AGATE: Unifying Languages and Operating Systems

John Aycock
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4
aycock@cpsc.ucalgary.ca

TR 2002-703-06

Abstract

AGATE is a novel operating system which unifies disparate areas such as compiler construction, operating systems, data compression, formal languages, inter-process communication, and program representation. It does this through a somewhat unusual means: context-free grammar rules. We describe the ideas underlying AGATE and our current prototype implementation.

1 Introduction

What constitutes an operating system? Embedded systems typically do not require the trappings of their general-purpose counterparts, and so a small operating system kernel which supports multitasking and inter-process communication may suffice.

In this paper we present one such operating system: AGATE.[1] By basing the system on context-free grammars, we are able to unify ideas in compiler construction, operating systems, data compression, formal languages, inter-process communication, and program representation. Despite the unusual nature of the underlying strata, the user never sees AGATE's mechanisms and is free to write programs for AGATE in any programming language.

Beginning with an explanation of the notation and terms we use, we describe the ideas behind AGATE, our prototype implementation, and some empirical results.

[1] *A*ll *G*ood *A*cronyms *T*aken, *E*vidently.

2 Notation and Definitions

Formally, a *context-free grammar* is a quadruple $G = (N, T, R, S)$ where N is a set of *nonterminal* symbols, T is a set of *terminal* symbols ($N \cap T = \emptyset$), R is a set of *grammar rules* of the form $N \times (N \cup T)*$, and S is a distinguished nonterminal called the *start symbol*. When discussing context-free grammars in the abstract, we use uppercase letters early in the alphabet to denote nonterminals, lowercase letters to represent terminals, and uppercase letters late in the alphabet stand for either terminals or nonterminals. The empty string is written as ϵ, and α stands for $X_1 X_2 \ldots X_k, k \geq 0$. Grammar rules are written $A \rightarrow \alpha$.

Beginning with the start symbol, a nonterminal A may be *expanded* by replacing A with α for some grammar rule $A \rightarrow \alpha$. This may be repeated until only terminal symbols remain, at which point we have a *sentence* in the language described by G.

3 Code as Grammar

Context-free grammars (and various subsets thereof) are primarily used in parsers, which take an input string and attempt to determine if the input is valid according to a set of grammar rules. However, such grammars[2] may be interpreted in at least three other ways [19]. One way to see a grammar is as a generative device, where the grammar rules are applied to generate a sentence; effectively, this is the reverse of what parsers do. This generative interpretation is used in AGATE.

Compiled code for a program can easily be represented by a grammar. Compiled code is a static, finite-length sequence of machine instructions. Using machine instructions as terminal symbols, at least one grammar rule suffices to describe code.[3] Grammars can also be used to describe feasible execution paths in programs, for purposes of program analysis [25]. But in both cases, this is a static description: static code, static analysis.

Instead, we use grammars to describe a dynamic, executing program. The grammar rules are used to generate the program's code on demand as it is executing. Program execution thus alternates with code generation from grammar rules; every time execution reaches a nonterminal, control reverts back to AGATE's kernel which expands the nonterminal, then resumes execution.

Figure 1 illustrates this new execution model, for the grammar:

$$\begin{array}{rcl} \mathrm{S} & \rightarrow & \mathrm{i}_1\ \mathrm{i}_2\ \mathrm{A}\ \mathrm{i}_3 \\ \mathrm{A} & \rightarrow & \mathrm{i}_4 \end{array}$$

Here, i_1, i_2, i_3, and i_4 are instructions, and S is the start symbol. At each time step, the box represents the current state of the program code in memory; we call

[2]Unless otherwise noted, we use "grammar" to mean "context-free grammar" in the remainder of the paper.

[3]We ignore the possibility of self-modifying code and run-time code generation [18] for the time being.

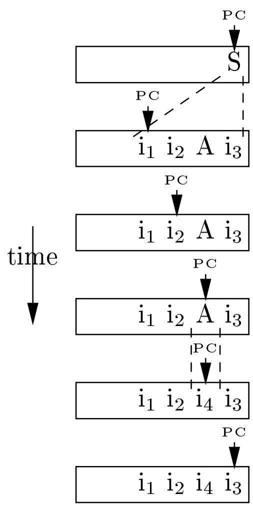

Figure 1: AGATE execution model over six time steps. Dashed lines represent the expansion of a nonterminal by the kernel, and "PC" is the program counter.

this state the *sentential code form*, because it corresponds to a sentential form of the grammar [1]. The initial execution state simply has the start symbol as the sentential code form, with the program counter pointing to the start symbol. As the start symbol is a nonterminal, this causes the kernel to expand the nonterminal and adjust the program counter accordingly. In this particular example, the kernel performs two nonterminal expansions.

Our execution model allows for code decompression as part of its basic operation, an attractive feature for memory-limited embedded systems. Employing a grammar-based scheme for data compression, repetitive code sequences would become new grammar rules, with the original locations of these sequences compressed to a single nonterminal [24]. AGATE would decompress code compressed this way automatically during program execution.

There are other interesting implications of our grammar-based approach, which we have divided into four areas: control flow, context switching, interprocess communication, and run-time code generation.

3.1 Control Flow

One unusual aspect of AGATE is that AGATE programs have no explicit control transfer instructions; there is sufficient mechanism in the grammar model to handle control flow changes.

Let us assume that user programs are compiled to a typical intermediate representation (IR) which looks similar to a machine-independent assembly code. Using such an IR is typical of optimizing compilers. Putting data compression aside for the moment, we convert the IR into a grammar by creating a separate grammar rule for each basic block in the IR. There are no user-visible

$$
\begin{array}{ll}
\text{L1:} & \text{i}_1 \\
 & \text{i}_2 \\
 & \text{goto L2} \\
 & \cdots \\
\text{L2:} & \text{i}_3
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lll}
\text{L1} & \rightarrow & \text{i}_1\ \text{i}_2\ \text{L2} \\
\text{L2} & \rightarrow & \text{i}_3
\end{array}
$$

(a) Goto

$$
\begin{array}{ll}
\text{L1:} & \text{i}_1 \\
 & \text{i}_2 \\
 & \text{call foo} \\
 & \text{i}_3 \\
 & \cdots \\
\text{foo:} & \text{i}_4
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lll}
\text{L1} & \rightarrow & \text{i}_1\ \text{i}_2\ \text{foo}\ \text{i}_3 \\
\text{foo} & \rightarrow & \text{i}_4
\end{array}
$$

(b) Function call

$$
\begin{array}{ll}
\text{L1:} & \text{i}_1 \\
 & \text{if cond goto L2} \\
 & \quad\text{else goto L3} \\
 & \cdots \\
\text{L2:} & \text{i}_2 \\
 & \cdots \\
\text{L3:} & \text{i}_3
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lll}
\text{L1} & \rightarrow & \text{i}_1 \\
 & & [\ \text{cond ? gen L1}' \rightarrow \text{L2}\ ] \\
 & & [\ \neg\text{cond ? gen L1}' \rightarrow \text{L3}\ ] \\
 & & \text{L1}' \\
\text{L2} & \rightarrow & \text{i}_2 \\
\text{L3} & \rightarrow & \text{i}_3
\end{array}
$$

(c) Conditional goto

Figure 2: Translation of control flow into grammar rules. Where needed, details of an AGATE instruction are enclosed in brackets.

changes because we convert from the IR level; users write their programs as they normally would, in any language.

Given this grammar encoding, a goto statement in the IR is simply replaced by a nonterminal naming the basic block that was the target of the goto. This is illustrated in Figure 2a. When execution reaches the end of $L1$ and the program counter points to $L2$, the kernel will expand $L2$, yielding the net effect of the original goto.

Function calls are dealt with in a similar manner, as Figure 2b shows. A function call in the IR is replaced by the nonterminal corresponding to the start of the callee's code. Essentially, AGATE is performing procedure integration [23][4] of the callee into the caller's code.

Unconditional gotos and function calls are static, in that their targets are unchanging. What about conditional gotos? The answer lies in dynamic generation of grammar rules, shown in Figure 2c. Here, the nonterminal $L1'$ is initially without a definition. Execution of $L1$'s code results in the generation of either $L1' \rightarrow L2$ or $L1' \rightarrow L3$, depending on the value of the conditional. To ensure that the correct rule is generated, AGATE instructions may be *guarded* – an instruction is only executed if its guard expression is true [15]. This solves only half the problem, however, because the program may need to take a different path the next time the conditional is evaluated. The semantics of dynamically-generated grammar rules in AGATE are that a dynamically-generated rule is deleted immediately after being used in a production by the kernel, which allows conditional gotos to function as expected.

[4] Also known as inline expansion or, historically, open linkage [8].

3.2 Context Switching

When control flow is encoded as described in the last section, the kernel is frequently invoked to expand nonterminals. Also, AGATE programs cannot loop without the kernel being called, because AGATE programs do not have explicit control flow statements. The kernel is thus in an ideal position to implement nonpreemptive context switching.

In AGATE, each process is represented by a grammar and associated run-time state: the sentential code form, registers, stack, and data space. The set of processes is assumed to be fixed – a feasible constraint for many embedded systems. Every time the kernel is invoked, it suspends the current process, then finds and resumes the next unblocked process in round-robin fashion (blocking is discussed in the next section).

The execution time allotted to a given process, then, is bounded by the longest time it takes to execute any of the process' basic blocks. For a process i with k basic blocks this time may be expressed as $t_{P_i} = max(t_{BB_1}, t_{BB_2}, \ldots, t_{BB_k})$. This makes the worst case time for which a process may be preempted

$$worst\ case\ preemption(P_i) = n \times t_{cs} + \sum_{p \in P \setminus \{P_i\}} t_p$$

where $P = P_1, P_2, \ldots, P_n$ is the set of all n processes and t_{cs} is the context switch latency. Barring dynamically-generated code, this value may be computed prior to run-time.

Naturally, variations on the scheduling algorithm or context switching policy (e.g., context switching every Kth invocation of the kernel by a process) are possible.

3.3 Inter-Process Communication

Inter-process communication in AGATE hinges on "useless" nonterminals, specifically undefined nonterminals. Normally the appearance of an undefined nonterminal in a grammar rule would signal a potential error, and the grammar rule would be deleted [13]. In AGATE, an undefined nonterminal makes it impossible for the kernel to expand the sentential code form further. We use this idea as the basis of blocking – a process whose program counter points to an undefined nonterminal is blocked. That semantics, plus the fact that dynamic rule generation is an atomic kernel operation, allows the construction of inter-process communication mechanisms.

Semaphores are exposed to the user via `signal` and `wait` function calls at the source language level. The compiler maps these into a dynamic rule generation and an undefined nonterminal, respectively, as shown in Figure 3. In the user's code, "mutex" is used as the semaphore's name; this becomes an undefined nonterminal in the translated AGATE code. Grammar rules for processes are maintained globally by the kernel, so a process may add a rule to another process as easily as to itself.

```
L1:  ...                            L1 → ...
     wait("mutex")                       mutex
     critical section       ⇒            critical section
     signal("mutex")                     [ gen mutex → ϵ ]
     ...                                 ...
```

Figure 3: Semaphore translation.

```
L1:  ...                        | L2:  ...
     send("queue", 123)         |      x = recv("queue")
     ...                        |      ...
            ⇓                   |             ⇓
L1  →  ...                      | L2  →  ...
       [ gen queue →            |        queue
            [ R0 = 123 ] ]      |        [ x = R0 ]
       ...                      |        ...
                                |
         Sender                 |      Receiver
```

Figure 4: Message translation.

The kernel stores and uses dynamically-generated rules using a FIFO scheme, so multiple signals on a single semaphore are handled properly. The ordering property of FIFO is useful for the second type of inter-process communication in AGATE, messages.

Messages are useful for passing values between processes. The user simply sees two operations at the source language level, `send` and `recv`. Their translation is given in Figure 4, where the integer value 123 is being sent. The receiver waits on the "queue" nonterminal for a message, after which it saves the message into `x`. The compiler translated `recv` as a function returning a value, and function return values are always in register `R0`. The sender sends its message by generating a rule which causes "queue" to be replaced by a single instruction that loads the appropriate value into `R0`.

Currently, only integer messages are allowed, although this is a limitation of the prototype implementation, not the design. Of course, the message need not be a constant; the full details are presented in Section 4.

3.4 Run-Time Code Generation

Essentially, message passing in AGATE is implemented by having one process dynamically modify another process' code. What we inadvertently have created is a formal framework for self-modifying code as well as run-time code generation [18]. AGATE allows a process to declare where it can be modified, and provides a mechanism for doing so: grammar rule generation. We do not yet know the full implications or the possible (ab)uses of this.

add R_d, R_1, R_2	$R_d \leftarrow R_1 + R_2$
assign R_d, R_s	$R_d \leftarrow R_s$
div R_d, R_1, R_2	$R_d \leftarrow R_1 \div R_2$
epilogue	restore register window
eq R_d, R_1, R_2	$R_d \leftarrow R_1 = R_2$
ge R_d, R_1, R_2	$R_d \leftarrow R_1 \geq R_2$
gen	see text
gent	see text
gt R_d, R_1, R_2	$R_d \leftarrow R_1 > R_2$
input	read character into R0
le R_d, R_1, R_2	$R_d \leftarrow R_1 \leq R_2$
loadb $R_d, const_b$	$R_d \leftarrow const_b$ (boolean constant)
loadi $R_d, const_i$	$R_d \leftarrow const_i$ (integer constant)
loads $R_d, const_s$	$R_d \leftarrow const_s$ (string constant)
lt R_d, R_1, R_2	$R_d \leftarrow R_1 < R_2$
mod R_d, R_1, R_2	$R_d \leftarrow R_1 \ modulo \ R_2$
mul R_d, R_1, R_2	$R_d \leftarrow R_1 \times R_2$
ne R_d, R_1, R_2	$R_d \leftarrow R_1 \neq R_2$
neg R_d, R_s	$R_d \leftarrow -R_s$
not R_d, R_s	$R_d \leftarrow \neg R_s$
print R_s	print R_s as string/integer/boolean
prologue	shift register window
sub R_d, R_1, R_2	$R_d \leftarrow R_1 - R_2$

Figure 5: AGATE virtual machine instruction set.

4 AGATE Implementation

We have implemented a prototype of AGATE in Python, an object-oriented scripting language. The prototype consists of a virtual machine and the kernel itself, which manages dynamic addition of grammar rules and grammar rule expansion. Upon booting, the kernel loads grammars corresponding to its set of processes, constructs a list of processes, and begins its round-robin scheduling. There are two main issues: the design of the virtual machine's architecture and the management of the sentential code form.

4.1 Virtual Machine Architecture

The AGATE virtual machine (AVM) is a register-oriented, primarily three-address architecture. The AVM has 23 instructions, which are listed in Figure 5. All the instructions are guarded by a register, whose contents may be nonzero (true; the instruction is executed) or zero (false; the instruction is not executed). One register permanently has its value set to a nonzero value to support the common case where the guard is true.

Notice the absence of control flow instructions. Even a "halt" instruction in an IR is converted into grammatical form, as an undefined nonterminal which is never defined.

Each function has its own set of registers; the AVM uses a register window scheme. Parameters are passed to functions in registers which are shared between caller and callee, and a function's return value, if any, is passed back through a shared register as well.

The two I/O instructions, `print` and `input`, were added to the AVM as a temporary convenience. The best way to integrate I/O into AGATE is not immediately apparent, a problem also noted by Fraser and Hanson for their EZ system [12]. Output could, for example, be generated by dynamically generating grammar rules for a nonterminal with a well-known name. The system would recognize this nonterminal as a special case, and perform the requested output. This is somewhat lacking in terms of elegance, so we have sidestepped the issue for the time being by adding I/O instructions directly to the AVM.

Grammar rules are generated with the `gen` and `gent` instructions. To support control flow and inter-process communication as described in Section 3, we only need to be able to generate rules in the following three forms:

$$\begin{array}{ccc} \mathrm{A} & \rightarrow & \mathrm{B} \\ \mathrm{A} & \rightarrow & \mathrm{i} \\ \mathrm{A} & \rightarrow & \epsilon \end{array}$$

However, limiting dynamically-added grammar rules to these three forms would not allow all possible context-free grammar rules to be expressed. At the very least, we would need to be able to add rules of the form $A \rightarrow BC$ to maintain full generality.

Because we want to be able to add any kind of context-free grammar rule, the AVM's `gen` instruction allows adding grammar rules with zero or more nonterminals on the right-hand side, i.e., $A \rightarrow B_1 B_2 \ldots B_k, k \geq 0$. The `gent` instruction adds a grammar rule with a single terminal symbol (an AVM instruction) on its right-hand side, or $A \rightarrow i$. In theory, one virtual machine instruction would suffice; the right-hand side of the grammar rule could be encoded to distinguish between terminal and nonterminal symbols. In practice, however, a terminal symbol, being an instruction, is comprised of many parts. We felt the rule-generation instructions would be easier to encode and implement if they were separated in this way.

Returning to inter-process communication, messages are passed by generating rules with the `gent` instruction. The operands of `gent` are registers, and the new grammar rule is built from those registers' contents at the time the `gent` instruction is executed. Through this mechanism, non-constant messages may be sent between processes.

4.2 Sentential Code Form

How should the kernel maintain the sentential code form? While the representation in Figure 1 is a useful conceptual model, it has some practical problems:

1. The sentence being generated by a program's grammar is not the original code, but an execution trace which logs each instruction executed. Clearly

there is not going to be enough memory in an embedded system to hold this trace for all but the most trivial programs.

2. Naïve expansion of grammar rules requires existing symbols to be moved in memory, to make room for the expanded rule, as well as copying of the symbols on the rule's right-hand side. This copying would introduce considerable overhead.

One solution derives from a property of our execution model. The AVM cannot re-execute an instruction, because there are no instructions that permit backward jumps in the sentential code form. This means that symbols to the left of the program counter need not be retained, a considerable space savings. Furthermore, because these leftmost symbols are expendable, we can avoid moving existing symbols when performing rule expansion if we expand to the left of the program counter. If `SCF` is a zero-indexed array for storing the sentential code form, then the kernel would initialize it with the pseudocode:

```
SCF[|SCF|-1] = S
PC = |SCF|-1
```

The pseudocode to expand a rule $A \to \alpha$ would be:

```
assert SCF[PC] == A
SCF[PC-|α|+1 .. PC] = α
PC = PC-|α|+1
```

Another solution, which we use in our prototype, addresses both the problems above. Considering that the kernel has a copy of the grammar rules stored already, the AVM can execute instructions directly from the kernel's copy of the grammar rules. A stack of (rule,index) pairs is needed to keep track of the AVM's current position in each rule. For example, let R_1 be the rule $S \to i_1 i_2 A i_3$ and R_2 be $A \to i_4$. If the grammar is augmented with a new start rule $R_0 = S' \to S$, then the stack trace corresponding to the six time steps in Figure 1 would be:

$$(R_0, 0)$$
$$(R_1, 0)$$
$$(R_1, 1)$$
$$(R_1, 2)$$
$$(R_1, 3)(R_2, 0)$$
$$(R_1, 3)$$

The maximum stack depth needed for a program depends on the structure of its grammar. Assuming a translation of basic block to grammar rules as outlined in Section 3, the maximum stack depth would be proportional to the call depth of the original program.

5 Experimental Results

Our AGATE prototype was intended as a proof-of-concept implementation and has not been optimized in any way. Still, it is instructive to perform some timing

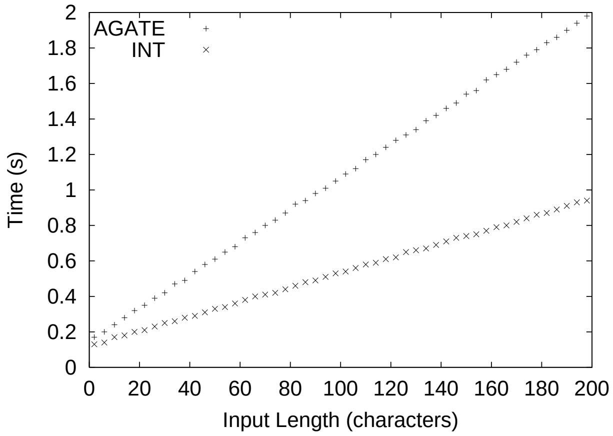

Figure 6: Timings for `calc`, for inputs $(\texttt{1+()})^i\texttt{1())}^i, 0 \leq i < 50$.

comparisons. The experiments described in this section were performed on a 700 MHz Pentium III with 256 M RAM running Red Hat Linux 7.1 and Python 1.5.2. Times reported represent combined user and system times.

The two benchmarks we use are `calc`, a recursive-descent calculator, and `fib`, which computes the n^{th} Fibonacci number. Comparison is made between AGATE and INT, an interpreter which executes code similar to AGATE, except using traditional control flow instructions. INT avoids the kernel overhead of AGATE for grammar rule expansion and context switching.

Figures 6 and 7 show the results for `calc` and `fib`, respectively. (Note that Figure 7 uses a logarithmic scale for the y axis, so that all the data points are visible.) On both these programs, AGATE runs half as fast as INT. Modifying AGATE so that it did not perform context switching improved its time on these benchmarks by 6–7%.

Where does the time go? We profiled AGATE executing first `calc`, and then `fib` with their longest inputs. Table 1 summarizes the data. The relative percentages of time spent in the kernel and interpreter are unsurprising, and reflect the timing differences between AGATE and INT. Within the kernel, the most time is taken finding the next process to run and performing grammar rule expansion. Other kernel activities require much less time. This includes looking up a grammar rule corresponding to a nonterminal, and dynamic rule generation (handling of the `gen` and `gent` AVM instructions in the kernel). As mentioned, our prototype is not optimized, and we expect these numbers to drop substantially in future versions of AGATE.

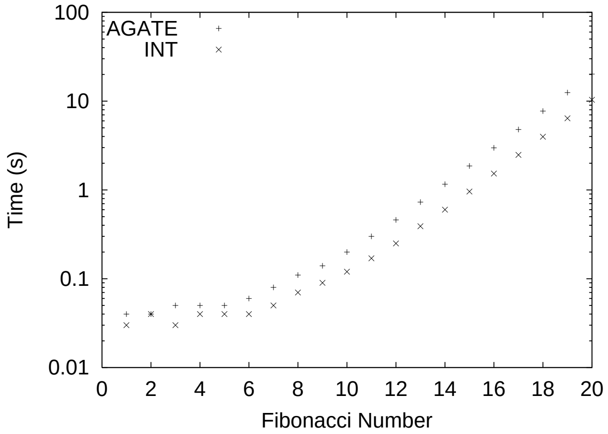

Figure 7: Timings for `fib`, computing *fib*(1) through *fib*(1).

Table 1: AGATE profiling results.

Time	`calc`	`fib`
In interpreter	53%	60%
In kernel	47%	40%
Finding next runnable process and grammar rule expansion	18%	16%
Grammar rule lookup	8%	8%
Dynamic rule generation	3%	3%

6 Related Work

Other unification efforts have been made. Symbolics Lisp machines permeated all software with a single programming language, and allowed a user to replace any component, including ones in the operating system [27]. The EZ system involved a programming language which subsumed operating system functionality into language abstractions mapping, for example, files into strings [12]. Jones [16] suggests ways to exploit concepts common to both languages and operating systems, and also advocates that user programs and the operating system be written in the same language. By changing only the low-level program representation, AGATE unifies many areas without impinging on the user's choice of programming language.

Dynamic grammars have been the subject of occasional research, dating back to the early 1960s [10]; Christiansen [7] gives a survey of work prior to 1990. However, this work [4, 5, 6, 7, 10, 14] is uniformly concerned with developing a formalism that captures static semantic properties in syntax. To the best of our knowledge, no one else has explored applications of dynamic grammars in a generative sense.

Nonpreemptive, or cooperative, multitasking has been used previously in operating systems, most notably in early versions of MacOS [2, 22] and Windows [3, 17]. In these systems, programs are event-driven and assumed to frequently call an API routine to get the next event; the operating system relies on this to regain control and context switch to another process. Any violation of this assumption by a program – a long-running computation, a misbehaving process – and the system as a whole becomes unresponsive. In contrast, AGATE's nonpreemptive multitasking does not require programs to be written in a particular fashion, cannot lose control of the system, and demonstrates that successful nonpreemptive multitasking derives from the underlying program representation, not the program behavior.

Software-based code decompression has been explored, for example, in [9, 11, 20]. Our approach is loosely related to Liao et al., who extract common code sequences and invoke them via function calls at run-time [21]. Our current contribution is not a new compression method, but a natural way to incorporate decompression into program execution.

7 Future Work

There are many avenues for further work. Our next step will be to build an AGATE system for a board with an ARM processor [26]; terminal symbols will be native ARM instructions rather than interpreted code. This will allow us to better measure the kernel overhead compared to traditional program execution.

We are also interested to see how much code compression can be obtained with grammar-based techniques. Decompression will likely increase the need for efficient generation of sentential code forms, another area which we will be looking at.

Finally, there are some lingering conceptual questions. How should I/O be represented in this model? Must we stop at representing code, or can data also be represented using the dynamic grammars?

8 Conclusion

AGATE is a novel operating system based on a simple, well-understood formalism: context-free grammars. In AGATE, the kernel simply expands grammar rules. Each process' compiled code is represented with a grammar, and by dynamically adding grammar rules we can model control flow, inter-process communication, and run-time code generation.

We have shown that basing an operating system on these grammars has several benefits, unifying ideas from a diverse range of fields. Our prototype implementation has established a proof of concept, and identified areas for future improvement.

9 Acknowledgments

Shannon Jaeger and Tim Williams made many helpful comments on this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1988.

[2] Apple Computer, Inc. *Technical Introduction to the Macintosh Family.* Addison-Wesley, second edition, 1992.

[3] N. Baran. Operating systems now and beyond. *BYTE*, 16(11):93–98, 1991.

[4] P. Boullier. Dynamic grammars & semantic analysis. Technical Report 2322, INRIA, 1994.

[5] B. Burshteyn. On the modification of the formal grammar at parse time. *ACM SIGPLAN Notices*, 25(5):117–123, 1990.

[6] S. Cabasino, P. S. Paolucci, and G. M. Todesco. Dynamic parsers and evolving grammars. *ACM SIGPLAN Notices*, 27(11):39–48, 1992.

[7] H. Christiansen. A survey of adaptable grammars. *ACM SIGPLAN Notices*, 25(11):35–44, 1990.

[8] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compilers (Preliminary Notes).* Courant Institute of Mathematical Sciences, 1970.

[9] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 139–149, 1999.

[10] A. C. di Forino. Some remarks on the syntax os symbolic programming languages. *Commun. ACM*, 6(8):456–460, 1963.

[11] W. S. Evans and C. W. Fraser. Bytecode compression via profiled grammar rewriting. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 148–155, 2001.

[12] C. W. Fraser and D. R. Hanson. Integrating operating systems and languages. Technical Report TR 84-2, Department of Computer Science, University of Arizona, 1984.

[13] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.

[14] K. V. Hanford and C. B. Jones. Dynamic syntax: A concept for the definition of the syntax of programming languages. In *Annual Review in Automatic Programming*, volume 7, pages 115–142, 1974.

[15] P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 386–395, 1986.

[16] A. K. Jones. The narrowing gap between language systems and operating systems. In *Information Processing 77*, pages 869–873, 1977.

[17] R. C. Kennedy. The elegant kludge. *BYTE*, 20(8):54–60, 1995.

[18] D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. Technical Report 91-11-04, University of Washington Department of Computer Science and Engineering, 1991.

[19] R. Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic, 1993.

[20] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with runtime decompression. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, 2000.

[21] S. Liao, S. Devadas, and K. Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 4(1):12–38, 1999.

[22] G. Little and T. Swihart. *Programming for System 7*. Addison-Wesley, 1991.

[23] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[24] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *Computer Journal*, 40(2/3):102–116, 1997.

[25] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, 1998.

[26] D. Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley, second edition, 2000.

[27] J. H. Walker, D. A. Moon, D. L. Weinreb, and M. McMahon. The Symbolics Genera programming environment. *IEEE Software*, pages 36–45, 1987.