LARGE-SCALE SYSTEM DEVELOPMENT IN SEVERAL LISP ENVIRONMENTS!

Sanjal Naraln, David McArthur and Philip Klahr

The Rand Corporation
1700 Main Street
Santa Monica, California 90406
USA

ABSTRACT

ROSS [7] is an object-oriented language developed for building knowledge-based simulations [4I. SWIRL [5, 6] is a program written in ROSS that embeds knowledge about defensive and offensive air battle strategies. Given an initial configuration of military forces, SWIRL simulates the resulting air battle. We have implemented ROSS and SWIRL in several different Lisp environments. We report upon this experience by comparing the various environments in terms of cpu usage, real-time usage, and various user aids.

1. INTRODUCTION

Over the past six months we have been engaged in implementing ROSS (an object-oriented, know ledge-based simulation language [7]j and SWIRL (an air battle simulation [5, 6]) in five different Lisp environments We discovered that the environments varied considerably in how well they supported system development. We report upon our experience by comparing the various environments in terms of cpu usage, real-time usage, and user aids (e.g., editors and file packages). Our aim is to critically examine several Lisps available today, in order to help those who are building large systems to make an informed choice about which dialect may be most appropriate for their application.

2. FIVE IMPLEMENTATIONS OF ROSS AND SWIRL

2.1. Maclisp on a DEC-20

Maclisp, running under TOPS-20, was the original language in which ROSS was implemented. Approximately one man-year was required to make ROSS a reasonably mature object-oriented programming environment. Two factors greatly contributed to the speed of ROSS development. First, Maclisp has a very efficient implementation on the DEC-20 that made it possible to quickly experiment with additions and changes to the ROSS design. (Of course, the speed of Maclisp also contributed to the efficiency of the SWIRL simulation written in ROSS. As may be seen in Table 1, the implementation in the Maclisp-ROSS version is, overall, the fastest of all implementations of SWIRL.) Second, within Maclisp, we could make use of Emacs, a powerful screen-oriented editor, that runs locally within Maclisp through the use of the LEDIT package. Emacs had two features that were crucial to efficient program development. First, Emacs understands Lisp structures (e.g., it does automatic s-expression indentation, pretty-printing and parentheses balancing). Second, when using Emacs to edit a function definition, one directly edits the file containing the function. The user is in full control over not only the format of function definitions, but also exactly how the functions are arranged in files, and therefore in hardcopy when printed.

What hindered the development of ROSS in Maclisp was the poor documentation of many essential language features. Often novice Maclisp users regard Maclisp as primitive, especially compared to Interlisp, which advertises such user packages as the Clisp iterative facility, the record package, the filepackage, and the PRINTOUT printing facility In fact all these features exist in Maclisp, but until recently, unless you had a Maclisp wizard around, the only way to find out about them was by browsing through the files your machine's Maclisp directory. This kind of search cost us at least a month in development time.

† The views expressed in this paper are the authors' own and do not represent thope of Rand or any of its research sponsors.

The other main problem with the Maclisp implementation was that it resided on a DEC-20. This machine is proving increasingly inadequate for large AI programs, because of its small 18-bit address space.

2.2. Franrllsp on a VAX-11/780 [2]

Because of Franzlisp's advertised compatibility with Maclisp, we thought it would be straightforward to convert our Maclisp versions of ROSS and SWIRL to work within the Franzlisp environment. This was almost the case. About two man-weeks of effort over a one month period was required for the conversion process. We did notice certain inconsistencies between Franzlisp and Maclisp but they were fairly easy to fix. Although ROSS and SWIRL are large systems even when compiled (see Table 1), they ran acceptably fast within Franzlisp, even when running large simulations.

Our Franzlisp runs under the UNIX operating system and provides us with a very useful link to our C-based graphics programs. Compiled versions of these programs can be directly loaded into Franzlisp. This feature allows us to dynamically view a SWIRL simulation as it is running.

Franzlisp also scores high marks for some user development tools. Like Maclisp, one can use Emacs as a screen-oriented local editor. Here the interaction is a little different. The user is actually in Emacs and sends forms to Lisp for evaluation. The advantage of this is that within Emacs, the user can move backwards and forwards across forms, modifying them and resubmitting them to Lisp as he chooses. We find this to be a powerful environment for rapid code modification.

2.3. Interllsp-D on a Dolphin [12]

Our first Interlisp version of ROSS was developed on the Xerox 1100 (Dolphin) Processor. This version underwent substantial redesign not only because of the difference between Maclisp and Interlisp but also because we wanted to improve upon the Maclisp version. It took us about four man-months for the new implementation, which was more time than expected. The major problem was speed. As the statistics in Table 1 indicate, the Dolphin was over an order of magnitude slower than the Maclisp-20 version for running a standard simulation. But the Dolphin fared worse in speed of software development than speed of performance of a developed system. File operations (e.g., loading files or editing a function then saving it on file, see Table 1) were the major bottlenecks. Further, since interpreted code on the Dolphin usually executes too slowly to be tolerable, all code had to be compiled before being tested. This substantially increased the time between identifying a bug, fixing it, and testing out the fix.

Perhaps the most outstanding feature of the Dolphin is its graphics capability. We found it to be very powerful for quickly implementing a wide range of graphics tasks. Also several user aids like the trace and break packages have been augmented graphically and enable one to display and hold far greater information on the screen than with conventional terminals. The Dolphin's graphics capabilities, bitmap display, mouse and windowing are well exploited by DEDIT, the principle function editor. It is especially powerful when used with the TTYIN facility which, like Emacs, has some very useful features such as parenthesis matching, and moving forwards and backwards over s-expressions. We have one maior caveat: DEDIT is a function editor, not a file editor. With DEDIT one changes the definition of a function, but MAKEFILE (the Interlisp function for writing out a symbolic file) determines the appearance and location of function definitions within files. In contrast, when editing functions with Emacs in Maclisp, one changes both the definition and appearance, since one is editing a file and a function simultaneously.

The Dolphin operating system does not provide a tree structured directory system, so that files cannot be stored as logically as one would like them to be. The high bandwidth (3 Mbits/sec) ethernet communication network connecting the Dolphins and the VAX-11/780 allows one to quickly transfer large files between different stations.

2.4. Interlisp on a VAX-11/780 [1]

Bringing the ROSS/SWIRL system up in VAX-Interlisp did not turn out to be as straightforward as we had expected even though we had already developed ROSS/SWIRL in Interlisp-D. Though Interlisp-D source code was almost completely transportable, certain subtle incompatibilities were fairly time consuming to discover. It took about one and a half man-months to bring the ROSS/SWIRL system up.

As with Interlisp-D, speed was the main factor slowing both program development and hindering subsequent effective use of the SWIRL simulation. This slowness is partly due to working in a time-sharing environment and partly due to the sheer size of Interlisp and the overhead involved in evaluating an expression (e.g., DWIM, CLISP). We regard the Interlisp-VAX version of ROSS as marginal for serious simulation applications.

One inconvenience of Interlisp-VAX, not shared by the other Interlisp implementations, is that UNIX file names are restricted to at most 14 characters and do not have version numbers. Although Interlisp does attempt to simulate version numbers, it does so at the UNIX level. These files all look alike and what results can be seriously confusing at times. Also, considerable caution must be exercised while manipulating files in UNIX since it is very easy to delete or overwrite them.

Since the conversion task involved a non-trivial amount of file manipulations (editing, copying, deleting) the process was helped immensely by the availability excellent file editors (Emacs and the local Rand editor E), ani by the UNIX software support. We rarely had occasion to use the Interlisp structure editor.

2.5. Interllsp on a DEC-20 (10)

ROSS/SWIRL has been most recently developed for Interlisp on the DFX-20. It took about one man-week to bring up this version. Yet, several factors made this a surprisingly painful process. Among other things, the backquote facility (for defining macros) had to be redefined. We also found that files bad to be massaged in format considerably so that Interlisp-10 would accept them. However, because of the extremely efficient disk operations on the DEC-20, the powerful Emacs editor, the friendly TOPS-20 operating system (which has among other features, excellent backup support, tree structured file directories, altmode completion), and useful TOPS-20 resident software, it was possible to go through several iterations of the system fairly rapidly. Even so, SWIRL ran substantially slower than the corresponding implementation in Maclisp on a DEC-20.

3. DISCUSSION

Through several years of experience in developing large Lisp-based systems we have formed certain opinions about features in various Lisp environments, and their effect upon programmer productivity. We summarize these below.

First, as perhaps with any software development effort, the total turnaround time between writing a piece of code and testing it out should be minimal. This implies that the cycle of obtaining fresh Lisp sessions, loading files, executing code, modifying code, and storing changes should be as rapid as possible. We found that Maclisp, Franzlisp and Interlisp-10 satisfied this requirement, but Interlisp-D and Interlisp-VAX did not. A major bottleneck in Interlisp-D program development concerned file operations. Not only did they take a long time to complete, but there was also a lack of a good file editor. We had to use the file editors on the VAX, particularly when converting some of SWIRL files from Maclisp into Interlisp, and suffered a substantial increase in the number of time-consuming steps needed to make a change on disk. Also, Interlisp is much bigger than Maclisp or Franzlisp, and its large size causes frequent swapping at run time leading to considerable deterioration in execution speed. The presence of user aids (e.g. DWIM, Mas terse ope, Programmers Assistant) if not deactivated, further adds to the time needed to produce a response. Even while programming in Int lisp, we found we had rare occasion to use many of these featur In the rase of Interlisp-D, for example, DWIM operated so slow that it was quicker for the user to detect and correct his own sp ling errors.

Secondly, we found it more desirable to create our code by direc writing into files, instead of defining functions within Lisp, and usi MAKEFILE to write them out to disk, as is done in Interlisp. our opinion, good programming style refers not only to the str ture of functions, but also to the organization of functions in larger, logically connected units of code. If it is difficult for the p grammer to maintain full control over the appearance of lar hunks of code or files, it eventually becomes very hard for him keep a good cognitive map of his programs. Without such a m the programmer's understanding of his program, and the rate which he can modify it, diminishes as the system grows larg Maclisp and Franzlisp give the programmer complete control ov the organization of code. Obtaining such control in Interlisp is pc sible, but indirect, awkward, and time consuming.

Finally, we have come to increasingly believe that speed is as impo tant a tool for software development as any sophisticated user int face. AI has been notorious for ignoring efficiency issues. Ma research applications run in interpreted Lisp. Often there is decent Lisp compiler, and where there is, even good Lisp hackers not know the special compiler declarations needed to make good of compilation. As AI moves from developing theoretically intere; ing but computationally light systems into a market place th demands computationally-intensive, scaled-up systems that m run in real time, there will be an increasing need for "industri; strength Lisps". These would include, for example, compilers th are well documented, heavily optimized, that run in the san environment as the interpreter, and that are guaranteed to. produ code that runs the same as interpreted code. Presently no compi meets all these requirements, although Common Lisp [9] hopes meet these goals.

4. REFERENCES

[l] Bates, R. L.. Dyer, D., and Koomen, .J. Implementation Interlisp on the VAX. *Proc. 1982 ACM Symposium on L and Functional Programming,* Pittsburgh, 1082.

(2) Foderaro, J.K., and Sklower, K.L. The Franz Lisp Manu University of California, Berkeley, April 1982.

[3] Gabriel, R.P., and Masinter, L.M. Performance of Lisp System *Proc. 19S2 ACM Symposium on Lisp and Functional P ming,* Pittsburgh, 1982.

|4] Klahr, P., and Faught, W. S. Knowledge-Based Simulatic *Proc. AAA1-S0,* Stanford, 1980.

(5] Klahr, P., McArthur, D., and Narain, S. SWIRL: An Obje Oriented Air Battle Simulator *Proc. AAAI-82,* Pittsburg 1982.

[G] Klahr, P., McArthur, D., and Narain, S. SWIRL: Simulati Warfare in the ROSS Language. N-1885-AF, The Rand Cc poration, Santa Monica, September 1982.

[7] McArthur, D., and Klahr, P. The ROSS Language Manual, 1854-AF, The Rand Corporation, Santa Monica, Septemt 1982.

[8] Sandewall, E. Programming in the Interactive Environme The Lisp Experience. *ACM Computing Surveys* 10(1), Mar 1978.

[9) Steele, G. An Overview of Common Lisp. *Proc. 1982 AC Symposium on Lisp and Functional Programming,* Pitts 1982.

(10] Teitelman, W. Interlisp Reference Manual. Xerox Palo A" Research Center, October 1978.

|11] Teitelman, W., and Masinter, L. The Interlisp Programmi Environment. *IEEE Computer* 14(4), April 1981.

[12| Xerox PARC, Interlisp-D User's Guide. Xerox Palo As Research Center, February 1982.

Table 1. Time and space statistics for five implementations of ROSS and SWIRL.
(Timings made during January and February, 1983)

	Maclisp[1]	Franzlisp[2]	Interlisp-D[3]	Interlisp-VAX[4]	Interlisp-10[5]
1. Start a fresh LISP/ROSS.					
seconds of real time:	4	1.5	100	22	30
2. Invoke editor on a function or behavior.					
seconds of total cpu time:	–	–	20.8/4.6	8.5/.18	2.38/.1
seconds of real time:	5/2[6]	20/10[7]	58.0/5.0[8]	70/<1[9]	12/<1[10]
3. Store changed function or behavior out to disk.					
seconds of total cpu time:	–	–	53	24	7
seconds of real time:	3[11]	1[11]	76[12]	45[13]	20[14]
4. Load files for compiled SWIRL behaviors.					
seconds of total cpu time:	19.5	22	180	161	NA[17]
includes garbage collect time:	10.9	4	7	0	NA[17]
seconds of real time:	35	30	286	230	NA[17]
5. Load a complete SWIRL simulation environment.					
seconds of total cpu time:	10.9	50	550.7	310	133
includes garbage collect time:	.031	9	24.6	0	9
seconds of real time:	21	66	622.0	365	231
6. Run SWIRL (simulating a three-hour battle).					
seconds of total cpu time:	63.0	156	1171.2	669	480
includes garbage collect time:	31.6	21	161.0	0	82
seconds of real time:	133	199	1197.5	1295	1117
7. Make a SWIRL sysout.					
seconds of real time:	25	27	330	225	3
bytes (size), LISP & ROSS & SWIRL:	669,696	812,095	2,236,928	5,047,364	1,882,112
bytes (size), LISP only:	174,080	440,383[15]	1,849,856	4,069,876[16]	1,247,232

Notes for Table 1.

(1)'Maclisp times are averages over three runs on a DEC-20 (approximate load averages of 1.5) with 1.25 megabytes of physical memory.

(2'Franzlisp version is Opus 38 (April, 1082). Timings are averages over three runs on a VAX 11/780 (load averages of less than 1.5) with 4 megabytes of physical memory.

(3)Interlisp-D is Chorus release of February, 1983. Timings are averages over three runs on a Xerox 1100 Dolphin with 1 megabyte of physical memory. (Timings on a Dolphin with 1.5 megabytes memory averaged between 6% and 19% faster.)

(4)Interlisp-VAX is April, 1982 release. Timings are averages over three runs on a VAX 11/780 (approximate load averages of 1.1) with 4 megabytes of physical memory.

(5) Interlisp-IO timings are averages over three runs on a DEC-20 (approximate load averages of 1.5).

(6)First value is for first time LEDIT is invoked; second is for subsequent invocations. First value is higher because the LEDIT subfork must be created. No cpu time is given because editing is done outside of Lisp.

(7)First value is for first time EMACS is invoked; second is for subsequent invocations. No cpu time is given because editing is done outside of Lisp.

(8)Time to invoke the DEDIT local editor (first/second times).

(9) First value is for first edit, second is for subsequent edits. First value includes time to fetch interpreted function from a file into an otherwise compiled environment.

(10) Time to invoke EDITF (first/second times).

(11) [Time taken to save a file in EMACS editor.

(12)Time required to execute MAKEFILE on a file of 33,550 bytes after one function definition had been changed. RC option was not used.

(13) Time required to execute MAKEFILE on a file of 35,177 bytes after one function definition had been changed. RC option was not used.

(14)Time required to execute MAKEFILE on a file of 35,177 bytes after one function definition had been changed. RC option was not used.

(15)'Opus 38.50 (February, 1983) is larger (938,614 bytes for SWIRL sysout, 605,302 bytes for Franzlisp) with about the same performance.

(16)'A pre-release Inter lisp-VAX version of April, 1983 is smaller (3,214,304 bytes) and seems about 40% faster.

(17) Even though SWIRL code could be compiled and run inside of Interlisp-10, the loader generated an error when loading compiled SWIRL files. We have not been able to fix this error.