Design and Implementation of EASEL
A Language for Simulating Highly Distributed Systems

David A. Fisher
Carnegie Mellon University
Pittsburgh, PA

This paper describes some of the motivation, design and implementation strategies for a new simulation language with distributed system semantics. The system is intended for simulating, depicting, and gathering information about networks, software agents, and other active entities of the physical, electronic and software worlds, about their interactions, and about their collective global effects. The simulator is intended as a tool for research toward security and survivability in unbounded systems, but may have wider applicability. The language allows simulation of hundreds to thousands of semi-autonomous actors cooperating in a simulated world without global visibility nor central control. It supports a loosely coupled distributed network model in contrast with shared memory multiprocessor or multiprogramming models of most discrete event simulation systems. Special features include mobile code, near neighbors based on either explicit communication links, physical proximity or line of sight, actor subtypes with multiple inheritance, and a declarative depiction facility. The translator and interpreter for the language are hosted and targeted to a uniprocessor Macintosh under Mac OS. The implementation involves techniques and trade-offs borrowed from instruction set architectures, distributed operating systems, and compiler construction, as well as discrete event simulation.

Background and Motivation

The need for the Emergent Algorithm Simulation Environment and Language (Easel) derives from ongoing research in the area of survivable systems at the Software Engineering Institute (SEI). *Survivability* is defined as the ability of a system to fulfill its mission in a timely manner in the presence of attacks, failures, or accidents [EFL97]. *System* in this context is used in the broadest possible sense to mean networks and large-scale systems of systems. Survivability cannot be measured or achieved in the absence of knowledge of the system's purpose or mission, and of the critical functionality and quality attributes required to achieve that mission. Although security and survivability have overlapping goals and methods, they differ in many fundamental ways. Survivability is concerned primarily with system availability and mission fulfillment, while security is concerned primarily with confidentiality of information. Security employs a fortress approach that attempts to prevent all successful intrusions and assumes two categories of participants: trusted insiders and untrusted outsiders. A fundamental assumption underlying survivability is that no individual component of a system can be immune to all attacks, accidents, and design errors. Furthermore, concerns for survivability often arise in the context of modern networked systems, such as the Internet, where everyone is an insider whether trusted on not.

Much of the current interest in survivability derives from concerns for infrastructure assurance [PCCIP97]. How can one guarantee the continued operation and service availability for critical national infrastructures such as banking and finance, electric power production and distribution, transportation, and communications, in the presence of successful intrusions, failures, or attacks against components of these systems? Concern for survivability is primarily a phenomena of highly distributed network based systems. It is also unlikely that there can be effective solutions in the absence of large numbers of communicating nodes to replace functions provided by compromised nodes. Thus, unbounded networks constitute both the problem and solution spaces for survivability. An *unbounded network* is characterized by distributed administrative control without central authority, by limited visibility beyond the boundaries of local administration, and at any individual node, by incomplete information about the network's topology and component functions [EFL97].

One direction of our own research has been emergent algorithms [FL99]. Emergent algorithms take their analogy from biological and social systems in which each participant performs a simple local action involving interactions with other participants but without complete knowledge of either who else is participating or their roles. In these systems, extremely complex global properties emerge from the simple actions and interactions of the participants in the absence of central control or administrative authority. Examples include birds flying in flocks, the culture created by people of a region, and the national economy. Internet examples include the use of chat rooms, the overall governance of the Internet, and the combining of independently developed search tool in ways unanticipated by their authors. An *emergent algorithm* produces global system-wide properties that emerge from the collective actions of the participating nodes. These global emergent properties may be arbitrarily complex, but emerge from the interactions of large numbers of individual nodes each autonomously performing simple local actions. Failures and compromises in a few nodes should inhibit neither the global functionality nor other global qualities of the system. Emergent algorithms offer the possibility of satisfying critical mission goals in the presence of component failures and compromises whether or not known, and in some cases even when small but unknown numbers of nodes are intelligent rogues.

It is difficult, however, to envision the global consequences that derive from the interactions of simple local algorithms. Developing effective strategies and techniques for the design of emergent algorithms and their protocols of interaction to achieve needed mission functionality and nonfunctional global properties is similarly difficult. Existing unbounded networks are generally unavailable for experimental use, and their distributed character would make monitoring impractical. A system is needed in which emergent algorithms can be tested, monitored, analyzed, and their execution depicted in the context of simulated unbounded networks.

For a variety of reasons, existing simulations are inadequate for our purposes. Discrete event simulation languages typically support a shared memory multiprogramming model with an interleaved semantics. For unbounded networks, however, a loosely coupled multiprocessing model with near neighbor communication with parallel semantics is needed. Even the Star-Logo [Res95] language, which is intended for simulating emergent-like algorithms, requires that simulations be expressed in terms of central control and global visibility, often in contrast with the activities they simulate.

Thus, Easel is being developed as a tool for research in security and survivability in unbounded systems. Key requirements include an execution semantics consistent with unbounded networks and a rich set of data types to support a broad spectrum of simulated applications as well as monitoring, data collection and analyses of simulations. Other key requirements are an easy to use depiction facility that will help users visualize emergent properties and algorithms, and several features unique to security and survivability. The latter include mobile code, user definable protocols of interaction, and the ability to share node characteristics in arbitrary user specified combinations. Other important characteristics are that algorithms and protocols used in the simulation can be identical to those of the actual distributed application being simulated, that it be possible to simulate systems at many different levels of granularity in their component structure and in time, and that the language be able to process abstract descriptions of actors at any desired level of precision.

Language Design

In many superficial ways the language is similar to conventional programming languages of the 1960s, '70s and '80s. It has strong user definable types, bounds checking on array references, no implicit type casting, static grammatically embedded block structured scopes, and a syntax similar to those of Algol, Pascal and Ada. These characteristics make programs more understandable, easier to maintain, and less error prone when developing applications.

Because the language is intended primarily for applications that simulate unbounded networks, loosely coupled systems, and highly distributed infrastructures, the simulation mechanism of Easel provides a simulated environment of loosely coupled actors interacting without central control nor global visibility. Central control and global visibility are available to observers and facilitators outside the simulation.

Actors. An actor is any active entity of an Easel program, simulation, or processor within a simulation. At the program level (but outside of simulations), actors act as multiprogrammed tasks with the shared memory environment of a Macintosh application. The semantics of these tasks includes interleaved uniprocessor execution with FIFO-by-priority scheduling, explicit real-time wait on any thread, access to the local real-time clock, user interaction, and access to actual peripheral devices and persistent storage. At the level of simulations, actors are simulated entities of the physical world (e.g., a system administrator, user, intruder, automobile, bird, or the moon), of the electronic world (e.g., a computer, router, or peripheral device), or of the software world (e.g., a software agent or task). Within a simulated processor of a simulated computer network, actors serve as simulated tasks analogous to the real multiprogrammed tasks at the program level with interleaved semantics, local shared memory, and local simulated time clock.

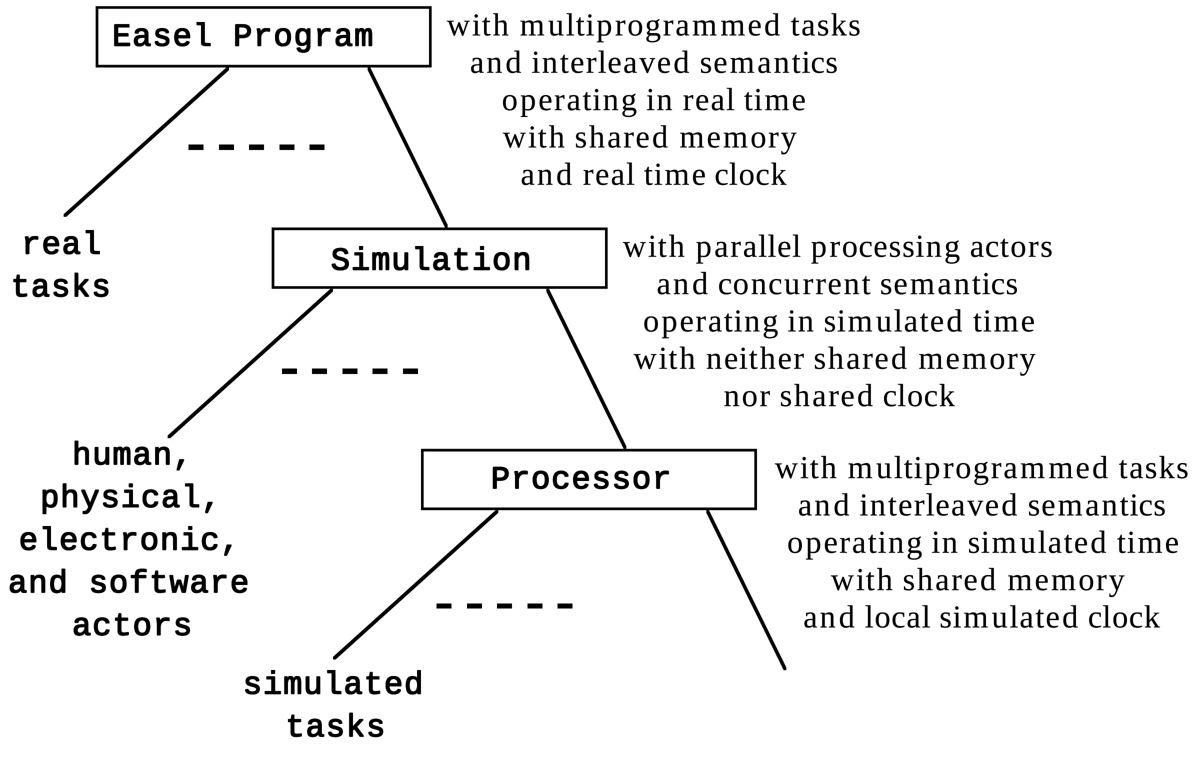
Easel Program
with multiprogrammed tasks and interleaved semantics operating in real time with shared memory and real time clock
real tasks
Simulation
with parallel processing actors and concurrent semantics operating in simulated time with neither shared memory nor shared clock
human, physical, electronic, and software actors
Processor
with multiprogrammed tasks and interleaved semantics operating in simulated time with shared memory and local simulated clock
simulated tasks

Figure A. Dynamic Structure of Easel Program Execution.

Neighbors. Each actor can interact directly only with its near neighbor and only in ways prescribed by their neighbor relationships. *Neighbor* relationships are protocols of interaction and are defined as types that can be associated with any actor. Thus, in a simulation of a communications network, a node's near neighbors might be only those nodes that are connected directly by communications links. The associated neighbor operations might include sending and receiving messages. In a simulation of birds in flight, a bird's near neighbors might be any bird or other object which the bird can see from its current position and heading. In Easel, near neighbor relation is a property of all parties to the relation, but for definitional purposes is generally described as a property of the actor affected by the relation. Thus, neighbor operations are defined in the local context of the type definition of the affected actor. In neighbor definitions, the actor type being defined is referenced by the pronoun "this", while the actor executing the definition at run-time is referred to by the pronoun "self". Neighbor relationships are often asymmetrical and can include any prefix predicate and computation involving "this" and "self".

Special Actor Types. An *observer* is a special type of actor that has global visibility throughout a simulation. An observer typically monitors and oversees the simulation, dynamically depicts the state of the simulation, or collects information about the simulation for later analyses, but does not correspond to any component of the system being simulated. An observer can read the state of any actor within a simulation. A *facilitator* is a special type of actor that can exercise central control over a simulation. A facilitator typically controls the simulation, sets initial states, varies characteristics and states dynamically, or serves as a surrogate for multiple or unimplemented actors when conducting simulations at more abstract levels. A facilitator can read or write the state of any actor within a simulation. A user interacting with the simulation from a terminal has the power, control and visibility of a facilitator. It can suspend the simulation, can execute an Easel statement in any context of the simulation, and can observe or change the state of any actor. A *processor* is a special type of actor that simulates a uniprocessor computer system with a multiprogramming operating system, priority scheduling, shared memory, and local clock. A simulated network may include any number of simulated processors. Each processor may have any number of local tasks. The overall dynamic structure of an Easel program is shown in Figure A.

Simulated Time. As might be expected in a discrete simulation languages for loosely coupled systems, actors in an Easel simulation execute in parallel in a world of simulated time. Unlike many discrete simulation languages, however, execution times in Easel are specified in the form of assertions associated with sequences of statements rather than as imperative delays at points between statements. This more abstract declarative specification corresponds closely to the intended interpretation and does not overspecify constraints on the implementation. It does, however, increase the likelihood of erroneous results from programs that attempt to exploit the actual implementation semantics. Although all actors of a simulation operate in the same simulated time frame, a shared global clock is not provided because it would violate the laws of physics in the world being simulated.

Built-in Types. Like most modern programming languages Easel provides a variety of common types and operations on scalars, composite data, and control structures. There are built-in scalar types for integers, floating point numbers, booleans and characters, and type constructors for arrays and records. Examples of mutable array and record types have a shared semantics and a copy operation that creates a similar structure but with separate identity. That is, mutable arrays and records correspond to what in some languages are called pointers to arrays and pointers to records. Easel also provides union types which are easier to use and less error prone than variant records. There are structures for conditional, iterative, selective, and recursive control. Pointers exist only at the implementation level and are not a language concept.

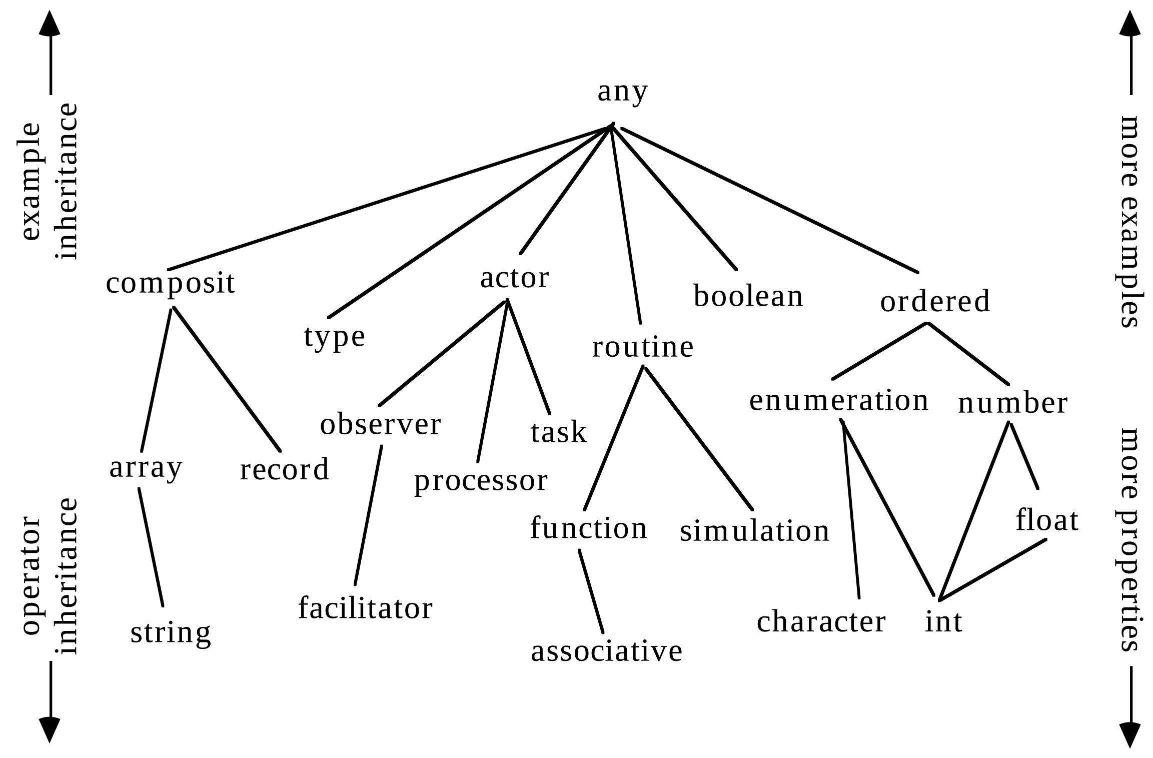

Figure B, Partial Type Structure of Easel.

Type System. The Easel type system is a generalization of strong user defined types. It is based on a principle of property-based types [BFM87, MS91] in which *types* are sets of properties instead of sets of objects. Each *property* describes some characteristic of the type and can be interpreted as a theorem about examples of the type. An *example* of a type is any object that satisfies all of the type's properties. Examples of a type may have, and usually do have, other characteristics not included in their type description. Thus, any abstraction can be represented as a type. Easel types closely approximate the concept of type in everyday use of natural languages. Easel types do not necessarily impose any particular representation or implementation regime on examples of the type.

A portion of the Easel type structure is shown in Figure B. An object can be simultaneously as example of any number of types. The number 5, for instance, is an example of the types: positive, odd, and anything. Consequently, multiple inheritance is both transparent and automatic with each object inheriting the operations for all types for which it is an example. For example, all operations defined on positive numbers and all operations defined on odd integers may be applied to 5. The predefined types include the universal type; any and ype", the type of all types. Although types are in some ways first class objects of Easel, few operations are defined on types. These include type union, type intersection, and the is-an-example-of predicate "in".

Mobile Code. Because Easel will be used to simulate security aspects of computer systems that include mobile code (e.g., Java applets, viruses, Trojan horses, and worms), code is a built-in type of the language. Actors may compute code, send it in simulated messages, and execute it in any local context of an Easel program.

Special Protocols. Although any arbitrary neighbor relation can be defined within an Easel program, three particularly useful neighbor relations are predefined. These are directly linked neighbors in a communications network, near neighbors by physical proximity in a two- or three-dimensional space, and line of sight neighbors within a range of angles in two-dimensional space.

Depiction. Easel provides a depiction facility for dynamic display of the simulations. Visualizations use declarative specifications and clearly delineates between the responsibilities of application developers and those of users interacting with the application. Applications determine what views are available to the user. Each *view* is an unbounded two-dimensional depiction world. Drawing objects for each static and dynamic entity of the simulation can be placed at any point in a view. Typically, each actor would change the image or location of its depiction with knowledge of neither other actors' depictions nor which views are currently associated with user windows. A *window* is a portal displaying a portion of some view on the users screen. A user may create any number of windows and must associate each one with some view provided by the application. The user dynamically determines the region and magnification of the portal within the view. Applications generally do not have knowledge of the state of windows.

Implementation Strategy

The Easel implementation draws techniques from the design of instruction set architectures (ISA), distributed and uniprocessor operating systems, compiler construction, user interfaces, and discrete event simulation. Programs written in the language are compiled into byte-codes for a pseudo machine. The pseudo machine is emulated in PPC code. The system also has a source language debugger that is separate from, but interacts with, both the compiler and the pseudo machine emulator. The implementation structure of the Easel simulation system is shown in Figure C.

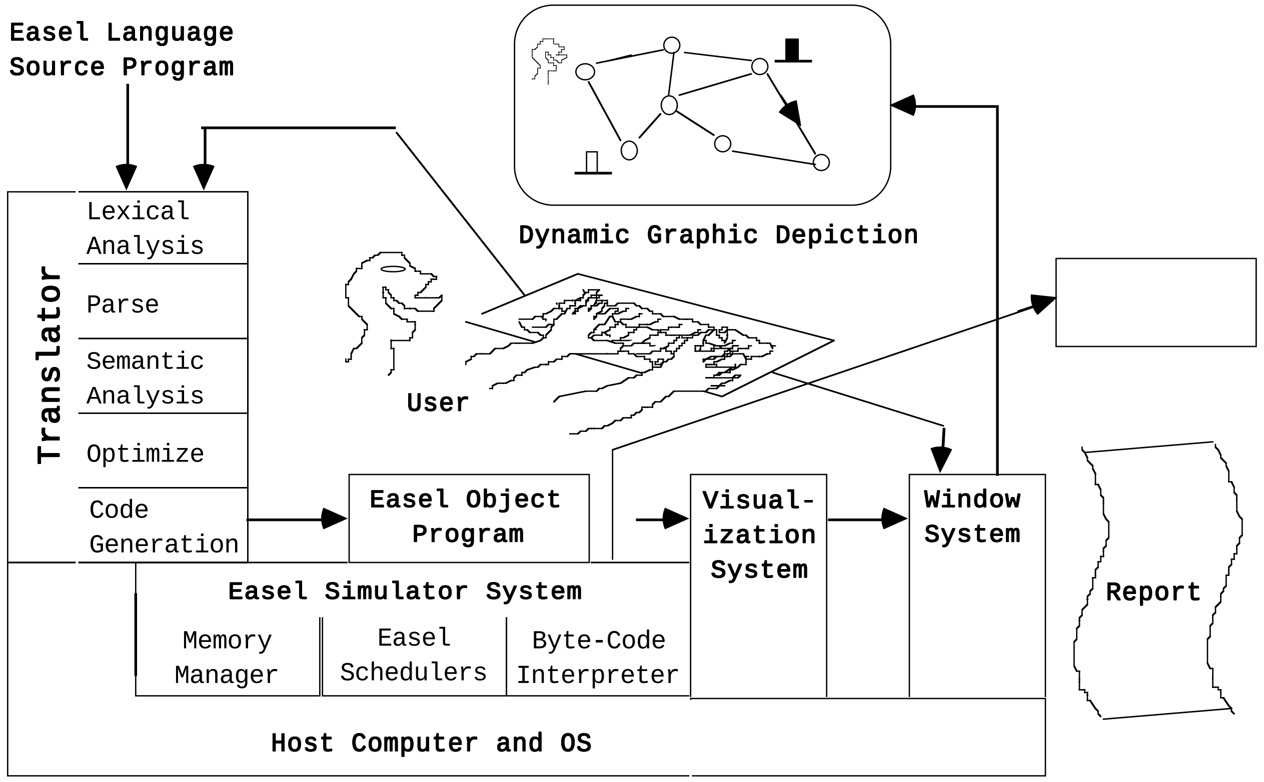

Figure C. Implementation Structure of Easel Simulation System.

```
property ::= property type                       -- included properties
  | [ var ] id : type [( = | is | := ) exp ]     -- state properties
  | when type then property                      -- neighbor properties
  | define [{ property ; }] end [ id ]           -- compound property
type ::= exp                                     -- type valued expression
  | [ type ] ( routine | function | simulation ) fpl -- routines
  | ( actor | observer | facilitator | processor | -- actors
      type ) [ fpl ]                             -- type type
  | record [{ id : type }] end record [ id ]     -- record type
  | type . . .                                   -- last formal parameter only
fpl ::= ( [ [ id : ] type { ; [ id : ] type } ] ) -- formal parameter list
stat ::= null                                    -- null statement
  | ref := exp                                   -- assignment statement
  | exp ( [ exp [{ , exp }] ] )                  -- call on routine
  | id exp                                       -- call on unary routine
  | wait until exp                               -- wait condition or time
  | [ define [{ property ; }] [ takes exp ]      -- duration specification
      begin { stat ; } end [ id]                 --      local block
  | if exp then { stat ; } [{                    -- conditional statement
     elsif exp then { stat ; } }] [ else { stat ; } ] end if
  | case exp of { when exp => { stat ; }}        -- case statement
     [ otherwise { state ; } ] end case
  | stat ( when exp | where property )
  | ( id : | for id : type ) loop { stat ; } end loop [id ] -- loop statement
exp ::= id | literal | ( exp )                   -- atomic expressions
  | exp . id | exp [ exp [{ , exp }] ]           -- array and record reference
  | id ( [ exp [{ , exp }] ] )                   -- function call
  | [ [ exp [{ , exp }] ] ]                      -- composite value constructor
  | uop exp                                      -- unary operator call
  | exp bop exp                                  -- binary operator call
uop ::= id | new | not | op                      --unary operators
bop ::= and | or | xor | in | op                 -- binary operators
op ::= ^ | & | * | / | + | ‘                     -- operator symbols
  | ≤ | = | ≥ | ± | - | . .
```

Figure D. Grammar for Easel Language.

Compiler. The compiler includes subsystems for lexical analysis, parsing, semantic analysis, optimization, and code generation. Lexical analysis uses a simple state machine structure. A recursive descent parser is used for performance reasons. The parser is table driven for greater compactness and for flexibility during development. For both performance reasons and to overcome the traditional error recovery problems of recursive descent parsers, the parser is very permissive in allowing any legal syntactic structure of the language to be used anywhere some legal structure is allowed. Thus, during the parse phase, statements can appear anywhere expressions or declarations are allowed and vice versa. User errors of this kind are reported later in the semantic analysis phase when more diagnostic information is available. The grammar is given in Figure D.

The semantic analysis phase of the compiler operates on the parse tree and performs all legality checks that can be done at compile time. It constructs the symbol table for the program, determines and enforces the visibility rules, annotates the parse tree with information that will aid optimization and code generation, and determines the representation and allocation scope for each program entity. To minimize the development time, optimization is not be included in the initial implementation. It is anticipated that optimization eventually will be applied in the form of transformations on the semantically analyzed parse tree representation. The optimizer could also evaluate any components of a program that depend only on literals and compile-time determinable values. This latter optimization together with the use of memo functions at run-time could help performance significantly in a system in which hundreds or thousands of actors are repeating, at least in part, the same computations.

Code generation involves a single pass over the semantically analyzed parse tree to emit the appropriate byte codes for each program component that is to be evaluated at run-time. Because the byte codes were chosen to support the Easel language, the correspondence will be one-to-one for most frequently used operations and with substitutions in code macro patterns for control structures and infrequently used features.

Pseudo Machine. The pseudo machine is a stack-oriented zero-address byte-code machine similar to those for Java and UCSD-Pascal, but with a more compact representation. The pseudo machine supports four levels of reference and assignment: Most variable references and assignments likely will be to local variables and formal parameters of the local environment of the current routine's stack frame (i.e., relative to pseudo machine register "lenv" in Figure E). Local state variables of the currently executing actor are accessed relative to the pseudo machine register "self". Local state variables of the neighbor when executing an operation defined within a neighbor are accessed relative to the pseudo machine register "this". Global variables of an Easel program are accessed relative to the pseudo machine register "genv". Most references in Easel programs will be to constants rather than variables. Constants are generally accessed relative to their respective code segments, with the code segments themselves accessed as global variables. Each program

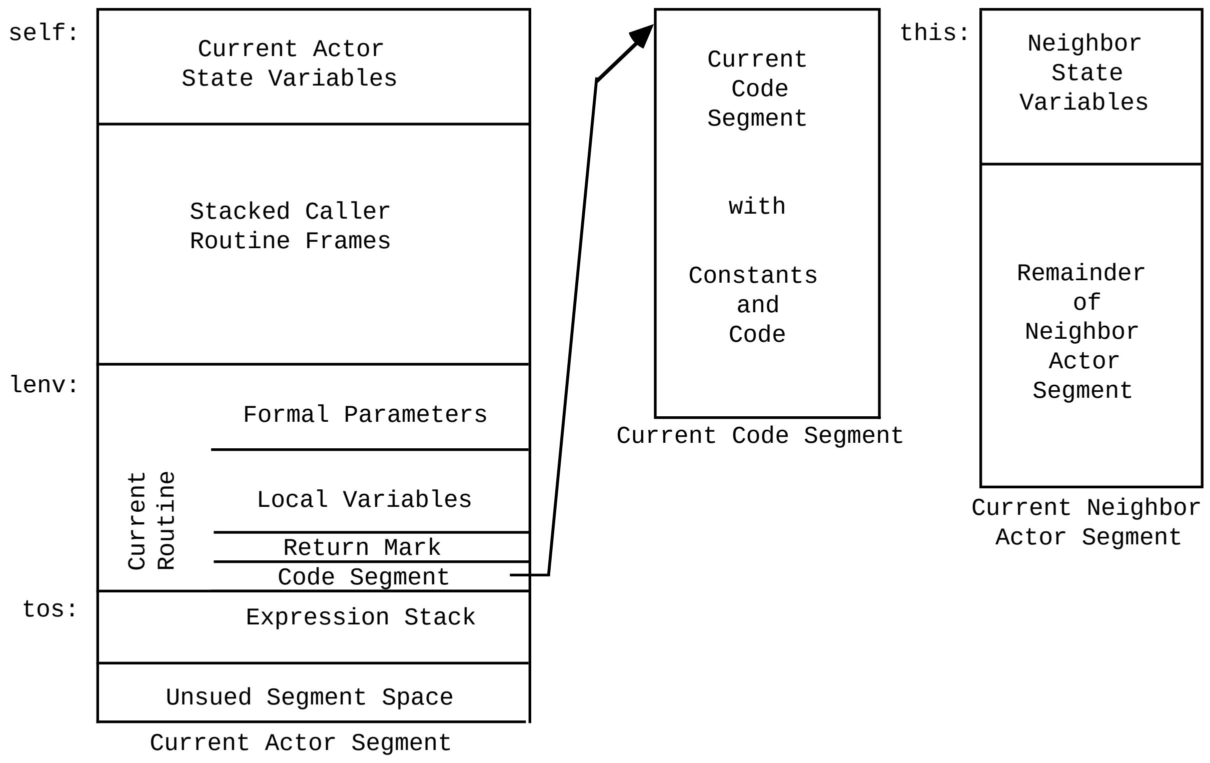

Figure E. Pseudo Machine Stack Structure.

variable occupies exactly one word with larger structures implemented through pointers (see Figure F). In combination, these characteristics enable most variable references and assignments to be encoded as a single byte using a four-bit relative address (see Figure F). Code size is important in an interpretative system because it correlates with the number of interpretation cycles required. It is important in any system because it significantly reduces main memory requirements, which is often a major factor in overall system performance.

Two-byte instructions typically are used for references to constants and compile-time program entities, for most control structure operations, and for calls on infrequently used language and program routines (see Figure F). Frequently called language and program defined routines are encoded as eight-bit instructions. Array reference is a byte instruction with both the base address and the index as computed parameters. Record reference is usually a two-byte instruction with a base address parameter and an offset as part of the instruction. All references and assignments are bounds checked.

The pseudo machine has base registers for the four standard access levels (i.e. local, self, this and global), the current top of the stack, the current code segment, the current processor environment, and the current simulation.

Each actor has a frame stack and an execution thread (See Figure E). Each frame corresponds to one level of recursion in the routine calling structure of an actor. Each frame consists of a formal parameter region, a local variable region, a return mark, and an expression stack. Actual parameters are computed into the expression stack of the caller where, at the time of call, they are reinterpreted as formal parameters of the called routine. Because the formal parameters and local variables are contiguous within each frame, they are implemented as a single region (i.e., local environment). The return mark contains the location of caller's local environment, the relative address of the caller's instruction, and the location of the caller's code segment. The first bytes of each routine's code contain the number of formal parameters and number of local variables.

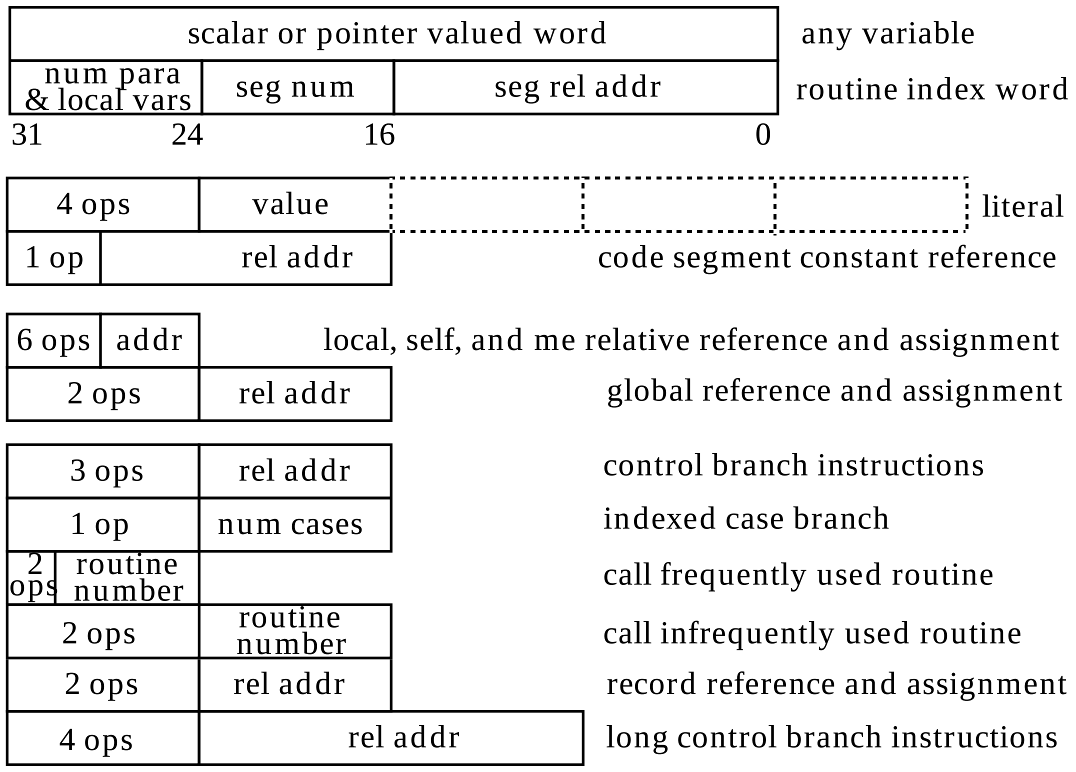
scalar or pointer valued word
any variable
num para & local vars
seg num
seg rel addr
routine index word
31
24
16
0
4 ops
value
literal
1 op
rel addr
code segment constant reference
6 ops
addr
local, self, and me relative reference and assignment
2 ops
rel addr
global reference and assignment
3 ops
rel addr
control branch instructions
1 op
num cases
indexed case branch
2 ops
routine number
call frequently used routine
2 ops
routine number
call infrequently used routine
2 ops
rel addr
record reference and assignment
4 ops
rel addr
long control branch instructions

Figure F. Variables and Byte Code Instruction Formats.

Operating System Level Features. The Easel system provides memory management with garbage collection, a multiprogramming real-time scheduler, a simulated distributed multiprocessor scheduler, a simulated multiprogramming scheduler, declarative access to quick draw, a specialized windowing and dialog system, and access to Macintosh files and devices. The memory manager allocates all storage used within the Easel system, except where the Mac OS requires its own allocation. The Easel managed memory can be any portion of the Macintosh memory and need not be contiguous. For execution efficiency, all allocated blocks are referenced through pointers, rather than handles, but are relocatable through a scheme that temporarily uses a handle between the time a block is relocated and the next garbage collection. Allocation is by first fit from the previous allocation. This minimizes memory fragmentation and maximizes performance. Separate routines are used for garbage collection and for compaction. Either or both can be used to obtain a contiguous block large enough for the current allocation. Garbage collection is a two-pass system which first marks all storage accessible from the program root followed by a sequential pass through managed memory reclaiming any unmarked areas.

The Easel system has several of the characteristics that support security and reduce vulnerabilities. These include bounds checking on memory accesses, only fixed size data in the stack, run-time knowledge of the representation of all words in allocated memory, and the use of logically segmented memory. At the implementation level, each memory segment has an explicit tag which can be used to determine the representation of each word within the segment. Segments whose words are nominally pointers to other segments, use the high order bit in each word to specify whether the word contains a scalar or an address. Frame stack words, for example, are type tagged in this manner. The actor visibility rules also provide compartmented data that can be violated only by neighbor relationships defined for that purpose.

Mobile Code. The interface to an interactive user acting as a facilitator or using the debugger is implemented by passing the user's textual input through the lexical analysis and parse phases of the compiler one statement at a time. This incremental compilation is then completed in the context of the appropriate preexisting local program context. The resulting byte code is interpreted in the context of the corresponding frame in the on-going execution. A similar strategy is used when simulated mobile code is computed and later executed in a simulation.

System Structure. The compiler, operating system, depiction system, debugger, user interface, and run-time language features are implemented as programs of the pseudo machine and managed by its operating system (see Figure C). For efficiency, many of them are represented in PPC code instead of Easel byte codes. In all cases, however, they conform to the implementation conventions for Easel applications and depend on the Easel memory manager and scheduler.

Platform Choice. The system could have been hosted on any platform. The primary requirement was for a widely available machine with reasonable system software and software development tools. After doing some preliminary development on both the PC and Macintosh using CodeWarrior, the Macintosh was our preferred choice. It provides a fast, affordable, reliable, easy to maintain, and easy to use hardware and software environment with superior graphics capabilities and consistent interface conventions. Also, our prior experience with Inside Macintosh and software development on the Macintosh, may help the development schedule.

The absence of protected memory and true multitasking in the MacOS are not additional impediments in this project because our requirements for specialized network semantics in these areas could not be met by any uniprocessor or shared-memory multiprocessor operating system. Also, Easel requires very high performance memory management and task switching which probably can be achieved only in a system specialized for Easel's requirements and is unlikely from an general purpose operating system.

References

[BFM87] D.Baker, D.Fisher, D.Mundie, J.Shultis and F.Tadman. *Toward Full Spectrum Languages: A New Approach to Software.* Incremental Systems Corp. Technical Report TR871002, October 1987, 44 pp.

[EFL97] R.J.Ellison, D.A.Fisher, R.C.Linger, H.F.Lipson, T.A.Longstaff, and N.R.Mead. *Survivable Network Systems: An Emerging Discipline.* Software Engineering Institute Technical Report No. CMU/SEI-97-TR-013. November 1997.

[FL99] D.A.Fisher and H.F.Lipson. *Emergent Algorithms -- A New Method for Enhancing Survivability in Unbounded Systems.* Proceedings of the Hawai'i International Conference On System Sciences, January 5-8, 1999, Maui, Hawaii.

[MS91] Proceedings of the Workshop on Informal Computing, May 29-31, 1991, Santa Cruz, California, D.A. Mundie and J.C. Shultis, editors.

[PCCIP97] *Presidential Commission on Critical Infrastructure Protection, Critical Foundations -- Protecting America's Infrastructures.* Presidential Commission on Critical Infrastructure Protection, October 1997, 173 pp.

[Res95] M.Resnick. *New Paradigms for Computing, New Paradigms for Thinking.* Computers and Exploratory Learning, A.diSessa, C.Hoyles and R.Noss, editors, Springer-Verlag (1995).

Acknowledgments

Howard Lipson has provided ongoing feedback on the language design and contributed especially to the design of the neighbor relation facility. Tim Shimeall participated in the experimental implementation done in the summer of 1998 and contributed especially to the design of the scheduling system. David Mundie and Bob Ellison have been generous reviewers of earlier versions of this paper and have suggested several improvements incorporated here.