StarLogo: Architecture and Implementation

Andrew Begel
University of California, Berkeley
abegel@cs.berkeley.edu

Brian Silverman
MIT Media Laboratory
bss@media.mit.edu

10th November 1999

Abstract

StarLogo is a programmable modeling environment for exploring decentralized systems. These systems, which can be described using simple rules, exhibit emergent behaviors which arise out of the interactions of hundreds or thousands of individual computational elements. This paper describes the design, architecture, and implementation of the Macintosh version of StarLogo. We implemented a set of high-performance, multi-processing Logo virtual machines which run on low-end desktop computers. We also explored subtle design issues in context switching and in the unification of several different parallel and serial models of computation. The resulting programming language and environment is efficient, simple to understand, and easy to use, even by novice programmers.

1 Introduction

StarLogo [9] is a programmable modeling environment for exploring decentralized systems. These systems, which can be described using simple rules, exhibit emergent behaviors that arise out of the interactions of hundreds or thousands of individual computational elements. Researchers have been turning towards these decentralized models to simplify their work and make them computationally accessible on today's computers.

For the past ten years, our research group has stressed the importance of decentralized modeling [8, 10], and worked towards making modeling environments accessible to students [1, 7]. In particular, our goal was to build an environment that encourages kids to build their own models as well as simply explore models built by experts.

To achieve this goal, we have designed, architected, and implemented the StarLogo programming language and environment for the Macintosh platform. StarLogo is a parallel dialect of the Logo programming language [6]. Logo, itself a dialect of Lisp, was invented in the 1960's by Seymour Papert. Logo was designed to make programming more accessible to kids. This was achieved through the use of simpler syntax and simpler data structures. In addition, the Logo Turtle, a virtual creature that responds to users' commands, was introduced to enable kids to explore computational geometry and programming in a more concrete way.

The StarLogo dialect of Logo was created by Mitchel Resnick in 1989. A large number of *turtles* roam a grid of *patches* while the *observer* watches. All three entities – turtles, patches and observer – may run Logo code, however, each has a slightly different set of primitive operations they may perform. Turtles can move around the grid of patches and communicate with other turtles, patches, and the observer. The patches can do everything a turtle can, but they can't move. There is only one observer, and it may query or affect the state of individual turtles or patches or the entire world. It can also control the global parameters that a user has programmed into their model.

StarLogo is an extremely parallel environment. There are thousands of turtles and patches all executing code at the same time. Its first implementation was on a Connection Machine massively parallel supercomputer with 16,000 processors. This gave us plenty of parallelism, but its accessibility to schoolchildren left something to be desired.

Our challenge was to bring this environment to the moderately under-powered, single processor, desktop computers that were common in schools at the time. The implementation had to be quite efficient because the turtles and patches were supposed to look like they're executing in parallel. Minimizing the memory requirement was also important because we needed to run thousands of processes on a computer with only 8 MB of RAM.

To run parallel processes on a uniprocessor, you need some form of multi-processing capability. Most operating systems at the time included such support, but their implementations were quite heavyweight. Their process abstraction was never intended to scale, either in performance or in memory usage, to thousands of active processes. Some operating systems supported multi-threading (a form of lightweight process), but even this was only meant for small numbers of threads. We needed a system with extremely fast context switching and very low memory overhead.

We implemented a set of multi-processing Logo virtual

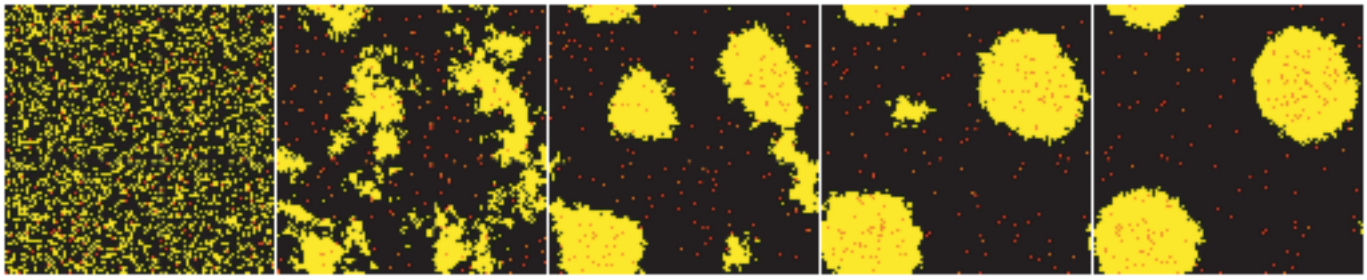

Figure 1: This figure shows the time evolution of the termites project. At the beginning (a), all termites and wood chips are randomly scattered over the patches. As the termites pick up and drop the wood chips (b), the number of piles begins to decrease (c). Small piles get smaller (d) and eventually disappear (e). If we run this further, all of the wood chips will end up in one pile.

machines in assembly language. We use a handful of high-level primitives to implement the Logo language, and use a large number of application-specific primitives to run the turtles and patches. Our context-switching policy removes the need for most user-visible synchronization from the language, which improves its ease-of-use for novice programmers. In addition, we present the user with a unified programming model that allows him to write code for all three entities without any additional syntax.

With the virtual machine and language in place, we wrote a StarLogo compiler in Macintosh Common Lisp and constructed a GUI around the pair. Our first release was in June, 1994, and so far around 10,000 people have downloaded StarLogo 1.0 and subsequent versions. Currently, we are working on a Java version of StarLogo with more features, but still based on a virtual machine model.

In the remainder of this paper, we motivate, describe, and reflect on the StarLogo architecture and implementation. First, we give an introduction to the StarLogo language through an example. Then, we present an overview of our basic virtual machine design before going on to explain how we realized this design on the Macintosh platform. Later, we describe in detail the implementation of our parallel virtual machines and discuss the models of parallelism that each uses. Finally, we talk about our recent development efforts with our Java version of StarLogo and conclude.

2 Termites Example

To familiarize readers with StarLogo, we present a simple StarLogo project about termites. This project is inspired by the behavior of termites gathering wood chips into piles. The termites follow a set of simple rules. Each termite wanders randomly. If it bumps into a wood chip, it picks the chip up, and continues to wander around. When it bumps into another wood chip, it finds a nearby empty space and puts its wood chip down. We show a run of this simulation in Figure 1.

In general, the number of piles decreases with time. Why? Some piles disappear, when termites carry away all of the chips. And there is no way to start a new pile from scratch, since termites always put their wood chips near other wood chips. So the number of piles must decrease over time. (The only way a "new" pile starts is when an existing pile splits into two.)

We look at the StarLogo source code to get a feel for what the language feels like. The following is the setup procedure for termites.

```
to setup
clearall
if (random 100) > 80 [setpatchcolor yellow]
create-turtles 200
setcolor red
setxy random (screen-edge * 2)
      random (screen-edge * 2)
end
```

First, we kill all of the turtles and set the patch colors to black. Then, on each patch, we throw a random 100-sided die. It it exceeds a threshold, we set the patch's color to yellow – we give it a wood chip. We then create 200 turtles, set their colors to red and scatter them around the screen.

```
to go
search-for-chip
find-new-pile
find-empty-spot
end
```

The go procedure is the main loop. First, a termite looks for a wood chip and picks it up. Then it wanders until it finds another wood chip in a pile and finds a place to put it down.

```
to search-for-chip
if patchcolor = yellow
  [stamp black jump 20 stop]
wiggle
search-for-chip
end
```

In search-for-chip, a termite wanders around, wiggling, until it is standing on a yellow patch. That means there's

a wood chip there. It picks up the wood chip and jumps 20 turtle steps away.

```
to wiggle
forward 1
right random 50
left random 50
end
```

A termite wiggles by moving forward one turtle step, then turning right and left a random number of degrees.

```
to find-new-pile
if patchcolor = yellow [stop]
wiggle
find-new-pile
end
```

The termite then wiggles around until it finds a pile to put down the wood chip.

```
to find-empty-spot
if patchcolor = black
  [stamp yellow get-away stop]
setheading random 360
forward 1
find-empty-spot
end
```

A termite doesn't want to put a wood chip down on top of another one, so it moves forward in a random direction until it finds an empty patch. Once it finds that spot, it stamps the patch to make it yellow (giving it the wood chip), and then jumps away to look for new wood chips.

```
to get-away
setheading random 360
jump 20
if patchcolor = black [stop]
get-away
end
```

To get away, a termite keeps jumping 20 steps in random directions until it lands on a spot without any wood chips. Then it starts the cycle over again, looking for another wood chip to pick up.

3 Design Overview

StarLogo has more going on in parallel than most other computer environments. In StarLogo, there are thousands of computing elements. Several thousand turtles move about on a grid of over ten thousand patches, each one executing code. On the CM2, we had 16,384 processors running at 8 MHz. There was plenty of parallelism for our needs, so not much thought was given to implementation efficiency.

In 1994, we moved to a Macintosh platform. Our primary hardware targets were systems at the performance level of a Mac IIfx, using a single 25 MHz 68020 processor with 8 MB of RAM. A single processor would have to be made to emulate thousands, something that we recognized would be a challenge from the start.

Our first prototype uniprocessor implementation of StarLogo was written in Macintosh Common Lisp. Turtles and patches were each Common Lisp objects which encapsulated their state and a reference to a function containing the user code. Our scheduler looped over each turtle and patch and executed each one's function code to completion before moving on to the next. Performance was pretty poor. Simulations with more than 100 turtles were almost too slow to be usable, and any operation on patches was so glacially slow that you could actually see the screen updates occurring whenever the patches changed their color.

Motivated by this failure, we resolved to think much closer to the metal. Process switching had to be much faster, at most tens of machine instructions. Processes also had to be extremely small. All of our data structures have a parallel cost – the addition of 1 byte in a patch costs 10 KB of real RAM.

We turned to assembly language programming and implemented a process scheduler in 68000 assembly language. This scheduler ended up being at the core of a virtual machine that we used to implement the turtles and patches.

This virtual machine is different than those used by other byte-coded languages like Smalltalk [3] and Java [4] because StarLogo is not a general-purpose programming language. The virtual machine uses just a handful of high-level primitives that match closely with the Logo language. There are also around 200 application-specific byte codes which comprise all of the turtle and patch functionality. Unlike Smalltalk or Java, StarLogo is purely procedural, so we didn't need to support object-oriented dynamic dispatch. And since StarLogo requires users to declare all variables as compile-time globals (only requiring a simple static memory allocation algorithm), we didn't need to add in a sophisticated dynamic memory allocator or garbage collector.

All of these features made our virtual machine extremely simple and enabled it to be much smaller than those of other languages. Our entire 68K virtual machine fits into 12K of object code. The PPC version takes 20K. We probably didn't have to go as small as we did, but we have to admit that we were influenced by other research projects in our group. For many years, our group has been building Logo virtual machines for extremely small electronic devices running on very tiny processors.

The StarLogo virtual machine is very similar to the one used in the Cricket project [5]. A Cricket is a tiny computer, powered by a 9 volt battery, that can control two motors and receive information from two sensors. Crickets are equipped with an infrared communication system that allows them to communicate with each other. Powered by a Microchip PIC

processor, the virtual machine in Cricket Logo takes about 1000 words of instruction memory and only a few dozen words of RAM.

Even though a Mac IIfx is much more powerful than a PIC microcontroller, it isn't unreasonable to use a similar virtual machine for the Cricket and StarLogo. Though the Mac is much faster than the PIC, the PIC only has to run one process, while the Mac has to run thousands. In both cases, the virtual machine has to be very efficient in terms of execution speed as well as memory footprint.

The next several sections describe our virtual machine implementation in detail. We start with the virtual machine as seen by a single turtle, and later extend the description to include multiple turtles, patches, and the observer.

4 One Turtle Virtual Machine

There are three main components to the virtual machine. The first is the turtle – an array of process state, turtle state variables, a statically allocated heap, and a stack. The second piece is the dispatch loop and primitive set, and the third is the scheduler, which controls execution of code from various sources in the StarLogo user interface.

4.1 A Look Inside a Turtle

The state of a StarLogo turtle consists of a fixed-size array divided into five parts.

The top of the turtle holds the process state: the instruction pointer, stack pointer, base of frame pointer, stack limit pointer, and the "done" state (set to true when the turtle is finished running its code).

Next come the turtle state variables: the coordinates of the turtle on the screen, its id number, and its color, heading, breed, shape, pen state (up or down), and visibility (shown or hidden).

After those are variables for keeping track of the turtle's connection to other turtles: a next and previous turtle pointer (all live turtles are kept in this linked list), an underme and overme pointer to keep track of turtles stacked on top of a particular patch, and a boolean that indicates if this turtle is alive (useful for generating proper error conditions).

Finally, the remaining two parts of the turtle consist of a statically allocated heap and an upward-growing unified control and data stack that begins at the bottom of the turtle.

Both our 68K and PPC implementations of StarLogo statically allocate 4096 turtles at boot-time. By default, each turtle is 256 bytes long, so we allocate a 1 MB array to hold them. This gives each turtle 17 user variables in its heap and 28 words of stack space. While this might sound small, in practice, most users hardly ever run out of user variables (this limit may be increased at run-time). Any stack overflows are almost always due to an infinitely recursing non-tail-recursive procedure (and in the case where a deeply-recursive procedure is desired, the user may increase the stack size at run-time).

4.2 Primitives

There are two sets of opcodes in this virtual machine: system opcodes and opcodes that correspond to user-level primitives. There are very few system opcodes. They are used for procedure invocation and returns, and for handling numbers and instruction lists.

The system opcodes are

1. `num` (pushes a fixnum immediate on the stack),
2. `ufun` (call a user-defined procedure),
3. `ufun-tail` (call a user-defined procedure tail-recursively),
4. `stop` (non-local exit from the current procedure),
5. `output` (non-local exit from the current procedure and return a value on the stack),
6. `getl` (read the value of a local variable (identified by a compile-time offset) and push it on the stack),
7. `setl` (set the value of a local variable).
8. `list` (identify the start of instruction list),
9. `eol` (identify the end of an instruction list), and
10. `eolr` (identify the end of an instruction list that returns a value on the stack).

There are over 200 user-level primitives that are specific to the turtle virtual machine. These include primitives for moving the turtle around the screen, communicating between the turtles, patches and the observer, observing and modifying the turtle's environment, reading and writing turtle, patch, and observer state and user variables, and executing mathematics and control functions.

These functions are described in the StarLogo reference manual [2] and will not be further explained here except in the illustration of a code example.

4.3 Dispatch Loop

The 68K and PPC versions of StarLogo take two very different tactics to the dispatch loop implementation. First, we explain the basic model of the 68K dispatcher and then indicate why we changed it for the PPC.

In the 68K StarLogo, we use a byte-coded jump table dispatch loop. All primitives, including the system primitives, are labeled in a jump table whose offsets are accessible from our runtime system.

Each turtle's instruction pointer points into an instruction list downloaded by the runtime system. An instruction list is a linear sequence of opcodes representing a postfix form

of the user-level program. For example, if the user types in `forward 3 + 4` to make the turtle go forward 7 turtle steps, the compiler translates this into this instruction list: [`3 4 + forward`].

Instruction dispatch reads the next 16-bit integer out of this buffer and uses it as an index into the jump table. After running each command we check whether or not to continue execution. Since we are running this virtual machine on a sequential computer in which we fully take over the processor, we need to "come up for air" every so often. We do this every 60th of a second. This gives enough time for the Macintosh event dispatch loop to process any pending user events.

Control flow primitives, such as `repeat`, `if`, and `ifelse` have instruction list parameters. For example, if the user types in `repeat 100 [setcolor color + 1]`, the compiler produces this instruction list: [`100 list 4 color 1 + setcolor eol repeat`]. The `list` byte code is compiled at the beginning of an instruction list argument. It pushes the starting address of the list on the stack and uses the compiler-supplied length of the list to jump past the `eol`. When the control flow primitive (in this case, `repeat`) wishes to execute the list, it saves the current instruction pointer on the stack, pops the list's address off the stack and uses that address as the new instruction pointer. Execution proceeds until the `eol` or `eolr` is encountered. This end of list instruction pops the old instruction pointer off the stack, restores it to active use, and jumps to it. This way, the turtle continues running where it left off.

When we ported StarLogo to the PowerPC, we read that its branch unit folded out branches and made static branches effectively free. So we chose a different tactic; we "unroll" the dispatch loop in the compiler. Instead of encoding each primitive call as an index into a jump table, we directly inline a PPC PC-relative branch instruction with the correct offset to the primitive. In addition, we inline several system opcodes (`num`, `list`, `eol`, `eolr`, `getl`) in PPC assembly, as well as inline the procedure prologue code. In essence, our instruction buffers do not hold byte codes anymore, but in fact, hold executable PPC code.

In the next section, we expand the description of the virtual machine model to support multiple turtles in parallel, as well as explain how we support parallel patches and the sequential observer in runtime system.

5 The Complete Virtual Machine

Each of the three kinds of entities in StarLogo have their own idiosyncratic rules for parallelism. This section describes these rules, how they operate, and how they interact.

5.1 Multi-Turtle Virtual Machine

StarLogo turtles are supposed to appear to be acting in parallel. In order to maintain the fiction of parallelism, we context switch among the turtles very quickly. There are two ways we could do this. We could support preemptive multiprocessing, in which a timer interrupt every n milliseconds would cause us to context switch, or we could use cooperative multi-processing and only context switch at carefully chosen control points.

StarLogo chooses the latter for several reasons. First, under preemptive multi-processing, synchronization issues would become exposed to the user. For instance, the following common StarLogo idiom would need user-visible locks:

```
if count-turtles-here > 1
  [mate-with one-of-turtles-here]
```

This turtle is looking for another to mate with. It looks on the current patch to see if there is any other turtle there. If there is, it mates with it by asking for its turtle id and calling a user-defined procedure `mate-with`. Under preemptive multi-processing, it is possible to context switch between the condition `count-turtles-here > 1` and the consequent of the `if` statement, `mate-with one-of-turtles-here`; if this happens, the other turtle might move before this turtle has had to chance to mate with it.

To avoid this kind of problem, we only allow context switches at "safe" times such as the end of each command. A *command* is what we call a primitive or user function that doesn't return a value. In contrast, a *reporter* is a primitive or user function that does return a value. In our cooperatively multi-processing virtual machine, we choose to context-switch between commands, but not after reporters. This gives the if statement above a guarantee of atomicity. The reporter in the predicate is guaranteed to still be true when the first command in the consequent executes. This policy also enables atomic fetch and update operations to work without user-supplied synchronization (e.g. `setfoo foo + 1`).

We had a second reason to choose a cooperatively multiprocessed environment instead of an preemptive one. In a preemptive environment, there would be far more machine and turtle state to save and restore on every context switch since it could happen at any time, even inside of a virtual machine primitive.

In both 68K and PPC StarLogo, our context-switch function consists of just 10-20 assembly language instructions. The first few are used to store the process state variables of the turtle that is being suspended. There is then a branch to check whether or not this was the last turtle to run during this iteration (so we can "come up for air" if necessary). If not, we use another bunch of instructions to install the process state variables of the next turtle to be scheduled into the registers and jump to its instruction pointer.

There is one primitive that interacts with context switches: `forward`. When a turtle wants to move in the direction of its current heading, it calls `forward` with a number of turtle steps. If we implemented this by context switching after each turtle went forward the full number of turtle steps, we'd see (if we slowed the computer down) individual turtles, one by one, scooting to their final locations. We wish to maintain an illusion of realistic-looking parallelism, so we want to see all of the turtles move forward at the same time. To do this, we context switch in the middle of the `forward` primitive after the turtle has taken one step. The instruction pointer is set up to return to the middle of the `forward` primitive when the turtle is rescheduled. Context switching on this granularity gives us the nice-looking parallel behavior we want.

Once we had completed the turtle virtual machine, we ran some simple performance tests to get a feel for how well we were doing. On a Mac IIfx in 1994, we were able to run around 60,000 Logo operations per second. When turtles were moving on the screen, the graphics subsystem slowed us down a bit; we could run 5,000 turtle steps per second. After five years, computers have gotten quite a bit faster; on a 233 MHz Power Mac G3, we can execute 2.15 million Logo operations per second and move 61,000 turtle steps per second.

5.2 Patch Virtual Machine

In StarLogo, the patches in the grid underlying the turtle world are active and may run Logo code. However, there are an extremely large number of patches to run (the default grid size is 101 by 101 patches – an odd number so we can have a single patch in the center). This has great ramifications on space requirements for the patch virtual machine.

If each patch contained as much state as a turtle, we would need around 2.5 megabytes of RAM just to hold them all. We only had an 8 MB computer; with the operating system taking up 2 or 3 MB, we just didn't have 2.5 MB to spare.

To conserve process state space, we run the patches through their instruction list in series, context switching only after each patch has finished the instruction list completely. This allows all patches to use a single instance of process state and one stack.

This reduction in parallelism might seem like a terrible price to pay for reduced memory usage. Had we had the memory resources, we would have used the same virtual machine and parallelism model as we did for turtles. However, in practice, this patch model is nowhere near as bad as it sounds. In the StarLogo projects we've seen, patch code is often only used for simulation setup, and for small display updates on each iteration of the simulation's main loop. Thus, the total amount of patch code that must be run at any given time tends to be small, on the order of one or two commands. Since this patch code is short and the period is long, the look of the parallelism is preserved.

Some interesting ramifications of this serial parallelism strategy come into play when looking at the interaction of loops with primitives that display colors on the patches. For example, the following code cycles all of the patches through the 140 StarLogo colors:

```
repeat 140 [ setpatchcolor patchcolor + 1 ]
```

In the parallelism model we are using, each patch runs this entire command in series. Thus, the user sees each patch change its color directly to the last color. This isn't quite the effect we were going for.

The same phenomenon affects tail recursive patch code. Consider the following simple tail-recursive loop:

```
to foobar
foobar
end
```

This code recurses infinitely. One might consider this a bug, but since many more complicated programs can be reduced to this simple function, we use it to illustrate our problem. If each patch must run to completion before we can context switch, we will get stuck running the first patch forever! In fact, in the current Macintosh version of StarLogo, if we turned on tail recursion in this patch code, we would actually hang the computer because we would never "come up for air" to process Macintosh events.

Ultimately, we fixed the problems arising out of serial patch execution by one, removing the `repeat` primitive from the patch virtual machine and only allowing it as an observer primitive, and two, forbidding patch tail recursion.

5.3 Observer

The observer, the only non-parallel entity in StarLogo, may view and modify global characteristics of the StarLogo model. It can create and kill turtles, gather statistics about the turtles and patches, as well as perform auxiliary functions in the StarLogo environment such as plotting, recording movies, and performing data collection and file I/O. The observer can be thought of a lifeguard watching over the world from a high chair at coordinate (0, 0) in the grid.

In StarLogo, the observer is not implemented in assembly language, but as a Lisp program in Macintosh Common Lisp. Since Logo is a dialect of Lisp, it is very easy to compile observer procedures into Lisp code.

The following section describes the "grand unification," a term we use to describe the combination of language, compiler, and runtime system techniques used to give the user the illusion of a seamless integration of the three machines into one environment.

6 The Grand Unification

The previous section described how turtles, patches, and the observer are implemented using fairly different techniques. We didn't want these differences to be obvious so we tried to unify the three components in a way that was seamless to a user. In the next three sections, we show how the compiler and runtime system are constructed to support the user's unified view of the world.

6.1 User's View

The user's view in Connection Machine StarLogo was very different from the Macintosh version. In CM StarLogo, there were three separate namespaces for user procedures and variables: one for the turtles, one for the patches, and one for the observer.

While this interface nicely separates the three different types of code, it is necessary, at times, to have a procedure in one virtual machine call a procedure in another. For instance, a procedure to setup a simulation might begin in the observer procedures page where it creates a bunch of turtles, and move over to the turtle procedures page where the turtles set their color to red and distribute themselves randomly across the grid of patches. For this purpose, CM StarLogo introduced the `foreach` command. The observer could say `foreach "turtle [setup]` to ask all of the turtles to run the setup procedure and wait for them to finish before continuing.

StarLogo is interactive, so it also has a command center where can issue one line commands to the runtime system. CM StarLogo only had one command center, whose focus was set, by default, to the observer. In order to ask the turtles to move forward 50 turtle steps, it was necessary to say `foreach "turtle [forward 50]`. Alternatively, a user could use the `talkto` command to set the default focus to either the turtles or the patches. After that, a user could type `forward 50` directly to make the turtles move.

This led us to a conundrum. StarLogo was supposed to be easy to use by novices, but understandably, the `foreach` command is not easily explained in the first five minutes of use. We wanted a person to be able to sit down at the terminal and type in commands for any entity:

`forward 50`	*turtle command*
`setpatchcolor blue`	*patch command*
`show count-turtles`	*observer command*

without knowing more syntax than they had used in Logo.

So, in the Macintosh version of StarLogo, we decided to try unifying the three virtual machines. The user would simply type a sequence of commands. The system would figure out which commands went to which machines and dispatch them appropriately.

This turned out to be fairly simple in theory and in practice. A few cases, though, required special attention. The simple cases make use of the fact that most commands unambiguously belong to a single virtual machine. Only turtles may go forward. Only patches may set their own patch color (turtles use the command `stamp` to color the patch beneath them), and only the observer can print out values to the screen.

But not all commands are unambiguous. Control flow commands, `repeat`, `if` and `ifelse`, are used in all three virtual machines. It does not make much sense to change their names in each virtual machine, so we disambiguate them by their use. Specifically, these commands have two (or three) parameters: a predicate reporter and an instruction list (or two). Reporters tend to be virtual machine agnostic (they work in all machines), so we use the instruction list to indicate to which virtual machine the command belongs. If there are only turtle commands in the instruction list, then the control flow command is a turtle command. If there are only patch commands, then it's a patch command. If there are any observer commands, or there is a mix of patch, turtle or observer commands, then this control flow command is an observer command (the only virtual machine that can call into the others).

The reader may notice that this disambiguation requires a control flow analysis through the entire program. In fact, we not only have to identify every command and reporter, but we also have to assign a virtual machine identity to each user-written procedure. This process will be explained in Section 6.2.

Variables represent another special case. In Macintosh StarLogo, all variables must be declared at the top of the procedures page. Turtle variables are declared inside `turtles-own`, patch variables inside `patches-own`, and observer variables inside `globals`.[1] So, when we see the use of a variable as a reporter, we know unamiguously to which virtual machine it belongs. However, when we set a variable, while we know who "owns" it, without more information, we would not be able to identify which virtual machine is performing the setting. Let's illustrate this using an example:

```
turtles-own [age]
patches-own [chemical]
globals     [time]

to foo
setage 10
setchemical 10
settime 10
end
```

In the procedure foo, `setage` refers to a turtle setting its

[1] Turtles-own and patches-own are plays on words. Not only are these the turtles' and patches' own variables, but the turtles and patches own (verb) the variables. Thus, whichever way the user wants to read these is correct.

own age variable to 10. But why can't this mean the patch under the turtle setting that turtle's age to 10? What about `setchemical`? This is a patch setting its own chemical value to 10. But why can't it be a turtle setting the value of chemical of the patch beneath it to 10? `Settime` refers to the observer setting the global variable time to 10. But it could as easily mean all the turtles and all the patches setting time to 10.

The ambiguity here is only in the command `set`, not in the variable that we are setting. We came up with the following solution. The most common usage is to have a virtual machine set its own variable. Thus, here we use the command `set`. When a virtual machine sets a different machine's variable, we identify the virtual machine performing the action by a prefix on `set`. Thus, `tset` is a turtle setting a variable, `pset` is a patch, and `oset` is the observer doing it. So, if we wanted to have a turtle setting the chemical variable of the patch beneath it, we write `tsetchemical 10`; to have a patch set the age of the turtle above it, we say `psetage 10`. The observer may not set all of the turtles or patches (which is what would be implied by `osetage` or `osetchemical`), but it may set a specific turtle or patch variable using `osetage-of <turtle id>` and `osetchemical-at <xcor> <ycor>`.[2]

6.2 Compiler's View

The StarLogo whole program compiler consists of four main phases. First, the parser takes Logo code from the single Procedures window and turns it into a Lisp-like intermediate representation. Next, we decide to which virtual machines each procedure belongs. The third phase takes care of identifying and separating turtle, patch, and observer commands from each other. Finally, we generate code for each virtual machine from the intermediate representation.

The first phase is relatively simple. It parses the Logo code into a Lisp-like representation, using whole-program knowledge to identify the type of every symbol entered. If a symbol is a primitive or procedure name, the parser collects the proper number of arguments and performs some loose type checking to make sure that the few compile-time type constraints are satisfied.

The next phase determines to which virtual machine each procedure belongs. Since there are many types of procedures that are virtual machine agnostic (e.g. mathematical functions), a procedure may, in fact, belong to several virtual machines. Due to this, our analysis actually computes the inverse – we determine to which virtual machines a procedure *can't* belong.

We run the following analysis on each user-written procedure. If there are any turtle commands, then the procedure cannot be a patch procedure. If there are any patch commands, it can't be a turtle procedure. If there are any observer commands, it can't be a turtle procedure or a patch procedure. When we reach a procedure call, we add the current procedure to the "called-by" stack of the target procedure and process these in a second pass.

We determine the identity of the control flow primitives `repeat`, `if`, and `ifelse` by examining their instruction lists. If they contain only turtle code, and the reporter argument is a valid turtle reporter, the control flow command is a turtle command. The same is true for patches concerning `if` and `ifelse`. `Repeat` is a special case and is never a patch command (see Section 5.2). If there is only observer code or a mix of patch, turtle or observer code, then the control flow command is an observer command. In this case, the reporter argument must be a valid observer reporter – it may not be run only by turtles or by patches.

After this phase is complete, we have three groups of procedures: turtle, patch and observer. Turtle and patch procedures are "pure"; they do not contain code for any other virtual machine and may be compiled straightforwardly into byte codes for their respective VMs. Observer code generation, however, is slightly more interesting.

Observer code may consist of "pure" observer code, or a mix of code from any of the three virtual machines. Just as CM StarLogo had a `foreach` statement to run instruction lists in the other virtual machines, StarLogo uses a similar command internally. Running through the code in the procedure, we separate out sequences of turtle and patch code into their own `foreach` commands. Once this is done, we convert the observer procedure into Lisp code (where `foreach` is defined as a macro to run the instruction list in the appropriate virtual machine) and have Lisp compile it.

6.3 Runtime View

StarLogo commands are run from either the command center or from buttons in the interface window (see Figure 2). Buttons can either run a command once or run a command repeatedly ("forever"). The interactions between these sources of commands[3] and between the three virtual machines leads to fairly complicated overall scheduling.

How are all these commands from various sources put together and executed on a sequential machine with one processor? "Forever" buttons control the main loop. On each iteration, we run the turtle machine for 1/60th of a second, then run the patch machine, then run one observer forever button. Each time we enter the turtle or patch machines, we run through the list of its forever buttons; if the forever button is on, we execute its code.

[2] These suffixes are widely used throughout StarLogo in all three virtual machines to identify specific turtles and patches.

[3] Runnable code also comes from monitors. These are widgets in the GUI which display the value of any observer expression. They update their display every half second.

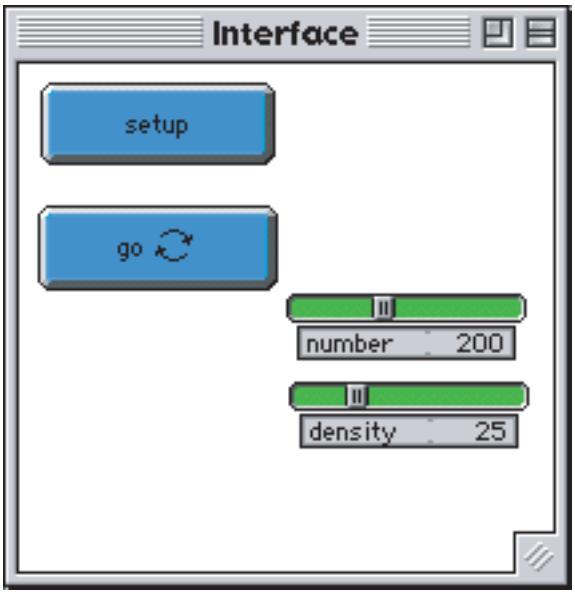

Figure 2: StarLogo interface for the termites project. There is a `setup` button and a `go` forever button. Two sliders allow the user to control global parameters of the project.

Code from the command center and normal buttons interrupt this loop. Observer code executes immediately, but turtle and patch code execute after the turtle and patch forever buttons have finished running once. Since forever button code is typically short, this allows you to play interesting tricks with the job parallelism. If you turn on a `back 100` button while a `forward 1` forever button is executing, the turtles jitter back and forth while the two buttons are executing simultaneously.

The integration of these three virtual machines may seem fairly complicated. Ironically, the complexity largely serves to make the system perform as users expect. The rules were worked out over a period of several years. We wrote and rewrote this main loop several times. Each time, the user perspective became simpler, at the cost of the internals becoming subtler. In the next section, we explain how we rewrote StarLogo in Java and radically revamped our parallelism models (and our main loop) to become much more powerful, much simpler to implement, and easier to explain.

7 Further Developments

In the past year, we have reimplemented StarLogo for more modern computers. This time, we assumed that we had a Pentium 133 with 32 MB of RAM, a huge leap from our Mac IIfx. We also wanted to take advantage of the Internet so that students could publish their projects on their home pages, as well as enable educators to write "active essays" (see *http://el.www.media.mit.edu/groups/el/projects/emergence*). A natural choice for our new implementation was to use Java, which runs in web browsers as well as stand-alone applications.

Since computers were faster and had much more RAM, we didn't have such tight hardware constraints as on the earlier computers. We wouldn't have to write our virtual machines in assembly language, and also figured that we wouldn't even have to use C native libraries either. Consequently, we ported our 68K virtual machine almost line for line to Java. Instead of a jump dispatch table, we used a switch statement; each primitive was implemented in a case statement inside the switch. All turtles, patches, and the observer were Java objects with roughly the same state that they had in our virtual machine implementation.

We decided to try a new parallelism model. Instead of looping over the turtles, running one command per iteration, we adopted a new multi-threaded model for both the turtles and the observer. Each entity's process state and stack was stored separately in a thread object. Our main loop simply ran over all threads, context-switching a thread by swapping in its process state into the entity's fields. Threads were created by running code from the command center and by clicking buttons. Forever buttons were now the same as normal buttons, but wrap the button's code with a loop primitive.

We took the occasion of the reimplementation as a chance to rethink several language design points. These were not based only on technical matters, but also on years of user feedback from children, educators, and novice programmers.

Our "grand unification," while elegant in its vision, was confusing to most users in practice. Everyone understood that `forward 50` was a turtle command, and that `set-patchcolor blue` was a patch command, but when the two interacted in the same function to make a hybrid observer command, people did not understand why their program exhibited unexpected behaviors. The observer, implemented in Lisp, was much slower than the turtle or patch virtual machine. So when a user mixed turtle and patch commands, StarLogo slowed down.

Users were also confused by this hybrid interaction through `if` statements. Often, the user meant for the turtle to be running the `if` statement's predicate, but used an observer command (or worse, an user-defined observer procedure that they had thought was a turtle procedure) instead of a pure turtle command in the consequent instruction list. We had hoped to hide the differences between our virtual machines, but they turned out to be too big to conceal.

In order to address these issues, we made several significant changes to the StarLogo language. First, we separated the command center into two command centers, one for turtles and one for the observer. In order to address the other entity from one of the command centers, it is now necessary to explicitly ask it to perform the action (e.g. `ask-turtles`

`[fd 10]` from the observer command center). This split extends to the procedures page as well.

Where are the patches? Part of the users' difficulties came from identifying which entity would run a particular piece of code. Having two different types of parallel entities was confusing, so we removed the ability of patches to run Logo code. They still exist as passive repositories of data, which, after examining the large corpus of StarLogo projects, seemed to cover just about all of its uses.

Users also had difficulty understanding turtle synchronization. While it was relatively easy to write programs where turtles ran completely out of sync, it was very difficult to make two turtles perform roughly the same actions in sync. We had added a synchronization operator to Macintosh StarLogo, the comma (,) operator. This operator reads as a pause, which is exactly its intention. It is an observer no-op, a barrier, which forces the turtle and patch virtual machines to complete their actions before continuing. People didn't really understand when to use the comma and when not to. We'd see it sprinkled around a program like so much voodoo when usually it was only needed in small doses.

In the new StarLogo, we make the `ask-turtles` and `ask-observer` commands the new synchronization points (we removed the comma operator). Whenever the observer asks a group of turtles (or a turtle asks the observer) to perform some action, it will block until the action has finished. Thus, to synchronize a group of turtles with the observer, the user programs the control into an observer procedure and asks the turtles to perform actions in groups of commands that represent the synchronization points.

A third issue was related to turtle-turtle communication. StarLogo enabled turtles to communicate asynchronously via reading and writing another turtle's variables. However, it wasn't the best idea for this communication to be asynchronous because when you are talking to another turtle, it does not mean that it is talking back to you. In fact, the other turtle may not even notice that you've communicated with it until the context has changed to a point where the communication has become irrelevant.

To solve this problem, we added a new control flow primitive named `grab`. This command enables a turtle or the observer to grab another turtle and run a command while the turtle is being held onto. `Grab` solves the most important problem in pairwise turtle-turtle interactions. While the communication is still asynchronous, `grab` will prevent the other turtle from grabbing, and therefore communicating, with someone else while the first turtle is talking to it.

Our Java implementation has not yet been completed, but the goal is to recreate many of the user interface features of the Macintosh version that made it fun and easy to use. A beta version is available from *http://www.media.mit.edu/starlogo*.

8 Conclusion

Our experience with StarLogo has shown that it is possible to use a moderately under-powered, low-end, desktop computer to perform the work of a massively parallel supercomputer. Through the construction of a set of extremely small, multiprocessing, Logo virtual machines, we were able to not only efficiently execute parallel StarLogo code, but, in the process, explore ways to make the StarLogo language and environment simple to understand and easy-to-use by novice programmers.

9 Acknowledgements

The authors would like to thank Mitchel Resnick for creating the StarLogo environment which provided endless fun and challenge to implement. We'd also like to thank Bill Thies, David Feinberg, and Carlos Mochon for contributing significantly to StarLogo's implementations. Vanessa Collela, Eric Klopfer, Amy Feinup and Monica Linden were instrumental in getting StarLogo out to students.

References

[1] Vanessa Colella, Richard Borovoy, and Mitchel Resnick. Participatory simulations: Using computational objects to learn about dynamic systems. In *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems (Summary)*, volume 2 of *Demonstrations: Interaction via Play*, pages 9–10, 1998.

[2] Vanessa Collela and Amy Feinup. *StarLogo Reference Manual*. Available from `http://www.media.mit.edu/starlogo`, April 1994.

[3] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, 1985.

[4] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[5] Fred Martin. *Cricket Project Page*. Available at `http://fredm.www.media.mit.edu/people/fredm/projects/cricket/`.

[6] Seymour Papert. *Mindstorms : children, computers, and powerful ideas*. Basic Books, New York, N.Y., 1980.

[7] Mitchel Resnick. Multilogo: A study of children and concurrent programming. *Interactive Learning Environments*, 1(3):153–170, 1990.

[8] Mitchel Resnick. Changing the centralized mind. *Technology Review*, 97, 1994.

[9] Mitchel Resnick. *Turtles, termites, and traffic jams: Explorations in massively parallel microworlds*. Bradford Books/MIT Press, Cambridge, Mass., 1994.

[10] Mitchel Resnick. Starlogo: An environment for decentralized modeling and decentralized thinking. In *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*, volume 2 of *DEMONSTRATIONS: Education: Modeling and Tutoring*, pages 11–12, 1996.