Visual & Interactive Simulation Package

Suleyman Sevinc
Petr Hejda

Technical Report 516
3 October 1997

BASSER DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF SYDNEY
Visual & Interactive Simulation Package

Süleyman Sevinç
Petr Hejda
Basser Department of Computer Science
University of Sydney
Australia
Visual & Interactive Simulation Package

Suleyman Sevinc, Petr Hejda

October 3, 1997

1 Introduction

This report presents work done for DSTO Maritime and Aeronautical Research Labs in Melbourne over the past one year. Some publications relating to work done in earlier years are also appended to the report because of the strong dependency in content and concepts.

The generic title of the research project is "An Object-Oriented Framework for Naval Mine Warfare Modelling and Simulation". Since the research collaboration started between the University of Sydney and the DSTO in 1992, number of people have been involved in discussions and developments of the ideas, the problems and potential solutions. In particular Dr. Don Richardson and Dr. Peter Ryan of DSTO have contributed immensely to our understanding of issues in naval mine warfare modelling. They provided valuable assistance in making sure that the project went on time and within its scope. Their contribution to the identification of the specific problems that this report presents and to the solutions of these problems is also acknowledged.

This report presents current theoretical developments for handling of Emergent Events which proved to be one of the key concepts to be dealt with in order to come up with frameworks for naval mine warfare modelling and simulation that generate valid simulations of reality. Our current work builds upon our earlier work but is simpler and more developed as we gained more understanding of this concept.

Our earlier work was resulted in a framework for which a prototype was implemented by extending Microsoft Visual Basic engine, i.e., designing and building of control files. Internet related technologies have been rapidly developed over the past few years which provided facilities for more cooperation and integration. Consequently we carried our work over to Java and World Wide Web. Theoretical concepts developed for emergent event processing are then added which resulted in Visual and Interactive Simulation Package (VISP). This report describes both design and use of VISP as well. An on-line version of VISP currently exists as cited in the text and the reader is encouraged to try it out.
Contents

1 Introduction 1
  1.1 Background .................................................. 5
  1.2 Challenges .................................................. 5
  1.3 Structure of this report ................................. 5

2 Problems and Solutions 7
  2.1 Collision Detection ......................................... 7
  2.2 Teaching Environment ..................................... 7

3 VISP Software Package 9

4 VISP Site 11
  4.1 Users' Visualisation ....................................... 11
    4.1.1 Main .................................................. 11
    4.1.2 VISP ................................................ 12
    4.1.3 Tutorial ............................................. 12
    4.1.4 Documentation ...................................... 12
    4.1.5 Feedback ............................................ 12
  4.2 Used Techniques ........................................... 12
  4.3 Common Layout ............................................. 12
  4.4 Dynamic Parts ............................................. 13
    4.4.1 Database of Pictures ................................ 14
    4.4.2 Database of Simulation Entities ....................... 15
    4.4.3 Database of Scenarios ................................ 18
  4.5 Security ................................................... 20
  4.6 Storing Data on Server .................................... 21
  4.7 Log Files .................................................. 22
  4.8 Re-compilation - javac.cgi ................................ 22

5 VISP Applet 23
  5.0.1 Conventions ............................................. 23
  5.1 Internal Structure ......................................... 24
  5.2 Scenario Builder .......................................... 24
    5.2.1 ClassPalette ......................................... 25
    5.2.2 WorkMap, Grip, and Icon ............................ 26
    5.2.3 List of Editables ..................................... 26
  5.3 Editable - Simulable Relationship ....................... 26
  5.4 Generic Entities ........................................... 27
    5.4.1 Parameters ........................................... 28
    5.4.2 Specification of Behaviour ............................ 29
    5.4.3 Templates ............................................ 31
    5.4.4 Components ........................................... 31
  5.5 Simulator .................................................. 31
    5.5.1 Event Queue ........................................... 32
    5.5.2 Emergent Event Computation ......................... 33
    5.5.3 Main Loop ............................................ 35
    5.5.4 Simulator User Interface ........................... 35
  5.6 Utilities .................................................. 36
    5.6.1 I/O Operation ........................................ 36
    5.6.2 User Interface Objects ............................... 36
6 VISP Functionality and Usage

6.1 Introduction ........................................ 38
  6.1.1 System Requirements and Functionality .......... 38
  6.1.2 Basic principles .................................. 39
  6.1.3 Where to Find More ............................... 40

6.2 Terms and Definitions ............................... 40

6.3 VISP Controls ........................................ 41
  6.3.1 Running the VISP ................................ 41
  6.3.2 WorkPlace ........................................ 42
  6.3.3 Button Bar ........................................ 43
  6.3.4 Links to Resources ................................ 46

6.4 User Interface of Entities ........................... 46
  6.4.1 Parameters ........................................ 47
  6.4.2 Buttons of Parameter Window ..................... 47
  6.4.3 Code Window ..................................... 48
  6.4.4 Export/Import Window ............................ 48
  6.4.5 Component Settings Window ....................... 49

6.5 Specification of Behaviour .......................... 50
  6.5.1 Entities versus Containers ....................... 51
  6.5.2 Creating a New Class ............................. 51
  6.5.3 Indicators and Documentation .................... 51
  6.5.4 Code Specification ............................... 52
  6.5.5 Events ............................................ 54
  6.5.6 Useful Methods ................................. 55

6.6 Examples and Experimentation ....................... 56
  6.6.1 Event types ...................................... 56
  6.6.2 Internal Transition ............................... 57
  6.6.3 External Transition .............................. 57
  6.6.4 Combined Internal and External Transition .... 57
  6.6.5 Emergent Events ................................ 58
  6.6.6 Keeping a List ................................... 59

7 Summary .............................................. 60
  7.1 Evaluation of the Project ........................... 60
  7.2 Future Research .................................... 60

A Approximation and Collision Detection ................. 65
  A.1 Definitions ......................................... 66
  A.2 Related Work ....................................... 66
    A.2.1 Static Approach ................................ 67
    A.2.2 Dynamic Approach ............................... 67
  A.3 Object Movement Description ....................... 67
  A.4 Incidence Detection Algorithms ..................... 67
    A.4.1 Fixed Step Search ................................ 69
    A.4.2 Algorithm Description ........................... 69
    A.4.3 Prove of Correctness ............................ 70
    A.4.4 Complexity Analysis ............................. 70
    A.4.5 Pros and Cons .................................. 70
    A.4.6 Binary Search ................................... 70
    A.4.7 Algorithm Description ........................... 71
    A.4.8 Prove of Correctness ............................ 72
    A.4.9 Complexity Analysis ............................. 72
    A.4.10 Pros and Cons .................................. 72
    A.4.11 Approximation .................................. 72
A.4.12 Algorithm Description ........................................ 73
A.4.13 Prove of Correctness ......................................... 74
A.4.14 Complexity Analysis ......................................... 74
A.4.15 Pros and Cons .................................................. 74
A.5 Approximation Functions ........................................ 74
A.5.1 Range Approximated by a Circle (Sphere) ............... 75
A.5.2 Range Approximated by a Square Parallel to a Coordinate System ........................................... 75
A.5.3 Range Approximated by a Half-Space ..................... 76
A.5.4 Conclusion ...................................................... 76
A.6 Classification of Objects ........................................ 76
A.6.1 Constant Acceleration ........................................ 77
A.6.2 Limited Acceleration ......................................... 78
A.6.3 Trajectory Defined by Points ............................... 79
A.6.4 Movement Defined by an Analytic Function ............ 79
A.7 Performance Issues ............................................. 79
A.8 Implementation Issues ......................................... 80
A.8.1 Definitions .................................................... 80
A.8.2 Time-table .................................................... 80
A.8.3 Interest Matrix ............................................... 82
A.8.4 Object's Methods ............................................ 82
A.8.5 Simulation .................................................... 82
A.8.6 Certain Aspects of this Approach ......................... 83
A.9 Application Fields .............................................. 83
A.10 Conclusion ..................................................... 84

B Server - Directory Structure & Scripts ........................ 85
B.1 URL conversions and Directories .............................. 85
B.2 cgi-bin/check.c ................................................ 87
B.3 cgi-bin/image.cgi ............................................. 89
B.4 cgi-bin/javac.cgi ............................................ 90
B.5 cgi-bin/saveFile.c ........................................... 91
B.6 cgi-bin/saveSecure.cgi ...................................... 93
B.7 sim/doc.cgi .................................................... 94
B.8 sim/inf.cgi ..................................................... 95
B.9 sim/lib.cgi ..................................................... 97
B.10 sim/setpwd .................................................... 100

C Java Packages and Classes Hierarchy ......................... 101
C.1 Package Structure ............................................. 101
C.2 Interfaces ...................................................... 104
C.2.1 basser/sim/comp/EditComp.java ......................... 104
C.2.2 basser/sim/comp/Generable.java ......................... 106
C.2.3 basser/sim/comp/SimComp.java .......................... 107
C.2.4 basser/sim/gui/Editable.java ............................ 108
C.2.5 basser/sim/lang/Moving.java ............................. 110
C.2.6 basser/sim/lang/Simulable.java ......................... 112
C.2.7 basser/util/net/Serializable.java ....................... 113

D Earlier Publications ............................................. 114
1.1 Background

Our research extends the earlier work in the field of modeling and simulation [SBR\(^+\), SB95, Sev96a], and especially [Sev96b]. The effort was directed towards developing a new methodology to model moving objects using discrete event/time approach [SB95]. The authors proposed the use of a discrete event simulator for scenarios involving moving objects. Interaction of moving bodies can then be described and processed based on emergent events (events not scheduled by any component, but caused by their interaction).

The other part of their research was the specification of generic components for naval mine warfare modelling and simulation [Sev96a] and developing a visual and interactive environment for mine warfare modelling and simulation - VISPMW - [Sev96b]. The simulation environment was implemented as an extension to Microsoft Visual Basic engine. It supports graphical debugging, editing, and visual preparation of simulation scenarios. In addition, the simulation environment can interface to almost any existing software package which provides the user with good means for report generation, data acquisition, and output analysis.

1.2 Challenges

To keep abreast with latest technology we have challenged ourselves to develop a teaching environment for simulation and modelling, which would be

- platform independent,
- interactive & user-driven,
- distributed and accessible on-line,
- with a user friendly, modelling oriented interface.

The platform independence is a programming technique where created software does not depend on one computer architecture or one operating system only. Multi-platform software lets a user to choose platform of operation. Another feature of such systems is their user interface which usually has some platform-specific appearance and helps users with understanding the system faster.

A system is considered interactive and user-driven if it reacts to user's actions and shows its results in a way understandable for a user. The reasonably short react times are one of the criteria for such systems. This poses another challenge: fast collision detection algorithm.

Distribution and on-line accessibility of an application means that the application possibly employs more than one computer to run. This is crucial for cooperation of users, information exchange, and sharing of knowledge. With regard to education, it can be ground for the management and distribution of lessons to students computers.

Importance and function of user friendly interfaces are self explanatory. The main purpose of the software package under study is modelling and simulation which is what should be reflected in the design of its user interface.

1.3 Structure of this report

This report describes an implementation of VISP (Visual and Interactive Simulation Package) using Java and related technologies. The information was structured as follows; section 2 lists several technical problems and their proposed solutions, section 4 describes the structure of the server side of the application. VISP itself is covered in section 3. Technical data relevant to this part are in appendix B.
Sections 5 and 6 reveal design and usage of the part of the system supposed to run on the client side. The implementation is based on theoretical analysis included in appendix A and some technical details can be found in appendix C. An evaluation of the project can be found in section 7. Each section usually describes its internal dividing at its start.
2 Problems and Solutions

The previous section mentions several problems and challenges we have approached while working on the project. Some of them simply narrowed down the choice of implementation tools, or of employed technology. This section lists the examinations and the solutions of more demanding tasks. At the start of the project, great effort was devoted to developing a fast and correct collision detection algorithm. The development of the VISP software package started after this initial stage. There, we concentrated on implementation of a system for teaching of simulation and modelling.

2.1 Collision Detection

Authors in [SB95] explain how emergent events can be employed in simulation. Their introduction simplifies the specification of entities' behaviour. It can be prescribed fully in terms of discrete events although locomotion is inherently a continuous process. The above mentioned article leaves open the question of computing of emergent events.

The demand for interactive simulation of moving entities defined a need for reasonably fast computation of these events. Times on which these events should occur cannot be scheduled by any entity involved in the simulation. There has to be a special unit for detection of such events. The challenge was to develop a collision detection unit efficient enough for interactive simulations.

There is a great deal of literature on collision detection (see A.2 for a survey). The optimal solutions vary with proposed field of application. From the definition of emergent events [SB94] it is clear that an entity can be approximated by a point in space (mass centre) and an effective area in which the entity can "see" and interact with other entities. Even such simplifications does not allow to compute directly emergent event times for arbitrary locomotion specification (see A.3). There are three kinds of well known solutions:

- Sampling of positions of entities in discrete time points. Rather expensive and inaccurate.
- Inaccurate methods based on search in 4D space.
- Accurate computation of collision of object. Entity locomotion has to be of a limited complexity.

The last group of methods lead us to the following solution: each entity approximates its movement with defined accuracy and this information is used for collision detection. The detected emergent events approximate the "ideal" ones with at least specified accuracy (see A.4.11). Appendix A elaborates on tasks of our solution in greater detail.

The tasks were: to find a set of functions appropriate for locomotion approximation, to specify methodology for approximation of arbitrary movement, and develop an algorithm which ensures correct simulation. The results of our work on these tasks are described in sections A.5 (approximation issues), A.6 (the methodology), and A.8 (implementation related issues). The performed tests showed our approach is the most efficient under the given conditions (see A.7).

2.2 Teaching Environment

The implementation of an environment suitable for teaching is a complex problem. One of the challenges is to develop a framework for co-operation of tutors and
students. Other one might be to provide users with powerful but simple to learn user interface to the system. The relevant tasks are:

- distribution and management of software
- management of shared data
- management of support information
- graphical user interface
- multi-platform execution

The management of distribution is crucial for providing students with working material. The software they may need during each lesson should be available on their computers as well as data and support information. On the other hand, the system should allow students to evaluate their work, keep record of their actions and activities.

The main objective of working with the system should be learning more about simulation and modelling. The need for platform independent system is somehow related to this. A multi-platform system usually keeps some of the special features of the host system, which improves and speeds up assumption(s) of the system. It also widens the number of potential users.

Current technology offers several solutions to the tasks stated above. Sharing of resources is much more common than it used to be a few years ago. The decision of which formats and standards to use was based on several criteria.

The need for management of distribution lead us to employing a client-server architecture. It allows to store data and software centrally, which eases the management and maintenance. Users working with client part of the system (see 3). To avoid the need for complicated installation of a client part, the client was implemented as an applet (see 5) embedded in the information pages. This ensured the multi-platform capability.

Great effort was dedicated for designing and implementation of the user interface of the application running on the client side. As it is described in section 5.2 and later in the section 6, although the system exhibits fairly complex functionality, we believe the interface is easy to understand, and synoptic.
3 VISP Software Package

This part of the report describes the design and implementation of software package named VISP. The purpose of this package was to create an environment suitable for teaching of modelling and simulation as specified by the project input. Various requirements led us to the following decisions in the design process.

The application has client-server architecture based on WWW (World Wide Web) technology (see Figure 1).

![Client-Server Architecture of the VISP package](image)

Figure 1: Client-Server Architecture of the VISP package

The server stores all information relevant to the project. They are information pages, a tutorial, documentation pages, a database of entities, a database of simulations, and a database of icons. The structure of directories, design strategies, and organisation are described in section 4. Part of the server space is occupied by the source files, documentation, and compiled classes of the VISP applet. It is a program for exploring simulation and modelling of moving objects. Its detailed description can be found in section 5.

The main purpose of a client is to download the information from the server and interpret it. The information pages are usually shown as a formatted text with pictures and typographical marks. The client also serves as a host for the VISP applet application.

Some existing software and protocols were employed to achieve high stability and compatibility. The server part of VISP is based on the WWW server of the Basser Department of Computer Science \(^1\). The current configuration is

- Sun 200MHz dual processor UltraSparc server
- SunOS 5.5.1.
- Apache\(^2\) Server 1.1.3 (HTTP server)

To avoid confusion of over-used term server, the data dedicated to VISP will be called site in the rest of this document.

Part of the site is created dynamically, using Unix shell scripts, which interface with the HTTP server via standard CGI - Common Gateway Interface. They include some Unix utilities as awk, cat, echo, basename, dirname, tr, tail and

---

\(^1\)http://www.cs.usyd.edu.au/

\(^2\)http://www.apache.org/
a binary executable compiled from source written in C. Detailed description can be found in Appendix B.

If there is a need to move the server to other computer it is recommended to use a similar configuration, though it may not be impossible to run it on a completely different architecture. A special care was taken to use only well standardised and established interfaces. From this point of view we do not see any problems in moving the server to any computer running Unix or similar operation system (Linux, Irix, SCO Unix etc.).

The communication between the server and clients is based on the HTTP - Hyper Text Transfer Protocol [Fly95] only. This allows to use any WWW browser as a client. The function of a client is twofold;

- download and show data pages from the server,
- download and host the VISP applet.

The user can either use a single browser acting as a browser and a host, or two separate, more specialised programs. Examples of a client capable of both function are Netscape Navigator, Version 3.01 (used for testing) or Microsoft Explorer. One of the reasons why not to use these browsers as a host for the applet is their tough security policy. In this case Applet Viewer by Sun can be used to host the application to mention just one. All this software is available for all common architectures.

The other advantage is the browsers can be used free for non-commercial purposes.

While installation and using a browser as a client does not pose any special requirements, for the full functionality of Applet Viewer is necessary to down-load some classes. This procedure is described in on-line manuals of the project and later in the Section 6.
4 VISP Site

This section describes the design and structure of the information related to the VISP project stored on the server. It explains what is the users' visualisation of the site, how it corresponds with its implementation, and design. The directory structure, scripts and source codes can be found in Appendix B.

4.1 Users' Visualisation

The information delivered to the user is structured to several pages. To improve the orientation and understanding all the information pages share some common characteristics. The background colour and typesetting characteristics are the same throughout the site. Each page is divided into three major parts:

- a title,
- a body,
- an icon bar with shortcuts.

While the first two parts deliver information, the icon bar serves for better orientation and understanding. All pages are virtually divided into five groups. The icon bar offers shortcuts to the main pages of these sections. Their names are Main, VISP, Tutorial, Documentation, and Feedback.

![Figure 2: Main page](image)

4.1.1 Main

This section consists only of a single page informing about the main project goals, the structure of the site, and legal issues (see figure 2). Its location is http://www.cs.usyd.edu.au/~hejda/sim/doc/cgi/index.
4.1.2 VISP

This is the actual simulation and modelling environment. The main page of this part contains an applet-based application allowing for a user to build simulation from prepared and/or user-defined components. Resources necessary for simulation are referenced from this page as well. There are links to the database of classes of entities, database of simulations, and database of icons.

The user is instructed to keep the central page open while working on simulation, and to use other windows for support information. An alternative way is to use an Applet Viewer for running the application and the browser itself just for screening the support information.

4.1.3 Tutorial

A quick introduction for beginners to the system. It contains working examples and explains basic principles. The topics cover work with the VISP, security limitations, creation of simple simulations, specifying of behaviour of generic objects and input/output operations. The tutorial contains useful links which lead to simulation examples.

4.1.4 Documentation

This rather extensive part of the site contains description of classes used in the VISP, instructions how to write special classes, and links to Java tutorials.

4.1.5 Feedback

This section consists of a single page which contains contact addresses to the designers of VISP site.

4.2 Used Techniques

A number of techniques have been employed to keep the look of the pages consistent and synoptic. A consistent layout of all pages is ensured by "packing" the information into a standard header and a footer. Dynamic parts were created by scripts interpreting directory structures as databases. Since users (clients) are allowed to save data on the server, simple security policies were implemented. All scripts have been implemented as Unix shell scripts (for the /bin/sh standard shell). Next sections explain the details.

4.3 Common Layout

The common layout of all pages on the server should help users with orientation and understanding of presented information. Its implementation relays on two files:

- doc.cgi,

- footer.

While doc.cgi is a shell script based on the CGI, footer contains an HTML-formatted bottom part of pages. It specifies a table containing links to the five site's main sections (see the section above). The code of the doc.cgi follows:

```
1 : #!/bin/sh
2 :
3 : PATH=/usr/sbin:/usr/bin
4 :
```
The script uses the information in PATH_INDEX variable as a description of relative location of a data file. First, it prints a HTML header setting colours of the background and hypertext links (lines 13-24). Then, if the data file is found, it is included into the document. In the case of a failure, appropriate message is printed (lines 26-32). At the end, the footer is printed.

This script is called with each demand of URL, with the form:


The DATA.FILE part is stored in the PATH_INDEX variable, during the execution of the script.

4.4 Dynamic Parts

The server contains several parts, which can change from time to time. User actions can alter the structure and content of database of pictures, simulation entities, and scenarios. This section describes their implementation and organisation.

The dynamic implementation ensures that all changes (newly saved data) can be observed immediately simply by re-loading a database page from the server. This can, among others, acknowledge that data arrived at the server.
4.4.1 Database of Pictures

The simplest one of the databases is a storage of pictures, which can be used as icons of simulation entities. The database consists of a directory full of pictures in a "GIF" format and a script `image.cgi` interpreting the names of files in the directory as data entries.

The script has the following structure:

```
1: #!/bin/sh
2: 
3: PATH=/usr/sbin/usr/bin
4: 
5: BASENAME="../$PATH_INFO"
7: NAME=$PATH_INFO
8: 
9: cat << EOM
10: Content-type: text/html
11: 
12: <HTML><HEAD><TITLE>Images in $SERVER$PATH_INFO</TITLE></HEAD>
13: <BODY BGCOLOR="#ffffff"
14:   TEXT= "#000000"
15:   LINK= "#a00000"
16:   VLINK= "#000000"
17:   ALINK= "#ff0000"
18: EOM
19: 
20: echo "<H1>Directory: $SERVER$PATH_INFO"
21: echo "</H1><HR>"
22: 
23: echo "<P>To use an image copy its location to dialog window." 
24: echo "You can also use a <A HREF="/hejda/cgi-bin/
          imageBrief.cgi$PATH_INFO"> compressed list</A>".
25: echo ""
26: 
27: ALLNAMES='ls -1 $BASENAME*.gif'
28: 
29: echo "<PRE> "
30: 
31: for EACH in $ALLNAMES
32: do 
33:   echo "<A HREF=$SERVER$PATH_INFO$EACH><IMG HEIGHT=40 WIDTH=40
          SRC=$SERVER$PATH_INFO$EACH></A> $SERVER$PATH_INFO$EACH"
34: }
35: done
36: 
37: echo "</PRE>"
38: 
39: cat << EOM
40: 
41: <HR>
42: <P align=center>
43: Go to the
45: <A HREF="/hejda/sim/">VISP</A>, or to the
```
The script specifies the layout of the page and prints a short hint on how an icon can be used (lines 1-25). Lines 27 to 35 read the content of a directory specified in PATH.INFO variable and interpret this information as a table. In the left-hand side of the table the images are shown (with size scaled to 40x40 pixels regardless of the actual size), in the right-hand side there is a URL of each image.

4.4.2 Database of Simulation Entities

The structure of this database is fairly complex. It stores structured information on simulation entities. From the technical point of view these entities are Java classes and their database follows the structure of package/class database as defined by the JDK. The $HOME/java/java/lib/ is the root directory of classes. Classes of a aaa.bbb.ccc package are stored in directory aaa/bbb/ccc/. Information on a class is divided into several files; if a class is named colour, the names and content of related files is as follows:

colour.author  - authors name, URL, e-mail
colour.class    - Java "executable" code of the class
colour.className - name of the user interface class
colour.doc      - detailed description
colour.handler  - code of methods specific for simulation
colour.info     - short description
colour.java     - java code of the class
colour.params   - structure and initial values of parameters
colour.pict     - URL of class’ icon

From the users’ point of view, a tree hierarchy can be observed, which consists of the three following types of pages:

**Type**    **Description**

A  List of packages, with a short description and hints on using the classes. There is only one page like this.

B  List of classes in a selected package; it starts with the name of the package and its security status (whether or not it is protected by a password). An icon, a name, and a short description of each class follows. These pages are linked to the type A page.

C  Document page of a class consisting of name, icon, short description, author, date of the last modification, indication of completeness, long description of functionality, and a part of the code specifying the behaviour of the entity. These pages are accessible through type B pages.

All pages exhibit the layout specified earlier. Specialised dialog windows allow users to add and alter information in this database (see section 6.4.4). The database has been implemented as a directory and file structure. Unix shell script lib.cgi interprets this structure as HTML formatted pages. To select packages relevant to the problem only, the first level (type A) page is static. It is shown when the PATH.INFO is empty (see lines 24-31 of the lib.cgi script).
If the PATH_INFO variable is not empty, the content is interpreted as a name of a package or a class and a page type B or C is constructed. Variables NAME, FILE, and SHORT are set to name of the package or class (PATH_INFO without the first '/'), file name (all '/' are substituted with '/'), and the last part of the name, respectively. See lines 9 to 11.

```sh
#!/bin/sh
#

PATH=/usr/sbin:/usr/bin

./log.cgi

NAME=`echo $PATH_INFO | tail -2`
FILE=../java/echo $PATH_INFO | tr "." "/
SHORT=`basename $FILE`

cat << EOM

Content-type: text/html

়<HTML><HEAD><TITLE>Database of classes...$NAME</TITLE></HEAD>

<BODY BGCOLOR="#ffffff"

TEXT= "#000000"

LINK= "#a00000"

VLINK= "#000000"

ALINK= "#ff0000">

EOM

if [ -z "$NAME" ]
then {
cat doc
cat footer
exit 0
}
fi

# The object is a class
if [ -r "$FILE.author" ]
then {
    echo "<IMG SRC="
    cat "$FILE.pict"
    echo " ALIGN=right HEIGHT=40 WIDTH=40">
    echo "<A href="#TOP"><H1>Class name: $NAME</H1></A><UL>
    cat "$FILE.info
    echo "</UL><HR>
    echo "<TABLE><TR>
    echo "<TD><P><STRONG>Author: </STRONG><TD>
    cat "$FILE.author
    echo "<TR><TD><P><STRONG>Last modified: </STRONG><TD>

"
ls -l $FILE.java | awk '{print $6 "\" $7 "\", $8 : ;}'
echo "<TR><TD><P><STRONG>Ready to use:</STRONG><TD>"
if [ -r $FILE.class ]
then {
  echo Yes
}
else {
  echo No
}
fi

echo "</TABLE>"
echo "<P><STRONG>Description: </STRONG>"
cat $FILE.doc
echo "<P><STRONG>Code: </STRONG>"
echo "</PRE>"
cat $FILE.handler
echo "</PRE>"
}
#
elif [ -d $FILE ]
then {
if ./check $FILE > /dev/null
then {
  echo "<IMG SRC="/hejda/images/unlock.gif ALIGN=right>"
}
else {
  echo "<IMG SRC="/hejda/images/lock.gif ALIGN=right>"
}
fi

echo "<H1>Package: $NAME </H1><HR>"

ALL NAMES='ls -l $FILE |
  awk -F'\' '{if ($2 == "doc") print $1} '

echo "<TABLE>"
for EACH in $ALL NAMES
do {
  echo "<TR>"
  echo "<TD><IMG HEIGHT=40 WIDTH=40 SRC="
cat $FILE/$EACH.pict
echo "">"
echo "</TD>"
echo "<A HREF="
echo $NAME.$EACH
echo "">"
echo $NAME.$EACH
echo "</A>"
echo "</TD>"
cat $FILE/$EACH.info
}
Both type b and type C pages are created by the script. The condition on the line 34 recognises, whether there was required a page about a package (the packages does not have any *author file) or about the class. In the case of a class (type C), the lines 36-66 arrange the data from files related to the class (see above).

In the case of a package, lines 73-103 are executed. The condition on the line 73 checks the security status of a package (see 4.5), and shows icons of a closed lock, and an open lock, respectively. Line 84 reads all class names stored in the package. The rest of this part creates a table, which contains icons, names, links, and short description of the classes.

If the request (PATH_INFO) specifies non-existing package, or class, the lines 108 to 116 print an error message.

As it will be described later, the users can add their own classes into packages. The dynamic creation of the pages ensures, that the re-loaded page will be automatically updated to include the new information. A new package can be added too, but in this case this procedure has to be followed:

- Add a directory
- Set the password for the directory (see section 4.5)
- Edit the $HOME/lib/html/sim/doc file to include the new package

To keep the script simple and robust, we have narrowed down the structure of packages and classes in this case. Unlike in Java, the tree has been limited to two levels only; the root directory contains package directories, but not any classes. The package directories contain classes but not packages.

### 4.4.3 Database of Scenarios

This database is intended to store scenarios and entities built by users (see 6.3.3). This section discusses internal structure and user interface of this database. Other sections cover the security issues (see 4.5) and storing of data (see saveFile).

The implementation is based on the Unix file system. A directory tree contains sub-directories, data files and description files. Unlike in database of entities, the sub-directories can be nested. There is a description file for each (sub)directory and
each data file. The name of these files is created appending ".inf" to the name of sub-directory or file. This information is used for orientation.

Unix script inf.cgi interprets the directory structure and allows users to browse through the database.

```bash
#!/bin/sh
PATH=/usr/sbin:/usr/bin
./log.cgi
BASE=./$PATH_INFO

cat << EOM
Content-type: text/html

<HTML><HEAD><TITLE>List of directory $PATH_INFO</TITLE></HEAD>
<BODY BGCOLOR="#ffffff"
TEXT="#000000"
LINK="#a00000"
VLINK="#0000ff"
ALINK="#ff0000">
EOM

if [ ! -d "$BASE" ]
then

cat << EOM
<HEAD>ERROR: Directory
<UL><B>$PATH_INFO</B></UL> not found on this server</HEAD>
<body>Explanation: The specified directory does not exist on this server. Check for the name and the spelling, or start from the root of database</body>
EOM

cat footer
exit 0

fi

if ./$BASE > /dev/null
then

echo "<IMG SRC="/hejda/images/unlock.gif ALIGN=right>"

else

echo "<IMG SRC="/hejda/images/lock.gif ALIGN=right>"

fi

EOM

<HR>
```
50 :
51 : ALLNAMES='ls -l $BASE'
52 :
53 : echo "<TABLE>"
54 :
55 : for EACH in $ALLNAMES
56 : do {
57 :   if [ -f "$BASE/$EACH.inf" ]
58 :     then {
59 :       echo "<TR><TD>"
60 :       if [ -d "$BASE/$EACH" ]
61 :         then {
62 :           echo "<A HREF="/hejda/sim//inf.cgi$PATH_INFO/$EACH>"
63 :           echo $EACH
64 :           echo "</A>"
65 :         }
66 :       else {
67 :         echo $EACH
68 :       }
69 :     }
70 :   fi
71 :   echo "<TD>"
72 :   cat $BASE/$EACH.inf
73 :   fi
74 : }
75 : done
76 :
77 : echo "</TABLE>"
78 :
79 : echo "<CENTER>"
80 : echo "Back to the

   <A HREF="/hejda/sim//inf.cgi/sim/data>root.</A>"
81 :
82 : cat footer

The content of variable PATH_INFO is used as a directory specification (line 7). If there is not such a directory, an error message is generated - lines 20 to 31. Otherwise, a page is created, which contains a security status of the directory (lines 35-42) and a table with names of sub-directories and files, and contents of the information files. The names of sub-directories are presented as hyper-links. At the end, the standard footer is included.

4.5 Security

There is a simple security system implemented to protect the parts of the server, which can be changed by users - the database of simulation entities (see 4.4.2) and the database of scenarios(see 4.4.3).

Each directory which is dedicated for storing user data have to contain a password file ".p". The permissions of these files have to be set so as to avoid reading and changing the content by anyone but administrator (600, for example). The administrator can use utility $HOME/lib/html/sim/setpwd to change passwords and set the permissions correctly. The usage is:

   ./setpwd directory password - sets the password
   ./setpwd directory - sets the password to be void
A utility called `check` is used to compare its content with claimed password. Since all scripts called by the HTML server run with UID set to nobody, the check utility has to have 'set user ID on execution' to be able to read the password files. To avoid possible break into the system, this utility is not a shell script, but executable binary. Its code in the C language can be found in the Appendix (see B.2).

The utility can have up to two parameters. The first one is compulsory, it specifies the directory. The second one is optional, and can contain the claimed password. The utility tries to read the `.p` file from specified directory. If there is no such file, an error message is displayed and the return value is -2. The directory is not available to the user.

If the `.p` file is found, and is empty, the directory is supposed to be unprotected, and the utility ends with exit status 0. Otherwise, the content is compared against the claimed password. If these two words fit, the password is validated and exit status is set to 0. Otherwise, an error message is displayed and exit status is set to -3.

The check utility has been used in scripts for storing data on server (see 4.6) and database scripts (see 4.4.2 and 4.4.3).

### 4.6 Storing Data on Server

The requirements of collaboration and co-operation amongst users of the VISIP pose claim of saving data on the server. This section describes a part of the server, which receives and stores data into a file. The client part (sending the data) is described in Section 5.6.1. The implementation is based on two utilities: `saveFile`, and `saveSecure.cgi`.

The `saveFile` is a simple binary executable, which saves data from its `stdin` under the name specified by the first parameter. The second parameter can optionally specify the length of the data. The success of the operation is indicated by a message. The possibilities are:

- OK, data saved.
- ERROR on server, data not saved.

See B.5 for details. The `saveSecure.cgi` script is an environment for the `saveFile` utility.

```bash
1 : #!/bin/sh
   ...
7 : PATH=/usr/sbin:/usr/bin
8 :
9 : ./log.cgi
10 :
11 : echo Content-type: text/plain
12 : echo
13 :
14 : FILE_NAME=$PATH_INFO
15 : DIR='basename $FILE_NAME'
16 : FILE='basename $FILE_NAME'
17 :
18 : if [ $FILE = ".p" ]
19 : then {
20 :   echo ERROR, cannot rewrite the specified file
```
21:   exit
22: }
23: fi
24: :
25: if RES='.check $DIR/ $QUERY_STRING'
26: then {
27: ./saveFile $FILE_NAME $CONTENT_LENGTH <&0
28: }
29: else {
30:   echo $RES
31: }
32: fi

The script first analyses the PATH_INFO (lines 14-16). The name of the file has
to differ from "p", which are dedicated to store passwords. The script checks the
claimed password sent as QUERY_STRING (line 25). If it agrees with the password
of the directory the saveFile utility is called to save the data.

The string sent back to the client contains result of the operation. The convention
is, that anything starting with "ERROR" is considered as a failure. The rest
of the string specify details.

4.7 Log Files

Log files were implemented to monitor the activity of the VISP site. With each
script execution date, time, script name, page name, name of the client, and clients
IP are appended to the log file. The fields are separated by semicolons for further
processing.

4.8 Re-compilation - javac.cgi

Java was chosen as a language for specification of entity behaviour. The user can
write and edit the code of an entity on his/her client, and store it on the server (in
the entity database) and compile it. The compiler runs on the server, on clients
demand. The javac.cgi script is a CGI interface of a standard javac compiler
delivered with the Java Development Kit (see appendix B.4).
5 VISP Applet

This section describes part of the VISP package which runs on the client side. It is an applet based application implemented in programming language Java, under the Java\textsuperscript{TM} Developers Kit Version 1.0.2 (JDK). This language was chosen for its advantages:

- portability and platform-independence
- a de-facto standard for Web applications
- modern and well defined syntax
- advanced features and a large object library
- free availability and good support from manufacturer

Java is an object oriented programming language developed by Sun Microsystems. Its syntax is very similar to C++, but it is clean from problems introduced into the C++ language by evolution; different brands, compiler compatibility and so on. Some advanced features, for example system of exceptions, or multi-threaded applications, make programming easier and more systematic.

It was probably the platform independence, which granted Java such a credit. This is achieved by combination of compilation and interpretation. The source code in Java is compiled to so-called class code (stored in files with \texttt{.class} suffix), which is platform independent. This code is then interpreted on a simple, platform dependent Java Virtual Machine. This solution compromises portability and speed of the application. Additional improvement is Just-In-Time (JIT) compiler, which speeds up the execution of a class by translation to the native machine code. Further information on Java can be found for example in [CMG+96].

Today internet browsers contain a Java class code interpreter. A Java-based application can be part of a page. The interface between the browser and the application embedded in a page is defined by an \texttt{applet - java.applet.Applet} class. VISP applet is an extension of this class. The client downloads the classes (their class codes) and runs the application as a host (and on Java's Virtual Machine).

This section elaborates the internal structure of the application, explains used principles and relationships among classes. Further information can be found in the Appendix C and in the extensive on-line documentation of the VISP project (http://www.cs.usyd.edu.au/~hejda/sim/doc/cgi/docs/packages).

5.0.1 Conventions

There are some conventions used in this section. Unless specified otherwise, they are:

- Names of the packages and classes are printed slanted
- Names of methods, parts of the Java code are printed with a non-proportional typeface
- Names of classes are capitalised (Applet)
- Names of packages do not contain any capital letters
- Full class names are written as package.name.ClassName

\footnote{Visit http://java.sun.com/products/JDK/}
5.1 Internal Structure

Due to the complex task the VISP applet was designed for, its internal structure is quite complicated. Figure 3 illustrates the following description.

![Diagram of VISP applet structure]

Figure 3: Internal Structure of the VISP applet

The VISP applet exercises two main tasks. On one hand it is an environment for composing simulation scenarios, and specifying the behaviour of entities. This part is called **Scenario Builder**. Classes implementing it are grouped in the `basser.sim.gui` package. The central class of this part is the `Environment`. It contains objects implementing a palette of entities (`ClassPalette`), list of entities used in scenario (`EditableList`), a map for editing entity positions in 2D space (`WorkMap`), and others. Detailed description is in the 5.2.

On the other hand the VISP applet allows to simulate created scenarios. Package `basser.sim.lang` contains some of the employed classes. Objects of class `Simulation` run on a dedicated thread to support independence and parallel execution. Each `Simulation` object contains its own list of entities (`SimulableList`), queue of events (`SimEventQueue`) and emergent event detection unit (`TimeTable`). This allows to run several versions of one scenario in parallel. More can be found in section 5.5.

5.2 Scenario Builder

As it was mentioned earlier, the scenario builder is a part of the VISP applet dedicated to build the scenario, to specify behaviour of entities, to exchange data with the server, and to launch new simulations. Since the VISP was intended as an educational project, a sizeable effort was spent on implementing a graphical user interface. Classes implementing scenario builder are aggregated in package
basser.sim.gui. As a rule, if the name of the class' package is not mentioned, the class belongs to basser.sim.gui.

The structure of scenario builder was kept as simple as possible. The central class - Environment - extends the class java.applet.Applet and it implements the basic user interface. The instances of Environment contain references of other involved classes, such as basser.util.palette.ClassPalette, WorkMap, EditableList, NetFrame, ClassFrame, and instances of java.awt.Buttons and java.awt.Panels used in the user interface. Methods of the Environment class implement button call-backs, and a few utilities.

The following sections describe the co-operation and internal structure of parts of the scenario builder, namely the palette of classes, 2D editor, and a list of entities.

5.2.1 ClassPalette

The basser.util.palette package contains classes implementing a palette of classes. This palette is employed to contain downloaded wrapper classes for entities involved in simulation. The main function is to download a class from a server, check its correctness and completeness. Other function of palette is to re-load a previously loaded class. This function depends on the environment hosting VISP. Discussion in section 6.1.1 explains more about re-loading of classes. No longer desired classes can be removed from the palette. Picture 4 shows the internal structure.

![Diagram of ClassPalette - internal structure](image)

Figure 4: ClassPalette - internal structure

The basser.util.palette.ClassPalette includes menu “Item”, with “Add”, “Remove”, and “Reload” options. If the “Add” option is chosen then a new instance of basser.util.paramedit.StringDialog class is created and shown. If the user fills in a class name, this string is included in the event, and posted to the ClassPalette with ID set to “DATA”. This causes the ClassPalette to create one instance of ClassHolder class and one of the ClassButton class. The ClassHolder object tries to load the required class, and the ClassButton object is added to the user interface
to display the result of download operation. If the class was loaded successfully, its icon is shown, otherwise an error is indicated (see 6.3.3 to learn more about icons).

Each correctly loaded class is represented in the user interface of the ClassPalette as a ClassButton - a button with an icon of the class (see picture 12). The buttons can be in two states: activated or deactivated. Only the last clicked-on button is considered activated. There is a mechanism allowing to use the class presented by activated button by other classes (Environment class object in VISP). Each time a button is activated, an event is sent to the Environment object and the button is displayed pressed. If the Environment object accepts the choice it calls the deactivate() method, as an approval, the class was used.

5.2.2 WorkMap, Grip, and Icon

A part of the scenario builder implements a 2D editor, which allows users to specify a position of simulation entities and their intended paths. The implementation is based on java.awt.Container and java.awt.Component classes. The first one was extended to a map, while the second one was used to implement editable points and icons.

The WorkMap extends basser.util.workspace.Workspace class. This class is a container with a void layout manager. It can contain a list of instances of the class basser.util.workspace.Grip. This class implements rectangular areas which can be moved across the map. These were used for marking the end of a path, initial position etc. The class basser.util.workspace.Icon extends the Grip class further. The icons can be either in active or passive state, and only one icon at a time can be active. This mechanism was employed for choosing an entity, which will be the subject of users' actions.

5.2.3 List of Editables

The scenario builder is designed to keep, manage and work with so called editable entities. Each editable entity represents a user interface of an object later used in simulations. The editable - simulable relationship is discussed in detail in section 5.3. All editable entities are kept in the basser.sim.guiEditableList class instance. This class extends the java.util.Vector class and implements methods for

- adding/removing an item to/from the list
- saving/retrieving the information in the list to/from a stream

5.3 Editable - Simulable Relationship

The representation of an entity in the scenario builder have to be different from that used in simulations. For example, each entity can be destroyed during a simulation. If the internal representation is shared by the simulator and the scenario builder, the entity would disappear not only from simulation, but from scenario builder too. This would not be what users expect.

Term editable entity is defined as the representation of the entity in the scenario builder. Each object implementing an editable entity has to define methods from interface basser.sim.guiEditable. This interface contains methods for communication with scenario builder (see appendix C.2.4).

Term simulable entity is defined as the representation of an entity in a simulator. Each object which implements a simulable entity has to define methods from interface basser.sim.lang.Simulable. This interface lists methods for communication with the simulator.
The relationship of editable and simulable entities is illustrated in figure 5. Method `produce()` from the `Editable` interface implements the transition from an editable entity to its simulable counterpart. The implementation of the method `produce()` can differ. For example, this method can simply create a copy of the editable entity. In other case, rather complicated mechanism compiles a new class, than creates a new instance of the class and initiates it. While complexity may differ, one condition must be satisfied: the produce must not return an object used in the scenario builder. This ensures that simulations are independent from the scenario builder once they are created.

Another aspect of this conception is that there can be concurrent simulations of the same scenario, independent from each other. Several versions of a scenario may be run simultaneously. These issues are elaborated further on in the next section.

![Diagram](image)

**Figure 5: Editable - Simulable relationship**

### 5.4 Generic Entities

This section describes generic editable entities, which were developed to allow users to specify the behaviour of an entity. Classes related to this mechanism can be found mostly in the `basser.sim.comp` package. If not stated otherwise, the classes mentioned in this section belong to this package.

In section 5.3 is elaborated the system of production of Simulable entities. The editable entities represent a wrapper object for the entities involved in simulation. Two simulation entities of different classes can be produced by two editable entities which belong to the same class. Generic editable entities are based on this principle. Figure 6 illustrates the internal structure of a generic editable class.

The central class used for generic editable entities is `GenericSimulable`. Members of this class can produce different classes of objects used in simulations according to the data provided by the users. Users can specify:

- numbers, types, and names of parameters
- behaviour in terms of code
- initial values of the parameters
- description used in the documentation pages
Class GenericSimulable is elaborated later on in this section. It was designed to generate entities without any material body. For entities with some 2D representation there exists the subclass GenericMoving.

![Class diagram](image)

**Figure 6: Generic Editable Class**

### 5.4.1 Parameters

The state of an entity during a simulation is kept in state variables. Variables of an entity can be employed to keep the name of the entity, a message string, whatever. Subclasses of class basser.util.paramedit.Parameter were designed to represent the variables of both editable and simulable entities. The superclass Parameter has five subclasses:

- **IntParam** - keeps integer values
- **DoubleParam** - keeps double values
- **ColorParam** - keeps an object of class java.awt.Color
- **ImageParam** - keeps an object of class java.awt.Image
- **StringParam** - keeps an object of class java.lang.String

Each subclass of Parameter defines methods for:

- setting and reading of the value
- saving the value to a stream, retrieving value from a stream
- conversion of the value to a string

Class ParamEdit from package basser.util.paramedit implements a manager of Parameters. Its user interface allows to add and remove Parameters. Each Parameter has a name and keeps values of certain type according to the subclass used.
The initial values of variables presented by Parameters can be set according to the type.

The StringParam allows users to enter any string. The IntParam and DoubleParam check after each change, whether the resulting string represents a value of integer or double precision number, respectively. If the string, entered by a user, is incorrect, the field displaying the name and the value of a variable turns red. An example can be seen in figure 7 - look at field "delay". Another item shown in this figure is the way a colour parameter is set. ColorParam allows to chose the initial value from a palette of colours. The value of ImageParam is to be initiated by URL description 4. Again, if the string does not describes valid location of picture, the fields turn red.

![Figure 7: User Interface of the ParamEdit.](image)

Class GenericParamedit from package basser.sim.comp extends ParamEdit and implements method produce() which makes an exact copy of all parameters managed by the object. This method has been used in transition from editable to simulable entity.

5.4.2 Specification of Behaviour

Each object of the class GenericSimulable contains an object of the GenericHandler class. It was designed to allow users to edit a part of the Java code of the simulation entity. When an object of this class is shown for the first time, it contains a template of methods, which specify the behaviour of an entity during simulation. Other part of the window summarises some of the often used methods of the simulator, and methods which work with parameters.

The code of a simulation entity is created from several sources. A typical example follows:

```java
1: /* Code generated by the Generic Simulable class */
```

4Uniform Resource Locator - address of a piece of data in cyberspace
```java
2: package def;
3: import java.awt.*;
4: import java.util.*;
5: import basser.util.*;
6: import basser.util.paramedit.*;
7: import basser.sim.lang.*;
8: import basser.sim.comp.*;
9:;
10: public class printer extends GenericSimulableTemplate {
11:    basser.util.paramedit.StringParam name;
12:    basser.util.paramedit.ColorParam color;
13:    basser.util.paramedit.ImageParam image;
14:    
15:    public void genericInit() {
16:        name = (basser.util.paramedit.StringParam)parameters.at(0);
17:        color = (basser.util.paramedit.ColorParam)parameters.at(1);
18:        image = (basser.util.paramedit.ImageParam)parameters.at(2);
19:    }
20:    
21:    public void init(double time) {
22:    }
23:    
24:    public double time_advance(double elapsed_time, double time) {
25:        return tick.value();
26:    }
27:    
28:    public void internal(double elapsed_time, double time) {
29:        sim.send(time, portA.value(), "Hello");
30:    }
31:    
32:    public void external(double elapsed_time, SimEvent event) {
33:        passivate();
34:    }
35:    
36: }
```

Lines 1 to 10 were included by the GenericSimulable class (during execution of method translate()). They specify the name of a class, the name of a package, and imported packages. Lines 12 to 20 are generated by the GenericParamedit according to the actual number, names and types of parameters. This results in the necessity to re-compile the code after a parameter is added, or removed. Lines 22 to 35 are entered by the user. They define the behaviour of the entity when it is introduced into the simulator (the init() method) and when it receives an event (the eventHandler() method). The rest of our example (lines 28-34) are included by the GenericSimulable again.

When the code is assembled, it is sent to server for compilation (see section 4.8). Server makes a new entry in the database of entities, tries to compile the code and returns the results of the compilation. This can be either an error message, which is displayed to the user, or a class data. In the second case, the class data are loaded by basser.util.net.ToptoLoader class object (depending on the mode, in which the VISP is running - see section 6.1.1).

When a new simulable entity is produced, the loaded class data are used to create a new instance of the class. This instance can be then initialised and used in a simulation. The initialisation involves the delivery of parameters (as described in
the `genericInit()` method), the delivery of components and calling of the `init()` method.

5.4.3 Templates

As it can be seen in the example of the code generated by `GenericSimulable` class, the generated class is a subclass of class `GenericSimulableTemplate`. This class implements basic methods for communication with a simulator listed in interface `basser.sim.lang.Simulable`. The function of `GenericSimulableTemplate` is to provide a kind of a hub for the generated classes. The generated class is supposed to overload methods specifying the behaviour of the entity.

5.4.4 Components

The system of producing generic entities turned out to be rather complex, especially after introducing moving objects. The interface of such entities should include not only objects related to code and parameter specifications, but objects describing 2D qualities. Interfaces `EditComp` and `SimComp` helped to sort things out.

This pair of interfaces is an analogy of the `Editable - Simulable` pair. The `EditComp` is supposed to be implemented by objects used as components of editable entities. The `SimComp` should be implemented by components of simulable entities.

More about described techniques, classes, interfaces and their mutual relationships can be found in appendix C, and in the on-line documentation of the VISP project.

5.5 Simulator

VISP consists of a discrete event simulator capable of simulating objects which interact with each other in 2D space. This section describes the internal structure of the simulator, the implementation of the collision detection unit and it explains related problems and terms. Appendix A, and its part A.8 in particular, present profound analysis of the problem, theoretical reasoning, and a description of some practical tests. All classes mentioned in this section belong to `basser.sim.lang` package, if not stated otherwise.

The simulator is rather independent from the scenario builder. It has its own list of editable entities, an event queue, collision detection unit, and a user interface. Figure 8 illustrates the following description. There are two basic types of entities involved in simulations. The first type is defined by interface `Simulable`, the second one by interface `Moving`.

`Simulable` interface is to be used for entities without any 2D representation. A timer is an example which generates events, but its position in 2D space is irrelevant. Interface `Simulable` defines prototypes of methods used in communication with the simulator (see C.2.6 too):

- `init(...)` - initialises the entity, when introduced to simulator
- `eventHandler(...)` - reacts to incoming events
- `destroy(...)` - removes the entity from its simulator
- `name(...)` - returns the name of the entity

Interface `Moving` is an extension of interface `Simulable`. It is to be used for entities, whose 2D representation is relevant to the modelled problem. An example might be a ship, a mine, or a car. Apart from methods mentioned above, methods

\[ ^5 \text{See http://www.cs.uwa.edu.au/~hejda/sim/doc/cgi/docs/packages} \]
for communication with the collision detection unit are defined. As it was stated in [SB95], if one collides with other, an emergent event is generated. Emergent events are sort of quantification of continuously changing system properties. It is an effective way of how to use discrete event simulation for hybrid scenarios.

Emergent events are not scheduled by any entity. A dedicated collision detection unit senses and schedules such events. This unit can be designed in many ways. The demand on interactivity and effectiveness leads us to a rather extensive study, which is described in appendix A. It elaborates a technique, which compromises precision of computation and speed. The basic idea is to let the entities to approximate their intended route with a partially linear approximation and use this information to predict the times of potential emergent events.

![Diagram](image)

**Figure 8: Internal Structure of the Simulator**

The function of the simulator is as follows. It receives a list of entities from scenario builder. Each entity in this list is initialised (the `init()` method). This may result in scheduling a number of events in the event queue. Entities implementing interface Moving register their intended routes with the `Timetable`. This results in computation of potential emergent event times.

After the initialisation, the simulator takes either the first event in the queue, or constructs an emerging event from the `Timetable`, whichever has the smaller time stamp. Such an event is sent to the addressed entity, the simulator calls its `eventHandler()` method. According to the behaviour specification, this may result in scheduling and queuing of events, or in change of its route. The latter one forces the `Timetable` to re-compute the emergent event times. But all changes can schedule some activity for future time only. This allows the simulator to advance the simulated time. Some details are described in following sections.

### 5.5.1 Event Queue

The event queue (class `SimEventQueue`) was designed as a doubly linked dynamic list of events (class `SimEvent`). This allows to add events “in the middle” of the list.
and keep the events sorted according to the scheduled time. The following table shows complexity of basic operations for a queue of length \( n \).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add event</td>
<td>( O(n/2) )</td>
</tr>
<tr>
<td>Remove event</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Find minimum</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

Table 1: Complexity of queue operations

If an event is to be added, two neighbouring events in the queue are searched for, one with a greater time stamp, the second one with its stamp smaller or equal. The search starts at the end of the list. The event is inserted into to found place. Keeping the events sorted, the event with the minimal time stamp is always in the front of the queue. Removing an event from queue involves re-setting references on its neighbours. Hence both remove and find minimal operations have constant complexity.

5.5.2 Emergent Event Computation

The collision detection unit is the most complex part of the simulator. The reader is advised to read appendix A first. This section gives a description of basic principles and implementation of the collision detection unit.

The implementation of the collision detection unit is based on term range. A range is a representation of spatial qualities of an entity. It consists of two parts; the first part is a description of the solid body of the entity, the second one specifies the active (sensor) area of the entity.

The description of the body specifies:
- the location of the entity (as a point in 2D plane)
- the group number of the entity
- types of sensors; what can detect the entity

The description of the active area specifies:
- coverage of the active area
- types of sensors possessed by the entity

The information recorded by ranges is used to sense and schedule emergent events. An emergent event is generated, when one range enters or leaves an active area of other range. The additional conditions apply: group membership and sensor capability.

Only such pairs of ranges with different group numbers are checked for possible collisions. This can be used to simulate one entity with more than one sensor. The detection would be useless.

A second condition is concerned with abilities of sensors. The emergent events are generated only if the range possess at least one sensor, which can detect the other entity. For example, a range with an audio sensor can detect only bodies producing some noise.

Class \texttt{Range} from package \texttt{basser.sim.lang} implements a range according to the description above.

When initialising, each entity which implements the Moving interface registers all its ranges with the collision detection unit. Class \texttt{Range} contains data and
implements methods for detection of collisions with other objects. There are two kinds of data; one describes the size and route of the active area (there are two basic shapes supported by subclasses CircularRange and RectangularRange). The other data structure creates a row in the Timetable. It consists of:

- **min** - minimum of all times kept by the row
- **Group** - group number of the range
- **Expired** - time the approximation of route becomes invalid
- **In, Out (n times)** - the time points some other range is entering or leaving the active range

Each entity with 2D representation is responsible to approximate its intended route in prescribed accuracy with partially linear movements. This information is passed to the registered ranges in Route2D objects. The end of the validity interval of such an approximation is kept in the **Expired** field. If the value in this field becomes the smallest across the Timetable it results in generating an UPDATE_ROUTE event, which is passed to the entity. The entity should compute a new approximation of its route. The Timetable receives the new Route2D and updates the row and the column of In Out values of that particular range. This may result in rescheduling, deleting of existing emergent events, or creating new ones.

<table>
<thead>
<tr>
<th>Min</th>
<th>Group</th>
<th>Expire</th>
<th>In 1</th>
<th>Out 1</th>
<th>In 2</th>
<th>Out 2</th>
<th>In 3</th>
<th>Out 3</th>
<th>In 4</th>
<th>Out 4</th>
<th>In 5</th>
<th>Out 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 9: Emergent Events and Timetable**

Both In Out fields are set by the range’s methods for detection of other entity entering, or leaving the range. Of the proposed time of entering is less than current time, it is set to +∞. So the double \(< In = 123, Out = 456 >\) means there is some entity entering the range at time \(t_{in} = 123\) and leaving at \(t_{out} = 456\), while double \(< In = +∞, Out = 456 >\) means the entity is already within the range and is going to leave the range at \(t_{out} = 456\). If the smallest value in the Timetable is in one of the In Out fields, it means an emergent event has to be generated and passed to the entity. After that, the value is re-set to +∞.

There is an auxiliary structure to help to devise the minimum time values from the Timetable. Each row contains min and minCell fields, which keep the minimum value, and its position, respectively. When some values are changed in the row, these two fields are kept consistent. The complexity of update operations ranges from \(O(1)\) (new value is bigger than min) to \(O(n)\) (the minimum was removed and
a new one has to be found). The Timetable contains similar fields, min and minRow. The value of the min field has a special application as it is described in the next section.

5.5.3 Main Loop

A correct function of the simulator is qualified by proper interlacing of events in the queue and the ones generated by the collision detection unit. This is reflected by the main simulation loop. Here is method run() of class Simulation.

```java
61: public void run() {
62:     tracing = true;
63:     init(time);
64:     suspend();
65:     while(time < endTime) {
66:         ui.display(time);
67:         if(timetable.minTime() < eventQueue.minTime()) {
68:             if(tracing) {
69:                 time = timetable.minTime();
70:                 timetable.generateEvent();
71:                 suspend();
72:             }
73:             else
74:                 if( waitTo(timetable.minTime()) )
75:                     timetable.generateEvent();
76:             }
77:         else {
78:             if(tracing) {
79:                 time = eventQueue.minTime();
80:                 eventQueue.generateEvent();
81:                 suspend();
82:             }
83:             else
84:                 if(waitTo(eventQueue.minTime()))
85:                     eventQueue.generateEvent();
86:         }
87:     } /* end of while */
88:     ui.end();
89:     this.suspend();
90: }
```

Field tracing indicates whether to stop after each processed event, or to block the execution for some time proportional to time lag between the current time and the time of a next event. Lines 62 to 64 initialise the system. The while loop at lines 65 - 85 is executed repeatedly until the current time \( \text{time} \) exceeds the specified endTime. The condition on line 67 decides whether the next event is scheduled (from the event queue) or emergent (from collision detection unit). In the first case, the first event from the queue is passed to its addressee, and it is removed from the queue (lines 77 - 83). In the second case, the Timetable is the one to generate the event (lines 68 - 74).

5.5.4 Simulator User Interface

Each simulation has a dedicated object from SimulationUI class which allows user to control the execution of the simulation. It is a simple frame with three buttons
(see figure 10), time display, and text area. It shows messages sent to the simulator (see method print()). All other windows are generated and “owned” by simulation entities.

When creating a list of simulable entities, an object of the class Viewport from package basser.sim.core. This is a simulation entity which has a range covering all used 2D area. During initialisation it shows a frame displaying entities and schedules an UPDATE event. The event scheduling mechanism is used to update the frame to create an animation of entities moving through the 2D space.

![Simulation User Interface](image)

Figure 10: Simulator User Interface

5.6 Utilities

This section lists miscellaneous utilities from the packages basser.util subtree which have not been covered above in section 5.2. The utilities implement different tasks like data exchange with the server, classes used in the user interface. More can be found in the on-line documentation, and/or the appendix C.

5.6.1 I/O Operation

Package basser.util.net consists of classes related to input/output operations and dynamic loading of classes.

The class Net contains mostly static utilities for loading data from server, or saving data on server. The methods cooperate with the scripts saveSecure.cgi (see section B.6) and javac.cgi (see section B.4) running on the server.

TopLoader class extends java.lang.ClassLoader for dynamic loading of classes during the run if the application. Its functionality depends on the security manager, which is a part of the host environment. This issue is discussed later in the section 6.1.1.

Interface Serializable assists with saving and retrieving data from objects. An object which implements this interface can use the NetFrame class object to send its data to server and to retrieve them later. The NetFrame implements a user interface to set the location on the server, password, and text commentary. ClassFrame class is an extension, which saves the name of the class of the object and it uses this information while loading; it creates a new object of the named class and restores its data.

5.6.2 User Interface Objects

They are several classes in the basser.util package implementing user interface of simulation entities. They are all based on class java.awt.Frame - independent window. We believe users can manage large amounts of information by organising
several smaller windows easier then panning a large window, which contains all the data. The following classes can be used while specifying behaviour of entities:

- **CloseFrame** - simple frame with a "Close" button
- **CanvasFrame** - a frame with a canvas inside
- **TextFrame** - a frame with text area inside
- **Chart** - a frame displaying a Histogram

More information about package structure and classes can be found in appendix C and on-line documentation.
6 VISP Functionality and Usage

This section deals with functionality and user interface of VISP system. It explains principles, describes main functions of VISP environment, and contains links to examples. We assume the reader is familiar with the basics of client-server architecture and can work with a Web-browser. A more detailed version can be found on-line at page


Links to examples are given in the form of URL (Uniform Resource Description). Readers with Internet access are encouraged to down-load pages specified by URLs. This can be done by selecting Open location from menu File and using the addresses specified by links.

6.1 Introduction

VISP is a Java based application for education and exploring of hybrid simulations. The scenarios can involve not only discrete event models, but continuously moving objects, too. The simulator was designed to detect collisions between such objects. Therefore, VISP can be used to model combat situations, traffic scenarios, or technological processes, to name only a few examples.

Our goal was to build an environment suitable for teaching. This resulted in an open client-server architecture based on the WWW technology. The server stores classes and examples which are instantly available to all clients. There is no need to install any software to client’s computers, or worry about distribution of examples and data. The Java was chosen for implementation and specification for its portability and other features. The resulting environment is portable and easy to use.

6.1.1 System Requirements and Functionality

The functionality of VISP may be constrained by the browser which serves as a host for the VISP applet. This is because of varying security limitations implemented to safeguard user’s local data by different browsers.

The VISP uses two kinds of operations, which may be recognised by some browsers as insecure: communication with more than one server, and re-loading of classes. The first one is used for providing the application with data. Some browsers consider data to be secure only if they were down-loaded from the same server as the application. Data from other servers are forbidden, or treated with care. The second task, re-loading of classes, is essential for editing of object behaviour.

The VISP was tested in two environments: the applet viewer shipped with JDK 1.0.2\(^6\) and the Netscape Navigator Version 3.10\(^7\).

Applet Viewer There is almost no limitations. The applet viewer allows to load resources from any available server, or re-load classes. It seems to be faster and less error prone than Netscape. The installation and use of an applet viewer is quite easy and we recommend its use to allow full functionality and better efficiency. The VISP can work in so called enhanced mode.

The installation of applet viewer is simple. It can be down-loaded for free from its on-line source\(^8\). The instructions for installation, conditions of use, and support information are provided there too. After installing the applet viewer, the VISP

\(^6\) http://www.javasoft.com/80/products/jdk/1.0.2/
\(^7\) http://help.netscape.com/
\(^8\) http://www.javasoft.com/80/products/jdk/1.0.2/
classes have to be down-loaded from ftp-source\(^9\), and their location added to the Java lib path (e.g. CLASSPATH). The application is started pointing the applet viewer to URL http://www.cs.su.oz.au/\~hejda/sim/java/sim.html.

Netscape  This browser allows to exchange data only with the server the application was downloaded from. The second limitation concerns the reloading of classes. Since this is not allowed you would not see the changes of behaviour. The VISp works in so called fundamental mode. The code of an entity cannot be edited, parameters cannot be added or removed, and some other functions are disabled too.

The following table shows the features of fundamental and enhanced modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Functionality</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fundamental</td>
<td>Down-loading data from the main server</td>
<td>Netscape</td>
</tr>
<tr>
<td></td>
<td>Drag &amp; drop objects to build a simulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I/O operations with the main server</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changing parameters of objects and simulation</td>
<td></td>
</tr>
<tr>
<td>Enhanced</td>
<td>Down-loading data from any server on the net</td>
<td>Applet Viewer</td>
</tr>
<tr>
<td></td>
<td>Drag &amp; drop objects to build a simulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I/O operations with any server</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changing parameters of objects and simulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specifying object behaviour</td>
<td></td>
</tr>
</tbody>
</table>

6.1.2 Basic principles

VISp is a multi-threaded application which runs in a host environment (either a browser or an applet viewer). Some browsers are designed to run applets if the page is in an open window. If the window is minimised or closed, the application may be shut down. Due to a complex function the essential resources are relatively extensive. Therefore, it is reasonable to open the VISp as a separate window and to display support and resource pages in other windows. How to open link in a new browser window depends on the browser, generally pressing the right button of a mouse (a mouse button while holding the Alt key) over a link will display a menu with option "New Window with this Link" (or similar).

To build a model even of a small system requires to specify a large amount of information. VISp manages to display it by using frames; the stand-alone dialog windows. A pop-up of a frame is a usual response to user actions. These frames are parts of VISp too, therefore their forced closing (e.g. using xkill in X-Windows) may close the browser. Though, frames can be minimised and maximised and closed as usual windows.

It is useful to know, which part of the job is done by a client and which one by a remote server. Tasks are distributed this way:

- Client runs the environment and your simulations and shows you results.
- Server compiles user defined classes, stores code and data.

The rest of this section is structured as follows; section 6.2 explains terms and definitions used in the text, section 6.3 describes the environment and its basic controls, section 6.4 explains the user interface of an entity, section 6.5 shows how to specify the behaviour of an entity, and section 6.6 deals with a few examples.

6.1.3 Where to Find More

The World Wide Web is a solid source of information. The following links can serve as a starting point:

Java:
http://java.sun.com/products/JDK/1.0.2/index.html
http://www.gamelan.com/index.html
http://www.gamelan.com/pages/Gamelan.programming.html

Simulation and Modelling:
http://www.tasc.com/simweb/papers/simulyzer/backgrnd.html
http://www.orcaComputer.com/
http://www.pad.uwaterloo.ca/~mjfrazer/drtis/
http://www.cise.ufl.edu/~fishwick/webconf.html

... or you can search the Web:
http://www.altavista.digital.com/
http://www.yahoo.com/

6.2 Terms and Definitions

This section provides more precise definitions to simulation terms such as entity or simulator. The event-driven nature of discrete event simulation is also presented in detail.

Entity is a part of a simulation, which has defined state and a specific behaviour. The specification of behaviour of an entity follows a formalism of discrete event system specification (DEVS) from [Zei84], chapter 4. According to author, the DEVS is a structure consisting of:

- the set of external events,
- the set of sequential states
- the transition specification, which consists of two parts:
  - internal transition function
  - external transition function
- time advance function

Time advance function is a mapping from the set of sequential states to non-negative reals (including infinity). The function specifies a time interval an entity is allowed to stay in stable state if no external events occur.

Event is a message with a specified time of delivery and addressee. It should be the only way two entities can communicate. Entity receives events to its input port. Events generated by entity are sent to one of the output ports.

The internal transition function specifies actions to be taken after the time specified by the time advance function has elapsed (provided no external event
occurred at the input port). This function is a mapping on the set of sequential states.

The external transition function defines a new sequential state after an external event arrived at the input port. It is a function of the current sequential state, elapsed time since the last event and the arrived event.

An entity behaviour specification is done by describing the state of the entity and by specifying methods implementing above mentioned functions (time advance function, internal and external transition function). The format of the methods and practical issues are discussed in section 6.5.

The sequential state of a system is determined by state variables. In VISP the state variables are variables and objects owned by the entity. Practical issues of state variables are discussed in section 6.4.

Each entity has an input port, which name is specified by pre-defined parameter Name. The addressee of an event is specified by a string. An event is delivered to all input ports which name matches with the addressee. If there is no input port of such name, the event is discarded. Output ports are defined by the user as variables, who contain name of coupled input ports.

Material entity is an extension to the term entity. It describes an entity with a 2D representation. The spatial qualities are defined by the term range, as it was defined in section 5.3.2.

Two material entities can communicate either by sending events to each other, or by interaction in space. This interaction is described by means of emergent events [SB95]. These events are not scheduled by any entity, they emerge as a result of entering or leaving active area of an entity.

A simulation is defined as a set of entities, which cooperate and interact during the simulation. The simulation is modular but not hierarchical. Each entity represents a module with defined input and output ports and behaviour. Although the practical benefits of hierarchical approach are immense [Zei84], VISP does not implement any hierarchy of entities. The main objective of the research was detection and processing of emergent events.

6.3 VISP Controls

6.3.1 Running the VISP

Page “Visual and Interactive Simulation Package” 10 on the VISP site hosts the VISP system. The content of this page depends on the browser employed. It is either a warning message, or an applet. The warning message means, that the browser is either not capable or not allowed to run Java applications. In the case your browser cannot run Java, the instructions provided should be followed. Running the VISP in a browser is only one of the ways of how to work with it.

In the more favourable case, the browser will down-load and run VISP. The amount of code and data is quite extensive, so it may take a little time. When down-loading is finished, a page similar to the one in figure 11 appears. It consists of the following parts:

- WorkPlace
- Button Bar
- Links to resources
- Icon Bar

6.3.2 WorkPlace

The blue part of the applet, which is shown in the browser is called a WorkPlace. Its main purpose is management of icons and grips. Icons represents simulation entities. At most one icon can be active at a time. The active icon is framed with a black rectangle. If the entity models a moving object, after activation all its grips are shown. Grips specify points in the plane. Figure 11 shows grips as small black rectangles. A single mouse click can activate (and deactivate) an icon. The double click on an icon shows the parameter window of the entity.

Figure 11: VISP Controls and Simulation Parameters

A double-click on the background displays parameters of the simulation (the small window in figure 11). Simulation parameters specify:

- Name - name of the simulation
- Start time - initial time
- End time - time when the simulation stops
- Time scale - scale of real and simulated time (value "20" means the simulation runs twenty times slower than real time)
- Tolerance - accuracy of route approximation (in pixels)
- Background - colour of the background

6.3.3 Button Bar

Below the WorkPlace is a button area. The buttons call

- Palette - shows a palette of classes.
- Up, Down - the active icon is painted over (under) all overlapping icons.
- Delete - destroys the active entity
- Simulate - creates a simulation and shows the simulation windows.
- Save/Load - shows the window to save or load set-up of the WorkPlace and Palette.
- Save/Load Item - a dialog window is shown which allows to save or load a single entity.

A more detailed description of each function follows.

![Palette of Classes](image)

Figure 12: Palette of Classes

**Palette** This button displays class palette - see figure 12. VISP is an object oriented system. Each simulation entity belongs to a certain class (or pseudo-class). A class can be understood as a definition of variables and behaviour. The palette keeps classes which are used in a simulation. The server manages a database of classes \(^{11}\), where classes are grouped in packages. To use a class in the simulation

\(^{11}\)http://www.cs.usyd.edu.au/~hejda/sim/lib.cgi/
it has to be down-loaded to the palette first. For example, to down-load class

test.counter\textsuperscript{12} means:

1. Click on the Palette button to show the Palette.

2. From the Palette menu Items choose Add.

3. Copy the name of the class (e.g. test.counter) from the information page and
   paste it to the dialog box.

4. Click on Accept.

A new button should appear in the palette. If down-loading was successful this
button is labelled by the icon of the class. It might happen, that the class was
deleted, is not available at the time, or the name is simply not correct. In this case
the new button is labelled with one of these icons:

\begin{itemize}
\item The class does not implement the desired interfaces.
\item The class cannot be found.
\item The class is O.K. but its icon cannot be found.
\end{itemize}

In the first two cases the class cannot be used. To delete it, click on it and choose
Remove from the menu Items. The third picture indicates the class is fine, only the
picture is not available. The section 6.4 shows how to specify which picture is used
as an entity icon. More about classes, documentation and the default palette can
be found in this page\textsuperscript{13}.

\textbf{Simulate} A click on this button causes VISP to create a simulation and show
its two windows: simulation console and a map. The console controls the run of
the simulation and shows messages sent to simulator by entities. The map shows
all moving entities as they travel across the area.

The buttons in the console allow to initiate a simulation, run it, stop the run
of the simulation, or advance in steps. Each step processes one event from the
event queue. As in the real world, the time can only advance. When simulation
time reaches the value of parameter \textsf{End time}, the simulation ends. The \textsf{Destroy}
button stops the simulation, destroys all entities and windows. Simulation runs as
a separate thread with its own set of entities. This allows to generate and explore
more than one simulation at a time.

\textsuperscript{12}http://www.cs.usyd.edu.au/~hejda/sim/lib.cgi/test.counter
\textsuperscript{13}http://www.cs.usyd.edu.au/~hejda/cgi-bin/doc.cgi/
Save/Load  Pressing of this button displays the input/output dialog window. The function of this window is to communicate with the database of users' data on the server (see figure 13). Name of the file have to always be specified. The password is required only for saving into protected directories. A text description of the simulation can be entered into the text area.

The database has a tree structure, the main page is a root of the server space dedicated for users' data. This page provides users with some more detailed information and first level directory entries. Some of the directories are write-protected by a password. An image in the right-hand side corner of a directory page displays the status of the directory:

The directory is protected by a password.

The directory is public and unprotected.

![Figure 13: Save/Load Dialog Window](image)

Each piece of data stored in the database can have a short description and its type can be indicated by the ending, recommended extensions:

- .sim - for simulations
- .ent - for entities

An example can be loading a simulation multi from directory /sim/data/tutorial. The procedure is:

1. Press Save/Load to show the Save/Load dialog window
2. Fill in the File Name field. The name consists of the name of the directory, slash, name of the file. In our example it is /sim/data/tutorial/multi.sim.
3. Press Load and wait until the dialog window disappears.

4. Notice the Palette may have changed.


As the new entities are added to the existing ones, the Save/Load operation can be used to merge two or more simulations. This way, you can build more complex simulations from existing ones. To save the data you need to fill the File Name field as well as Password. The information in the text area is optional, but it is useful to revert to an older setup.

Save/Load Item This button calls a dialog window similar to the one in the previous section. The function is also similar except that it works only with the active entity. The loaded entity is added to the existing set, so its copy can be made this way. All user actions in the scenario builder have some feedback. The dragging mouse over the icon causes the icon to move. The click on Save/Load shows a dialog window. If nothing goes wrong, the feedback is intuitive. Some actions may produce a warning or error message. Feedback of such actions is provided through Java console.

6.3.4 Links to Resources

There are three resource links bellow the VISP applet. Their destinations are:

- class database,
- database of users' data
- icon database

The main pages of these databases instruct users how to browse through and work with their content.

6.4 User Interface of Entities

Each entity in the simulation is presented by an icon in the WorkPlace. Double click on the icon displays a parameter window. The title of the window shows the class to which the entity belongs. The menu serves to add and remove parameters. This window has two parts. The upper part keeps parameters of the entity. Names are listed in the left column, values in the right one (see 14).

![Parameter Window](image)

Figure 14: Parameters

The second part of the parameter window consist of several buttons - (Export/Import, Code, and Close in the example). When clicked, these buttons display
windows describing the entity further. The last one ("Close") is an exception, it closes the parameter window.

6.4.1 Parameters

The state of the entity is described by the values of the entity variables. A variable can be any object exclusively owned by the entity. Although users can specify these objects directly in the code, it may be convenient to use the parameter manager.

The parameter manager keeps objects implementing basic types of parameters: an integer, a real number, a colour, a string, and an image. There are methods available for setting and reading the value of each parameter type. For example, for parameter “face” of type “java.awt.Color” exist these methods:

- `java.awt.Color face.value()` - returns the value
- `java.awt.Color face.value(java.awt.Color c)` - sets the value to “c” and returns it
- `void face.print(Graphics g, int x, int y)` - displays the value in simulation

The difference between variables defined by objects in the code window and parameters in a parameter manager is that the parameter window allows to change the initial value of a variable without the need for re-compilation of the code. This re-compilation is inevitable if there is a need to change initialisation of an object specified in the code.

There are two kinds of parameters: user-defined and pre-defined. Pre-defined parameters deal with general variables of entities, such as name of the input port, icon etc. The names of pre-defined parameters are shown in the parameter manager with a ‘?’ in front of them (see figure 14). Meaning of pre-defined parameters is specified later. User-defined parameters describe class-specific part of entity variables. For example, the parameter name specifies the name of entity input port, parameter image contains image used as an entity icon.

The enhanced mode allows to define the behaviour of entity. Among other things, names and types of user-defined parameters have to be declared. The menu bar of the parameter window contains options for adding and removing of parameters.

When in the fundamental mode, the behaviour of entities cannot be changed. This results in that only the initial values of parameters can be specified. It is not possible to add or remove parameters.

6.4.2 Buttons of Parameter Window

The number of buttons in the parameter window varies. Non-material entities have the following buttons:

- Export/Import
- Code
- Close

Material entities has one more button, Component Settings. With the exception of “Close”, each of this buttons displays a window with additional information related to the entity.
6.4.3 Code Window

The Code button displays Code window. It is divided into two halves (see figure 15). The lower half keeps the Java code describing the behaviour of the entity. This code can be changed in the enhanced mode only. In the fundamental mode it is read-only.

The upper part of the window shows information which can be useful when writing code. It lists type of the parameters, and some useful methods of the simulator. The main purpose of this window is to help with generating code.

![Code Window](image)

Figure 15: Code Window

Button Compile at the bottom of the Code window causes the code to be transported to the server, compiled, and the class to be re-loaded.

6.4.4 Export/Import Window

The Export/Import button displays a Export/Import window. This window contains the documentation of the entity. There are two bigger text areas in the window (see figure 16). The upper one should contain a short description of the entity. The lower text area should contain detailed description of parameters, code etc.

The second function of the Export/Import window is to interface the database of classes. It allows to specify name of the class and password (if the class belongs to a password-protected package). The buttons Save on Server and Load from Server in the bottom of the window export can create a new entry in the database,
and load an entry from the database, respectively. These functions are available in the enhanced mode only.

![Image of export/import window](image)

**Figure 16: Export/Import Window**

### 6.4.5 Component Settings Window

Window **Component Settings** is displayed after the button **Component Settings** in parameter window was clicked on. This button is available only for moving entities. The window is dedicated to range characteristics and to the way the entity moves (see figure 17).

Options under label **Range type** allows to decide between either Circular or Rectangular shape of the range active area. The size of the area is set in the parameter window. The window contains two pre-defined parameters **Width** and **Height**. The size and the shape of the active range is reflected in the WorkMap.

According to the description in section 5.5.2 sensors can be of certain types:

- Light
- Magnetic
- Radio
- Sound
- Thermo

The options under label **Sensibility** allow to chose which types of sensors the range has. For example, only ranges with a magnetic detector are sensible to magnetic objects. Any combination of types can be selected.
The options under label Affects allow to specify, which types of sensors can detect the entity. Entity with no such option set is invisible for any other entity. Any combination of types can be selected.

![Component Setting Window](image)

Figure 17: Component Setting Window

The right-hand side of the Component Settings window specifies the way the entity moves. There are six kinds of route types available:

- Fixed - entity does not move
- Linear Route - entity moves with constant speed
- Parabolic Route - entity moves with constant acceleration
- Spline Route - entity moves along a B-spline curve
- Navigated - entity is navigated during the simulation, manually
- Random - entity travels randomly in an area specified by a rectangle

The changes of the route type are reflected in the WorkArea. Starting position of the entity is always specified by entity icon. Types Linear, Parabolic, and Spline use grip(s) to determine the exact shape of the route. There are two pre-defined parameters in the parameter window which determine when the entity starts moving from the starting position and when it arrives at the final position; Start Time and End Time, respectively.

Type Random uses the position of the icon and a single grip to specify a rectangular area. The entity travels only within this area.

6.5 Specification of Behaviour

This section describes how to create a new class, specify its behaviour, and store the class in the database of classes. Actions mentioned are possible in the enhanced mode only.
6.5.1 Entities versus Containers

The objects "behind an icon" which represent an entity can be thought of as a container with a user interface, documentation and code specification. When a simulation is created, the container produces the entity and sends it a copy of all necessary data (for example the parameter values).

There are two kinds of containers:

- `basser.sim.comp.GenericSimulable` - a simple container allows to specify behaviour of a non-material entity (no range). Classes based on this container are referred to as non-material classes.

- `basser.sim.comp.GenericMoving` - an extension of the first one. It allows to work with a single range. Classes based on this container are referred to as material classes.

GenericMoving is a sub-class of GenericSimulable, therefore anything said about the later one applies to the GenericMoving class too.

6.5.2 Creating a New Class

New class can be created by exporting a template class under a new name. The procedure follows:

1. Decide on type of the class. The new class can describe either material or non-material entity.

2. Load a template class in palette. Template class for non-material classes is `basser.sim.comp.GenericSimulable`, template class for material classes is `basser.sim.comp.GenericMoving`. Optionally, other class from database of classes can be used as a template, provided it is of the same type. Material class can never be changed to non-material, and vice-versa.

3. Set new class name. The new class name is to be entered into field "Name" in the Export/Import window. The class name consists of name of the package and name of the class, as specified in Java. The name of the package can consist from one word only. For example, `spinning.Wheel`.

4. Set password. To export class to password-protected package, the password have to be set.

5. Export the class. The new class is exported to the class database after pressing the "Save on Server" button.

The name of the class should not collide with any other class name in the database unless you want to re-write it. The Java key words should not be used, it confuses the compiler.

6.5.3 Indicators and Documentation

Java Development Kit 1.0.2 have been used as the implementation environment for VISP. It is used to specify the behaviour of entities. This results in requirement to compile each class before it can be used in VISP. Due to the Java class detection mechanism, if the name of a class changes, the class has to be recompiled. To indicate if a class can be used the following cues are used:
The code has been changed since the last compilation. The entity cannot be used in a simulation. Solution: Use Compile button in the Code window. A text window appears to display possible errors.

The entity was not saved on the server since the name change. It can be used in a simulation. Each time someone loads it, this class has to be recompiled. Solution: Open Export/Import window, choose a name for your class (if necessary), set password if using a protected directory and press Save on Server button.

The entity was neither compiled nor exported recently. Solution: Open Export/Import window, choose a name for your class (if necessary), set password if using a protected directory and press Save on Server button. This exports the class and calls the compiler.

Only properly exported classes can be used in VISP running in fundamental mode. The code cannot be compiled in this mode, so the class has to be ready for use on the server. Information page of the class in the database of classes indicates, whether the export was successful.

6.5.4 Code Specification

The code window of a new class contains templates of methods, which implement the transition and time advance functions. The bodies of this methods have to be written by users to specify the behaviour of the entity. Their templates follow.

```java
public double time_advance(double elapsed_time, double time) {
    return -1.0;
}

public void internal(double elapsed_time, double time) {
}

public void external(double elapsed_time, SimEvent event) {
}

public void paint(Graphics g, int x, int y) {
    image.paint(g, x, y);
    g.setColor(color.value());
}
```

Method `time_advance(...)`, specifies, how long the entity remains passive if no events arrive. Negative value means the entity is in passive state. The method has two parameters, the first one specifies the elapsed time from the last event, the second one contains current simulation time. This method can use values of variables of the entity. A simple example can be `time_advance` method of a timer; it should simply return constant value: the length of one tick of the timer.

Method `internal(...)`, determines the internal transition. The method is called each time, the entity has not received any events in time specified by the
last call of time_advance method. It is supposed to use values of variables, change them (to make state transition), and/or schedule events for other entities. It is not allowed to schedule any events for itself. A timer can serve again as an example: internal method will schedule an event.

Method external(...) implements the external transition method. It is called whenever an event is sent to the entity input port. It can be either emergent or scheduled event. The method's first parameter is the elapsed time since the last received event. The second parameter is the received event. This method is supposed to change the value of variables and/or schedule events for other entities.

Method paint(...) should be defined for material entities only. It specifies the visual appearance of the entity during the simulation. It is called by Viewport objects when there is the need to update the viewport window. Parameters of method paint(...) describe where to paint: g denotes a drawing area, x and y the current position. Common use of this method is to paint values of entity variables. For example, the following sample code displays value of variable counter twenty pixels to the right from the entity icon.

```java
public void paint(Graphics g, int x, int y) {
    image.paint(g, x, y);
    g.setColor(color.value());
    counter.paint(g, x + 20, y);
}
```

There are two other methods users can over-load:

```java
public void init(double time))

public void destroy())
```

Method init(...) is called when the entity is created and passed to the simulator. The parameter time is set to current simulation time. Method destroy(...) is called after the entity was removed from simulator (or the simulation was destroyed). These two methods can be used to open and close additional windows implementing user interface of an entity, or creating and destroying additional variables. For example, class base.frame show received events in a frame. Code dealing with the window follows:

```java
TextFrame t; /* Text frame from base.util */

public void init(double time) {
    t = new TextFrame("In"); /* Creates a window */
    t.show(); /* Shows it on the screen */
}

public void destroy() {
    super.destroy(); /* Calls default destroy */
    t.dispose(); /* Hide and destroy the window */
    t = null; /* Free the reference */
}

public void external(double elapsed_time, SimEvent event) {
```
t.appendText(event + "\n" + id + "\n");
}

6.5.5 Events

Events are the only way how two entities can communicate. This section describes the internal implementation of class SimEvent. Understanding of the internal structure of this class is essential for analyses of received events.

Events are messages with specified addressee and time of delivery. There are several types of events according to kind of messages they carry. Public member public int id of class SimEvent specifies to which type an event belongs. The reserved values are:

0 - SimEvent.TIME_INFO - event carries only the time information
1 - SimEvent.UPDATE_ROUTE - internal use only ...
2 - SimEvent.ENTER_EMERGENT - entity "subject" entered the active range
3 - SimEvent.LEAVE_EMERGENT - entity "subject" left the range
4 - SimEvent.STRING_VALUE - event contains a String in the stringValue member
5 - SimEvent.INT_VALUE - event contains an integer in the intValue member
6 - SimEvent.DOUBLE_VALUE - event contains a double precision number in the doubleValue member
7 - SimEvent.GENERIC_VALUE - event contains an object in the generic member

Other members of the class SimEvent are used according to the type of the event:

- public double time - time of delivery (always used)
- public Simulable subject - reference to the entity involved in emergent event (for ENTER EMERGENT or LEAVE EMERGENT type only)
- public String stringValue - a string message (for STRING_VALUE type only)
- public int intValue - an integer (for INT_VALUE type only)
- public double doubleValue - a real number (for DOUBLE_VALUE type only)
- public Object generic - a general object (for GENERIC_VALUE type only)

Example in section 6.6.1 illustrates how id member can be used to print messages carried by events. Additional information can be found in on-line documentation\textsuperscript{15}.

6.5.6 Useful Methods

This section describes methods which can be used while writing code. They are divided into four groups. Methods in each group can be used to manipulate different objects:

- simulation
- entity itself
- simulator event queue
- parameters

Methods for manipulating simulation:

```java
public void sim.stop()  // interrupts the simulation.
It has the same impact as pressing Stop button during simulation.
```

```java
public void sim.step()  // advances to the next event.
It has the same impact as pressing Stop button during simulation.
```

```java
public void sim.cont()  // continue without tracing.
It has the same impact as pressing Simulate button during simulation.
```

```java
public double sim.time()  // returns the current simulation time
```

```java
public void sim.print(String msg)  // prints msg to simulation console
```

Methods for manipulating the entity itself:

```java
public boolean isPassive()  // returns true, if the last call
of time.advance method
specified negative value.
```

```java
public void passivate()  // cancels the next internal transition
```

Methods for manipulation with simulation event queue:

```java
public void sim.send(SimEvent);
    // schedule an event
```

```java
public void event.remove();  // cancels an event sent by sim.send
```

```java
public void sim.send( double time, String name, String s);
public void sim.send( double time, String name, int i);
public void sim.send( double time, String name, double d);
public void sim.send( double time, String name, Object o);
    // schedule an event with specified message type
    (one of String, int, double, or Object)
    to specified time
```
Methods for manipulation with parameters correspond with the type of a parameter. There are three methods for each user-defined parameter. For example for parameter "size" of type DoubleParam there are:

```java
double size.value() returns value of the parameter
double size.setValue(double) sets value of the parameter
void name.print(Graphics, int, int) displays parameter value
```

Pre-defined parameters are read-only during the simulation. Only two methods for each parameter can be used. For example, parameter "name" of type "String" can be manipulated with:

```java
String name.value() returns value of the parameter
void name.print(Graphics, int, int) displays parameter value
```

Examples of usage of the methods mentioned above can be found in the next section, the on-line tutorial and the on-line documentation.

6.6 Examples and Experimentation

This section elaborates some examples of code specification. Each example contains a description of functionality, a list of user-defined parameters, and code listing.

6.6.1 Event types

Description: This example illustrates extraction of information from received events. Each event received by external(...) method is analysed and according to its type the value of one of the fields is printed with a comment.

User-defined parameters: none.

Code:

```java
public void external(double elapsed_time, SimEvent event) {
    String m;
    switch(event.id){
        case 0: m = "Time is " + event.time; break;
        case 1: m = "You will never see this line."; break;
        case 2: m = event.subject + "entered."; break;
        case 3: m = event.subject + "left."; break;
        case 4: m = "Got string: " + event.stringValue; break;
        case 5: m = "Got integer: " + event.intValue; break;
        case 6: m = "Got real: " + event.doubleValue; break;
        case 7: m = "Got object: " + event.generic; break;
        default : m = "Unknown type received";
    }
    sim.print(m + "\n");
}
```

56
6.6.2 Internal Transition

Description: This is an example of entity implementing internal behaviour only. It is a timer, which generates events and sends them to addressee.

User-defined parameters:

String addressee
double tick

Code:

```
public double time_advance(double elapsed_time,
                        double current_time) {
    return tick.value();
}

public void internal(double elapsed_time) {
    sim.send( sim.time, addressee.value(), "TICK");
}
```

6.6.3 External Transition

Description: This example shows a passive entity, which re-sends received events. The sent events carry value of entity parameter color.

User-defined parameters:

String addressee

Code:

```
public void external(double elapsed_time,
                        SimEvent event) {
    sim.send( event.time, addressee.value(), color.value());
}
```

6.6.4 Combined Internal and External Transition

Description: This example shows a class which implements a delayed delivery of an event. It requires to specify both internal as well as external transition functions.

User-defined parameters:

String addressee
double delay
Code:

```
SimEvent msg;

public double time_advance(double elapsed_time,
                           double current_time) {
    return delay.value();
}

public void internal(double elapsed_time) {
    if (msg == null) return;
    msg.time += elapsed_time;
    msg.s = sim;
    msg.targetName = addressee.value();
    sim.send(msg);
    msg = null;
}

public void external(double elapsed_time,
                      SimEvent event) {
    msg = event;
}
```

6.8.5 Emergent Events

Description: The emergent event can be employed to model many phenomena. A few examples can give you an idea how much they are powerful. This example shows a simple class, which count the number of entities, which are in its range. The information can be used to trigger some action if the counter exceeds a specified value. The implementation is simple. The count is incremented with each received entering emergent event, and decremented with each leaving emergent event.

Used-declare parameters:

```
int count
```

Code:

```
public void external(double elapsed_time, SimEvent event) {
    int c = count.value();

    switch(event.id) {
      case SimEvent.ENTER_EMERGENT:
        c++;
        break;
      case SimEvent.LEAVE_EMERGENT:
        c--;
        break;
      default:
        break;
    }
```
count.setValue(c);
}

describe:

public void paint(Graphics g, int x, int y){
    image.paint(g, x, y);
    g.setColor(color.value());
    count.paint(g, x + 20, y - 10);
}

6.6.6 Keeping a List

Description: Sometimes it may be useful to keep a list of visible entities, e.g. the entities which are in the active area of entity. Emergent events can be used to update the list according to the present situation. The following class keeps a list and send it to the addressee whenever it is changed.

Used-defind parameters:

String addressee

Code:

/**
 * List of the subjects
 */
Vector list;

public void external(double elapsed_time, SimEvent event) {
    int c = count.value();
    if(event.id == SimEvent.ENTER_EMERGENT)
        list.addElement(event.subject);
    if(event.id == SimEvent.LEAVE_EMERGENT)
        list.removeElement(event.subject);
    sim.send(event.time, addressee.value(), list);
}

Other examples of class behaviour specification can be found in the on-line tutorial\[16].

7 Summary

7.1 Evaluation of the Project

A complete, fully functional environment for modelling and simulation was designed, implemented, and tested. It comprises of two co-operating parts, the VISP server and the VISP client application. Part of the implementation was based on our theoretical work on emergent event computations.

The theoretical work analyzes the problem of two colliding bodies. It derives conditions for entity's route description for the easy and accurate computation of emergent events. These conditions are used to approximate object's route by a piecewise linear motion with a given tolerance. It is shown, that using this approximation the complexity of detection algorithm is generally better than the one of static approach. From the practical point of view the construction of heterogeneous simulator is described too.

The VISP server facilitates software, data archives, and sources of relevant information. It was developed on existing and wide spread technologies and standards to ensure the widest possible use. The resource management allows co-operation of users and exchange of information.

The VISP client application together with its documentation, tutorial, resource databases provides users with a scenario builder and a simulator. The scenario builder allows to combine entities from resource databases with the ones created by a user. Specification of entity behaviour is described in a modern programming language. The scenario builder server as an interface for altering the information in the resource databases.

Other part of the VISP client application is an event driven simulator which implements results of our theoretical work. This results in a fast simulator which works with good accuracy. Other features of the simulator are multi-threaded execution, generic user interface, interactive control and observation.

7.2 Future Research

Discrete event simulators inherently built for simulation of transaction based systems are clearly not going to be able to handle simulations involving sophisticated object models which themselves move in space. Such scenarios are encountered in nature far too often; our example of ships moving on the ocean surface interacting with each other and with naval mines, aircraft approaching an airport and can communicate with other aircraft and ground stations are examples for systems which require special treatment.

What is common to this class of systems is that their components come in contact with each other in space in a way that is not necessarily planned or foreseen by any of these components. In naval mine warfare simulation models for example, it is crucial to account for situations where a ship encounters a naval mine on its course as a results of which the ship or the mine (or both) may be fully or partially incapacitated. In reality such encounters occur routinely and unexpectedly from the individual components' point of view. In simulation models, to ensure validity, such events can not be handled in the traditional way where each event is scheduled by a specific component and remains under its control and ownership.

In our research we termed such events as 'emergent events' indicating their nature that they emerge from the interactions of the components of the system being simulated rather than being proposed by a specific component. We also determined to handle an emergent event one must first recognize and schedule it and then manage the interactions of components involved in it at the time of the emergent event. There can be more than two objects involved in an emergent event,
e.g., three-way interaction. Furthermore, emergent events may be merged with other concurrent emergent events and they could be split into two or more emergent events, e.g., four-way event can be split into two two-way events. In addition, normal traditional events must also be handled appropriately and concurrently with the emergent events if necessary.

In this research, we abstracted an object by a point in space and an effective area which usually is a circle in which the object is capable of sensing other objects. An emergent event arises when another object intrudes this area. The simulator has capability to detect such potential intrusions and schedule emergent events accordingly. If the emergent event eventuates, that is neither of the objects change their course, the simulator manages handling of the emergent event. We feel our solution to handling such events is simple but powerful; Each object is notified of the event and of the components around it upon the start of the emergent event. Components are also notified upon their departure of the critical region. Effectively, management of an emergent event is closely associated with space; each event is essentially a critical region in space which is monitored, managed and dynamically updated.

Computation of critical regions and times where and when emergent events may occur are conducted by reasonable approximations as described in the text. The simple algorithm roughly described in the above paragraph and described in more detail earlier in this text handles both arbitrary merging and splitting of emergent events. The approach imposes no further restrictions on the forms or structures of objects involved. The VISI system which subsequently was implemented as a sophisticated prototype provides a proof that indeed the algorithm works well. This generalization of emergent events and their handling for their simulations is sufficient to handle all the practical cases we can think of, i.e., cases where sophisticated system components move in space and interact with each other. These ideas should be easily applicable to other application domains other than naval mine warfare.

The Visual and Interactive Simulation System (VISI) is a state of the art software package which combines traditional concepts of discrete event modeling with newer approaches involving emergent event processing. For future, concepts of emergent event processing may be explored further. For example, objects which have multiple sensors are common in reality. In this work each object has a specific type of sensor. It would be a straightforward extension then to this work to manage a single object having multiple sensors and thus multiple critical regions. Work should also be carried out to develop further and more accurate approximations to routes objects follow.

VISI system facilitates an experimental environment to further investigate various scenarios of naval mine placing and clearing. VISI can help not only to assess resource implications of a particular clearing strategy and rapid decision making but also to provide assistance with micro-strategies of the actual clearing process, identification of patterns, etc. This would require a lot more knowledge of strategies and scenarios. From a pragmatic perspective, scenarios would not be of unlimited numbers but rather of nature that can be abstracted, characterized, stored on a digital computer and be manipulated and examined. In rapid decision making "What if...?" kind of questions can be answered through a combination of series of simulations and proof and disproof of reasonably sophisticated propositions. Management of scenarios and identification of strategies also have a tremendous value for training situations where typically an instructor puts out a challenge in terms of a scenario for which a strategy must be developed to undertake the challenge. Studying such scenarios is crucial then and is possible because VISI provides a sound experimental base. In addition to simulations, storage and retrieval techniques are needed to be developed for scenario creation and strategy development.

Finally, VISI is deliberately built using the newly emerging Internet enabled
technologies. This fact enables cooperation amongst individuals and groups, however, to facilitate real cooperation and distributed group decision making further improvements would be necessary. For example VISP allows people to share models but not workspace. Proper group work, one would imagine, would typically involve scenario creation and strategy development and it is this level of group work that VISP does not provide any specific improvements for. The value of such interaction must be clear for training situations. It can also help rapid decision making especially when decisions are made by a group geographically distributed.
References


A Approximation and Collision Detection

Discrete event simulation is an efficient method for the implementation of certain classes of models on a "classical" computer. This is mostly caused by the simple fact these computers work with discrete information at a discrete points of time. With a bit of exaggeration we can say today's programming models are based mostly on event processing which fact is now reflected even in the computer hardware architecture.

The use of a computer for the simulation of continuous or hybrid models is more demanding. Objects in such a model change its state not only at discrete points of time but continuously. The models which contain moving and colliding objects can be classified as hybrid. But the act of getting into the incidence is usually discrete. It happens only in discrete time points. The problem is the events emerging from mutual position changes cannot be planed by any object. This is the base of the idea the discrete event simulator can be used for such a model if it contains a special incidence detection unit.

Discrete event simulator have been successfully used to implement a naval mine warfare modelling and simulation environment [Sev96b]. Objects have been defined as a points with a defined active range. An object (called observer) can operate on other object in its active range. If an object enters (or leaves) the observer's active range an emergent event (see [SBR+] too) have to be generated and passed to the observer. It can react appropriately by handling such an event.

The approach described in [SBR+] allows the detection of emergent events where one of the participants is motionless and both ranges are approximated by circles. This paper concentrates on conditions for objects' location functions, fast emergent event computation and the implementation of incidence detection unit capable to produce emergent events where both parties are moving. Interactive steering of objects and object's classification are also discussed.

We hope the specialisation of our paper do not limit its use for other implementation fields like 3D modelling and simulation environments, virtual reality, or traffic and combat models.

The reminder of this paper is structured as follows. Section A.1 defines the problem and some basic terms for a later use. Section A.2 contains short survey of related articles and conceptions. Section A.3 discusses objects' motion description, and limitations of its different specifications. Section A.4 introduces three basic algorithms for the emergent event detection. Sections A.5 and A.6 deals with classification of objects motions and complexity issues. Section A.7 shows some practical results measured by simulations. And finally Sections A.8 and A.9 discuss some of the implementation aspects of a discrete event simulation.

---

17 The Van Neumann architecture is meant by a "computer" in this article.
18 The collision of hard bodies, subject entering or leaving observer's active range.
A.1 Definitions

Simulations usually involves more than two objects but the incidence of objects can be processed by pairs. More complicated situation have to be handled by the object's implementation of emerging event handler. An object can keep a list of already visible subjects which can be updated by emergent event handler. This result in a need of two types of emergent events: entering and leaving ones.

Though both objects in a pair can have an active range, the emerging events can be generated sequentially for each object. The object which is the addressee of emerging events is called observer, and the object which is to be observed is called subject, respectively. The problem can be clearly stated as:

“How to define observer’s range, its desired motion, and subject’s desired motion to be able to compute emergent events fast?”

The second topic of this paper is how to implement emergent event detector and integrate it to a discrete event simulator. To be able to express some more complicated ideas following terms are to be defined.

\(a, b\) Objects are marked by letters, \(a\) is used for the observer, \(b\) for the subject, if not stated otherwise.

\(\bar{p}_a\) Position of an object \(a\) is denoted as a point in space \((\bar{p}_a \in R^n)\).

\(\bar{p}_a(t)\) Route of an object \(a\) is specified by a continuous location vector function which determines the positions of the object at a given time; \(\bar{p}_a(t) = f(t) : R \rightarrow R^n\).

\(R_a\) Range is an object's active area \((R_a \subset R^n)\). If a range is defined as circular (or spherical) its radius is denoted as \(r_a\).

\(t^{\text{ent}}_{a,b}\) is a time in which subject \(b\) is entering range of observer \(a\). An entering emergent event have to be generated and sent to the observer \(a\).

\(t^{\text{left}}_{a,b}\) is a time in which subject \(b\) is leaving range of observer \(a\). An leaving emergent event have to be generated and sent to the observer \(a\).

A.2 Related Work

Despite a specialised definition of the subject of this paper we have found a mass of related material in other scientific fields. Emergent event detection can be seen as a subset of a collision or clash detection under certain mild circumstances. However the definition usually differs.

Collision detection algorithms have been attracted researchers from different fields ranging from computer graphics, virtual reality to molecular physics and statistical chemistry. Employed objects vary from spheres to complex polyhedra and implicit surfaces combinations. Solutions often depends on the application, there is not any ideal algorithm suitable for all needs. The following survey is showing how goals and solutions can vary. The listing of algorithms can be roughly divided into two groups. First one contains improvements of a static collision detection algorithm. Methods in the second group are based on the fact that objects are moving.
A.2.1 Static Approach

The elementary algorithm is simple and robust. The number of times within our time-span of interest is chosen first and than a static interference test is performed at each time. There are several methods how to improve the efficiency of this method. They employ objects' bounds, a time-step varying, the caching of data and the reduction of number of necessary tests.

Garcia-Aleman et. al. introduces in [GASF94] an interest matrix, containers, boundaries, and voxel spaces as an approximation for fast estimation of non-existing collisions. Pobil and Serna in [PS95] elect interior and exterior sets of spheres for the approximation since the sphere is invariant under translation and rotation transformations.

A solution for convex objects is described by Baraff [Bar90] where Newton's method and caching have been used. The interval method in [SWF+93] allows to include not only convex objects.

Space-Time Bounds in [Hub95] can be taken as a bridge between static and dynamic approach. The collisions are checked in time points, but some of the checks are eliminated using information of object's acceleration. Objects' complexity problem is addressed by a sphere-trees. Hierarchical Time-Critical Computing is focussed on giving as accurate results as possible in the given processor time.

A.2.2 Dynamic Approach

The methods mentioned here are based on the fact objects are moving through the space, they have defined a velocity, an acceleration, or a location function.

Hard bound models are probably the most traditional application used in molecular physics and theoretical chemistry. Hence the objects are usually moving in a vacuum without considering any force fields, state of an object between two collisions can be described fully by means of an initial position and a constant velocity. Berne uses time-tables for collision's planning. The later article by Lubachevsky [Lub91] shows that this tables are not essential for hard bound models because each collision have an influence to both participants. A space partitioning helps to reduce the number of object's pairs.

An accurate calculation for polyhedra with a limited location function degree is shown in [Can86]. Author transforms the problem to the search of a point in a configuration space. It is shown, that the collision can be detected analytically, for a uniformly moving and rotating object and a stable obstacle. However if both objects are moving uniformly, an approximate iterative method have to be used.

The method described by Cameron in [Cam90] is based on the intersection checking of 4D objects made by extrusion of 3D object in time dimension. Objects are described by constructive solid geometry (CSG) trees. The transformation of an object is distributed to simple shapes at the leaves of its CSG tree and the collision problem is transformed to the search of a null set. Similar approach is described in [Duf92]. Objects are built using a CSG tree of implicit surfaces. Detection collision is solved as the searching of the minimal set of points in a four dimensional space using subdivision techniques.

A.3 Object Movement Description

Each object in the simulation have to have a location function which specifies its desired motion. Whether or not an object will follow its desired route depends on the simulation, but object's aims can be changed only by events the object is able to handle. Between two successive events the object's route corresponds to its desired motion.
This section focuses on the qualities of the location function. The static collision detection algorithm is based on checking for collisions in specific time-points. Does this mean, the point knowledge is sufficient enough? However not. It will be shown that this information is not sufficient enough for detection of all emergent events correctly.

For the sake of simplicity the range of the observer is approximated by a circle with a radius $r > 0$. Emergent events’ times are determined by the following simple equation:

$$|\bar{\mu}(t) - \hat{\mu}(t)| = r$$

(1)

The left side of (1) is a scalar function of a scalar variable. It is widely known that only point knowledge is not sufficient to compute solution even with some tolerance. The problem is caused by unknown behaviour of the function between the known points.

This situation is illustrated in Figure 18. If two successive values of distance are bigger than critical value ($r$), it cannot be concluded that the solution cannot be somewhere in-between. Collision detection implementation based only on point knowledge can compute the $t_{min}^i$ as $B$ (or in a worse case as $C$) while the correct value should be $A$.

A satisfying conditions which grants that a solution need not to be searched between two distant points is the knowledge of monotony sections. The monotony enunciate that unknown function’s values are limited by the known surrounding points. This result in a “good” behaviour of the function between the known points. However recognising and computing of monotony sections is solved only for both objects moving with constant velocity ($\dot{r} = \text{const} \in R_n$).

The second and widely used approach gain from the knowledge of objects maximum velocity in some time span. The maximum speed defines maximum value of the first derivation of distance function. This value limits maximum change of the value and can be used to determine the “sampling” step of positions. Combination of position point knowledge and maximum velocity has been used for fixed step algorithm (FS), see Section A.4.1.

![Figure 18: “Well” and “bad” behaving function.](image)

The third approach is based on a distinct scheme. Rather then asking object about its position and than reconstructing its route, each object is asked to approximate its path with a given tolerance. This approximation can be used for emerging
event detection. Which class of functions can be used for approximation and how efficiently approximate is the topic of the following sections.

We can summarise this section by stating the minimal information about a desired object’s motion the incidence detection unit has to have in order to be able correctly compute time of a possible emergent events. The point knowledge is not sufficient enough, either maximum velocity or the understanding of monotony sections is required. Some other way is to let each object to approximate its route by some simpler function and use this information.

### A.4 Incidence Detection Algorithms

Section A.2 suggests an idea there are plenty of different methods approaching the incidence detection problem. Surprisingly, all of them are based only on few basic algorithms. In this section we have tried to pick up and describe the most basic ones. The first two are merely for comparison, we focus more on the third one later on. In the text below an observer is usually marked by a while b denotes a subject. Additionally, the $|\vec{v}_{max}|$ is for the maximum speed of the faster from both objects.

None of the three methods is accurate, all of them are able to find the solution (if it exists) with some tolerance $\varepsilon > 0$. A method is said to work correctly with a given tolerance if it is able to find every solution $t_{sol}$ for which:

$$\max_{t_a, t_b \in \langle t_{a}, t_{b}, t_{a}^{\varepsilon}, t_{b}^{\varepsilon} \rangle} |\vec{p}_a(t_a) - \vec{p}_a(t_b) - \vec{p}_b(t_a) + \vec{p}_b(t_b)| > \varepsilon$$

$$|\vec{p}_a(t_{a}^{\varepsilon}) - \vec{p}_a(t_{sol})| < \varepsilon$$

$$|\vec{p}_b(t_{b}^{\varepsilon}) - \vec{p}_b(t_{sol})| < \varepsilon$$

The first condition eliminates situations where the subject’s penetration of the range is shorter then tolerance $\varepsilon$. The second two conditions limit maximum distance between found and ideal entering times. Similar conditions have to be satisfied for the leaving time.

#### A.4.1 Fixed Step Search

Fixed Step Search (FS) is the oldest and most naive algorithm. Object’s incidence is checked for a chosen set of time points $\{t_0, t_1, ..., t_n\}$. Usually this points are equidistant, there is a fixed time-step $\Delta t > 0$ and

$$t_i = t_{i-1} + \Delta t, i \in \langle 1, n \rangle.$$  

The maximal speed ($|\vec{v}_{max}|$) is employed to obtain correct results. According to [UOTS3], to ensure given tolerance $\varepsilon$, an incidence have to be checked at each $t_i$ time with the time-step

$$\Delta t = |\vec{v}_{max}| / \varepsilon.$$  

#### A.4.2 Algorithm Description

```plaintext
//** Pseudo-code for Fixed step algorithm */
01
02 Each object have to implement following method:
03
04 maxSpeed - returns maximum object's speed
05
06 Fixed Search is
07```
max_speed = max (maxSpeed of a, maxSpeed of b)
time_step = max_speed / tolerance
time = start_time

while (simulation) do {
time = time + time_step
if ('b' is in range of 'a') return time;
}

/** end of pseudo-code for Fixed step algorithm*/

A.4.3 Prove of Correctness

This algorithm checks for an incidence at each $\frac{\bar{v}_{max}}{\varepsilon}$ time. In $\Delta t$ each object can move only to a place which is distant tolerance $\varepsilon$ or less from its previous position. If an incidence is detected, it is sure not further away from an ideal point than the value of tolerance. Other consequence of this approach is that the incidences of an observer with lesser range than accuracy need not to be find.

A.4.4 Complexity Analysis

Complexity of this algorithm is compound complexity for one step times the number of steps. One step has a constant complexity. If the observer's range is approximated by a circle it takes approximately less then 10 computational operations (+, -, *) for one incidence detection. The number of steps during the simulation is determined by the tolerance $\varepsilon$ and the maximum speed in the scene $\bar{v}_{max}$. The complexity is therefore:

$$O(\varepsilon/\bar{v}_{max}).$$

More about complexity issues can be found in efficiency measures (Section A.7).

A.4.5 Pros and Cons

Although FS is the slowest and non-efficient algorithm it is still used for its robustness and simplicity. The complexity is rather high (in comparison with other algorithms) but it is stable and predictable during the simulation and can be lowered by speed-up techniques like a space-partitioning in [GASF94] or space-time bounds in [Hub95].

A.4.6 Binary Search

Binary search (BS) was developed as an attempt to speed up the fixed step algorithm. The idea is simple, knowing a time point $t_{out}$, the subject is outside the observer's range, and $t_{in}$, it is inside, the place of approximately accurate position of entering can be found fast by a binary refining. In each step the mid time ($t_{mid}$) position is checked and the interval $< t_{out}, t_{in} >$ is narrowed down using $t_{mid}$ as a lower or an upper bound, if the subjects is out or in the range, respectively.

This implies conditions to a preprocessing. It have to provide the BS algorithm with:

- $t_{out}$ - a time point in which the subject is outside the range
- $t_{in}$ - a time point in which the subject is inside the range.
The FS algorithm with a modified step was chosen for the preprocessing. The step was adjusted so the range of the observer covers the route with a given tolerance \( \varepsilon \). See Figure 19.

Figure 19: Covering an object’s route with a given tolerance.

The modified time step is then (assuming a circular range with the radius \( r_o \gg \varepsilon \)):

\[
\Delta t_{\text{mod}} = \frac{\sqrt{2r_o\varepsilon - \varepsilon^2}}{|\vec{v}_{\text{max}}|} \leq \frac{\sqrt{2r_v\varepsilon}}{|\vec{v}_{\text{max}}|}
\]

Among more objects the minimum range and maximum \( |\vec{v}_{\text{max}}| \) in the scenario should be used for correct results.

A.4.7 Algorithm Description

Again, for the sake of simplicity only a circular range is considered. For more complex ranges the distance of the successive solutions can be used in line 16.

```c
/** Pseudo-code for Binary search algorithm */
01 Each object have to implement following method:
02 03 maxSpeed - returns maximum scalar speed of the objects
04 05 /* Preprocessing */
06 07 use Fixed Step with
08 09 time_step = Sqrt(2*range*tolerance) / max_speed
10 11 time_in = time found by FS
12 time_out = time_in - time_step
13 14 while (!)
15 16 do {
17 18 19 t = mid time of time_out and time_in
20 21 22 distance = distance of objects in time t
23 24 25 if ( distance - range < tolerance ) return t;
```

71
if ( distance > range ) time_out = t
else time_in = t

/** end of pseudo-code for Binary search algorithm */

A.4.8 Prove of Correctness

This algorithm is not generally correct. It is mainly caused by the following fact: 'only point knowledge of objects path is not sufficient for correct estimate of objects behaviour.'

The preprocessing is the main weakness of this approach. It cannot work correctly on cases the object is moving forwards and backwards, simply because it may not take in mind the highest extend of the object's "pass". To determine it all points of the path have to be explored at least with the given tolerance, exactly as the fixed step algorithm does.

Once the time_out and time_in values are find, the binary refining can quickly determine the entering time with the given tolerance. The other problem is that this time need not to be the time of the first contact. Again, this is a result of the point description weakness.

A.4.9 Complexity Analysis

As with the FS algorithm, the complexity of BS algorithm is compound complexity for one step times the number of steps. The total number of steps is a sum of the steps of preprocessing and the steps of binary refining.

The complexity of preprocessing is (circular range is considered):

\[ O \left( \frac{|V_{max}|}{\sqrt{2r_a \epsilon}} \right) = O \left( \frac{|V_{max}|}{\sqrt{r_a \epsilon}} \right). \]

In comparison to FS, with more narrow tolerance deflection the complexity is growing slower. Binary refining starts with deflection given by modified time step and ends with tolerance \( \epsilon \). Its complexity is therefore:

\[ O \left( \log \sqrt{2r_a / \epsilon} \right) = O \left( \log \frac{r_a}{\epsilon} \right). \]

The total complexity of this algorithm is consequently

\[ O \left( \frac{|V_{max}|}{\sqrt{r_a \epsilon}} \right) + O \left( \log \frac{r_a}{\epsilon} \right) = O \left( \frac{|V_{max}|}{\sqrt{r_a \epsilon}} + \log \frac{r_a}{\epsilon} \right). \] (3)

Some practical results can be found in Section A.7.

A.4.10 Pros and Cons

Despite the problems mentioned above, this algorithm shows in practical use a stable behaviour. However, it is of a little use for the applications in which the finding of all incidences is crucial.

A.4.11 Approximation

The approximation algorithm (AA) is based on the idea that each object knows its behaviour the best. It can therefore approximate its route by finite number of fragments with a simplified location function. An approximate point of collision can be computed using these functions. For the first look this approach does not seem to have a big benefit. But if the functions used for approximation are simple enough to allow fast and accurate solution, it is worth of consideration.
A.4.12 Algorithm Description

The algorithm calls forth objects for their intended route and uses this information for the incidence detection. It can be extended to handle objects changing its intentions (see Section A.8). One of its nice features is that objects can approximate its position by the most efficient way.

![Diagram showing true and approximated solutions.](image)

Figure 20: The true and approximated solutions.

```c
/** Pseudo-code for approximation algorithm */
01 Each object have to implement following method:
02
03 intendedRoute(t) - returns the approximation of route
04 Route is structure holding:
05    object's range
06    t0 - start time
07    t1 - end time - how long the approximation describes
08    object's motion with the given tolerance
09
10 Route a, b;
11
12 t = start_time
13 a = A.intendedRoute in t
14 b = B.intendedRoute in t
15
16 while(simulation)
17   do {
18      if a coincide with b return the time
19      time = min (a.t1, b.t1)
20   }
21
22 if (t >= a.t1) a = A.intendedRoute in t
23 if (t >= b.t1) b = B.intendedRoute in t
24
/** end of pseudo-code for Linear approximation algorithm */
```

73
The algorithm lets each object approximate its route. If the end of the approximation is reached it asks for a new one (lines 22, 23). The issues of routes' coincidence (line 18) are discussed in Section A.5.

A.4.13 Prove of Correctness

Each object describes its intended route by an approximation with the given tolerance $\varepsilon$ so that its approximated position is less then tolerance value apart from its real position, see Figure 20. If an incidence time is computed using these approximations, it is for sure not further then $2\varepsilon$ from its real position.

A.4.14 Complexity Analysis

The AA algorithm is based on the approximation by piecewise simplified motion. The number of the fragments is essential for the complexity of collision detection. It is equal or less the sum of fragment's numbers of both objects.

The number of fragments during the simulation depends on the behaviour of the object. It can vary in following bounds:

- one - stationary objects or objects which movement is as simple as its approximation.
- $O\left(\sqrt{|\text{max Acceleration}|/\varepsilon}\right)$ - for objects with a known maximum acceleration (see Section A.5).
- $O(|\bar{v}_{\text{max}}|/\varepsilon)$ - route of the object is determined by a function which have to be sampled (see Section A.5).

For many objects modelling a real world entities this number is usually small. This is caused by energy laws of the real world. Each acceleration (the change of a speed or a direction) costs some energy. The higher the acceleration is the more energy have to be spent. Objects in the real world tend to preserve their energy. More about object's classification can be found in the next two sections.

Computational effort spent on one step is constant and takes less then 40 operations $(+, - , \cdot , \sqrt{ })$ for a circular range. Both entering and leaving emergent event times can be computed at once.

A.4.15 Pros and Cons

AA works the most efficient way with the information of objects movement. Letting objects to approximate their route, the detection algorithm have not only point knowledge of intended path, but also knowledge how the objects behave "between the points". On the other side this method is harder to implement, each object has to provide the simulator with an intendedRoute function, which is sometimes not as easy to devise.

A.5 Approximation Functions

In the definition of the AA algorithm was mentioned it is efficient provided the functions used for approximation are simple enough to allow an easy, exact and fast incidence computation. The iteration processes (binary search, Newton method) for solving more complicated equations was rejected since they are slow and usually need some additional conditions. To discover the maximum set of useful functions the following reasoning has been done.
As in the previous text, let \(a\) be the observer with the range \(R_a\) and \(b\) the subject. Routes of both objects can be approximated by the first \(k\) members of Taylor series of \(\vec{p}_a(t), \vec{p}_b(t)\) functions, respectively. For \(a\) can be written:

\[
\vec{p}_a(t) = \sum_{j=0}^{k} \frac{\vec{p}_a^{(j)}(0)}{j!} t^j = \sum_{j=0}^{k} t^j \vec{a}_j
\]  

(4)

Where \(\vec{a}_j = \vec{p}_a^{(j)}(0)/j!\). This vectors have a practical meaning, \(\vec{a}_0\) is the initial position, \(\vec{a}_1\) is the initial velocity, \(\vec{a}_2\) is the initial acceleration, \(\vec{a}_3\) is the change of acceleration, and so on. The number of members of the Taylor series which can be used depends on the definition of the range. The following three cases are considered: a circle, a square, and a half-space.

A.5.1 Range Approximated by a Circle (Sphere)

Range is a circle (sphere) with a radius \(r_a\). Collision detection can be described as the solution of the following equation:

\[
|\vec{p}_a(t) - \vec{p}_b(t)| = r_a
\]

Applying the Taylor series . . .

\[
| \sum_{j=0}^{k} t^j \vec{a}_j - \sum_{j=0}^{k} t^j \vec{b}_j | = | \sum_{j=0}^{k} t^j (\vec{a}_j - \vec{b}_j) | = r_a
\]

\[
\sum_{d=0}^{n} \left( \sum_{j=0}^{k} (\vec{a}_j.d - \vec{b}_j.d) t^j \right)^2 = r_a^2
\]

By \(\vec{a}.d\) is meant a \(d\)-th component of vector \(\vec{a}\). Since the metric for Euclidean space uses the second powers only the first two members of Taylor series can be used for the location function approximation. This results in at most cubic equation which can be solved using standard approach. It can be concluded, movement of objects have to be approximated by partly linear routes on which the object is moving with a constant speed, \(\vec{v} = \text{const}\).

A.5.2 Range Approximated by a Square Parallel to a Coordinate System

The range of the object is approximated by a square parallel to coordinate system which sides are \(s\) long. Again, using the dot convention for vector’s components the incidence can be described by the equation:

\[
\max_{d=0}^{n} (\vec{p}_a(t).d - \vec{p}_b(t).d) = s/2
\]

An object is entering, or leaving the square-shaped range if the maximum of component differences is equal to a half of the side. If Taylor series is applied we get:

\[
\max_{d=0}^{n} \left( \sum_{j=0}^{k} t^j \vec{a}_j.d - \sum_{j=0}^{k} t^j \vec{b}_j.d \right) = \max_{d=0}^{n} \left( \sum_{j=0}^{k} t^j (\vec{a}_j.d - \vec{b}_j.d) \right) = s/2
\]

The solving of previous equation needs more sophisticated approach then the one in previous section, but this one allows to approximate the movement by functions with members of the Taylor series up to 3rd power. This means broad set of curved routes and movements described by an initial position, an initial velocity, an initial acceleration, and linear change of the acceleration.
A.5.3 Range Approximated by a Half-Space

A half-space is a very useful structure for polyhedra construction. It is limited by a cutting plane (in 3D, or by a line in 2D case). All points separated by this plane form a half-space. The cutting-plane can be easily defined by one of its points (\( \vec{a} \)) and a normal vector (\( |\vec{n}| = 1 \)). The half-space can be described as the following set:

\[ \{ \vec{x} \in \mathbb{R}^n : \vec{n} \cdot \vec{x} - \vec{n} \cdot \vec{a} > 0 \} \]

Where the "\( \cdot \)" denotes a dot product of vectors. More complicated polyhedral and convex ranges can be constructed by the combination of half-spaces.

Suppose now, the \( \vec{a} \), the normal vector \( \vec{n} \), and a subject \( b \) are all moving, their position are defined by a location functions \( \vec{p}_a(t) \), \( \vec{p}_n(t) \), and \( \vec{p}_b(t) \), \( t \in \mathbb{R} \). Similar reasoning to the one above is used to determine which parts of the location functions' Taylor series can be used. Emergent event have to be generated for each root of the equation:

\[ \vec{p}_a(t) \circ \vec{p}_n(t) - \vec{p}_a(t) \circ \vec{p}_b(t) = 0 \]

A substitution is to be applied:

\[ \sum_{j=0}^{k} t^j \vec{n}_i \circ \sum_{j=0}^{k} t^j \vec{s}_i - \sum_{j=0}^{k} t^j \vec{b}_i \circ \sum_{j=0}^{k} t^j \vec{b}_i = 0 \]

\[ \sum_{d=0}^{n} \left( \sum_{j=0}^{k} t^j \vec{n}_i \cdot d \cdot \sum_{j=0}^{k} t^j \vec{s}_i \cdot d - \sum_{j=0}^{k} t^j \vec{n}_i \cdot d \cdot \sum_{j=0}^{k} t^j \vec{b}_i \cdot d \right) = 0 \]

It is obvious that for a \( k > 1 \) would be this equation of degree more than three. Again, this limits the approximation to the first two members of Taylor series.

A.5.4 Conclusion

Although the previous reasoning involves only the simplest object ranges, more complex ones can be obtained by their combination. The results can be summarised by the statement:

If the motion of objects is described by a piecewise linear function, the emergent event times can be computed directly, using a procedure with a constant and small complexity.

A.6 Classification of Objects

The previous reasoning tell us something about which range types and location functions can be used for an approximation in order to be able to compute easily and fast the changes of objects' incidence. This section concentrates on fast and efficient methods for the location function approximation. It can be used as a methodology for the implementation of an approximation for more special motion kinds.

It is usual to start with the most simple cases. If the object is stable, or it is moving only with a constant speed, there is no need for approximation, objects motion can be described fully. Also, objects with a piecewise linear route (the velocity vector changes only in discrete time points) are determined without deflections. As an example of such an entity should serve an object in space not influenced by any power field (molecular physics, theoretical chemistry), also billiards and so-called hard-sphere models can be implemented this way [Lub91].
More complex movements need to be approximated with some level of accuracy. Let $\varepsilon > 0$ be a small fixed positive number which describes a tolerance. According to definition of the correct algorithm the approximation is correct if the maximum distance of real and approximated position is less than the tolerance $\varepsilon$ for each time $t$.

A.6.1 Constant Acceleration

As an example of objects from the real world with a (piecewise) constant acceleration can serve space-crafts, or objects moving in the gravitation field in vacuum, or heavy enough that the resistance of air can be neglected. It can bring more realism to a billiard model (add a friction) too.

The motion with a constant acceleration can be defined by an initial position $\vec{s}$, an initial velocity $\vec{v}$, and a constant acceleration $\vec{a}$. The location function is then:

$$\vec{p}(t) = \vec{s} + \vec{v}t + \frac{1}{2}\vec{a}t^2$$  \hspace{1cm} (5)

To make the concatenation of linear segments more simple bisectors are used. Each segment starts and end in the point of the real object’s route. Let $t_0 < t_1$ be a start and end time point, respectively. Approximation will be denoted by a $\vec{q}(t)$, and it is described as a straight move from the object position in time $t_0$ to the position in time $t_1$ (see Figure 21).

![Figure 21: Motion with a constant acceleration.](image)

Approximation is therefore:

$$\vec{q}(t) = \frac{1}{t_1 - t_0} \left( \vec{p}(t_0)(t_1 - t) + \vec{p}(t_1)(t - t_0) \right)$$

After the substitution and simplifying:

$$\vec{q}(t) = \frac{1}{t_1 - t_0} \left( \vec{s}(t_1 - t + t - t_0) + \vec{v}(t_0(t_1 - t) + t_1(t - t_0)) + \frac{1}{2}\vec{a}(t_0^2(t_1 - t) + t_1^2(t - t_0)) \right)$$

$$\vec{q}(t) = \vec{s} + \vec{v}t + \frac{1}{2}\vec{a}(t_0t - t_0t_1 + t_1t)$$  \hspace{1cm} (6)

The difference between the real and approximated position should be not bigger then a given tolerance $\varepsilon$. 

77
0 ≤ |\vec{r}(t) - \vec{q}(t)| ≤ ε,  \quad t \in [t_0, t_1]

This condition constrains the end time \( t_1 \) for each fixed start time \( t_0 \). If the left side of (7) equals to the right side we get the maximum span of these times. The maximum value on the left side can be computed as a point with the first derivation equal to zero.

\[
\frac{d}{dt}|\vec{r}(t) - \vec{q}(t)| = 0
\]

\[
\frac{d}{dt}\left[ \frac{1}{2}|\vec{a}(t)|^2 - t_0 t + t_0 t_1 - t_1 t \right] = 0
\]

\[
\frac{1}{2}|\vec{a}|(2t - t_0 - t_1) = 0
\]

\[
t = \frac{1}{2}(t_0 + t_1)
\]

Hence the maximum value is:

\[
|\vec{r}_1(t_0 + t_1) - \vec{q}_1(t_0 + t_1)| = \frac{1}{8}|\vec{a}|(t_1 - t_0)^2 < \varepsilon
\]

The maximum time span for which the object's route is "covered" by linear approximation is:

\[
\Delta t = t_1 - t_0 = \sqrt{\frac{8\varepsilon}{|\vec{a}|}}
\]

This is a quite nice result, which influences the complexity of the A.A algorithm. The complexity of incidence detection of two objects with a (piecewise) constant acceleration is

\[
O(1/\Delta t) = \sqrt{\frac{|\vec{a}|}{8\varepsilon}}
\]

which means it is growing less then linearly with the precision (if the tolerance gets 4 times smaller, the number of steps doubles) and the acceleration (4 times bigger acceleration needs only twice as much steps).

### A.6.2 Limited Acceleration

Almost all objects surrounding us have a limited acceleration. According to a basic Newton law any acceleration consumes some energy. Usually objects can dispose with limited energy. It is the most apparent with vehicles and vessels. The acceleration is limited by the load of engine and mass of the vehicle (except for casualties, which can be simulated as discrete events).

It may not be the most effective way how to approximate objects movement, but sometimes it is efficient enough to use the estimated value of maximal acceleration for the nearest future and use it to determine the \( \Delta t \). On this time interval the objects movement can be approximated by the bisector as it is described in the previous Section.
A.6.3 Trajectory Defined by Points

The most naive solution is to define a linear fraction by each pair of consecutive points. For small number of definition points this can be fairly effective. If the number grows, linear parts can be constructed (as a part of preprocessing) with the following method. For simplicity we assume the fraction starts in a point \( d_0 \). The algorithm is searching a point with a higher possible index which satisfies the condition, that all intermediate points are approximated with the given tolerance \( \varepsilon \).

```plaintext
01 low - index of the start point
02 high - index of the last point
03
04 while(low <> high)
05   do {
06       mid = (low + high) / 2
07
08       if all points from 0 to mid are approximated by bisector
09       (d₀, d_mid) with the given tolerance
10       low = mid
11     else
12       high = mid
13   }
14 return low
```

Since only point knowledge is provided the fastest possible method is the binary search. Complexity of such a preprocessing is less then \( O(k \log k) \) where \( k \) is the number of points.

A.6.4 Movement Defined by an Analytic Function

To approximate motion specified by an analytic motion all methods used in analytic algebra can be used. Hence function describing a motion should be continuous, its derivatives can serve either as a acceleration limit (see A.6.2 or as condition for use of higher-order methods (regula-falsa, Newton's).

A.7 Performance Issues

The theoretical conclusions in Section A.4 state something about the complexity of incidence detection. Practical tests of performance were done to reassure it.

The number of steps was measured for each algorithm. We have implemented several object's classes with different behaviours. The Table 2 show the ratios of algorithms performance. Used classes cover all types shown in the previous section except for position specified by a function (and stable objects). There are also two types of point defined route: first case is a set of points on a straight line, the second one on a spline-like route. Only circular ranges were tested.

The performance measurement results show a good coincidence with theoretical conclusions. Approximation algorithm needs usually 20-500 times less steps than the basic FS algorithm, and 2-20 times less than the BS which is not robust.

The following section shows how to implement a discrete event simulator using the AA incidence detection algorithm. Among other features it is suggested how to allow interactive object control and linked bodies.
<table>
<thead>
<tr>
<th>Objects</th>
<th>FS : BS : AA ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant velocity</td>
<td>1 : 0.0948 : 0.0023</td>
</tr>
<tr>
<td>Constant acceleration</td>
<td>1 : 0.0980 : 0.0148</td>
</tr>
<tr>
<td>Limited acceleration</td>
<td>1 : 0.0821 : 0.0430</td>
</tr>
<tr>
<td>Points on linear path</td>
<td>1 : 0.0948 : 0.0023</td>
</tr>
<tr>
<td>Points on spline path</td>
<td>1 : 0.0663 : 0.0033</td>
</tr>
</tbody>
</table>

Table 2: Performance of FS, BS, and AA algorithms

A.8 Implementation Issues

Standard discrete event simulator can be described in terms of generating and processing events. One of its main features is existence of an event queue. Objects modelling entities can generate events, pass them into the queue, and process events received from the queue. If the simulation involves moving objects the event processing unit have to be a bit more complicated than a usual queue. Especially, if objects are allowed to change their intended routes, the structure have to be more sophisticated. In this section is shown how to implement and use a time-table (see [EW77]), and an interest-matrix ([GASF94]).

A.8.1 Definitions

Let \( n \) be number of objects involved in the simulation. Each object has an active range and an implementation of its behaviour. The later one defines objects desired motion, external transition function (the reaction to incoming events), and internal transition function ([Zei84]).

The simulator has an event queue, which stores non-emergent events and a time-table. It also keeps a simulator’s time. At the start of the simulation the time is set to an initial value and all objects are initialised. This can produce some events. During the simulation the simulator repeatedly takes out an event scheduled to the nearest time point to time from both the event queue and the time-table. The time is set to the event’s one and the event is passed to the appropriate object for a processing.

The time-table is a structure which stores expected emergent events. Unless the events time is equal to simulator’s time we cannot be sure the event will happen. Unlike Lubachevsky in [Lub91] we suppose not each emergent event changes object’s intention. Some of them can just register a “friendly” object etc. The reason why to keep all expected emergent events is there is a high probability of successful prediction. However each object can as a product of some event handling change its intentions, which has to be taken in mind while implementing the time-table.

Using hierarchical models ([Sev96b]) allows existence of linked objects. This is a group of objects which models one bigger entity. It is usually unnecessary to check for incidence among objects from one group. The incidence matrix (see [GASF94]) carries information which pair of objects have to be checked for incidence.

A.8.2 Time-table

The implementation of the time-table has to fulfil the following criteria:

- It stores expire times of approximation validity.
- It stores entering and leaving emergent event times.
- Fast row and column update.
- Fast minimum search among all time values.

The first two criteria are obvious. The third and the last ones are essential for the implementation of object’s reaction.

The time-table can be implemented as follows: there is a row in the table for each (atomic) object. Each row contains n + 2 columns, one for each object, one for expire information and the last one contains a row’s minimal value (see Table 3 for an example, n = 4).

<table>
<thead>
<tr>
<th>Observer's number</th>
<th>Approximation expire</th>
<th>1 out</th>
<th>2 in</th>
<th>2 out</th>
<th>3 in</th>
<th>3 out</th>
<th>4 in</th>
<th>4 out</th>
<th>min</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
</tr>
<tr>
<td>2</td>
<td>234</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>423</td>
<td>037</td>
<td>096</td>
<td>037</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>432</td>
<td>213</td>
<td>572</td>
<td>100</td>
<td>356</td>
<td>inf</td>
<td>inf</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>024</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>354</td>
<td>987</td>
<td>381</td>
<td>024</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Time table

In our example the object 1 is not going to receive an emergent event in the nearest future. The inf value in its expire column suggests it is stable or moving with a constant speed. Object 1 is going to enter the active range of the object 3 in time ‘213’ and to leave it in time ‘572’. This is possible because of the bigger range of the object 3. Object 3 is already observed by the object 2, and is going probably to leave its range in time ‘423’. All this information can be changed by an events, which can alter object’s behaviour. The first one is planed to the time ‘024’, which is the minimum of min column.

At the start of the simulation all cells in the table are set to “inf”, which means the event will not ever happen. During the initialisation each object is asked to provide the simulator with its intended route. This fills the first column as well. The last step in the time-table’s initialisation is the computation of emergent event times for all pairs regarding the incidence matrix. The appropriate min value has to be re-considered with each cell’s update.

According to the definition of discrete event simulator the event with the minimal time value have to be processed each time. Introducing the min column the searching of the next event has complexity O(n) instead of O(n²). The min values have to be kept consistent, which is rather intricate in the case of removing an event from the table. However this operation have complexity O(n) too.

Event processing can result into the change of object’s intentions. In this case the object sends its new intended route to the simulator and the appropriate row and column have to be updated. This can alter the information in cells and again min column have to be updated.

The time-table have to implement the following functions:

- Update object’s route - updates the adequate row and column, and the min column.
- Get the minimal time - returns minimum of all values stored in min fields. It does not change the state of the time-table.
- Get the event with the minimal time - constructs event according to the minimal time value, sets value of time to the inf, update the min values and returns the event.
- Add an object.
• Remove an object. The implementation of this two methods is critical for distributed time-tables covering the overlapping areas of simulated space.

A.8.3 Interest Matrix

According to [GASF94] the interest matrix has a flag for each pair of objects which states the necessity of an incidence check. Since the link relation is reflexive and transitive each object can belong to at most one group of linked object. This allows a simple and effective implementation of the interest matrix.

Each group of objects is denoted by a unique number. While creating a new object its group number is set to a new, "unused" one. This number can be reset with an object's method when needed. Finally, the simulator checks for the incidence of two objects only if the objects have a different group numbers.

A.8.4 Object’s Methods

Each object have to implement at least the following set of methods.

• Initialise - prepares the object for the start of the simulation. It can be used for a temporary structure allocation, the resetting of object's user interface, a tolerance setting, etc.

• Get Intended Route - gets proposed approximation of the object’s location function.

• Event handler - implements object's reactions to incoming events. The handling can result in a change of object’s intended route and a call of the update of the time-table.

• Run - implements an object's behaviour. It can just wait for the incoming events to process them, perform a procedure or do some actions periodically.

• Set group number.

Additionally, each object can keep a list of subjects it is able to ‘see’. This option allows the object to keep track of all subjects in its range and trigger some action depending on number or type of the subjects. The list is updated by a emergent event handler.

A.8.5 Simulation

The simulator’s main function is to keep and handle events. There are two structures for event storing: the event queue, and the time-table. Whereas the event queue works with the “ordinary” events scheduled by objects, the time-table produces emergent events (notifying the entering and leaving a range, the need for a new approximation). The simulator have to choose the event with a minimal time from both structures and pass it to appropriate object’s event handler.

The simulator works according to this algorithm:

```c
/** Pseudo-code for Simulation */
01 The simulator possesses:
02   event_queue
03   time_table
04   time, old_time
05
06 Function Simulate is {
07```
call initialise of all objects

initialise the time_table

/** simulation loop */

while (time_table.get_the_minimal_time <> inf or
        there are some events in event_queue) {

    if(time_table.get_the_minimal_time
        < event_queue.first_event_time)
        then get the event with the minimal time from time_table
    else get the event from the head of event_queue

    old_time = time  /** clock can advance */
    time = event.time
    animation from old_time to time

    dispatch the event

}  

/** End of pseudo-code for Simulation */

The animation method have to allow an interrupt by the events created by user
to allow interaction. The other way is to treat all animation objects as a part of
the simulation and let them generate and handle special update events. This might
be a more sophisticated way, by also the more computing-expensive one.

A.8.6 Certain Aspects of this Approach

Simulation described in the previous section differs from an ordinary discrete event
simulation:

• It needs a special implementation of event queue, which was solved by the
adding of the time-table.

• Emergent event processing is interleaved with other events because the event
with a minimal time from both event queue and the time-table is to be han-
dled.

• The time-table is always consistent with first collision(s) of objects. Actual
changes of route as a respond to (emergent) event update this table.

• The splitting and merging of emergent events can be provided by the appro-
priate object's event handlers, which can update the information in the list of
visible subjects.

A.9 Application Fields

Approach described in this paper can be classified as a heterogeneous simulation.
Using the discrete event simulation it can model continuous behaviour of moving and
interacting objects. However our research concentrates on naval combat simulation
and modelling, the approach has a number of other application fields.

We believe it can be useful in virtual reality systems, games, combat simulators,
traffic models and robotics. Simple object's ranges were used to make the article
more readable, algorithm modification for large number of objects was omitted for
the same reason. We suggest to use sphere trees and uniform space division if
needed. Some helpful ideas and valuable references can be found in [GASF94] and
[Lub91].
A.10 Conclusion

This article derives conditions for object's route description for the easy and accurate computation of emergent events. This conditions are beard in mind approximating object's route by a piecewise linear motion with a given tolerance. It is shown, that using this approximation the complexity of detection algorithm is generally better than the one of static approach. From the practical point of view the construction of heterogeneous simulator is described.
B Server - Directory Structure & Scripts

B.1 URL conversions and Directories

The following table shows relationship between URL as seen in the client and the directory on the server:

<table>
<thead>
<tr>
<th>URL translation on the host server:</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL:</td>
</tr>
<tr>
<td>Location:</td>
</tr>
<tr>
<td>$HOME/lib/html/</td>
</tr>
</tbody>
</table>

Main page of the project:

<table>
<thead>
<tr>
<th>URL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location:</td>
</tr>
<tr>
<td>~hejda/lib/html/sim/</td>
</tr>
</tbody>
</table>

The following table shows the directories and files related to the VISP project (a star after a name denotes the file is listed later, the @-character denotes link):

- hejda/lib/html/
  - cgi-bin/
    - check password protection utility
    - image.cgi* generates the database of icons
    - imageBrief.cgi shorter version of image.cgi
    - javac.cgi* Java compiler CGI interface
    - log current log file
    - log.cgi script generating log file
    - saveFile save data utility
    - saveSecure.cgi* save data CGI interface
  - sim/
    - check password protection utility
    - doc root of the entity database
    - doc.cgi* generator of HTML pages
    - feedback body of the feedback page
    - footer body of the footer
    - index body of the main page
    - index.html static version of the main page
    - inf.cgi* generator of the database of scenarios
    - lib.cgi* generator of the database of entities
    - log current log file
    - log.cgi script generating log file
    - noCompile.gif images used in tutorial
    - noCompileNoExport.gif
    - noExport.gif
    - param.gif
    - rangeA.gif
    - rangeB.gif
    - setupd* set password utility
    - tutorial body of the main page of tutorial
    - tutorial12 body of parts of tutorial
    - tutorial13
    - tutorial14
    - tutorial15
    - tutorial16
    - tutorialJDK
    - tutorialJava

85
- data/   root of the scenario database
- docs/   contains some document pages
- images/ contains icons
- java@   link to the $HOME/java/java/lib/
          root of the Java files, source ...
- logs/
- src/

The structure of classes and files related to the part of the project implemented in Java are described in Appendix C. The rest of this section lists CGI scripts and source codes of utilities.
B.2 cgi-bin/check.c

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <ctype.h>

void usage() {
    printf("check dir [password] \n\n");
    printf(" Checks if the directory is protected by a password.\n");    
    printf(" An empty .p file indicates the directory is public.\n");    
    printf(" Otherwise, the data from .p file are compared with\n");    
    printf(" the password. The exit code is 0 if the directory\n");    
    printf(" is unprotected or the password matches, <>0 otherwise\n");    
    exit(-1);
}

void cut(char *pass) {
    int i;

    while(isalnum(pass[i])) i++;
    pass[i] = '\0';
}

void main(int argc, char *argv[]) {
    FILE *file;
    char *word;        /* claimed password */
    char pass[256];    /* password from the .p file */
    char dir[256];

    switch(argc) {
        case 1: usage(); exit(-1);
        case 2: strcat(dir, argv[1]);
            word = "";
            break;
        case 3: strcat(dir, argv[1]);
            word = argv[2];
            break;
           
        default: usage(); exit(-1);
    }

    strcat(dir, "/.p");

    if((file = fopen(dir, "r")) == NULL) {
        printf("ERROR: Wrong directory or missing protection file.\n");
        exit(-2);
    }

    pass[0] = '\0';
    fread(pass, 256, 1, file);
    cut(pass);

    if(strlen(pass) == 0) {
        exit(0); /* unprotected directory */
if(strcmp(pass,word) == 0) {
    exit(0);    /* password matches */
} else {
    printf("ERROR: wrong password.\n");
    exit(-3);
}

B.3  cgi-bin/image.cgi

#!/bin/sh

PATH=/usr/sbin:/usr/bin

BASENAME="../$PATH_INFO"
NAME=$PATH_INFO

  cat << EOM
Content-type: text/html

  <HTML><HEAD><TITLE>Images in $SERVER$PATH_INFO</TITLE></HEAD>
  <BODY BGCOLOR='#ffffff' TEXT='#000000' LINK='#0000ff' VLINK='#000000' ALINK='#ff0000'>

EOM

  echo "<H1>Directory: $SERVER$PATH_INFO"
  echo "</H1><HR>

  echo "<P>To use an image copy its location to the dialog window."
  echo "You can also use a <A HREF="/hejda/cgi-bin/imageBrief.cgi $PATH_INFO">compressed list</A>.
  echo ""

  ALLNAMES='ls -l $BASENAME*.gif'

  echo "<PRE>"
  for EACH in $ALLNAMES
  do
    echo "<A HREF="$SERVER$PATH_INFO$EACH>
      <IMG HEIGHT=40 WIDTH=40 SRC="$SERVER$PATH_INFO$EACH>
    </A> $SERVER$PATH_INFO$EACH"
  done
  echo "</PRE>"

  cat << EOM

  <HR>
  <P align=center>
  Go to the
  <A HREF="http://www.cs.su.oz.au/~hejda">Hejda's page</A>,
  <A HREF="/hejda/sim/">VISP</A>, or to the
  <A HREF="#TOP">top</A>.</P>

</BODY></HTML>

EOM
B.4  cgi-bin/javac.cgi

#!/bin/sh
# Compile file named in QUERY_STRING with javac compiler.
# The file has to be saved in the right place.
# See saveJava.cgi for saving.

PATH=/usr/sbin:/usr/bin
JAVAHOME=.:/java

echo Content-type: text/plain

echo

/usr/bin/rm -f $JAVAHOME/lib/$QUERY_STRING.class

$JAVAHOME/bin/javac -d $JAVAHOME/lib -classpath "$JAVAHOME/lib/classes.zip:$JAVAHOME/lib" $JAVAHOME/lib/$QUERY_STRING.java

echo Compilation finished
B.5  cgi-bin/saveFile.c

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

/*
  * Simple filter, which saves data from stdin to file with a name
  * argv[1]. In argv[2] have to be the length of the data. If not,
  * filter saves up to the end of the data.
  * It prints a "OK, data saved.\n" and "ERROR on server, data
  * not saved.\n", respective as a result.
  * This filter is intended to save POSTed data from application.
  * the params are prepared by saveFile.cgi.
  */

char *okMsg = "OK, data saved.\n";
char *errMsg = "ERROR on server, data not saved."

void error(char *str) {
    printf("%s (%s).\n", errMsg, str);
    exit(-2);
}

void usage() {
    printf("saveFile [filename] <length of the data>\n\n");
    printf(" Saves data from stdin to the specified file.\n");
    printf(" If length is specified, it limits the number of read\n");
    printf(" characters. Otherwise it reads until EOF is reached.\n");
    exit(-1);
}

main(int argc, char *argv[]) {

    FILE *file;
    int ch;
    int ch;

    switch(argc) {
    case 1: usage();
    case 2: cl = -1;
        break;
    case 3: cl = atoi(argv[2]);
        if(cl <= 0) error("Bad file lenght");
        break;
    default: error("Wrong number of arguments");
    }

    file = fopen(argv[1], "wb");
    if(file == NULL) {
        error(argv[1]);
        perror("ERROR");
    }

    if(cl>0) {
        if(cl > 17000) cl = 17000;
        if(cl > 0) {
            int i;
            ch = fgetc(stdin);
            while ((ch = fgetc(stdin)) != EOF)
                fwrite(&ch, 1, 1, file);
        }
    }
    else
        while ((ch = fgetc(stdin)) != EOF)
            fwrite(&ch, 1, 1, file);

    fclose(file);
    exit(0);
}
while(cl && (EOF != (ch = getchar()))) {
   putc(ch, file);
   cl--;
}
fclose(file);

printf(okMsg);
B.6  cgi-bin/saveSecure.cgi

#!/bin/sh
# Saves posted data to a specified file relative to the document
# base. The query string contains authorisation. It have to
# match the string in the .p file in the directory the data is to
# be saved. If there is no such a file, or the passwords does not
# match, it returns error.

PATH=/usr/sbin:/usr/bin

./log.cgi

echo Content-type: text/plain
echo

FILE_NAME=..$PATH_INFO
DIR='dirname $FILE_NAME'
FILE='basename $FILE_NAME'

if [ $FILE = ".p" ]
then {
    echo ERROR, cannot rewrite the specified file
    exit
}
fi

if RES='./check $DIR/ $QUERY_STRING'
then {
    ./saveFile $FILE_NAME $CONTENT_LENGTH <&0
}
else {
    echo $RES
}
fi
B.7 sim/doc.cgi

#!/bin/sh

PATH=/usr/sbin:/usr/bin

./log.cgi

# Simple wrapper which inserts HTML body specified by #PATH_INFO
# into a template.

BASENAME="$.PATH_INFO"
NAME=$PATH_INFO

cat << EOM
Content-type: text/html

<HTML><HEAD>
    <TITLE>VISP - $NAME</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffff"
    TEXT= 
    "#000000"
    LINK= "#000000"
    VLINK="#000000"
    ALINK="#ff0000">
EOM

if [ ! -r $BASENAME ]
then
    cat "$BASENAME"
else {
    echo "Error: $BASENAME not found. try footer to bring you back."
}
fi

cat footer
echo </BODY></HTML>
B.8 sim/inf.cgi

#!/bin/sh

PATH=/usr/sbin:/usr/bin

./log.cgi

BASE=\$.PATH_INFO

cat << EOM
Content-type: text/html

<HTML><HEAD><TITLE>List of directory $PATH_INFO</TITLE></HEAD>
<BODY BGCOLOR="ffffff"
   TEXT= "#000000"
   LINK= "#000000"
   VLINK="#000000"
   ALINK="#000000">
EOM

if [ ! -d "$BASE" ]
then {
   cat << EOM
   <H1>ERROR: Directory $PATH_INFO not found on this server</H1>
   <P>Explanation: The specified directory does not exist on this server. Check for the name and the spelling, or start from the root of the database</P>
EOM
   cat footer
   exit 0
}
fi

if !./check $BASE > /dev/null
then {
   echo "<IMG SRC="hejda/images/unlock.gif ALIGN=right>"
}
else {
   echo "<IMG SRC="hejda/images/lock.gif ALIGN=right>"
}
fi

echo "<H1>Content of $PATH_INFO</H1>"
echo "<P><UL>

cat $BASE.inf

echo "</UL><HR>"

ALLNAMES='ls -1 $BASE'"
echo "<TABLE>

for EACH in $ALLNAMES
do {
    if [ -f "$BASE/$EACH.inf" ]
    then {
        echo "<TR><TD>"
        if [ -d "$BASE/$EACH" ]
        then {
            echo "<A HREF="/hejda/sim/inf.cgi?PATH_INFO/$EACH>"
            echo $EACH
            echo "</A>"
        }
        else {
            echo $EACH
        }
    fi
    echo "</TD>"
    cat $BASE/$EACH.inf
}
fi
do

echo "</TABLE>

echo "<CENTER>"

echo "Back to the <A HREF="/hejda/sim/inf.cgi/sim/data">root.</A>"

cat footer
B.9  sim/lib.cgi

#!/bin/sh
# - the v8 is supposed to be more secure, but does not work for me.
## /v8/bin/sh -p

PATH=/usr/sbin:/usr/bin

./log.cgi

NAME='echo $PATH_INFO | tail +2c '
FILE=./java\'echo $PATH_INFO | tr ":" "/"\'
SHORT='basename $FILE'

cat << EOM
Content-type: text/html

<HTML><HEAD><TITLE>Database of classes ... $NAME</TITLE></HEAD>
<BODY bgcolor="#ffffff"
  TEXT= "#000000"
  LINK= "#a00000"
  VLINK=="#000000"
  ALINK=="#ff0000">

EOM

#############  If there is no name specified, show the doc page.
if [ -z "$NAME" ]
then {
  cat doc
cat footer
  exit 0
}
fi

#############  The object is a class
if [ -r "$FILE.author" ]
then {
  echo "<IMG SRC="
cat $FILE.pict
echo "  ALIGN=right HEIGHT=40 WIDTH=40>"

echo "<A name="#TOP">\<H1>Class name: $NAME</H1></A><UL>"

cat $FILE.info
echo "</UL><HR>"

echo "<TABLE><TR>
echo "<TD><P><STRONG>Author: </STRONG><TD>"
cat $FILE.author
echo "<TR><TD><P><STRONG>Last modified: </STRONG><TD>"
ls -l $FILE.java | awk '{print $6 " "$7 " "$8 ;}'}
echo "<TR><TD><P><STRONG>Ready to use: </STRONG><TD>"
if [ -r $FILE.class ]
  then {
elif [ -d $FILE ]
then {

    if ! [ -e $FILE ]
    then {
        echo "<P><STRONG>Description: </STRONG>
        cat $FILE.doc
        echo "<P><STRONG>Code: </STRONG>"
        echo "<PRE>
        cat $FILE.handler
        echo "</PRE>
    }

    fi

    echo "<H1>Package: $NAME </H1><HR>

    ALLNAMES='ls -l $FILE | awk -F'./' '{if ($2 == "doc") print $1}'

    echo "<TABLE>"
    for EACH in $ALLNAMES
    do {
        echo "<TR>"
        echo "<TD><IMG HEIGHT=40 WIDTH=40 SRC="
        cat $FILE/$EACH.pict
        echo ">
        echo "<TD>
        echo "<A HREF="
        echo $NAME.$EACH
        echo ">
        echo $NAME.$EACH
        echo "</A>
        echo "<TD>
        cat $FILE/$EACH.info
    done
    echo "</TABLE>"
}

else {

}
The specified package or does not exist

```
#
cat << EOM

<H1>ERROR: Class or package</H1>
<UL><B>"$NAME"</B></UL> not found on this server

<P>Explanation: The specified object or package does not exist
on this server. Check for the name and the spelling. To
enter the database of classes follow this

line</A>.

EOM
```

```
B.10 sim/setpwd

#!/bin/sh

# Usage: setpwd dir [passwd]
# Create a .p file in the specified directory and writes the
# passwd into it. If the password is omitted, the file is
# empty, which means no protection.

if [ $# = 1 ]
then
    echo > $1/.p
    chmod 600 $1/.p
    ls -la $1/.p
    echo Directory $1 is open.

elif [ $# = 2 ]
then
    echo $2 > $1/.p
    chmod 600 $1/.p
    ls -la $1/.p
    echo Directory $1 is protected by a password.

else
    echo Usage: setpwd dir [password]
    echo sets password of the directory.
fi
C  Java Packages and Classes Hierarchy

C.1 Package Structure

All Java related data are currently on this location:

directory tree: `hejda/java/java/
computer: staff.cs.usyd.edu.au.
It structure is as follows:

apibook/     - documentation of Java 1.0.1 API
bin/         - binary executables of Java compiler and utilities
doc/         - online documentation (see packages.html)
include/     - part of the JDK
lib/         - root of the database of classes (*.class files)
src/         - root of source files
src.zip      - compressed source files of the java.* packages

The src/ directory contains all source codes related to the project. The star (*) indicates the file contains interface description, and is listed in section C.2. The directory contains these files and subdirectories:

basser/       --- name of the organization
basser/sim/    --- packages dedicated to VISP only
basser/sim/comp/  --- generic mechanism, components

DocFrame.java
* EditComp.java
* Generable.java
  GenericComp.java
  GenericHandler.java
  GenericMoving.java
  GenericMovingTemplate.java
  GenericParamedit.java
  GenericRange.java
  GenericSimulable.java
  GenericSimulableTemplate.java
  GenericSteering.java
Line.java
Navigate.java
NavigationFrame.java
Parab.java
Randomized.java
* SimComp.java
  Spline.java
  Steering.java
  TimedSteering.java

basser/sim/core/ --- special entities for user interface
Sample.java
Simple.java
SimpleRange.java
ViewFrame.java
Viewport.java

basser/sim/gui/  --- user interface classes
* Editable.java
  EditableIcon.java
  EditableList.java
basser/sim/lang/ --- classes implementing simulator and timetable

Environment.java
IconButton.java
Params.java
Simple.java
SimulationUI.java
WorkMap.java

CircularRange.java
Interval.java

* Moving.java
Range.java
RectangularRange.java
Route2D.java
SimEvent.java
SimEventQueue.java

* Simulable.java
SimulableList.java
Simulation.java
Timetable.java

basser/util/ --- classes used in the project, but not dedicated

CanvasFrame.java
Chart.java
CloseFrame.java
CompFrame.java
Histogram.java
MsgDialog.java
Point2D.java
TextFrame.java
Util.java

basser/util/net/ --- communication with server, I/O operations

ClassFrame.java
Net.java
NetDialog.java
NetFrame.java
NetLoader.java

* Serializable.java
TestApplet.java
TopLoader.java

basser/util/palette: ClassButton.java
ClassHolder.java
ClassPalette.java
Holder.java
IconButton.java
Palette.java

basser/util/paramedit/ --- parameters, parameter editor

ColorButton.java
ColorField.java
ColorPalette.java
ColorParam.java
DoubleParam.java
ImageField.java
ImageParam.java
IntParam.java
ParamDialog.java
ParamEdit.java
Parameter.java
StringDialog.java
StringParam.java

basser/util/workspace/ --- 2D workplace
FreeLayout.java
Grip.java
Icon.java
NoPeer.java
Test.java
Workspace.java

R.I.P/ - older versions, test classes
...

java/ - source code of default java.* packages
...

sun/ - source code of default sun.* packages
...
C.2 Interfaces

This section contains listings of all interfaces used in the design of VISP Applet.

C.2.1 basser/sim/comp/EditComp.java

package basser.sim.comp;

import java.awt.Component;
import basser.sim.lang.*;
import basser.sim.gui.*;

/**
 * This interface specifies methods which have to be implemented
 * by components of Editable object.
 * @author <A HREF="http://www.cs.su.oz.au/~hejda/">Petr Hejda</A>
 * @version 1.0  Feb 26 1997
 */

public interface EditComp {

 /**
 * Substitues the constructor. The usage of <PRE>Class.forName()</PRE>
 * does not allow a constructor with parameters. This method has to be
 * called instead.
 * Another methods which can be used are Editable.getIconPoint(),
 * Editable.getParam().
 */
  public void create(Editable parent, Environment env);

  /**
   * The Component was deleted. It have to free all components, parameters
   * and used resources.
   */
  public void delete();

  /**
   * Produces and returns a SimComp object. This method allows to
   * separate edition and simulation.
   */
  public SimComp produce();

  /**
   * Creates the editor(s) for edition of object's properties.
   * Parameters are usually edited separately.
   */
  public Component edit();

  /**
   * Returns a description of component capabilities. The parameter name
   * is the name of the object. It should look like this:
   * <P><PRE>info("boo")</PRE>; produces:
   *   boo.create(Editable, Environment);
   */
boo.delete();
boo.edit();
</PRE>
*/
public String info(String name);
C.2.2 basser/sim/comp/Generable.java

package basser.sim.comp;

import basser.sim.lang.Simulable;

/**
 * This interface have to be implemented by the each object
 * generated from users data description. It allows to pass
 * SimComp objects from creator to the created Simulable.
 * This cannot be done using a template, java recognises between
 * classes with the same name but loaded by different ClassLoader.
 */

public interface Generable extends Simulable {

 /**
  * Provides the Simulable with a set of components.
  */
  public void setComps(SimComp[] comps);

 /**
  * This method have to be generated by Editable and it have to
  * initialize parameter fields etc...
  * @see GenericSimulable#translate
  */
  public void genericInit();
}
C.2.3  basser/sim/comp/SimComp.java

package basser.sim.comp;

import basser.sim.lang.*;

/**
 * This interface specifies methods which can be used by generic
 * Simulable object in the simulation.
 */

public interface SimComp {

/**
 * Called from the Simulable.init() to initialize the component.
 */
    public void init(Simulable parent, Simulation s, double time);

/**
 * The object was removed from the simulation. This method have to end
 * the functionality of the component.
 */
    public void destroy();
}

C.2.4 basser/sim/gui/Editable.java

package basser.sim.gui;

import java.awt.*;
import java.io.*;
import basser.util.*;
import basser.util.net.*;
import basser.sim.lang.*;

/**
 * This interface have to be implemented by each object, used by
 * Simulation builder as a component. It defines methods for
 * producing an Simulable object.
 * @author <A HREF="http://www.cs.su.oz.au/~hejda/">Petr Hejda</A>
 * @version 1.0  Feb 26 1997
 */

public interface Editable extends Serializable {

/**
 * Substitues the constructor. The usage of <PRE>Class.forName()</PRE>
 * does not allow a constructor with parameters. This method have to be
 * called instead.
 * @param x, y coordinates of the clicked place
 * @param img name of the image used in palette.
 */
  public void create(Environment env, int x, int y, String img);

/**
 * This Editable was deleted by environment. This method have to free
 * all components and used resources.
 */
  public void delete();

/**
 * Produces and returns a Simulable object. This method allows to
 * separate editation and simulation. All components have to be cloned,
 * bindings to other Simulables have to be set by init method.
 * @see Simulable#init
 */
  public Simulable produce();

/**
 * Returns last Simulable generated by this Editable. Can be used by
 * Simulable.init() method to make connections between Simulables.
 * @see Simulable#init
 */
  // public Simulable lastSimulable();

/**
 * Paints the Editable and its components. The coordinates shows

108
* the area inside the Editables Icon. It can be painted outside too.
*/
    public void paint(Graphics g, EditableIcon icon) 

/**
 * The icon was activated. Can be used to activate components, show 
 * grips.
 */
    public void activate();

/**
 * The icon was deactivated. It have to undo all that was done by activate.
 */
    public void deactivate();

/**
 * Updates state and appearance of the object. This method is called 
 * by source object, which caused a change of a property.
 */
    public void update();

/**
 * Recalls editor(s) for editation of object's properties. Called by 
 * the icon.
 */
    public void edit();

/**
 * Each Editable has got an icon.
 */
    public Point2D getIconPoint();
//    public EditableIcon getIcon();

/**
 * Each Editable has got a Parameditor.
 */
    public Params getParams();

/**
 * Each editable remembers its class name.
 */
    public String setClassName(String s); 
    public String getClassName();
}
C.2.5  basser/sim/lang/Moving.java

package basser.sim.lang;

import java.awt.Graphics;
import basser.util.Point2D;

/**
 * This is an interface which have to be implemented by all objects
 * which have some a range. Methods addRanges() and
 * intendedRoute() are to be overloaded by subclasses.
 */

public interface Moving extends Simulable {

/**
 * Keeps the last used group number. Can be used to generate unique
 * group number for each Moving object. Ranges in the same group do
 * not check each other position.
 * @see Range
 */
  public static int groupCounter = 0;

/**
 * Returns all ranges which belong to the Moving object. This are
 * added or removed to/from Timetable. The number of Ranges cannot
 * vary during the simulation.
 */
  public Range[] range();

/**
 * Returns proposed route of the object. To be overloaded.
 * The route's time span have to contain the \texttt{time}.
 * If the
 * simulator runs correct it can be relied upon each \texttt{time} value
 * is not lesser than the one in any previous call during
 * the simulation.
 * @see Simulator
 */
  public Route2D intendedRoute(double time);

/**************************************************************************/
* Inspection functions.
 */

/**
 * Paints the Moving object as it looks in \texttt{time}.
 */
  public void paint(Graphics g, double time);

/**
 * Returns the position of the object in \texttt{time}. This information
 * together with the Simulables name can be used to display and inspect
 * behaviour of Moving objects.
 */
*/

public Point2D point(double time) {
}

package basser.sim.lang;

/**
 * This interface have to be implemented by the each object involved
 * in the simulation. The connection with a Simulator is established
 * in the init(Simulation, double) method. Each Simulable can receive
 * SimEvents; method eventHandler have to implement internal and external
 * behaviour. The destroy method ends the lifecycle of the Simulable.
 */
public interface Simulable {

/**
 * Initialize the object for the simulation. It is intended to make
 * binding between Simulable objects.
 * @see Editable#lastSimulable
 */
public void init(Simulation s, double time);

/**
 * Event handler.
 */
public void eventHandler(SimEvent ev);

/**
 * Ends the validity of the object. It have to dispose all components,
 * remove object from all lists and the timetable and prohibit Simulable to
 * react and send events. This method have not to be called directly.
 * If the Simulable decides to destroy itself, it have to call
 * Simulation.destroy(Simulable) method.
 * @see Simulation#destroy
 */
public void destroy();

/**
 * Returns identification name of the Simulable.
 */
public String name();

/**
 * Returns simulation object. It can be used after (or during)
 * initialization to read public members of the simulation such as
 * tolerance, current time.
 */
public Simulation simulation();
}
C.2.7  basser/util/net/Serializable.java

package basser.util.net;

import java.io.*;

/**
 * This interface have to be implemented by all objects which
 * uses NetFrame to save and load their state.
 */
public interface Serializable {

/**
 * Object describe itself by a String and returns it.
 */
 public void save(DataOutputStream out) throws IOException;

/**
 * Object reads the data and restores its state.
 */
 public void load(DataInputStream in) throws IOException;

 public String getClassName();
}

D  Earlier Publications
AI, Simulation and Planning in High Autonomy Systems

AIS '96

March 23-27, 1996
La Jolla, California, USA

Sponsored by:
The University of Arizona
The San Diego Supercomputer Center
The Encyclopedia of Life Support Systems

In Conjunction with
The International Society for Intelligent Automation
Rand Corporation
VISP-MW: A Visual and Interactive Modelling and Simulation Environment for Naval Mine Warfare

Suleyman Sevinc
Basser Department of Computer Science
University of Sydney, NSW 2006
Australia

E-mail: sevinc@cs.su.oz.au

ABSTRACT

VISP-MW is a visual and interactive simulation package for simulating naval mine warfare. The research described in this article has been conducted at the University of Sydney in collaboration with Maritime Operations Division of Aeronautical and Maritime Research Laboratory in Melbourne.

VISP-MW is a specialized enhancement of its more generic version VISP. The emphasis of the research is to provide a visual and interactive environment supporting object-orientation using which models for entities involved in mine warfare can be developed. The objective is to be able to produce a wide variety of models by on-screen mixing and matching assets and sensors.

The environment is designed to allow modellers to focus on their models rather than idiosyncrasies of the model development environment. Microsoft Visual Basic (VB) was selected to be extended to be the environment because of its features such as its good development environment and compatibility with other modern software applications.

BACKGROUND

Naval mine warfare modelling typically addresses modelling of one or more areas contaminated by mines. Mines are usually placed according to a pattern to disrupt the sea traffic in those areas. In addition, mines with an empty shell are also placed to further complicate the search and the cleaning process. (Lawson and Richardson 1994) describes mine sweeping as a game in which the challenge to a player is to design a sweeping strategy for better results. (Sevino 1994) provides some background information on mine warfare model structures.

Mine warfare involves many cooperating entities. These entities include ships of various types, divers, remotely operated vehicles used for passivating mines, mines, etc. A typical scenario consists of a strategy which the components at the lowest level execute. Middle layers of mine warfare modelling consists of communication and cooperation between component models to realize a strategy.

THE VISP ENVIRONMENT

We recommend our audience to get a good grasp of VB programming model in order to comprehend the Visual and Interactive Simulation Pack (VISP) environment, fully (see Microsoft Press, 1994). Important aspects of the VISP environment include object model, object interaction model and model building (Sevino and others, 1995).
Figure 1 displays essential parts of every VISP object. In general, there are three kinds of VISP objects; model objects, report objects and simulator objects. Model objects are those that can take part in models of mine warfare. Ship objects, mine objects, diver objects are examples of model objects. Report objects are related to visualization or reporting aspects of simulation studies. Report objects do not participate directly in models. They are primarily used for report generation and visual effects. Animation boxes, text boxes, command buttons are examples for report objects. In general, it would not be wrong to say that a report object is a pure VB object, i.e., a standalone object which does not model a real entity. Simulator objects are used to manage model objects and their interaction. Simulator objects also provide a library of facilities which make model objects' tasks easier.

All three types of objects have a similar structure as shown in Figure 1. Each object has a set of properties, a set of events and a set of methods. In addition, although not shown in the figure, each object has an icon for its visual representation. A property may be related to visual appearance of an object such as height of its icon, backcolor of its icon, etc., or it may be related to simulation parameters of the object such as the time of last event, the time of next event, etc., for the object. Some properties are accessible during initial visual building of a model and some are only accessible during simulation runs and from within simulation code.

Each object also has a set of events that it can handle. To handle an event, code must be provided for the appropriate event procedure. Graphical editing and debugging of event procedures are provided by the VB environment. Some events are related to visual effects or interaction with the end users of models. For example a click event may be used to query the state of a model object or it may be used to set simulation parameters. The VISP environment supports fully interactive simulations using this approach. Other events are related to modelling reality. For example, each mine object has an event procedure Detect_Signal (x : Single) which is presented by signals from its environment. Modellers must provide code to decide what to do with the information presented. This depends on the type of the mine. A simple example will be presented in the later sections of this article.

An object also has a set of methods. Methods are like events except that each method has a pre-specified behavior which can not be changed by modellers. For example a passivate method is used by objects to detach themselves from a simulator in case they are destroyed by an explosion during the course of a simulation study.

Almost the first thing to do in a simulation study is to specify each and every object involved. Specification of an object involves setting its properties, fixing up its event routines and making sure that the object operates accurately in relationship to its real counterpart.

Object Interaction Model

Object interaction is central to object independence a concept which is an important design decision for the VISP
environment. VISP supports two kinds of object interaction. In one, two objects come in contact with one another either permanently or temporarily. In this case the source and the destination objects are known. Objects are allowed to make such a connection and terminate it as they wish. We call this coupling of two objects. Coupling of objects is managed through use of input and output ports (see Figure 2). This is done to preserve independence of objects so that objects can use the same port to couple themselves to many other objects either at the same time or at different times.

Figure 2. Coupling of Objects

The second kind of object interaction can not be predicted in advance. For example when a ship enters a mine field without being aware of the fact that there are mines around is such an event. We call this event an emergent event. Discrete event simulators are designed to handle events that are scheduled by a component. Emergent events are not scheduled by any component (or object) in the system. Therefore they can not be handled unless modifications are made to the simulator.

Handling of Emergent Events

The VISP environment models each model object as a point in space with a circle around it (Sevink and Bagde, 1995). The circle represents the effective area in which the object can see and interact with other objects. The radius of the circle depends on the particular object but also on the current environmental conditions, e.g., a mine can detect objects so far away but this distance may get shorter due to bad weather. For simplicity we assume that the effective area radius remains the same throughout simulation for each object. Figure 3 shows such an object moving.

Figure 3. The Object Model

As it is demonstrated in Figure 3, route of an object can be approximated by a series of lines. Given a particular route, the area which is visible to the object throughout that route can be easily computed using the effective area radius for the object and the series of lines used to represent the route.

Each object must inform its simulator with its intention of moving before it does so. Simulator maintains a map which shows routes of objects and it can determine from that information whether there is a potential collision between any of its objects. Figure 4 presents a potential collision situation.

Figure 4. Potential Collision.

Once the simulator determines that there is a potential collision, the simulator must also determine if and when that collision event will occur. Each object involved in the potential collision is asked when it intends to be at the location where the collision
might occur. If the simulator
concludes that a collision will take
place, it schedules an emerging event
for the time of the collision. The
emergent event is characterized by its
time and the components involved.

Processing of an emergent event starts
when its scheduled time is the current
simulation time. Once processing of an
event starts, each object
involved can send to its environment its
signature and can receive signatures of
the other objects involved. This
happens at time points separated from
each other by a fixed distance ‘d’.
What to do with the information
received is up to the individual objects.

It is probable that there will be other
events to be processed concurrently
with an emergent event. These may be
other scheduled events or other
event events. The simulator moves
onto processing the next event,
regardless of its type, with the smallest
time stamp. Processing of an emergent
event goes on until the components
involved resolve the situation.
Continuation of an emergent event is
ensured by scheduling another such
event after d time units. The end of an
emergent event is marked if either all
the objects are faraway from each other
that they can no longer see each other
or there is at most one object left
among the ones involved in simulation,
e.g., others are destroyed and pulled
out of simulation.

Model Building

The VISP environment supports visual
model building. A typical modelling
exercise starts by preparing visual
appearances of model objects, report
objects and simulator objects involved.
The second step involves setting up
simulation properties for each object
involved. All initial setups can be done
using VB’s graphical interfaces. Each
modelling exercise may require
preparation of one or more windows.
However, it is possible to have multiple
and independent simulations going on
at the same time. Therefore, each
model object must be bound to a
particular simulator object.

Once the initial scenario preparation is
ready and all the object properties are
set, the modeller needs to fill in the
event procedures. As noted before,
some events are related to model
behavior and some are used to support
user interaction when simulations are
conducted. Here is a simplified code
for a mine object:

```vb
important properties:
threshold = 10
amplitude = 50
time_of_next_event = 0
time_of_last_event = 0

Sub Detect_Signal (x as Single )
    if x > threshold then
        explode (amplitude)
        passivate
    end if
End sub
```

When there is a signal detectable by
this mine object within its effective
area, event procedure Detect_Signal is
called with an indication of the strength
of the signal detected. If the signal
strength exceeds a pre-specified
threshold, the mine object explodes
with a strength equal to its amplitude
property and then passivates. It
passivates because it no longer exists
after its explosion. Similarly a very
simplified code for a ship object which
might be intruding the mine’s effective
field might look like the following.

```vb
important properties:
type = 1 ; Destroyer
```
capacity = 100 ;Percent

Sub Get_Hit ( x as Single )
capacity = capacity - (x - 40) * 10
if capacity < 40 then passivate
End sub

Other event procedures; etc.,

Get_Hit event procedure first sets the capacity of the ship. If this capacity goes below 40% as a result of an explosion, the ship detaches itself from its simulator by passivating.

Handling Events

Each object can receive a set of events. A procedure exists for each event which must be filled in by the modeller to handle that event. Default behavior of the system usually does as little as possible. Click, drag, resize are examples of a few standard events and their definition can be found in VB Manuals. Each object receives a different set of events depending on the purpose of the object. However, two event procedures are common to all model objects.

The first of these, the e-function event procedure is used to handle inputs on input ports coupled to other models. Consider Figure 2 where the e-function of the model on the right must be filled in to handle the inputs from the model on the left hand side.

When multiple objects are coupled to a single port, the simulator will send an e-msg to all the receiver objects, and will wait for their acknowledgment before the next event is processed. Note that all the VISP objects can be designed as receiver except for the generator and simulator.

Here is an example for e-function;

Sub SeaPortTower_efunc (Contents As String, time_cur As Single, Phase As String, Port As String)
 ;body of the function
End Sub

Normally, the sender would enclose a text string to the message. This text is delivered to the user using Contents parameter. The global current time at the time of e-function call is kept in time_cur single parameter, and the Phase parameter states the object's current situation and its value is defined by modeller. For example, it may toggle between idle and busy. This Phase is unique to each object and is shared with the i-function which is the second event procedure that is common to all model objects. The last one in the parameter list, Port, is set by the simulator. Port parameter carries in it the name of the input port on which the input message has arrived. The input and output ports are defined using the attach method which will be presented later.

The second event procedure that is common to all model objects is i-function which is used to wake up the object for the next event processing.

The i-function event procedure has the following parameters; The State parameter is a string and it holds the current state of the model object. The time-cur parameter keeps the current global time. The Phase parameter is as explained before.

Methods

In addition to event procedures, a set of methods are defined for some objects. Methods are like event
procedures but are initiated from within the code and can not be re-defined by the modellers whereas event procedures need to be defined by the modeller. "Move", which moves an object to a given point, and "Print", which prints on a picture box, are examples of methods. Many other methods exist in VB resulting in a powerful visual development environment with graphical debugging and editing facilities.

We have the following methods for modelling purposes. ResetSim method is used to reset the simulator.

If ResetSim(Simulator) <> DONE
    Then End
End If

We also use the Activate method to set models active.

If Activate(Generator) <> DONE Then
End
End If

It is also possible to passivate a model object. The Passivate method is used for this purpose. The Attach method couples one component’s output port to an input port of another component. For example Attach(a, p1, b, p2) couples port p1 of component a to port p2 of component b. Normally the return value must be checked to make sure that the coupling has indeed taken place. Use of the Attach method establishes communication channels between various objects dynamically as well as statically.

After establishing communication channels, to send a message from object a to object b is possible using Send_Msg method.

Send_Msg(a, p1, message_text)

This statement is used to deliver the message text to whichever object is connected to the port p1 of a (in this case it is port p2 of object b). After the execution of this method the e-function of the object b is called with its Contents parameter set to the message_text and its Port parameter set to string “p2”.

There are two different modes of breaking established communication channels; single and multiple termination. In single termination, the Detach method is used with the same parameters as its corresponding Attach method which was used to establish that connection in the first place. For multiple termination, an object name and its port id would be sufficient as arguments to Detach method to break all the communication that the object has established through that particular port.

SUMMARY

Modelling and simulation environments have been studied, extensively (Tanir and Sevinc, 1994). Many environments have been developed as an extension of a general purpose programming language (Zeigler and others 1989, Sevinc and Tanir 1990).

Since the VISP environment is built to work within the VB programming environment, graphical debugging, editing and visual preparation of simulation scenarios are supported. In addition, the simulation environment can interface to almost any existing software package through its VB interface. This high degree of interconnectivity provides infinite means for report generation, data acquisition and output analysis.
One of the basic design principles of the Visual and Interactive Simulation Pack (VISP) (Bagde 1994, 1995) described in this article is object independence. Object independence preserves independence of mine warfare objects in simulation models. Object independence states that an object does not have any more information about its environment than its real counterpart would have under similar circumstances. This condition requires changes be made to a standard discrete event simulator.

VISP is largely modelled after (Zeigler 1976). It supports simple component models as well as more complicated components made out of simpler ones. Primary means of communication between objects is message passing facilitated by VB and Windows. However, this is transparent to the modellers.

Models and communication channels can be created and destroyed dynamically. This is an essential feature as the nature of mine warfare modelling requires such facilities.

Graphical editing, debugging and visual model creation are all supported by the VB environment (Microsoft Press, 1994). VB environment also supports creation of stand-alone executable files and provides facilities to design graphical and menu-based user interfaces. The VISP environment can be interfaced to all third party products designed as an extension to VB. Third party products available include animation, database, spreadsheet, network, etc., applications. This provides a vast repertoire for preparing and storing simulation results in whatever format they may be needed.

Discrete event simulators are mostly built for transaction analysis for systems with a static structure. They process events that are scheduled by a part of the system under study. This approach does not work if parts of the system move freely and may come in contact with one another unexpectedly. Such events are not scheduled by any component inside a system and yet they are very important to capture. We call such events emergent events. In this article, we discussed an approach to deal with the emergent event problem efficiently.

REFERENCES


Zeigler, B. P., Kim, T. G., Sevinc, S., Zhan, G. Implementing Methodology-Based Tools in DEVS-Scheme, Modelling and Simulation Methodology: Knowledge System Paradigms.
VISUAL BASIC EXTENSIONS FOR MINE WARFARE MODELLING AND SIMULATION

Suleyman Sevinc
Basser Department of Computer Science
University of Sydney NSW 2006
Australia

ABSTRACT

This article describes a research and development effort in computer modelling and simulation in support of naval mine warfare countermeasure activities. The significance of mine warfare has become especially apparent after recent military experiences gained in Middle East. The issue is even more important for countries like Australia whose well-being is particularly dependent on sea trade. The research described in this article has been conducted at the University of Sydney in collaboration with Maritime Operations Division of Materials Research Laboratory in Melbourne.

The emphasis of the research is to provide a visual environment supporting object-orientation using which models for entities involved in mine warfare can be developed.

Microsoft Visual Basic (VB) was selected to be the environment for its features such as its compatibility with other modern software applications and its visual development environment.

BACKGROUND

(Lawson and Richardson 1994) describes mine sweeping in a graphical way designed as a game. The game designed by Lawson and Richardson is aimed at training skills of a player in mine sweeping given various mine types, locations, probabilities, etc. Mines are placed in a random fashion whereas the role of the player is to design a sweeping strategy for better results.

Mine Warfare Model Structure involves four levels of models. The first level of models involve individual components used in mine warfare. These include mines, remotely operated vehicles, divers, ships and sonars. Since these individual component models are independent from each other in reality, the simulation models should preserve this autonomy. For example, a mine model should be a standalone entity which comes into contact with a ship model at an unknown future time. Similarly a ship model is an entity which does not necessarily know about the existence of mines within its vicinity.

The second level of mine warfare involves coordinated activities involving the individual models of level 1. For example, a set of divers form a team which is coordinated in neutralizing mines. Another example would be mine hunting which requires sonar model outputs to be analyzed to locate mines and to hunt them down using either remotely operated vehicles or other techniques. Other second level activities involve a collection of level one entities. A mine field is such an entity comprising of many mines, false and real, distributed according to a pattern in a geographical area.

The third level of activities involve higher level functions such as mine clearance force-threat models. At this level, the outputs of the level two entities are coordinated. For example, knowledge of mine sweeping models may be used along with the minifield models to model a convoy running through a minesfield. Retrospectively, level three might be interpreted comprising of models where many-to-many interactions between models are common place. Meanwhile level two might be interpreted as collection of models which interact in a one-to-many manner.

Level 4 is the top of the hierarchy and represents strategic models. Strategic models should allow representation of tactical decisions and resource allocation. The ultimate goal of building strategic models is to be able to ask “what if...” questions to the system for specific scenarios (Sevinc 93). Strategic models are representations of situations and command responses to those situations. For example, choosing a mine field and specifying the number of tracks for mine sweeping is a tactical decision. Similarly the command may choose to use convoy models and compare the two approaches for that specific scenario.

Modelling and Simulation

Much work has been done in modelling and simulation (see Tanir and Sevinc 94). Many environments have been developed usually as an extension of some programming language (Zeigler and others 1989, Sevinc and Tanir 90, Sevinc 91). More recently object-oriented models have gained popularity. Even though such environments are very powerful in modularizing system models and
organising them hierarchically, they necessarily fell short in their support for model development as they are not 'visual' development environments.

In this research we have taken the view that visual development environments would provide modellers with the ability to focus on their model development rather than having to deal with idiosyncrasies of the underlying programming language.

We refer the audience to (Zeigler 1976) for more information on simulation.

INTRODUCTION

Microsoft Visual Basic:

Microsoft VB was first released in 1991. Since its release VB has become a popular visual program development environment for Microsoft Windows. VB has many features for program development; graphical debuggers, extensive help facilities, project file management features, options for customising screens, text search and substitution facilities and many other modern features to support a good programming environment (See Microsoft VB manuals).

VB supports an event-driven programming model using a set of objects which users are provided with. Users are provided with buttons for example which can respond to a set of predefined events such as mouse click, double click, dragging the button, etc. Examples of other objects include check boxes, option buttons, text boxes, picture boxes and others. Each one of these objects can respond to a set of pre-defined events depending on their type. Text boxes can respond to key press events, option buttons can respond to click events, picture boxes can respond to resize events, etc. Programmers typically create these objects visually. Behaviour of these objects are determined by filling in the event procedures of each object.

VB is an expandable framework. Many third party developed objects are available in the market place. Such objects are developed for database integration, image processing, networking, voice and almost anything that today's computers are used for. The third party objects can be developed using C and C++.

VB engine supports VB objects using a message passing mechanism. A click event, for example, is captured by the VB engine and forwarded to the event handler of the appropriate object.

The concept of an object in VB is a good basis for representing models of mines, divers, remotely operated vehicles, ships, etc. A mine model, for example, has a number of event procedures such as detect_signal. This event is fired when a ship comes within a distance of a mine model. In addition to specific models to address mine warfare modelling and simulation, a set of generic models are developed which can be used in modelling and simulation of any system.

The development of a new object requires a definition of an icon which determines the look of the object. A set of properties can also be defined to characterise the object and its look. Mine object type for example has a property which indicates its actuation probability and one which indicates its strength if it goes off. A set of events can also be defined. The ship model/object, for example, has an event procedure get_hit which is called when a mine explodes within its vicinity. This event procedure is intended to be filled in by the modellers depending on what kind of ship they are modelling. To complete the definition of a new VB object, a set of methods may be defined. For example, it is possible to passivate a mine by calling a method called passivate which disables the mine.

VISUAL MODELLING ENVIRONMENT

Let us first develop a simple scenario in which a ship model moves towards a mine model. The first step involves creating a mine object and a ship object by clicking on icons and drawing them on the screen.

The second step involves developing models for the scenario. The mine model has a detect_signal event procedure through which it receives signals. In this case it is simple and the mine goes off when the signal level exceeds a certain amplitude, i.e., a ship comes close enough. When the mine model goes off, this is detected by the simulator. The ship model has a soil procedure by which it navigates in whatever direction it is programmed to be. It also has a get_hit procedure which is called by the simulator when a mine within its vicinity goes off. The get_hit procedure decides how much harm is done to the ship depending on the strength of the explosion, the distance from the mine and the type of the ship. The ship and the mine are totally unaware of one another until they come in close contact. The message exchanges between the models are handled by a simulator. Figure 1 shows the event procedures described above in pseudo-VB code.

SHIP_SAIL
move on a straight route

SHIP_GET_HIT
do damage assessment
if damage > tolerance then
   SHIP.DESTROY

MINE_DETECT_SIGNAL
   if signal > threshold then
      MINE.EXPLODE
      MINE.PASSIVATE

Figure 1. Event Procedures for the Simple Scenario

Using this approach it is possible to develop many scenarios.

Once a scenario has been developed, it is possible to turn off the visual features of the models to accelerate processing. If the visual feature stays on, the ship icons move around the screen consistently with their actual route and they disappear as they hit a mine and explode. The visual features are good for training and verification of models developed.

MODELS

Mine warfare modelling and simulation hierarchy involves individual models such as for mine types, ship types, divers, remotely operated vehicles, etc. In the environment described in this article these models are implemented as extended VB objects. A typical modelling study starts with identification of object types involved, drawing these objects on the screen, setting object properties related to their visual appearance and actual types and writing event procedures representing objects' behaviour. Level two of mine warfare modelling and simulation hierarchy involves coordination of these components. For example the detect_signal event procedure of a mine should be activated when a ship enters in its area, or a get_hit event procedure should be activated for a ship model if it happens to be intruding into a real mine's area and that the mine goes off.

A mine model has a number of properties. Some of those are related to the visual appearance of a mine such as whether it has borders or not, background colour, etc. Other properties are aimed at defining the behaviour of a mine model. For example a property Mine.Definition states whether the mine is a true mine or a false mine. Another property called Amplitude is the strength of the explosion if a mine goes off. Yet another property Distance indicates the distance in which the mine will cause some damage to the passing sea traffic.

In the simplified mine model, we have defined one event procedure: detect_signal. Detect_signal receives a signal each time it is called and assesses the signal to decide whether to go off or not. A method Explode is defined which causes a mine model to generate a shock wave and then disappear.

A simple ship model has a ship type property which allows a number of choices including a tanker, destroyer, speed boat and a mine sweeper. The Speed property sets the maximum speed of a ship. It has two event procedures; Sail and Get_Hit. Sail event procedure is designed so that the ship can calculate its route when it is needed. Get_Hit event handles the situation when there is a mine exploding within the vicinity of the ship. A method Destroy totally damages the ship and the ship disappears from the simulation models.

In addition to these we developed other models for mine warfare modelling and simulation including divers, remotely operated vehicles, sonars, etc. We also developed a set of generic components which can be used for general modelling and simulation. These generic models are designed in such a way that their appearance can be set by the modellers depending on the nature of the modelling exercise.

Models must be synchronised in time and space. For example, a mine model should receive signals from passing by ship models. There are serious problems with the computational efficiency of handling such synchronisation in such an environment where models move freely and come in contact with one another at unpredictable times. An example of such a contact is when a ship model briefly enters a mine model's effective area and immediately leaves before the mine decides to explode.

Synchronisation of our models are done by a simulator which keeps account of the location of each object. Discrete event modelling dictates that there are times of no activity, therefore, the global clock can advance safely to the next time. This is not strictly accurate for this case. The route through which the objects move is also important when the route is contaminated by mines in which case the object may never be able to make it to its final destination. Therefore a mechanism beyond a simple event scheduling and management is required.

Each object is defined to be a point in the space with a radius indicating the area which it is enabled to monitor. When such a circle moves on a plane, it produces a channel which can be approximated by straight rectangular paths. The requirement on the models' part is that they are responsible for informing the simulator of their routes between the last event they process and the next event they will be producing. If there is an intersection between the routes of any two objects, ship-to-ship or ship-to-mine, each object is informed of the
potential collision and asked to provide the timing details of the route. If the simulator determines that the objects will have to have intersecting routes on the time axis as well, it changes their time of next event time to just before the coincidence of routes starts and the simulation reverts to a discrete time mode. In other words, a maximum limit is placed on the next event time for the objects involved and they are only allowed to proceed within a certain interval. This addresses the problem we cited in a more efficient way than using pure discrete time based techniques.

STRATEGIES AND MODEL RETRIEVAL

Strategic models sit right at the top of the mine warfare modelling and simulation hierarchy. A strategy is perceived to be a scenario describing a situation.

Handling of scenarios is still at the planning stage. We are experimenting with various representation schemes such as natural language expressions and relational-like models. In this approach we assume many thousands of cases are loaded on a disk. Each case addresses a scenario with certain parameters to be set. The generic description of a scenario helps choosing the right case which then is followed by setting the parameters.

CONCLUSIONS

The article described a visual environment for mine warfare modelling and simulation. The environment is built by extending Visual Basic environment of Microsoft. VB is chosen because of its strengths in interfacing to other applications, ease of program development, graphical debugging and editing facilities and its event-based approach.

ACKNOWLEDGMENTS

I like to express my gratitude to Drs. Don Richardson, Peter Ryan and Richard Watson for their valuable input to this research. I like to thank other DSTO staff involved in mine warfare modelling for their valuable comments on this research. I also like to thank Dr. Kamil Bagdri who does the code development for most parts of the above project.

REFERENCES


Microsoft Visual Basic Manuals, Microsoft Press.
Defining Requirements for a Standard Simulation Environment

Oryal Tanir, Bell Canada and McGill University
Suleyman Sevinc, University of Sydney

Simulation is commonly an integral part of the product life cycle. The simulation project typically progresses through the logical, structured sequence of activities shown in Figure 1. To enhance this process, the concept of a simulation environment, encompassing the underlying simulation language and support tools, has evolved. In fact, a competent simulation environment addresses and supports most of the nine steps identified in Figure 1. For example, model and program design can be aided by graphical representation tools, custom model editors, and model management utilities. The simulation program development stages — program design, implementation, and verification — need editors, interpreters, compilers, and optimizers.

The world of simulation is really an amalgam of different environments striving to meet similar objectives. Although these environments provide similar features, they rarely use the same modeling language. In the past 30 years, over 200 different languages or environments, each presenting its own conceptual approach to simulating a given problem, have been published in the literature. A more standardized approach would overcome this difficulty, providing users with a richer set of tools to produce more reliable models and suppliers with a more rewarding market to explore.

In this article, we define a reference model for general-purpose discrete-event simulation environments, as well as the associated requirements for the model's functional layers, to be used as the basis for a future standard. We characterize environments by highlighting and grouping different tasks with common traits. Each group can then be related to the other groups in a logical, structured manner to create a model for a standard environment. Functionally, the standard environment would reside between the modeler (user) and the hardware platform's operating system, as shown in Figure 2.

Simulation languages

An underlying modeling language is a major contributor to the structure of a simulation environment. Because the environment manipulates components available from the language, the simulation language directly influences the environ-
ment's structure and form. Thus, to highlight similarities in the organization of different environments, let's compare some major features of four representative simulation languages (see Table 1). Three of the languages—SLAM-II, GPSS, and Simscript—have roots in Fortran; the fourth, Simula, is Algol based.

Although the level of model abstraction differs among the four languages, each is capable of representing a given conceptual system. The ease of representation is a direct consequence of the language's natural expressiveness with respect to a given application—that is, its ability to "naturally model" a conceptualized system with the least amount of mental transformation and mapping on the part of the modeler.

All simulation languages contain some underlying implementation language with a model representation mechanism. SLAM-II (version two of the Simulation Language for Alternative Modeling, developed by Pritsker and Associates) models employ stylized flow diagrams. SLAM's network representation mechanism constructs a system model through routing and control information. Because a SLAM network model is usually more comprehensible if it's shown graphically, the language-support environment facilitates graphical model building.

GPSS, which stands for the General-Purpose Simulation System developed by IBM Corp., has its own graphical representation in "blocks," which are essentially subroutines. Models are constructed using the process interaction view where entities (called "transactions" in GPSS) flow through different blocks. Blocks are executed in turn until the transaction terminates.

Simscript II.5 from CACI Inc. models are procedural representations, which are entity oriented. This implies defining models as entities possessing selected attributes. Basic actions are dictated by scheduling commands.

The Simula language models developed by the Norwegian Computing

![Figure 1. A typical simulation project cycle.](image1)

![Figure 2. The role of a standard simulation environment.](image2)

<table>
<thead>
<tr>
<th>Language</th>
<th>View</th>
<th>Debugging</th>
<th>Animation</th>
<th>Report Generation</th>
<th>Submodel Definition</th>
<th>Graphical Model Specification</th>
<th>Experimentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPSS</td>
<td>Process</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>*Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SLAM-II</td>
<td>Event, process activity</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Simscript</td>
<td>Event</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Simula</td>
<td>Event, process activity</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1. Typical features of simulation languages.
Center are built around an object-oriented paradigm. The language provides a powerful “class” construct amenable to modeling. Simula regards models as communicating objects, and a class encapsulates all facets of an object under a common name. Thus, models can be constructed from submodels by inheriting their particular properties. Models can also be reused or replicated by creating an instance of an existing class. This leads to a tree-like relation of models.

The four languages provide the user with different forms of modeling power and different facilities for model specification. The choice of one language over another would depend on the application and the modeler’s own background and preference. Modeling can be accomplished by numerous methods, including graphics, hierarchical construction, syntaxes, and interactive prompting. SLAM-II models use nodes, branch statements, queues, resources, state variables, and processes. GPSS uses blocks and queues. Simscript uses sets, entities, events, procedures, and scheduling. Simula uses classes, objects, lists, and imperative scheduling.

All four languages provide for experimentation — that is, facilities for perturbing the model and observing changes at selected points in the model. SLAM-II and Simscript, for example, can monitor events within a simulation and control experimentation, although actual experimentation must be developed by the modeler.

Validation and debugging are closely linked with these ideas, and different capabilities are available within commercial environments. SLAM-II and GPSS allow graphical construction of network models that provide one level of debugging, as well as snapshots and event tracing. Simscript provides declaration checking, snapshots, and event tracing. Simula performs type checking at the class level, but additional aids must be implemented by the modeler.

Clearly, the four languages — and a majority of other simulation languages — have common organizational features. This commonality provides the basis for a reference model, which can then be used to define future standards that would permit the development of language-independent tools.

**Reference model**

The various constructs of simulation environments can be better understood by defining layers that characterize their functional behavior. Figure 3 shows such a representation, or reference model. The model consists of five distinct layers. The top layer, or application layer 4, can access all layers so that developers can add application-specific constructs to their environments. The lower layers include properties that enable implementation of similar features at higher levels. The lowest layer 0 provides the basic language-level support for the environment and can be accessed by all other layers. Layer 1 defines the requirements for model specification. Layer 2 deals with model knowledge management. Layer 3 is the system design layer.

To establish requirements for a simulation environment, we will address each layer’s requirements and its dependence on the previous layer. In particular, we will establish requirements for the model specification layer 1, which supports the upper layers. The requirements for the upper layers are discussed only briefly; they are topics for ongoing research.

**Layer 0: Host language.** Layer 0 is the host language that provides the basic facilities for model specification. This layer can comprise almost any general-purpose programming environment; however, to properly support the higher level activities of modeling and simulation, the language must support high-level data structures and operations.

The scope of the reference model does not define a particular language for layer 0. In fact, the language should be chosen carefully, since it can limit the environment’s modeling capabilities and affect the modeler’s freedom at higher levels of simulation. For example, a language with no dynamic memory management primitives could restrict higher level operations, and a strictly typed language might impose unnecessary requirements that do not improve modeling.

There might not be one unique language for layer 0. Instead, it may well be specified as a set of languages, any of which could be used as the host language. Even after the host language is chosen, one must still tailor the specific language implementation so that it will not restrict higher level simulation activities.

To illustrate this point, let’s consider the object-oriented class of programming languages. An object-oriented language can be implemented as an extension of a traditional procedural language. However, problems can arise when a message-passing construct is implemented with traditional procedure calls. Each message consumes some program stack space, and a large simulation can require considerable message passing. Consequently, the modeler might encounter unforeseen resource or performance problems during simulation runs.

Thus, when layer 0 is abstracted as a set of service primitives, the semantics of each service must be clear. To avoid a new syntax, the specified host language should be similar to a standard general-purpose programming language.

**Layer 1: Model specification.** Modeling, the most basic activity within simulation, expresses real or imaginary worlds in a formal language’s symbol space. It is more abstract than coding in that coding is required only if the formal language expressing the model must be translated to be understood by a particular machine.

Layer 1 specifies most of the functionality and features that can be accessed by and incorporated at the next two higher layers (2 and 3). With this in mind, the following requirements constitute the minimum basis for flexible, powerful modeling. (Note, however, that for added flexibility, the upper layers can also access layer 0.)

Even when requirements for layer 1 are defined and discussed below.

**Model abstraction.** A model represents a segment of the domain under study. It replicates segment behavior in symbol space by generating trajectories consistent with those of its real counterpart. Since a model is the most basic construct available to a simulationist, support for defining a model is fundamental in a simulation environment.

A model is analogous to an object in the object-oriented paradigm. Therefore, using object-oriented notation, let’s define a construct of the form
class MODEL
operations
data (representation)

A model can be described by a set of operations on data. A data structure, often referred to as a state or representation, can take the form of a stack, list, or queue. Operations manipulate the data and need to be consistent with the structure. Although operations are intended primarily for data manipulation, other operations for pre- and postprocessing purposes, such as window drawing and filtering, may be desired. However, auxiliary operations that manage the presentation (such as visualization) should not be considered part of the model, and support for such operations is not mandatory. These operations may be supported through default dispatch mechanisms, with provisions for allowing modelers to define their own.

Since a model also needs to interact with its environment, class MODEL should be modified as follows:

class MODEL
operations
data
interfaces

Interfaces ensure that models can interact with their environments, including other models, in a predefined manner. Typically, communication is established by traditional message-passing mechanisms between classes. This enhances modularity, a powerful and necessary requirement in a simulation environment. Modularity and message passing provide the basis for constructing distributed simulators to improve the performance of complex simulations.

The class MODEL construct lends itself to very elegant and natural modeling foundations. The class can be reused in other models, inheriting all or part of its parents’ attributes. For example, MODEL can be a super class of a submodel by inheriting its properties. The numerous benefits include:

- development of a database of component models,
- model reusability,
- hierarchical construction of models, and
- increased degree of testability.

The above presents the basic constructs to support modularity and decomposition in a simulation environment, but so far, we have not considered essential timing and synchronization issues. A simulation environment

Simulation terms and definitions

Because the simulation community spans many disciplines, terminology often varies, and some terms, such as model and entity, are used loosely.

Basic terms. Below are basic terms and definitions generally accepted within the discrete-event simulation milieu.

- **Model.** A representation of a conceptualized system implemented within a simulation language.
- **Entity.** Any component in a system that requires explicit representation in a model. Entities possess attributes denoting specific properties.
- **State.** All the variables that are required to uniquely define a system at any point in time.
- **Event.** An occurrence of an operation at an isolated point in time; it may change the system state.
- **Activity.** An operation over a duration of time of finite length that causes change in the system state.

To simulate a simple banking operation with customers either depositing or withdrawing money, the definitions could be interpreted as:

| System: Bank |
| Entities: Customers (attributes: bank balance), tellers |
| Activities: Depositing, withdrawing, waiting |
| State: Teller idle, teller busy, customer waiting |
| Event: Customer arrival, customer departure |

**Modeling views.** Simulation languages employ one or more of the three prevalent modeling views: event scheduling, process interaction, and activity scanning. These are defined in the literature as follows:

- **Event scheduling.** The simulated system is constructed by observing the interaction of events and their respective effects on the system state.
- **Process interaction.** The system is seen as a set of overlapping activities, causing events as they begin or end, but the activities form related groups that are processes.
- **Activity scanning.** This view involves defining the conditions necessary to schedule each activity in a system.
must provide a means for synchronizing model activities. At the atomic model level, a primitive of the "hold" (duration) form may be used for scheduling the model's subsequent temporal activities.

Hierarchical model composition and decomposition. Building complex models from simpler ones or decomposing a complex model into its submodels is a powerful and necessary requirement. Models can be constructed from previously defined submodels or a composite of several submodels. This level of construction is the model's interface specification: a composite model can maintain information with regard to its submodels' interfaces. This can be regarded as an extension to class MODEL.

Decomposition facilitates model verification and validation. For example, a large model can be decomposed into simpler submodels, thus reducing the problem of verification to one of verifying the simpler submodels. Modularity and decomposition are complementary concepts. Therefore,

```plaintext
class MODEL
  component models
  operations
  data
  interfaces
```
becomes an extended description of the model. However, it's useful to separate models that are decomposed from those that are not. A model with no components is an atomic model; otherwise, it is a coupled model. Thus,

```plaintext
class COUPLED-MODEL
  component models;
  atomic or coupled coupling scheme
  interfaces
  operations

and

class ATOMIC-MODEL
  operations
  data
  interfaces
```
The major distinction of coupled models is a scheme that connects the interfaces of its component models. Coupled models may have operations for pre- and postprocessing, as well as for manipulating their component models and outputs.

At the coupled model level, the problem of synchronization may appear as a conflict between two or more component models simultaneously scheduled for an internal activity. This problem can be resolved by a tie-breaking mechanism. Resolution is generally indicated by the particular implementation. For example, a distributed simulation environment might use optimistic or pessimistic scheduling techniques to maintain synchronization. Other environments might enforce an internal-delta simulator clock delay before scheduling so that all events are eventually processed. The details of particular resolutions should not concern the modeler, but the semantics should be clear so that the

**Model specification language.** The language selected to represent the model limits the ease of representation. The specification language can be distinct from the implementation language, allowing the modeler to work independent of the underlying language syntax.

**Modularity.** Models are distinct objects that communicate with each other and the environment through their interface specifications. Adherence to that concept and structure implies modularity. The level of communication between models will be defined by the attributes inherited from layer 0.

**Verification.** An important and often neglected aspect of a simulation environment is the ability to test and verify the behavior of each submodel of a larger model. If modelers can test each model before building a larger, more complex one, they can minimize verification problems in the larger model. The model's interface specification must permit viewing a given model's behavior with respect to controllable stimuli. Decomposing a large model into smaller constituents generally makes managing the verification process easier.

---

**The modeler should be freed from the implementation details of high-level constructs in the environment.**

---

**Experimentation facilities.** Experimentation is the phase that follows modeling in a design or analysis task. The purpose of experimentation is to learn more about the system under study by subjecting its model to various input sequences selected from legitimate inputs.

The process of constructing experiments resembles modeling; it can be thought of as modeling the environment of the system under evaluation. Consequently, additional models can be defined to provide the required stimuli to the modeled system. These models, called experimental frames, allow controlled exercise of the system model through different simulation scenarios. When coupled, models and their experimental frames constitute a closed world. Observation of models during experimentation and checks for experiment acceptability may also be expressed using model theoretic means. Experimentation requirements must be available throughout layers 1 to 3.
provide the modeler with complete capabilities. The extent of experimentation support at layer 1 is variable. As a minimum, input/output states of the models at layer 1 must be observable so that visualization (statistics, animation, graphics) can be supported at higher levels.

A simulation environment must support designing and conducting experiments. It's helpful to separate experiment design from the modeling itself, so the environment should be able to make this distinction. However, developing experiments can be regarded as a modeling activity, so experimentation specifications used in the modeling environment should be applicable to design experiments as well. The key requirements are

- **Data or statistics gathering** — facilities for monitoring specific points in a model, that is, inputs and outputs;
- **Observation of interactions between the component models** — facilities for intercepting data (messages) exchanged between subcomponent models and the means to relate the cause and effect of the exchanges;
- **Redefining basic algorithmic components of the simulator** — flexibility in choosing internal random number generators or using an external algorithm or data file;
- **Presentation of information related to experimentation** — facilities for checkpointing, resetting, restarting, stopping, and so forth.

**Archive capabilities.** The layer must have provisions for storing a specific model as well as retrieving it.

**Model replication.** Models within a model base must be replicable. This implies that their respective attributes and integral components can be copied to a new model. This requirement enables model reuse within a project.

**Model reuse.** This requirement dictates the ability to inherit or reuse all or part of a given model. Modularity alone implies reusability at the model level. More support is required to ensure reuse of parts of a complex model at higher levels.

**Extensible models.** Model definitions (or specifications) must be extensible so that a new model can be generated from another by inheriting its specifications and incorporating additional features. The last two features are required to support model composition.

**Layer 2: Knowledge management.** Layer 2 provides the mechanisms for using and manipulating the models specified at layer 1. The current state of the art in simulation provides ways of organizing and using model knowledge at this level. However, developing requirements will demand considerably more investigation and analysis of the simulation community's interests and needs.

Modeling is seldom a stand-alone activity. Since it's usually part of a design or analysis project, the models may be used in many different ways. For example, a model may need to be stored and checked against other models for equivalence. Consequently, there must be higher level constructs to manage such functions in an efficient way, which we call knowledge management.

A typical example is a graphic and text-based tool that identifies model couplings and synthesizes the actual simulation model from graphical descriptions. Another example is a mechanism for organizing models in a structured manner, whereby model interconnection can be managed by a knowledge management and library support system. When a complex reusable model is requested by a parent model from the library, the environment can determine and create the necessary model interconnections and resources to effectively configure the model to function with the parent. These types of operations are valuable within abstract system design and synthesis environments such as DASE (Design Analysis and Synthesis Environment) where simulation is a key aid for computer and telecommunication system design exploration.

**Layer 3: System design.** Layer 3 deals with design-oriented user issues. The issues are complex and not clearly defined. This implies the use of lower layer primitives to support the design methodology. Important advances at this level have been reported. However, since there is no consensus for requirements at this level, specifying requirements for a reference model is difficult at this time.

**Layer 4: Application layer.** The environment at layer 4 is influenced by the application domain and needs to be specific to a given application. Thus, we are not suggesting standardization at this level. Instead, we perceive a set of domain-specific extensions to the standard. A telecommunications application, for example, would benefit from features for easy modeling of transmission lines, digital switches, or modulators, but most of these requirements would not apply to another domain such as a flexible manufacturing system. Hence, separate requirements would need to be generated for each domain.

**Future issues**

Some advanced issues for further study are listed below. These and other worthwhile concepts can be incorporated into the different layers of the reference model. The revision to any standard, however, will be at a slower rate.

- **Simplification** reduces a model's time and space complexity. We often simplify parts of a model that are not the focus of a particular study. An automatic means for model simplification would be beneficial.
- **Adaptive modeling** lets us study systems that change their behaviors over time. At worst, a simulation environment should not prevent simulationists from developing their own techniques for adaptive system simulation; at best, its features should support adaptive modeling.
AI and modeling often coexist to solve large problems. Eventually, mature methods applying AI techniques to modeling and simulation should be facilitated.

Visualization is another advanced issue that needs to be addressed. However, since visualization can vary from graph plotting to statistics to animation, it can potentially be addressed at every level of the reference model as its requirements become clearer.

Concepts such as parallel and distributed computing are leading to corresponding advances and changes in simulation technology. For example, compared to a structured programming language, an object-oriented language can be nicely implemented in a distributed simulation environment. Hence, the environment must remain flexible to allow integrating future requirements into the various layers.

It is sometimes argued that standardization hinders research. However, reasonable standards have historically focused research beyond the basic frameworks they define. Typically, a standard is adaptive and revised at regular intervals to accommodate technological advances and new ideas. From the onset of this work, our view was to use standardization as a means for advancing the field. Indeed, by providing a common foundation for basic requirements, a simulation environment standard could encourage further research. A standard could achieve a delicate balance between the two goals.

Acknowledgments

We wish to thank B.P. Zeigler and T. Ören for their valuable insight during the preparation of this article.

Standards update

For more information on how this standards activity is developing, readers can contact Oryal Tanir, chair of the IEEE P1173 Working Group on Simulation, at e-mail otanir@qc.bell.ca.

References


Oryal Tanir is the associate director of quality engineering and research within Bell Canada’s Acquisition Technical Services Division. His research interests are performance evaluation of computer system architectures, discrete-system simulation and modeling, and model synthesis. Tanir received his M.Eng degree from McGill University in 1985. He is currently a PhD candidate at McGill in the field of electrical engineering. Tanir is the chair of the IEEE P1173 Working Group on Simulation and has been a representative of the IEEE Technical Committee on Simulation. He is a voting member of ACM and a member of the IEEE Computer Society.

Soleyman Sevinc is a lecturer at the Basser Department of Computer Science at the University of Sydney. He was a consultant lecturer for Microsoft Corp. in 1991-92. His research interests include discrete-event modeling and simulation, and software engineering.

Sevinc received his PhD in electrical engineering from the University of Arizona in 1988. He is a member of the Society for Computer Simulation.

Readers can contact Tanir at Bell Canada and McGill University, Quality Engineering and Research, 3265 Roland Therrien Blvd., Longueuil, QC J4N 1C5 Canada.
Understanding Simulation Model Behaviour Using Event Descriptions

S. Sevinc
Department of Computer Science, University of Sydney, NSW 2006, Australia

Discrete event simulation models are computer programs that are isomorphic to their real counterparts. In their behaviour, they mimic the course of their intended real equivalents. Many simulation studies, therefore, are done to determine whether features based on previous estimations are presented in systems' behaviour.

The predictive power of simulation, however, goes far beyond hypothesis testing allowing discovery of unexpected patterns in systems behaviour. In other words, although simulation is generally used for showing the expected, it is a valuable technique for its ability to show the unexpected.

To be able to derive hypotheses from simulation runs, one needs to equip models with tools which can discover events and relate them to one another in variety of ways such as by causality, simultaneity, implication and exclusion. The current simulation environments, however, though very good at hypothesis testing, do not support hypothesis derivation from the simulation model runs.

As a first step towards this end, we extended DEVSCLOS, a general purpose modelling and simulation environment implemented using Common Lisp Object System, to specify events in a consistent way.

Keywords: Simulation; Event description; Behaviour analysis; DEVSCLOS; DEVSL

1. Introduction

Object-oriented technology is an advancement in implementing simulation methodologies providing better representations of real world entities. Because of the highly modular nature of object-oriented approach to programming, it works well in implementation of modular simulation methodologies. There is, however, no single view to object-orientedness. Therefore, we will briefly explain the Common Lisp Object System (CLOS) view which we selected to implement simulation concepts, in the next section of this article. DEVSCLOS is named after the implementation environment CLOS and the theory on which it is based, namely DEVSL [1]. In Section 3, we will briefly explain DEVSL and describe the design of DEVSCLOS. In Section 4 we present an argument for simulation as a problem solving technique. In Section 5, the facilities implemented in DEVSCLOS for event descriptions are explained in detail. Finally Section 6 looks at the future directions.

2. CLOS

CLOS integrates object-oriented programming into the functional programming paradigm. Since functional programming is largely based on the concept of functions, CLOS uses the concept of function to implement object-oriented features. In the Simula [2] style object-oriented approach the real world is modelled using objects communicating with one another through message passing. Messages passed from one object to another are processed by so called methods or member functions which collectively determine the behaviour of a particular
object. In this approach the problem arises if a function is needed which should be able to operate on multiple objects of different types. Such problems are usually overcome by breaking the modularity as with the *friend* functions of C++ [3].

In the CLOS style of object-oriented programming, there is no concept of passing messages. Instead generic functions and methods are introduced. Generic functions can be viewed as a collection of methods such that a valid call to a generic function is replaced with call(s) to one or more methods associated with it (methods associated with a generic function are named after that function and must have the same number of parameters). In the simplest case only one method will be dispatched: the one whose parameter types best match the arguments provided to the generic function call. Although we will not go into detail, an algorithm is described in detail in [4] which orders potentially applicable methods in place of a generic function call.

In CLOS classes of objects may be related to one another via sub- and superclass relationships. In the case where an object is a member of a class it is also assumed to be a member of every superclass of its class. Assume A is a superclass of B and B is a superclass of C. A call to generic function with a single parameter of type C could then generate up to three applicable methods, each one specialising on a different class. The most specific method or the best match would be the method specialising on the class C if it exists.

The concepts of generic functions, methods and inheritance between classes map quite nicely onto simulation constructs. In the next section we would like to present how such concepts are used in the design and implementation of DEVS-CLOS.

3. Design of DEVS-CLOS

DEVS-CLOS supports hierarchical and highly modular simulation. By modular simulation we mean models are organised into modules which are as independent from one another as possible. Two modules are said to be independent from one another if they do not refer to each other's variables [1]. Interactions between independent modules can only take place through well-defined and compatible interfaces. Modular approaches to simulation work well with hierarchical and top-down design techniques.

Modularity implies a greater degree of independence for various parts of a model. In many cases, simulation models are too big to be coded by a single person. Support for modularity in a simulation environment allows several people to work on different parts of a model. This is good for manageability of the project but creates additional issues to consider.

If parts of a model are built by different people, it is likely that no individual will have a very detailed understanding of the overall system. This means that the system model may present unexpected or emergent behaviour which needs to be discovered. In addition, compatibility problems may arise. Two modules are compatible if the output data generated by one can be processed by the other. Since parts of the simulation, in general, will appear as though they are operating concurrently there will be difficulties in managing this concurrency.

3.1. DEVS Formalism

DEVS formalism [1] is a theory allowing a better understanding of simulation system structure. It provides the mathematical basis on which to build valid simulators. The mathematical details are not relevant here, although the reader is encouraged to refer to [1] to appreciate fully the design details of DEVS-CLOS.

DEVS classifies systems into two classes; coupled and atomic. Coupled models are organisational structures for hierarchical modelling and have in them further coupled models and/or atomic models. Atomic models do not have any further structure in them, but associated with each atomic model is executable simulation code. The judgement of what is coupled and what is atomic is left to the modeller. In Fig. 1, cashier-1, cashier-2 and shopping-area are atomic models, whereas cashiers and supermarket-model are coupled models.

DEVS describes a coupled model in terms of its components and their couplings. Both atomic and coupled models have input and output ports through which they interact with the outside world.

![Fig. 1. Hierarchical model of a supermarket.](image-url)
coupled model keeps couplings of these ports to one another in terms of mappings.

An atomic model also has input and output ports, but since it must be an executable simulation code, it must also have functions which can be called when the component is scheduled to process an event. An atomic component may typically be processing three types of events: input, internal and output. Internal events are those that are scheduled by the component itself at an earlier stage. Output events are those that the component sends out through its output ports in terms of messages to affect its environment. Input events are the messages arriving at the input ports of the component sent by other atomic components. DEVS defines one function for each purpose: external transition function, internal transition function and output function.

3.2. DEVS-CLOS

DEVS-CLOS implements basic DEVS. It has representations of atomic and coupled models as well as their couplings. It is best explained in terms of an example.

Let us take the supermarket model in Fig. 1 as the example to present DEVS-CLOS concepts. A typical modelling and simulation exercise starts with a structural analysis of the system under study as in Fig. 1. The tree structure in Fig. 1, as the second step in modelling in DEVS-CLOS, would be detailed in Fig. 2.

The next step in model development is to develop the simulation code for the atomic models. In this case we note that the models cashier-1 and cashier-2 are both instances of a class, say cashier. So what we can do is define the cashier model and create cashier-1 and cashier-2 as instances of that model class. Figure 3 shows internal transition, external transition and output functions for cashier model class.

In Fig. 3, WORKING refers to a time-consuming state of the model and 'hold-in' is a macro which schedules an event for the module. Both 'passivate' and 'continue' are macros: 'passivate' schedules no future events for the module by driving it to a non-active state; 'continue' resets the model state to its value just before the arrival of the last external event. In the output function, OUT refers to an output port and send puts a message out on that port.

To create the cashier-1 atomic model and to integrate it into the overall system:

```
(defint cashier
 (setf (queue model)
   (cdr (queue model)))
 (if (queue model)
   (hold-in WORKING (random 10))
   (passivate)
 )
)
(defext cashier
 (setf queue model)
 (append (queue model)(list x))
 (if (> (length (queue model)) 1)
   (continue)
   (hold-in WORKING (random 10))
 )
)

(defout cashier
 (send (make-instance 'data-message
 :port 'OUT
 :content t))
)
```

Fig. 3. DEVS-CLOS definitions for cashier model class.

Fig. 2. A detailed equivalent of Fig. 1.
Similarly cashier-2 and shopping-area models can be created and integrated into the supermarket model. Notice the import of the model shopping-area needs to receive customers which in DEVSCLOS are generated by the experimental frame which is not shown in Fig. 2. For details of the experimental frame concept see [1].

Once all the atomic models and coupled models are defined in the above manner, the system is ready to conduct simulations. Simulations in DEVSCLOS are conducted in terms of rounds. Each round starts with a call to when-receive-\^\text{*} generic function which dispatches the methods associated with the highest level model in the hierarchy, which in this case is the model supermarket. The same call then has a rippling effect of calling when-receive-\^\text{*} methods of all the models on the path to the atomic model with the smallest time advance level (Fig. 1). Finally the method associated with the atomic model is also called which issues a call to its output function. The output function may generate a number of messages, each one of which is forwarded to its final destination in a hierarchical way. For each message delivered the system issues a call to the external transition function of the destination atomic model. The external transition functions would ensure correct state changes and operation of the model and also schedules internal events for the future for their components. Once all the destinations have finished processing the system generates a call to the internal transition function of the atomic model which originally started the processing of the current round. The internal transition function also ensures correct state changes and schedules a future event for its component. Now the system has completed processing one round and reached a stable state, therefore another round can start in the same way.

The original DEVS definition has an additional function called time advance function. In the previous example this function is not implemented as a separate entity. Instead it is buried inside the transition functions using the hold-in macro. DEVSCLOS allows a separate time advance function definition if required.

3.3. DEVS-CLOS Classes

Definition of new model classes and models are integrated into DEVS-CLOS class structure by gradual extension using inheritance. At the top level is the model class from which atomic-model, coupled-model and root-model classes inherit. In our example, cashiers is a coupled model, so it would inherit features from coupled-model class, whereas supermarket would inherit features from the root-model class. The model class cashier inherits from atomic models and it is designed to make use of the inheritance facilities in the definition of the cashier-1 and cashier-2 models as these models are identical to one another. The class shopping-area also inherits from atomic-model class. Cashier-1 and cashier-2 are more specific than the cashier class. The inheritance tree is presented in Fig. 4.

3.4. Strengths and Weaknesses of DEVS-CLOS

The use of object-oriented technology combined with the power of CLOS makes DEVSCLOS a very suitable environment for simulation methodology research. In addition, the use of DEVS formalism provides the formal basis for proofs of simulator correctness. On the weak side of DEVSCLOS, and any other simulator for that matter, is the difficulty of managing concurrent events. For example, if two events are simultaneous, DEVSCLOS could order them although this is under the control of the modeller. In non-serialisable events, where the simulation results will change according to the order of two events, this must be very carefully observed.

Also not allowed in DEVSCLOS is the pre-empting of events, unless explicitly managed by the modeller. The environment is entirely open for modification, although this must be exercised with care. For example, should one change the orders of method calls made to external transition functions and output functions, i.e. allowing generation of output while external transition function is being executed, it might invalidate the simulator for certain models. Models which might generate outputs at an unstable state and might receive a message back
will not operate correctly. This would be a condition very hard to detect, as it may happen occasionally with certain models, and the simulation system issues no warnings. In fact this is a general problem for any simulation environment and is not something specific to DEVS-CLOS.

4. Simulation as a Problem-Solving Tool

Whether discrete or not, simulation models are usually used to replicate the behaviour of the real world. The traditional approach to simulation, therefore, consists of modelling, simulation, experimentation, observation and analysis. The analysis is usually done using statistics, although lately expert systems and causal analysis [5] have also been proposed.

Simulation, however, can also be viewed as a problem-solving tool like analytical or declarative methods. Declarative methods model the reality in terms of true or false statements, analytical methods model it using arithmetic and simulation models the real world in terms of procedures or computer programs.

Let us borrow an example from the literature: a frog trying to get out of a hole which is 100 yards deep. The frog jumps 2 yards at each trial and takes 1 second to get prepared for the second jump. However, during this interval the frog slides down by one yard. How many jumps or how many seconds does the frog need to get out of the well?

This problem can be solved using analytical models, quite easily. For the sake of our argument let us demonstrate a solution using procedural models. Figure 5 captures the frog’s behaviour in terms of a procedure which can be coded into a simulator like DEVS-CLOS. This procedural model captures the frog’s attempts just as well as analytical models. In this particular case the model will print out the number of steps and seconds required to achieve the task and then it will passivate.

Simulation models have many advantages over the analytical models; they may be as detailed as one requires, whereas analytical models for medium size systems are usually beyond human comprehension and are full of approximations and simplifications. Simulation (or procedural) modelling is a more natural way to capture real world system operations.

Since a system can be represented using any of the modelling paradigms, i.e., mathematical, logical and procedural, they can be converted to one another [6]. In our interpretation, understanding simulation model behaviour is extracting true statements about the simulation model behaviour. In other words it may basically be interpreted as extracting equivalent logical theories from procedural model runs.

Extraction of equivalent logical theories from simulation models requires that the simulation system be able to understand events and be able to relate them to one another. The approach proposed in this article provides a framework for expressing such events. In the next section, we will describe this framework in more detail.

5. Event Descriptions in DEVS-CLOS

Suppose we are trying to simulate the supermarket used as example in the earlier sections. Furthermore, assume that the supermarket has a policy whereby a new cashier is promised to be made available whenever all the queues contain three or more customers (see Fig. 6).

To capture this more formally, whenever the

![Diagram](image)

**Fig. 6.** Adding a new cashier.
statement \( \forall m, m \in \text{MODELS}, \; m.\text{queue\_length} \geq 3 \) becomes true, a new cashier model needs to be created and integrated into the supermarket model.

IN DEVS-CLOS models may be referred to by name and can be created or destroyed dynamically. Suppose the cashier models are named in some consistent manner such as \text{CASHIER-1}, \text{CASHIER-2}, etc.

We added facilities to DEVS-CLOS which help us generate and select consistent names from various model sets. One such facility is called conditional-expand which essentially is a regular expression matching routine. It receives a list of symbols as its argument and returns a sublist consisting of only those that are consistent with a given pattern. For example (conditional-expand \text{CASHIER-~a (elements MODELS)}) would return a list of symbols containing \text{[CASHIER-1] [CASHIER-2]...}. The symbol \text{~a} indicates any character would match.

The conditional-expand facility is particularly good for identifying model classes. These model classes would have to be named in some consistent form. The following illustrates the use of conditional-expand in identifying all the models whose queue lengths are greater than or equal to three and whose names start with \text{CASHIER-:}

\[
(\text{mapcar } #\{ (\text{lambda} (x) \\
(>= (\text{queue\_length} \; x) \; 3)) \\
\text{(conditional\_expand \text{CASHIER-~a (elements MODELS))})}
\]

The above expression would return a list with all non-zero elements for those models whose queue length is greater than or equal to three.

Creation of a new model in DEVS-CLOS, dynamically, is a trivial exercise;

\[
(\text{defmodel } \text{model\_name} \\
(\text{super \; classes \; if \; any} \\
\text{coupings \; (\ldots)})
\]

Here the coupings provided are simply added to the hash tables of the parent model. These coupings are then used to route the messages exchanged between the newly made component and the other existing components.

Please note that the above code is not complete and edited for clarity where necessary. Additional operations may be needed to integrate a model fully into the current model structure.

Now we are in the position to put all this together to express the statement when \( \forall m, m \in \text{MODELS}, \; m.\text{queue\_length} \geq 3 \) becomes true, create a new cashier model:

\[
(\text{if (reduce #\{log\_and} \\
(\text{mapcar } #\{ (\text{lambda} (x) \\
(>= (\text{queue\_length} \; x) \; 3)) \\
\text{(conditional\_expand \text{CASHIER-~a}} \\
\text{elements \text{MODELS})})
\]

\[
;; \text{when } \forall m, m \in \text{MODELS}, \; m.\text{queue\_length} \geq 3
\]

\[
(\text{let} \\
(\text{(new\_name (car (expand \text{CASHIER-~a}} \\
\text{range}))})
\]

\[
(\text{defmodel } \text{new\_name} \\
(\text{CASHIER atomic\_models}) \\
\text{coupings ((\text{SHOPPING\_AREA \; OUT}} \\
\text{(new\_model IN))}
\]

The utility \text{log\_and} implements the operation logical-and of a list of truth values.

Similarly, once the need is removed, a cashier must be removed from the system. This merely requires removing the appropriate coupling specifications and freeing the model.

To complete the discussion we also like to show how existential quantifiers may be used in DEVS-CLOS environment. An existentially qualified statement such as \( \exists m, m \in \text{MODELS} \) and \( m.\text{queue\_length} \geq 3 \) (there is at least one model with a queue length greater than three) may be encoded in the following way:

\[
(\text{reduce #\{log\_or} \\
(\text{mapcar } #\{ (\text{lambda} (x) \\
(>= (\text{queue\_length} \; x) \; 3)) \\
\text{(conditional\_expand \text{CASHIER-~a}} \\
\text{elements \text{MODELS})})
\]

5.1. Other Examples

The integration of the above facilities to the DEVS-CLOS environment provides the basis to express events which then can be detected by the simulation system itself for further processing. For example, in a computer network simulation, collision is an event when two transmitting computers corrupt one another’s data. Collision levels are usually indicative of the network performance and are important to the modeler; the definition of the collision event would be \( \exists x,y, \; x \neq y, \; x \in \text{COMPUTERS}, \; x.\text{transmitting}, \; y.\text{transmitting} \). In other words, collision requires two computers to be transmitting at the same time. Stopping conditions of a simulation study may also be expressed using descriptions like those above, such as, \( \forall m, m \in \text{MODELS}, \; m.\text{queue\_length} = 0 \). Our last example is a definition for
deadlock, which is the case when all components in a system are waiting for something to happen, as in the case of simulation of the famous dining philosophers problem; $\forall x, x \in$ DINING-PHILOSOPHERS, $x.waiting$.

6. Future Directions

What has been described so far in this article is an event description technique by which the simulation systems can recognise the occurrence of events. There are two basic issues that need to be resolved in order to realise fully what was described in the abstract of this article.

The first issue has to do with relating events to one another in sensible ways. For example, deriving a hypothesis such as 'event A is always followed by event B' may be useful information not known to the modeller before the simulation study was done. There are other ways of relating these events to one another such as by causality, implication, exclusion, etc. One might argue that these terms themselves are very difficult to describe in a precise way, nevertheless provision of such a tool even without an exact theory would be a useful exercise.

The other issue is related to event descriptions. Currently, event descriptions are done manually; however, a fully automated system should have the capabilities to generate higher level event descriptions from sequences of lower level micro events. The past approaches to this include determining scripts consisting of a regular pattern and state abstraction. Describing events as scripts that are similar to one another is a time-intensive operation where states of the system are examined to see if patterns recur from time to time. The state abstraction approach [6] is about identification of states that are similar to one another and grouping them into abstract states. The state abstraction technique is weak where relative ordering of states becomes important for event descriptions.

Simulation, as discussed above, is a problem-solving tool and has more expressive power than the other modelling paradigms in a practical sense. Highly modular simulation models and the interaction between these modules often present behaviour which may not have been conceived by the modellers. This behaviour needs to be captured and brought to the attention of the modeller.

References

EXTENDING COMMON LISP OBJECT SYSTEM FOR DISCRETE EVENT MODELING AND SIMULATION

Suleyman Sevinc
Department of Computer Science
University of Sydney
NSW 2006 AUSTRALIA

ABSTRACT

We described DEVS-CLOS modelling and simulation environment. The environment combines Zeigler’s DEVS formalism with powerful constructs of CLOS. DEVS-CLOS allows dynamic model creation and modification and is capable of supporting advanced simulation research, as well as applications.

1 INTRODUCTION

Common LISP Object System (CLOS) is an extension of Common LISP to support object oriented programming (Steele 1990). There are various reasons behind our choice for using CLOS to do modelling and simulation. To start with, simulation environments do not require any programming constructs that can not be supported by a general purpose programming language such as CLOS. Moreover, CLOS consists of some attractive features that reduce model development time considerably. Most importantly, the resulting environment is entirely open to modifications, therefore capable of being used in research as well as applications. The extension is named DEVS-CLOS after the formalism it is based on (DEVS, Zeigler(1976)). We will briefly review relevant parts of CLOS in the next section. This will be followed by a summary of novel simulation concepts supported by DEVS-CLOS.

2 CLOS

Basic CLOS facilities include those for defining classes, generic functions and methods. A class definition consists of a set of slots, generic functions to access these slots and other classes (called direct superclasses) whose definitions are inherited. Generic functions are constructs with a set of methods and a dispatching mechanism. Methods are related to the classes via a relation called applicability. A method is said to be applicable only if its parameter specializers are satisfied by the current set of arguments. For multilevel inheritance, a class precedence algorithm is included in CLOS definition which produces a unique ordering of superclasses for a class. The output of the algorithm is used to order all applicable methods associated with the generic function call and the dispatching mechanism selects and applies a subset of these, in the order specified. For more information, the reader is advised to refer to Steele (1990).

3 DEVS-CLOS

DEVS-CLOS is an advanced implementation of DEVS concepts which have been developed by B.P. Zeigler in his seminal works Zeigler (1976, 1984). It is closely related to DEVS-SCHEME described in Zeigler (1990) in some detail (see also Cellier, Zeigler and Wang (1990) and Zeigler, Hu and Rozenblit (1989) It is implemented with two goals in mind: We used DEVS concepts and abstract simulator code to ensure correctness of our simulator and we tried giving our modellers a flexible basis to do applications or research. A typical simulation study in DEVS-CLOS starts with a series of decompositions. The decomposition process results in a hierarchical representation of the system being studied (see Figure 1 for an example supermarket model). The leaves of the system decomposition tree are called the ATOMIC-MODELS whereas intermediate nodes are COUPLED-MODELS.

Figure 1: Hierarchical Model Representation

SUPERMARKET

CHECK-OUT-AREA  SHOPPING-AREA

CASHIER-1  CASHIER-2 ... CASHIER-N
In contrast to the analysis process, the synthesis starts at the bottom. ATOMIC-MODELS are implemented first. COUPLED-MODELS are formed using coupling specifications which are mappings between input/output ports of other models. To completely define an atomic model, we have to define 3 functions; external transition function to process input events, internal transition function to process events scheduled by the model itself and output function whose return value constitutes an output which is forwarded to the proper destination using coupling specifications. Figure 2 shows all the related definitions for the CASHIER model of Figure 1. We will now provide an example to illustrate dynamic model creation controlled by logic statements in DEVS-CLOS.

(defclass CASHIER (atomic-models)
   ((queue-length :initial 0 :initarg :ql :accessor queue-length)))

(defext CASHIER
   (self (queue-length model)
      (+ 1 (queue-length model)))
   (if (> (queue-length model) 1)
      (continue)
      (hold-in WORKING (random 10)))
   )

(defint CASHIER
   (self (queue-length model)
      (- (queue-length model) 1))
   (if (> (queue-length model) 0)
      (hold-in WORKING (random 10))
      (passivate))
   )

(defout CASHIER
   (send (make-instance 'data-message
      :port 'OUT
      :content t))
   )

Figure 2. Definition of model CASHIER in DEVS-CLOS

3.1 Novel Model Oriented Operations in DEVS-CLOS

Suppose we are trying to simulate the three is a crowd policy of a supermarket in which a new cashier is promised to be made available whenever all the queues contain 3 or more customers (Figure 3).

More formally, we would like to create a cashier model whenever the statement \( \forall m \in \text{MODELS} \ m.\text{queue-length} \geq 3 \) becomes true. In DEVS-CLOS, models can be referred to by name and can be dynamically created or destroyed. Suppose the cashier models are named in some consistent manner such as CASHIER-1, CASHIER-2, etc. The statement above can be expressed by (conditional-expand CASHIER-a (elements MODELS)) which returns a list of symbols (CASHIER-11 |CASHIER-21 ...). The macro conditional-expand returns all the models in set MODELS whose names match the form CASHIER-"a where "a is don't care. There are many ways in CLOS to test a specific condition on list elements:

(mapcar('#'(lambda (x)
  (>= (queue-length x) 3)))

Figure 3: Creation and Integration of a Model
(conditional-expand CASHIER-a
   (elements MODELS))
will return a list with all non-nil elements when all the
queues have 3 or more customers. This list may be
reduced to a single truth value using reduce operation
of Common LISP (CL). The following is a complete
coding of the statement whenever \( \forall m \in \text{MODELS} \)
m.queue-length \( \geq 3 \) make a new model in DEVS-
CLOS.

\[
\begin{align*}
\text{if} & \quad (\text{reduce \#'log-and} \\
& \quad (\text{mapcar \#'\{(lambda (x) \\
& \quad (> = (\text{queue-length} x) 3)\}}} \\
& \quad \text{(conditional-expand CASHIER-a} \\
& \quad \text{(elements MODELS)}\)) \\
& \quad :: \text{when } \forall m \in \text{MODELS} \ m.\text{queue-length} \geq 3 \\
& \quad \text{let} \\
& \quad \text{((new-name (car (expand CASHIER-a range)))} \\
& \quad \text{(defmodel new-name} \\
& \quad \text{(CASHIER atomic-models)} \\
& \quad \text{::couplings (\{SHOPPING-AREA OUT} \\
& \quad \text{\{NEW-MODEL IN\}})} \\
& \quad \text{)}}
\end{align*}
\]

We would like to note that the above code is edited
for clarity where necessary. (expand string list) is a
macro that generates a list containing all combinations
of string substituted by the elements of list in a
manner similar to the format statement of CL. defmo-
del is an extended model definition way in which cou-
pplings can be specified in terms of (model port) pairs.
log-and is a function which performs logical and.

Similarly, \( \exists m \in \text{MODELS} \) such that \( m.\text{queue-}
length \geq 3 \) may be coded in the following way;

\[
\begin{align*}
\text{(reduce \#'log-or} \\
& \quad (\text{mapcar \#'\{(lambda (x) \\
& \quad (> = (\text{queue-length} x) 3)\}}} \\
& \quad \text{(conditional-expand CASHIER-a} \\
& \quad \text{(elements MODELS)}))))
\end{align*}
\]

5 REFERENCES
F. Cellier, B.P. Zeigler and Q. Wang, "A Five Level
Hierarchy for the Management of Simulation
Steele, G.L. Jr., COMMON LISP: THE LANGUAGE,
Zeigler, B.P., Theory of Modelling and Simulation,
Zeigler, B.P., Multifacetted Modelling and Discrete
Zeigler, B.P., Object-Oriented Simulation with
Hierarchical, Modular Models, Academic Press,
1990.
Zeigler, B.P., J. Hu and J.W. Rozenblit, Hierarchical
Modular Modelling in DEVS-Scheme, Proc. Winter

6 AUTHOR BIOGRAPHY
SULEYMAN SEVINC is a lecturer in the Depart-
ment of Computer Science at the University of Syd-
ney, Australia. His research interests are model
abstraction and theory-based modelling environments.

4 CONCLUSIONS
DEVS-CLOS, based on DEVS, is a flexible simulation
environment. The environment is model oriented and
can fully support both behavioural and structural simu-
lation. The structural changes are triggered by first
order statements and involve making new models and
integrating them into larger models, destroying exist-
ing models when they are not needed and changes in
the model structure when necessary. Current research
looks into ways of using first order logic to reason
about the simulation behaviour.
Index

.env, 45
.p, 20
.sim, 45

acceleration
  constant, 77
  limited, 78
applet, 23

basser.sim.comp, 27
basser.sim.core, 36
basser.sim.gui, 25
basser.util, 36
basser.util.net, 36
basser.util.Params, 28
behaviour, 29
binary search, 70
Button Bar, 41

CanvasFrame, 37
CGI, 9
Chart, 37
check, 21
CircularRange, 33
class, 23
ClassButton, 25
ClassFrame, 36
ClassLoader, 36
ClassPalette, 25
client, 9
client-server, 9
CloseFrame, 37
code window, 48
collision detection unit, 32
ColorParam, 28

Database of Entities, 15
Database of Pictures, 14
Database of Scenarios, 18
Destroy, 44
doc.cgi, 12
doubleParam, 28
dynamic approach, 67

Editable, 26
EditableList, 26
EditComp, 31
event
  entering, 66
  leaving, 66
Emergent Event, 33
event, 32, 40
event queue, 32
eventHandler(), 30

Feedback page, 12
fixed step, 69
footer, 12

GenericHandler, 29
genericInit(), 31
GenericSimulable, 27, 51
GenericSimulableTemplate, 31
Grip, 26

Histogram, 37
HTTP, 10

Icon, 26
Icon Bar, 41
image.cgi, 14
ImageParam, 28
inf.cgi, 19
Init, 44
init(), 30
interest matrix, 82
IntParam, 28

JDK, 23

lib.cgi, 15
log, 22

Main Loop, 35
Main page, 11
mass centre, 7
material entity, 41
maximum speed, 69
monotony problem, 68
Moving, 33

.Net, 36
NetFrame, 25, 36

On-line Documentation, 12
On-line Tutorial, 12

Palette, 43
palette, 25
ParamEdit, 28
Parameter, 28
produce(), 26

Range, 33
ISBN: 1 86451 329 2

Basser Department of Computer Science
University of Sydney
Sydney NSW 2006
Australia