Software Verification and Validation with Destiny: A parallel approach to automated theorem proving

by *Josiah Dykstra*

Abstract

This paper presents an introduction to computer-aided theorem proving and a new approach using parallel processing to increase power and speed of this computation. Automated theorem provers, along with human interpretation, have been shown to be powerful tools in verifying and validating computer software. *Destiny* is a new tool that provides even greater and more powerful analysis enabling greater ties between software programs and their specifications.

Introduction

Computer software development is a tedious process. Unlike hardware, software is easily, and therefore often, changed and updated. Consequently, several update ideologies have developed including "build first, fix later," "if it runs, ship it" and, "there will always be bugs, so only fix the worst ones." Additionally, the software development process has a tendency to yield an increasingly complex, and increasingly buggy product. Because of this, reliability is nearly impossible to attain; even when a primary design requirement. In high-assurance applications, this presents a grave situation. Not only must a program be free of errors, but equally important is the end program performing no more or less than originally intended. Writing a program that executes as planned is no simple task. Therefore, the software process includes validation and verification, where validation is the process of answering the question, "Did we build the right product?" and verification answers the question "Did we build the product right"[5]. G. J. Myers said, "Testing is the process of executing programs with the intention of finding errors." But, according to E. W. Dijkstra,"Testing can show the presence of bugs, but never their absence"[6]. Many years of research have been dedicated to latter goal: proving that errors do not exist.

Formal methods is the application of mathematical techniques for the specification, analysis, design, and implementation of complex computer software and hardware. Computational logic is a mathematical logic mechanized to check proofs by computation. Combining these two, one should be able to attack the problem of software and hardware validation and verification. In theory, one can mathematically prove that a program adheres to a specification. If, for example, a programmer implements an algorithm, an analyst can translate the source code into functional form and use a theorem prover to formally demonstrate that the code will satisfy the given requirements. It also means that we can identify whether the algorithm will crash, produce errors, or succumb to unexpected or improperly formatted input data. Despite this, one cannot guarantee against all dysfunction (deadlock detection, for instance); however, under the assumption that these requirements are met, the resulting program is mathematically guaranteed to execute error-free, and exactly as desired.

Theorem Proving to Date

Much of the early work related to theorem proving and the development of software utilizing theorem proving techniques came from Robert S. Boyer and J. Strother Moore at the University of Texas at Austin. *A Computational Logic*, by Boyer and Moore, published in 1979 and followed by a Second Edition in 1998, is a handbook that includes a primer for the logic as a functional programming language and a detailed reference guide to the associated mechanical theorem proving systems [1]. The system they describe, and widely distribute, is called Nqthm. The program is based on Common Lisp, and has endured several releases. Since the early 1990s, Matt Kaufmann and Moore have been working on a new theorem prover, called "A Computational Logic for Applicative Common Lisp," or ACL2 [7].

Nqthm has been widely accepted and used as a powerful tool, proving such things as a small multitasking operating system kernel complying with its specification[1], the invariability of the RSA public key encryption algorithm [3], and various communications protocols [4]. ACL2 is slowly being adopted by academic institutions and research centers around the world.

Destiny is an application framework designed with the hindsight of previous theorem provers in mind. Destiny provides a graphical front end and means of interacting with the results produced. Various levels of control-flow diagrams are generated providing the analyst with deeper levels of information, as needed. Additionally, it is designed for scalable parallel processing. This "divide and conquer" method means that as the problems become more and more complex, additional computing power can be levied against it. Essentially, the large problem domain is broken down into small problem sets to be solved individually, when combined yield the final answer. Finally, the current version of Destiny accepts Java code as input. While Destiny converts the program to functional form, Nqthm and ACL2 require the original input to be constructed in Lisp.

A Framework for Modeling Java

Java was chosen for several reasons. The Java language was designed by Sun Microsystems to run on a detailed specification of a virtual machine, which has been ported to various hardware architectures. The portability and implementation independence of the Java Virtual Machine (JVM) allows Destiny to function on a larger problem set. Any Java program can be modeled without thought to how the JVM has been implemented; to the programmer, the JVM instruction set is consistently the same. Within the JVM, executing code is organized into a stack of "frames." The frame of an executing method holds local variables and an operand stack, which mimics the registers of a hardware CPU.

Java classes are loaded dynamically as a program needs them. The first time a new class is accessed, the runtime environment loads the class and performs necessary operations to use it. Since Destiny never executes the inputted program, an initial pass is required through the source code, which notes all library calls and external references to be certain that whatever Java code is loaded will be executed. Using this feature, we can construct a directed graph demonstrating all possible paths for program execution. Each vertex of the graph represents a single opcode, and the edges describe the transition between instructions. The invocation and return of control given and surrendered by method calls within Java help give the program distinct form and discrete blocks. This knowledge can be used to construct a higher-level method graph, with each vertex representing an entire method and edges showing calls and returns. Analysis and evaluation can now be done on isolated methods or code segments. Each vertex is associated with a state transition, a formal expression describing how the execution of which will alter the virtual machine. Each edge is associated with a predicate, a conditional requirement in order to follow the given path.

For example, an if-then-else statement determines which branch to follow based on a conditional, as shown in Figures 1 and 2. A walk through the actual Destiny process demonstrates the progression from Java object code to byte code, to method and bytecode graphic interfaces. This process is illustrated in Figures 3-6. Interaction is achieved by allowing the analyst to click on any vertex for more information or to construct the graph of the next level of detail.

```
Boolean verified;
if(x==5){
        verified = true;
}
else{
        verified = false;
}
```

Figure 1: If-Then-Else statement showing code branching.

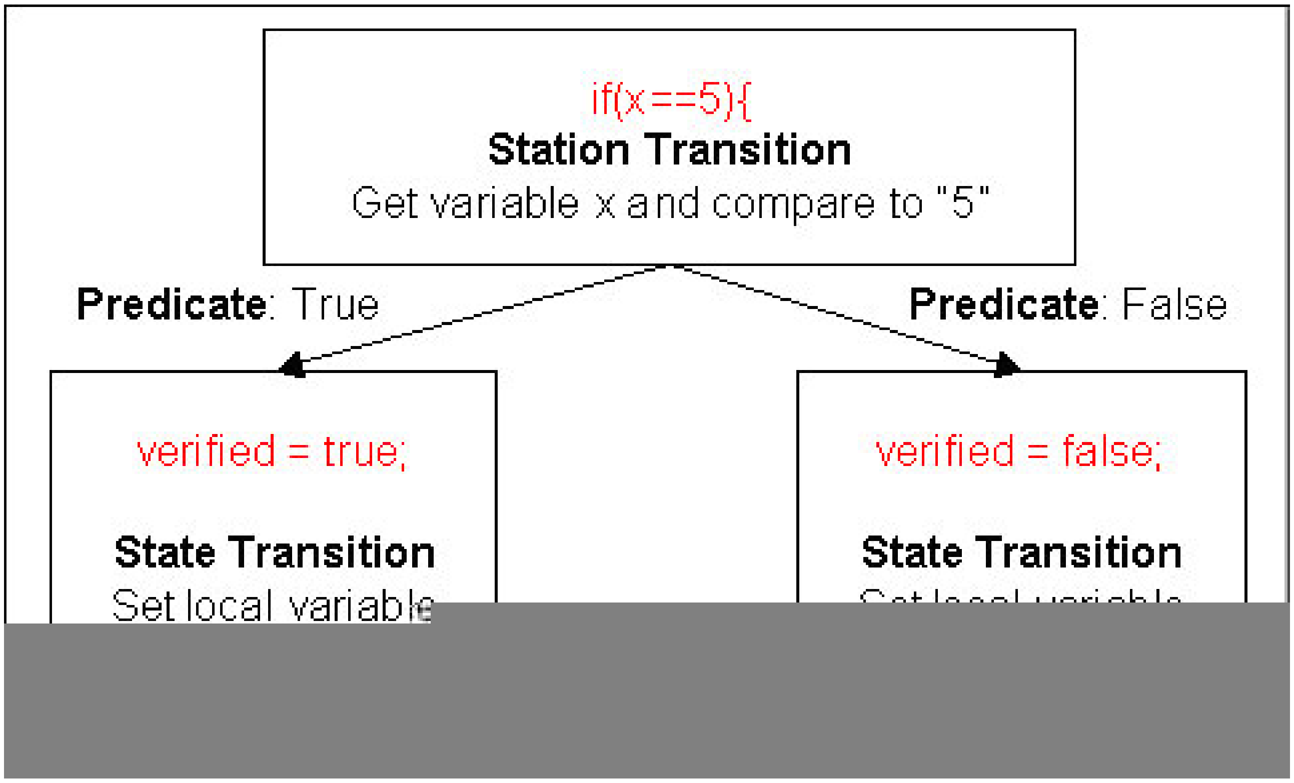

Figure 2: State transition diagram illustrating code in Figure 1.

```
import java.io.*;
public class SimpleProgram {
        public static void main(String args[]){
                String x = args[0];
                int n = Integer.parseInt(x);
                int sum = 0;
                boolean verified = false;
                for (int i=1; i<=n; ++i){
                        sum = sum+i;
                }
                if (sum = n * (n+1)/2){
                        verified = true;
                }
        }
}
```

Figure 3: Example program that verifies the sum of integers 1 to n.

```
0 aload_0                         // String x = args[0]
1 iconst_0
2 aaload
3 astore_1                        // int n = Integer.parseInt(x)
4 aload_1
5 invokestatic #2
8 istore_2
9 iconst_0                        // int sum = 0
10 istore_3
11 iconst_0                       // boolean verified = false
12 istore 4
14 iconst_1                       // for (int i=1; i<=n; ++i){
15 istore 5
17 goto 28…
```

Figure 4: Java byte code compiled from Java source code in Figure 3.

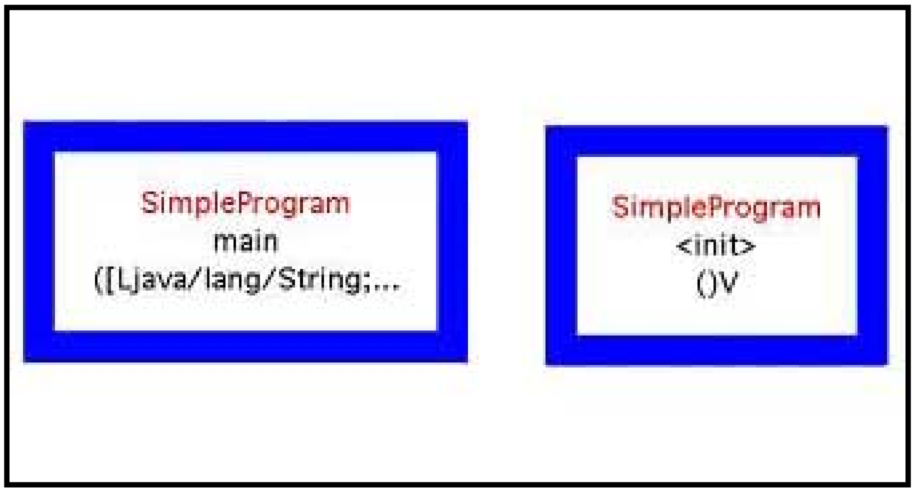

Figure 5: Destiny Method Graph (high level graph) showing one defined method (main) and the JVM constructed initializer.

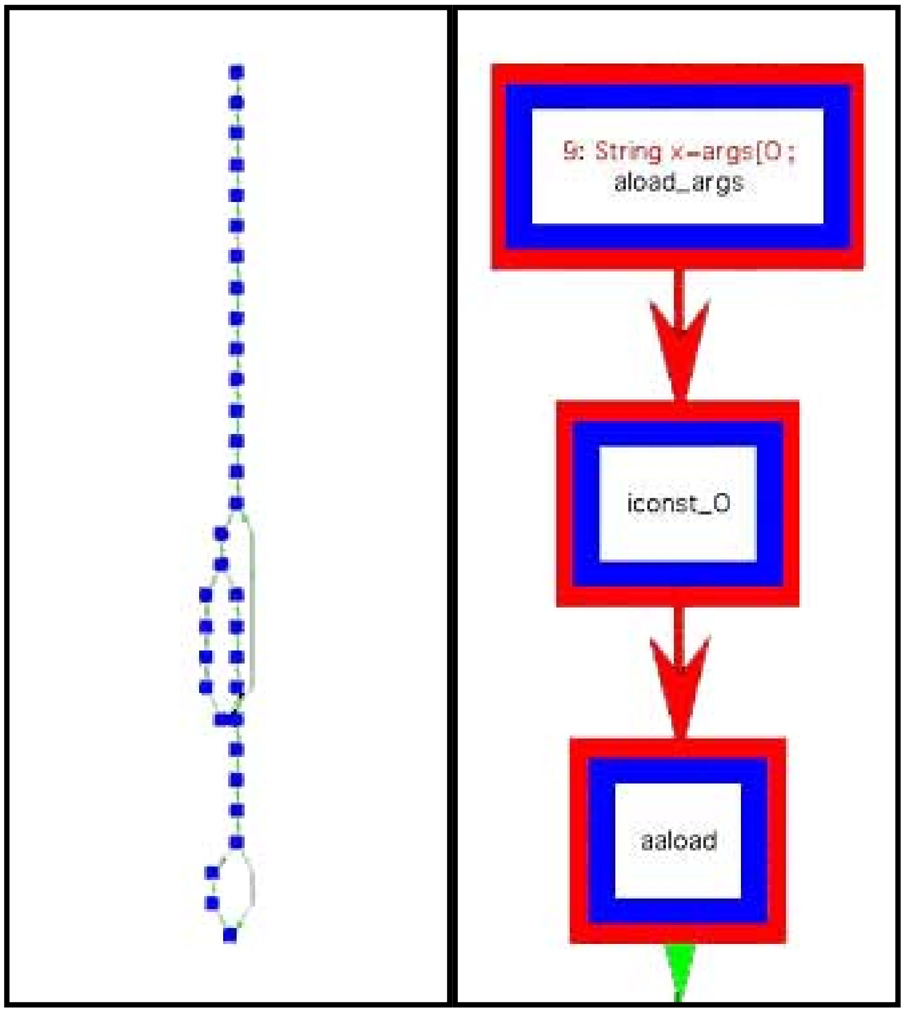

Figure 6: Destiny Bytecode Graph "(low level graph) full (left) and enlarged (right).

Parallel Theorem Proving

Destiny's use of the ideas behind a Beowulf cluster makes it uniquely powerful. A Beowulf cluster is a collection of commodity computers running in parallel via a private network, which rival supercomputer power at substantially reduced cost. The Destiny implementation of the cluster involves two autonomous network domains, the data ring and the load-balancing star. The supporting hardware consists of a master at the center of a star of slaves, joined together to form a ring (Figure 7).

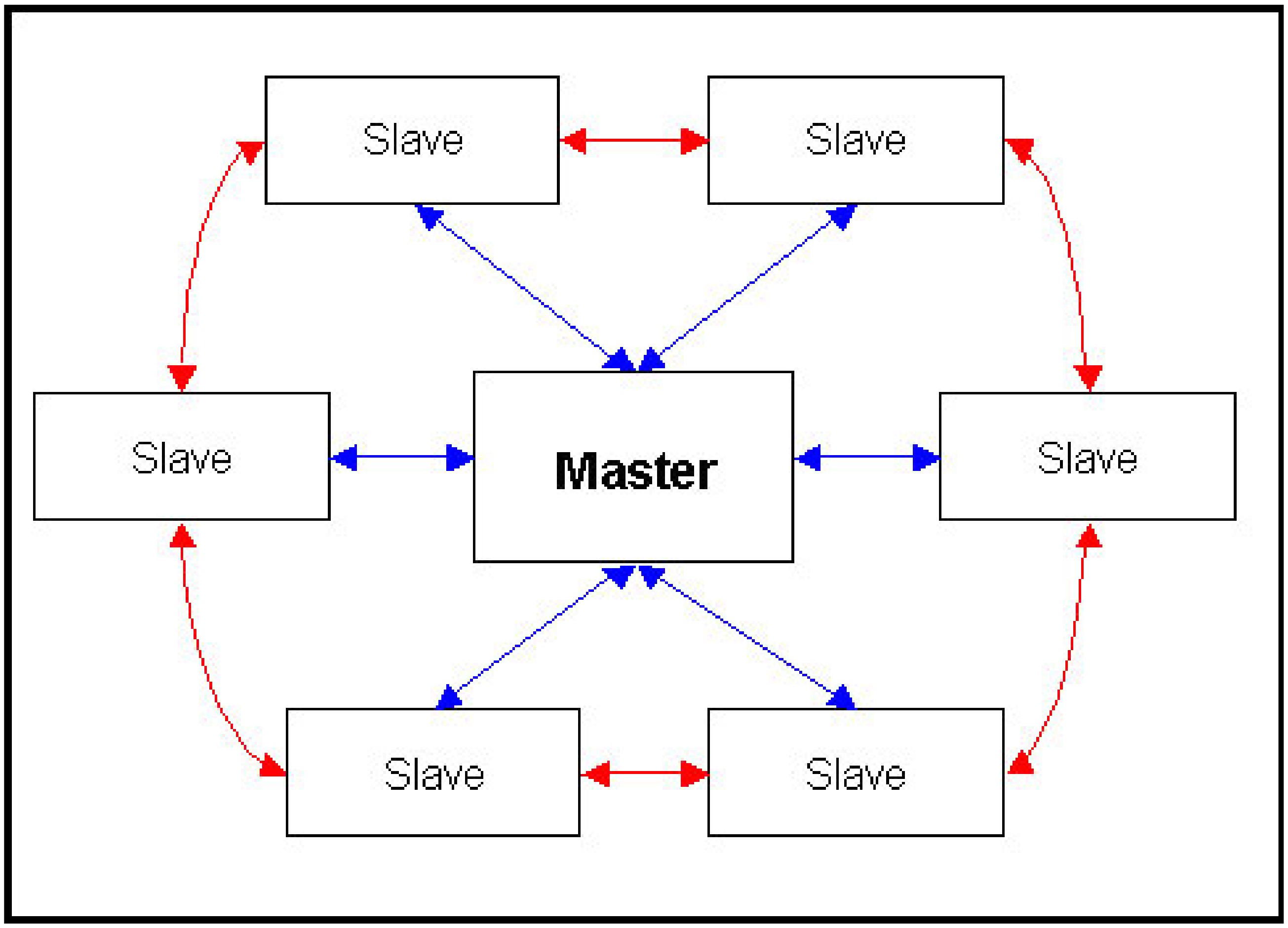

Figure 7: Destiny architecture showing balancing star and data ring.

Load Balancing Star

As Destiny runs, it generates events, which are modules comprised of states and code that are executable and persistent. Only those events that are capable of import and export may be candidates for load balancing across the slaves. The master moderates the load-balancing service. When Destiny first begins execution, events that need to be processed are passed to a single slave. As the load increases and the slave becomes saturated, events are forwarded to other slaves in order to lighten the load. The master maintains the sole event queue. The current implementation has been done with one master and twenty-three slaves using a high-speed 24-port switch and TCP/IP.

Data Ring

The data ring is for exclusive use of the slaves. Two applications run on each slave: a controller and a work processor. The controller acts as a liaison between the slave work processor and the master, leaving the work processor to concentrate solely on data processing. This setup is intended to support slaves with symmetric multi-processing capability, in our case dual-processor Pentium-class machines. At startup, the controller links to the worker, links to its upstream and downstream neighbor controllers, and establishes a connection with the master. The underlying hardware for our setup was another 24-port switch, though crossover cables could easily be used at a cost of an additional network adapter for each machine. The data ring utilizes the UDP protocol to distribute updated information amongst itself. Since packets travel around the ring and back to the sender, dropped packets can be detected via a time-out mechanism and resent. Relatively cheap reliability, compared to TCP/IP, can be achieved on this network. In the switch implementation, multicast might be possible, but this has not been investigated.

Because events on different slaves may rely on each other, or manipulate the same data, information must be shared amongst the slaves. Therefore, all slaves synchronize on a global name-space. Additionally, the slave must notify all others before changing an expression. Slaves also have the ability to block other events from occurring, until notified to continue. Finally, in the process of evaluating an expression, new events may be generated that are then passed back to the master for distribution.

Destiny Concepts in Depth

Two concepts are key to understanding how automated theorem provers such as Destiny work. First, is the concept of backing up a predicate. The second requires the understanding of a rewrite event and how an engine processes these events.

Backing Up Predicates

As was mentioned earlier, a predicate is the condition that must be satisfied in order to take a particular path in the execution of a program. By working from the bottom (end result or state) up, we can compose a series of states and predicates together to create an expression at the top (beginning) representing the entire path of execution. This process can be applied to a block of code, or an entire program. Figure 8 illustrates the original progression of a program, and how the predicates are backed up to the final version at the top using preorder notation. Note that it is possible to categorize the vertices into levels, based on number of branches, for instance, and compress (converge many vertices into one) sequentially and systematically. These options allow Destiny to offer varying levels of information in separate graphs, based on the wishes of the analyst. The actual implementation and mathematical logic behind backing up predicates is beyond the scope of this paper.

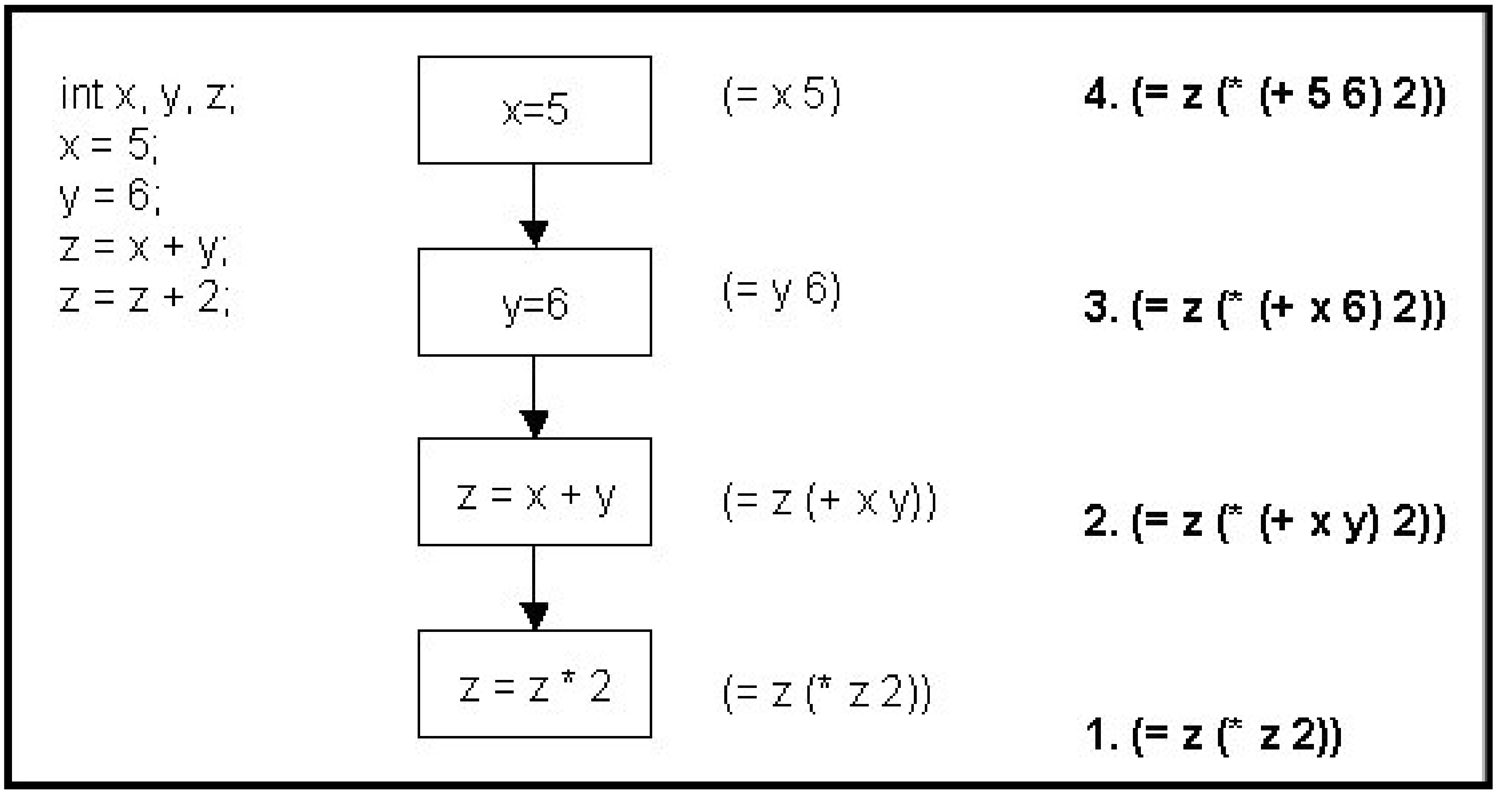

Figure 8: Backing up a predicate

The Rewrite Engine

Once a program or code block has been backed up, it must be simplified to produce readable and interpretable results. Stand-alone Java source code is the original input into Destiny. This code is compiled into byte code, and then translated into functional form embodied in Lisp notation. The rewrite engine then uses a database of user-defined rules to pattern match against the expression in order to simplify it. For instance, the rewrite engine first must be educated about the associative addition rules, so that (+ x y) is equivalent to (+ y x). Similarly, it must understand how to evaluate an expression, say (+ 2 3), to its calculated value, 5. Rewriting is done from the top down. At some point, no more rules will match against an expression, at which time it is declared stable. The human running the program has the ability, and is in fact encouraged, to manipulate the rules database, adding to it and rearranging the order in which rules are applied. In the example shown in Figure 8, the rewrite engine should evaluate the expression at the top to 22.

Conclusions

Theorem proving is not a novel or recent development in mathematics or computer science. However, the growth of modern computing power and the growing complexity of problems to be solved has necessitated a fresh approach to the solution of these problems. Destiny's three major assets are the beginning of a breakthrough in automated theorem proving. First, work is done interactively with a human analyst who can interpret and extend the results generated by the computer. Second, scalable parallel processing can be employed to attack any size problem. Finally, Destiny is directed at Java code, giving it a substantial problem set and a cross-platform, portable abstract machine.

Final development and extensive testing still remain before Destiny is widely available. Additionally, stress testing must be done to fine-tune the underlying architecture. However, the concepts introduced in this new design are powerful and unique, making it a strong contender as the next theorem prover of choice.

References

1
Bevier, W. *A Verified Operating System Kernel.* Ph.D. Th., University of Texas at Austin, 1987. University Microfilms and ftp://ftp.cs.utexas.edu/pub/boyer/diss/bevier.ps.Z.

2
Boyer, R.S. and J.S. Moore. *A Computational Logic Handbook, Second Edition.* Academic Press, 1998.

3
Boyer, R.S. and J.S. Moore. "Proof Checking the RSA Public Key Encryption Algorithm." *American Mathematical Monthly 91*, 3 (1984), 181-189.

4
Di Vito, B.L. *Verification of Communications Protocols and Abstract Process Models.* Ph.D. Th., University of Texas at Austin, 1982. University Microfilms.

5
ANSI/IEEE: IEEE Standard for Software Verification and Validation. ANSI/IEEE Std. 1012-1998, New York, 1998.

6
Dahl, O., E. W. Dijkstra and C. A. Hoare. "Structured Programming." Academic Press, New York, 1972.

7
Kaufmann, M. and J.S. Moore. "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp." *IEEE Transactions on Software Engineering 23*, 4 (April 1997), 203-213.

Biography

Josiah Dykstra (dykstra@cs.hope.edu) is an undergraduate computer science and music double major at Hope College in Holland, Michigan. He is a member of IEEE and is currently serving his second term as President of the local ACM chapter. He has held internships with Gateway, Inc. and the National Security Agency and conducted research with the NSF Research Experience for Undergraduates Program.

Research on Destiny was conducted in conjunction with the NSA.

Want more Crossroads articles about Parallel Computing? Get a listing or go to the next one or the previous one.