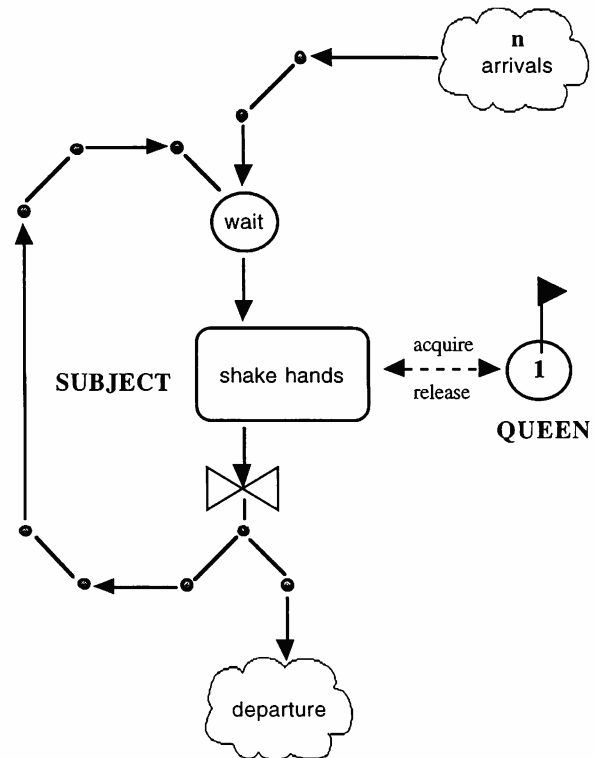


Figure 2: A Queen's Gardenparty

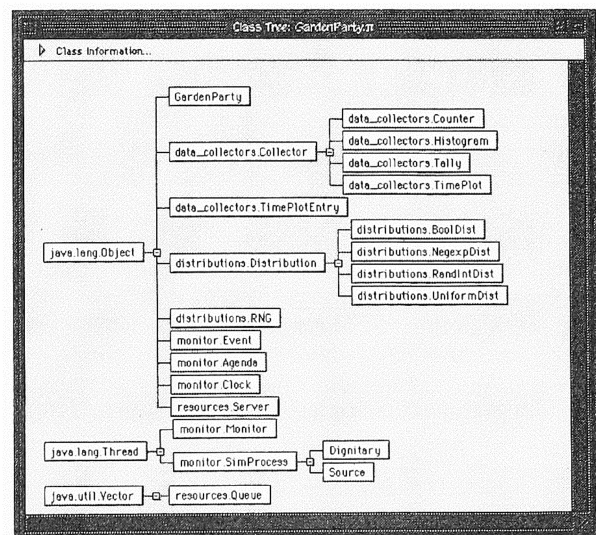


Figure 3: The JavaSIM Package

Before we can use the simJAVA framework's components in models we must import them. Figure 3

shows the relevant packages' structure and the following Java code implements the model. Terms shown in bold represent Java keywords, while italicised ones are part of simJAVA's built-in abstractions. Note that simJAVA collects and presents many standard statistics on lengths and delays in a queue as well as all resources' utilization. However, if we are also interested in the mean time a dignitary will idle away at the party we must record this explicitly by using a data collector (e.g. a Tally such as `time_at_party` - to summarise time series data).

```
// a garden party simulation.
// Times are in seconds.

import distributions.*;
import data_collectors.*;
import monitor.*;
import queues.*;

class Dignitary extends SimProcess {
    static Server queen;
    static NegexpDist shaketime;
    static BoolDist goback;
    static RandIntDist intershaketime;
    static Tally time_at_party;

    public Dignitary(String name, Monitor aMon)
    { super(name, aMon); }

    public void run() // a life cycle
    {do
        { queen.Acquire(this);
          hold(shaketime.sample());
          queen.Release();
          hold(intershaketime.sample());
          while (goback.sample());

          System.out.println(getName() + "enough ");
          time_at_party.Update(now() - birthTime); }
    }

class Source extends SimProcess
{ // used to generate dignitaries

    static NegexpDist interarrival;

    public Source(String name, Monitor aMon)
    { super(name, aMon); }

    public void run() // a life cycle
    {int i;
      Dignitary thisVIP;

      System.out.println("Gate opens!");

      for (i = 0; i <= 20; ++i)
        {hold(interarrival.sample());
         thisVIP = new Dignitary("fred" + i,
                                 aMon);
         thisVIP.schedule(now()); } }

class GardenParty
{ // the "driver class"
```

```
public static void main(String[] args)
{ // Instantiate & initialise entities
  Monitor james = new Monitor();
  Source gate = new Source("Garden Gate",
                           james);

  Source.interarrival
    = new NegexpDist("Inter-Arrival", 60);
  gate.run();
  Dignitary.queen = new Server("The Queen",
                               james);

  Dignitary.shaketime
    = new NegexpDist("Shake Time", 10);
  Dignitary.goback
    = new BoolDist("Go Back?", 0.5);
  Dignitary.intershaketime
    = new RandIntDist("Inter-shake time",
                      30, 300);
  Dignitary.time_at_party
    = new Tally("Time spent at party");

  // start a simulation
  james.begin();
  // (stops once all dignitaries have left)
  Source.interarrival.show();
  Dignitary.goback.show();
  Dignitary.intershaketime.show();
  Dignitary.shaketime.show();
  Dignitary.time_at_party.show();
  Dignitary.queen.show(); } }
```

This model is composed of three classes: Dignitary, Source and GardenParty. The GardenParty sets up and drives the simulation. Since this is a self-contained Java program and not an applet it contains a *main* method at which execution will start. In simJAVA the *main* method forms the "body" of the simulation, which instantiates (via *new*), initialises (via assignment) and activates (via *schedule* or *run*) other simulation objects. Here we have two process descriptions. The first, called Source, models a gate through which dignitaries arrive, governed by a specified interarrival time distribution. Once it is time, Source instances (here a single one referred to as *gate*) will generate and activate 20 arrivals (as wired into our Source's code) before they snap shut. Arriving dignitaries step through the life cycle shown in their *run* method, i.e. they try to acquire the queen, engage here in a shake, and release her again, until their energy or patience runs out. Note that a *hold()* message is used to model delays and that the resource representing the Queen will keep track of her queue and utilization. Note also that she is represented by one of class Dignitary's class variables (i.e. as static). Alternatively we could have chosen to make her a direct GardenParty component. A monitor (here referred to as *james*) must be started before the simulation begins moving through time. The command `james.begin()`; accomplishes this.

Active model components (i.e. those with a life cycle)

must be defined as a subclass of *SimProcess*, so that they can be delayed and scheduled in terms of model time. simJAVA models use a monitor to advance model time and control all events' execution, making sure that all actions, of all active phases, of all conceptually parallel processes are performed in the right sequence. simJAVA views a model's behaviour as defined by a collection of active entities executing their life cycles. Although this is a "natural" view of the world it needs support for expressing concurrency and process synchronisation (e.g. coroutines). While most modern simulation languages offer such features, they are usually lacking in general purpose programming tools, making implementation of process-orientation an awkward task. simJAVA draws on Java's *thread* concept to provide the necessary infrastructure for this approach. Figure 4 and the associated monitor's *run* method's code document the resulting architecture, which reflects a typical discrete event monitor pattern quite cleanly. Note that the term *event routines* refers to process descriptions here, each of which keeps a reference to the code it will resume at when it is activated next, while holding and scheduling processes will suspend them until they can be resumed by their monitor.

The *run* method's operation should be easy to understand, with the main interest focussing on how threads are being used. Note that Java relies on the underlying operating system's services to implement threads and that we must force them to give up control at strategic points to avoid relying on preemptive multi-tasking by the host's operating system - e.g. without a *Thread.yield()* statement the above code would not work on a MacIntosh (since the first thread would run to completion and stop the simulation before any other could gain control).

```
public void run() {
    Event next;
    while (!(agenda.events.isEmpty()) &&
           (clock.time <= simTime)) {
        // get first event on the agenda
        next = agenda.getNextEvent();

        // update the clock to this event's time
        clock.set(next.when());

        // continue next waiting process (event)
        if (clock.time <= simTime)
            { next.execute();
              Thread.yield(); }
        else
            { next.proc.resume();
              next.proc.stop(); } }
}
```

```
// simulation has stopped.
// kill all processes on agenda
for(int i=0 ; i<agenda.events.size() ;
    i++) {
    ((Event)agenda.events.elementAt(i)).
        proc.resume();
    ((Event)agenda.events.elementAt(i)).
        proc.stop(); }

// resume & stop all queued processes
for(int i=0 ; i<inTransit.size() ;
    i++) {
    ((SimProcess)inTransit.
        elementAt(i)).resume();
    ((SimProcess)inTransit.
        elementAt(i)).stop();}
inTransit.removeAllElements(); }
```

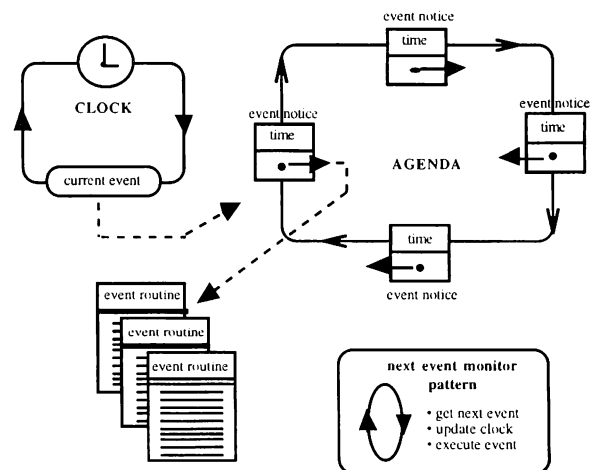


Figure 4: A Simulation Monitor's Components

3 CONCLUSIONS

The recent dramatic shift in the cost ratios of hardware, software and human resources has challenged traditional views of the goals and processes of model development. Since ambition invariably taxes the limits of technology there is a sore need for tools which allow us to cope with higher levels of model and program complexity in a reliable fashion. The modeling framework described in this paper is a step towards this goal in that it eases model development and serves to relieve the modeller from some of the many routine tasks associated with her craft. Java, a modern object-oriented language supporting multiple threads was chosen as a programming tool for this project, and all software has been written in an object-oriented style. The resulting simJAVA package is very flexible. Encapsulated and reusable components can be specialised and extended to make rapid composition of models an easy and painless task, while standard components can be reused and more specialised subclasses can be evolved to cater for specific requirements. In this way special purpose libraries can be

built, whose components are defined, tested and debugged incrementally and interactively. By using classes of objects to encapsulate closely related information as manageable and separately testable chunks we prevent mental complexity from growing exponentially with a model's size. Separate name spaces for all classes of objects ensure that naming of simJAVA's classes, their properties and messages do not prejudge any choices made in whatever context these objects will later be used in. Such polymorphism means that identifiers can be short and informative, e.g. all distributions respond to `sample` and all data collectors to `update`. Although implementations of these methods may differ considerably from class to class, this need not concern the user in any way. Inheritance of structure and behaviour enables us to import appropriate functionality from superclasses, which means that common aspects need to be defined only once and can be kept localised in a single place. This has the nice side effect that consistency becomes easier to ensure and any consequences of changes are less likely to be overlooked.

What should be appreciated is the ease with which complex simulations can be defined in this fashion, and how easily the framework can be assembled and extended once "the right" building blocks have been found. Identifying and casting them in convenient linguistic abstractions is an ongoing task for which much work is still needed; particularly with regard to graphical instrumentation and GUI-style model development tools. Our experiences during this project have firmly convinced us that object-orientation and a process-based perspective is an appropriate framework for this domain. Since discrete event models are best represented by sets of conceptually "active" entities we have made good use of Java's threads to support this view of the world.

While simJAVA's core has been patterned after a conventional discrete event simulation framework [e.g. see G. Birtwistle's (1979) Demos system] Java's support for graphical interfaces (i.e. its AWT package) and applets embedded in web pages will allow us to go some way beyond this. Java 1.2's support for nesting of methods and classes in the style of Beta and Simula will provide further opportunities to explore better linguistic abstractions for process synchronisation.

We strongly believe that it is futile to attempt to anticipate all the services a given class of users will ever require. Instead of freezing such a decision in rigid language designs we are convinced that good tools must provide safe and convenient means for their own extension; thereby allowing new layers to be added and existing abstractions to be refined. Simula's class libraries, the Smalltalk programming system and the simJAVA toolbox we have discussed in this paper demonstrate the power of this approach, which can only be harnessed safely if the conceptual gap between layers is small. Of course, this may require many such layers to coexist, which will have a negative impact on

computational efficiency. This problem is a pervasive and persistent one and needs to be addressed. Exploring the relevant tradeoffs more deeply and reflecting about the "appropriateness" of particular modeling styles (e.g. what levels, what entities, what functionality, how to partition functionality, how to describe interaction and synchronisation ...) requires much further thought and experimentation. Design, analysis, implementation, and empirical exploration of reusable and "higher level" application frameworks is an essential ingredient to this research and simJAVA as well as similar work in this area (see e.g. Howell and McNab 1996) tries to make such a contribution to the discrete event simulation domain.

REFERENCES

- Arnold, K., and J. Gosling. 1996. *The Java Programming Language*. Reading(MA): Addison Wesley.
- Birtwistle, G. 1979. *Discrete Event Modeling on Simula*. Basingstoke: McMillan.
- Fischer, G. 1989. Human-Computer Interaction Software: Lessons Learned, Challenges Ahead. *IEEE Software*, 44-52.
- Howell, F., and R. McNab. 1996. *A Guide to the simjava Package*. Department of Computer Science, University of Edinburgh, www.dcs.ed.ac.uk/home/hase/simjavasimjava-1.0/
- Kreutzer, W. 1986. *System Simulation - Programming Styles & Languages*. Reading(MA), Addison Wesley.
- Kreutzer, W. 1996. Towards a Family of Pattern Languages for Simulation Software Design. *Proceedings OOIS'96*, 387-398. Berlin: Springer.
- Winograd, T. 1996. *Bringing Design to Software*. Reading(MA): Addison Wesley.

AUTHOR BIOGRAPHIES

WOLFGANG KREUTZER is an Associate Professor of computer science at the University of Canterbury in New Zealand. His current research centres on software design, simulation modeling tools, visual programming languages and computer science education.

JANE HOPKINS is an Honours Student at the University of Canterbury in New Zealand.

MARCEL V. MIERLO is an Honours Student at the University of Canterbury in New Zealand.