INTRODUCTION
The nature of hardware is changing. Deep submicron technologies will increase the number of gates per chip by orders of magnitude in the next five to ten years, reaching a billion gates per chip in about 2004. Current design methodologies will not scale to that level of complexity.

We believe that the current situation in hardware is analogous to the situation that software was in the late fifties and early sixties: computers that had always been programmed directly in machine language were beginning to achieve sufficient scale that building large systems in such a low-level manner was becoming impractical. That period of time gave birth to languages like FORTRAN and COBOL and systems like the IBM/360, whose descendants are still in wide use today.

Now, forty years later, it appears that hardware is about to achieve a level of complexity that requires that it be programmed in a high-level language. Otherwise, systems will collapse under the weight of their own descriptions.

JavaTime is a dialect of Java designed to allow systems to be described in such a way that they can be automatically compiled into either software (in the form of Java bytecodes [Lindholm & Yellin 1997]) or hardware (in the form of some lower-level design language, such as Verilog [Sternheim et al 1990]) or VHDL).

With current language technology, a programmer implementing a new encryption algorithm would never start by implementing it in Verilog --- Verilog is much too low-level. Instead, the algorithm would be implemented, debugged, and distributed in a high-level language like Java. At some later time, if demand was sufficient and performance was critical, a hardware implementation might be fabricated. This implementation would then have to be designed and tested by a different person with expertise in hardware design and in a different language that would not allow any re-use of the original code.

The goals of JavaTime are much more ambitious than those of the current generation of hardware description languages (HDL's): JavaTime is intended to be a unified hardware/software language. The fundamental approach that we have taken in order to reach this very ambitious goal is to assume that most or all of the problems that make hardware design difficult are manifestations of deep problems which affect the design of all systems. We then looked for ways to solve these problems in the language design.
By viewing the problems sufficiently abstractly, we are able to develop solutions to both hardware and software problems.

The language JavaTime as described here is not complete; it does not solve all of the problems and we do not provide algorithms for compiling it into a lower-level HDL. We have deliberately worked top-down from high-level languages toward hardware implementation, instead of bottom-up, because we believe that this is the only way to design a language that solves problems for both hardware and software systems designers.

We must solve problems in both arenas because only then will both programmers and hardware designers use the same language.

**JavaTime Design Principles**

Each feature of JavaTime must solve an important problem in both hardware and software design. Each section of this document that introduces a new feature of includes discussion the hardware and software issues and how we attempt to solve them.

JavaTime is designed as an extension to the Java language that can be compiled into Java bytecodes. The extensions are designed to be relatively minor changes to the Java syntax; most are modifiers to class definitions.

The extensions have two effects: they instruct the JavaTime compiler to check additional static properties of the program, and reject it as ill-formed if it does not meet those requirements; and they instruct the JavaTime compiler to generate certain extra methods for a class, or to modify the code of some of the programmer-supplied methods.

The JavaTime compiler converts the JavaTime language extensions into standard Java, and performs static checks to enforce the semantic requirements of the JavaTime language on certain classes.

A JavaTime program is compiled statically with global information – all program source files and library class files must be available to the compiler. The JavaTime semantics are enforced on a static program call graph, which is conservative due to the fact that not all virtual and interface calls can be resolved at compile time.

The JavaTime compiler attempts to build a call graph that is as precise as possible. By default, it uses a linear expected time algorithm such as *Class Hierarchy Analysis* or *Rapid Type Analysis* [Bacon & Sweeney 1996, Bacon 1997] to determine the static call graph. There are more powerful call graph analysis algorithms [Grove et al 1997] which may be provided as options by the JavaTime compiler. These algorithms can be used to produce a more precise result, although in general they are too slow to be used by the compiler during development, and in general should be employed only when mapping the program into hardware.

**FINITE COMPUTATION**

The area of a chip is a finite resource; therefore a program that is compiled onto a chip must make finite use of chip area. Hardware systems are also subject to external timing constraints, such as
bus cycle time; therefore programs must have a demonstrable upper bound on their execution time.

Software systems, by contrast, are generally written in such a way that scarce resources, such as memory, are “virtualized” and made to seem unlimited. Virtualization is aided by the introduction of indirection (like a virtual-to-physical memory mapping). Of course, these systems also have finite resources, but virtualization allows a single program to run well across machines with varying amounts of physical resources. Virtualization also allows the finite resource problem to be conveniently deferred. Some systems will never exceed resource limits in practice; others will simply stop working (or stop working well) with inputs beyond a certain size. Ultimately, the responsibility for dealing with the finiteness problem is left to the user.

In some situations this is reasonable and even desirable, since it allows the user to go to the store and buy more hardware (RAM, disks, etc.) to expand the capabilities of the system.

Is finiteness a property that has general application to software systems? There are real-time systems, but they represent a very small segment of software. Finiteness has been avoided because it considerably constrains the expressibility of a language --- it is not Turing complete.

Our approach to introducing finiteness into JavaTime is to use it to solve performance problems in software created by the potentially limitless use of resources. In particular, the JavaTime compiler will be able to statically allocate memory for some types of finite objects.

**FINITE CLASSES AND METHODS**

The `finite` modifier can be applied to JavaTime methods or to entire JavaTime classes. The call graph of the method or of all of the methods and initializers of the class must satisfy the following restrictions:

- no while loops
- no recursion
- no invocations of `wait()`
- no calls to `synchronized` methods or blocks (except with provably thread-local objects)
- all `for` loops must have definite bounds

A `finite` class is translated into standard Java as a class that implements the interface `Finite` (which defines no methods). All subclasses of a `finite` class must also be `finite`.

When the JavaTime compiler is checking that a method is `finite`, it does not need to check the code of called methods which are themselves `finite`.

**Examples**

The class `Foo` has two methods declared as `finite` and is extended by class `Bar` which is declared as a `finite` class. All methods are provably finite-time computations:

```java
class Foo {
    private int a = 0;
    public finite int get() { return a; }
    public finite void set(int x) { a = x; }
}
```
finite class Bar extends Foo {
    public add(int x) { a += x; }
}

The class Baz also extends Foo but is not declared as a finite because its method waitForValue() contains a while loop and invokes the wait() operation, which may take arbitrarily long to complete. The class Foo was not declared as finite because it would otherwise be impossible for Baz to subclass it.

class Baz extends Foo {
    public void waitForValue() { while (a == 0) wait(); }
}

Finite Use of Space

JavaTime does not explicitly require that storage consumption be bounded in finite methods or classes. Instead, it merely requires that computations be finite in time, which implies that they must be finite in space as well.

When compiling to hardware (and in some cases when compiling to software as well), the JavaTime compiler automatically moves storage allocation by finite methods or classes into the class constructor. However, this may not always be possible since in general Java allows references to objects to be passed without restriction. Therefore, the lifetime of objects is not known in general.

We require some additional mechanisms in JavaTime before the storage resource problem can be fully addressed.

COPYABLE OBJECTS

In hardware, latching a value out of a register and transmitting it over wires “copies” the value in the register. Copying is therefore natural and cheap in hardware. On the other hand, multiple references to an object with shared state require wires to be run from each use of the object to the part of the chip where it has been placed. References are therefore very expensive, and will only become more so as wiring continues to consume more of the total chip area in deep submicron technologies.

In software, the exact opposite situation applies. References are cheap; they only require copying a one-word pointer to an object. On the other hand, copying values is expensive, because it consumes additional memory and time. For example, numerically intensive floating point performance is entirely dominated by memory copying costs.

Java only provides a mechanism for copying references, not values. In order to be able to create JavaTime objects that can be effectively manipulated in hardware, we must have a mechanism for copying values.

What software problem will this solve? Copying references is convenient and cheap in software, but it makes program analysis tremendously more complicated and severely
constrains compilers in their ability to optimize code. The computer language community has been struggling with this issue for at least 30 years.

There are two approaches that are used: (1) allow copying of references, and then use program analysis to determine when an object is private to some portion of the code (so that it can be placed in a register, allocated on the stack, etc.); or (2) disallow copying of references, and instead define the language semantics so that true copies are always made, and then use analysis and run-time techniques like reference counting to try to avoid actually copying the objects and instead just copy references. Clearly, these two approaches are duels of each other.

We introduce the notion of a copyable class into JavaTime in order to allow the two styles to be mixed in a disciplined way.

**COPYABLE CLASS DEFINITIONS**

In Java, parameters to methods are passed by value. However, when an object is passed, it is a reference to the object that is passed by value, and not a copy of the object itself. Therefore, the called method can change the state of the passed parameter.

While such semantics are often exactly what is desired, there are also significant advantages to using true pass-by-value semantics, in which a “deep” copy of the object is passed. When a parameter is copied, we can guarantee that there will be no side effects on the original parameter value. Absence of side-effects is a very fundamental property of hardware systems, so providing a mechanism for reducing side effects is an essential building block of the JavaTime language.

By declaring an object to be copyable, the programmer indicates to the JavaTime compiler that the object consists entirely of scalars and other copyable objects. The JavaTime compiler automatically generates a clone() method, which recursively invokes the clone() methods of all of the contained objects.

```java
copyable class Employee {
    public EmployeeID id;
    public int salary;
}
```

The above JavaTime class definition would be converted into the following Java code:

```java
class Node implements Copyable {
    public EmployeeID id;
    public int salary;

    public Object clone() {
        Employee e = new Employee();
        e.id = id.clone();
        e.salary = salary;
    }
}
```
If the class EmployeeID is not copyable, then a compile-time error will be generated. The interface Copyable is an extension of the Cloneable interface. The JavaTime compiler uses the Copyable interface to make copyable classes recognizable at the Java bytecode level.

It is good programming practice to make all JavaTime objects copyable unless there is a specific reason not to do so, such as use of recursive data structures or a need for a non-standard clone() method.

**ISOLATED OBJECTS**

JavaTime makes use of copyable objects to define a new class of objects called isolated objects. An isolated object has the property that all values that are passed into or out of its methods (including its constructors) are copied, and no global references are permitted. This means that an isolated object can not have any aliases with outside objects, which has many powerful implications for both hardware and software.

There are three kinds of isolated objects: immutable objects, which are values without modifiable state; pure objects, which are values with mutable state; and module objects, which are threads of control with mutable state.

**IMMUTABLE OBJECTS**

An object is immutable if it is a read-only class for which two references to the same object are indistinguishable from two references to different objects containing the same data values. We use the term immutable value to denote both scalars and immutable objects.

In the hardware domain, an immutable value is something that can be passed freely over a wire. In the software domain, an immutable value is something that can be copied very efficiently, by copying its pointer. An immutable value is also something that can be passed meaningfully between different Java virtual machines connected by a network.

An immutable object is a kind of copyable object (interface Immutable is a subclass of interface Copyable).

The transitive call graph of an immutable object must obey the following restrictions:
- instance variables are only modified in initializers and constructors
- all public data members are immutable values
- public methods only take as parameters and return as results scalars, copyable objects, or arrays of either scalars or copyable objects, or nested arrays of either scalars or copyable objects. Non-scalar parameters and return values that are not immutable will be cloned before being passed into or out of the method.
- no synchronized methods or blocks or invocations of wait() or notify()
- no references to non-constant (i.e. recursively final) global variables
- no creations of Thread objects or invocations of Thread methods.
- references to non-immutable data members of the immutable class may not be copied or passed as parameters to other functions.
In addition, the following restrictions are placed on the use of immutable objects by other classes:

- no assignment to public data members
- no use in a synchronized() block
- no use with the == and != operators

All immutable objects are copyable objects. But unlike more general copyable objects, immutable objects can be copied simply by copying their reference. In other words, a “shallow” copy and a “deep” copy are indistinguishable from one another.

The public methods of an immutable class may only take as parameters values that can be copied. The JavaTime compiler inserts extra operations to invoke the clone() method of any parameters that are copyable but not immutable; scalar and immutable objects are copied by the default “shallow” pass-by-value semantics of Java.

Subject to the above restrictions, immutable objects may perform new operations, and invoke methods of internally created objects that take non-immutable objects as parameters. In other words, the methods of immutable objects are free to create internal objects that have modifiable state information, so long as that state information is not visible outside of the method.

All subclasses of an immutable class are also immutable.

**Examples**

The simplest immutable object is simply a collection of scalars or other immutable values:

```java
immutable class Pair {
    public int a;
    public int b;
}
```

When the JavaTime compiler translates an immutable class definition into standard Java it adds a no-argument constructor which initializes all member data items without initializers to default values, as well as a constructor which takes all non-initialized data members as parameters.

The JavaTime compiler also creates a default clone() method, which simply returns a pointer to the object, an equals() method that checks for both pointer and value equality, and a hashCode() method that creates a hash code based solely on the value of the object, not its identity.

```java
class Pair {
    public int a;
    public int b;

    public Pair() {
        a = 0;
        b = 0;
    }

    public Pair(int a, int b) {
```
this.a = a;
this.b = b;
}

public Object clone() {
    return this;
}

public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (! obj instanceof Pair)
        return false;
    Pair comparand = (Pair) obj;
    return a == comparand.a && b == comparand.b;
}

public int hashCode() {
    return a ^ b;
}
}

ByteVector takes an array (which is not immutable) as an input to its constructor. The JavaTime compiler automatically inserts code to copy the array (which it can do because it is an array of scalars), so the assignment of data to vector in the constructor of ByteVector does not create an alias with the input parameter. Because arrays are not immutable values, the vector member must be private (or protected).

finite immutable class ByteVector {
    private byte[] vector;

    public ByteVector(byte[] data) {
        vector = data;
    }

    public byte get(int i) {
        return vector[i];
    }
}

The Packet class shown below has public data members, since those members are themselves immutable. However, any code assigning to those fields would be flagged as a compile-time error by the JavaTime compiler.

The filter method of the Packet class returns true if the specified byte is contained in the packet data, and false otherwise. The public methods of finite immutable classes can be implemented as combinational logic in hardware.

finite immutable class Packet {
    public ByteVector address;
    public ByteVector data;

    public boolean filter(byte b) {
        for (int i = 0; i < data.length; i++)
            if (data.get(i) == b)
                return true;
    }
Copying vs. Alias Analysis

An immutable object and its methods can not side-effect any external objects because all input parameters and return values are copied, thereby preventing aliasing. Another way to prevent aliasing would simply be to use an alias analysis algorithm, and to reject any programs that introduced potential aliasing.

The problem with using alias analysis directly is that it would violate fundamental principles of abstraction. A programmer might call a library function from one of the methods of an immutable object; this library function in turn might call other functions, eventually creating an alias (or the appearance of an alias). Therefore, the compiler would reject the input program as illegal, and the only available diagnostic would be: “possible alias generated in library function Class.method”.

We have chosen instead to require explicit copying of inputs and outputs. However, alias analysis can be used in a conservative fashion with JavaTime, to determine when it is redundant for the JavaTime compiler to insert calls to clone(), and remove those redundant calls.

PURE OBJECTS

A pure object is like an immutable object in that it has no ability to side-effect external objects, but it does have modifiable local state. All parameters to public methods are copied on input. A pure object is subject to the following restrictions of immutable objects:

• public methods only take as parameters and return as results scalars, copyable objects, or arrays of either scalars or copyable objects, or nested arrays of either scalars or copyable objects.
• no references to non-constant (i.e. recursively final) global variables

However, pure objects also have the following properties:

• instance variables may be modified by any method
• all data members must be private or protected
• the == and != operations may be applied to pure objects
• synchronized methods may be invoked

Here example of a pure class, a simple class that maintains state:

```java
pure finite class FloatingInteger {
    private int x = 0;

    public synchronized void add(int i) { x += i; }
    public synchronized boolean isZero() { return x == 0; }
}
```
PORTS
In Java, multiple threads communicate through shared objects. Such communication is anathema to hardware because it is highly non-deterministic and very difficult to locate through analysis.

Shared memory communication is also problematic in distributed systems. While shared-memory semantics can be implemented in such an environment, they tend to be extremely inefficient. Ultimately, the problems in hardware and software have the same fundamental cause: a chip is also a distributed system.

To allow for the creation of distributed systems (in hardware or software) we need a more disciplined method of communication between threads.

Ports are communication channels that can be used to transmit values between threads, whether those threads are running locally in the same Java virtual machine or in different virtual machines that may reside on different processors connected by a network.

Ports may only be used to transfer copyable values and ports. A port has two ends: an output and an input end. Ports are typed; ports of integers are declared as follows:

```java
input int in;
output int out;
```

The fundamental operation on the output end of a port is to send a value; the fundamental operation on the input end of a port is to receive a value. Ports are implemented as Java objects with certain methods. Here are some examples:

```java
in = new input int;
out = in.connect();
out.send(3);
int x = in.empty() ? 0 : in.receive();
```

The JavaTime compiler automatically generates the port class definitions for an integer port when it sees the `input int` and `output int` declarations.

MODULES
The module is the basic abstraction of computation with state in the JavaTime language. Modules interact with each other only through ports. Ports may be used to communicate immutable values or ports.

There are many reasons for building a system out of modules. In the software domain, a system built out of modules can be distributed across multiple processors much more easily than by using the Java remote method invocation (RMI) facility. With the exception of specifying the machines where the modules are created, all aspects of distributing the application are handled automatically.
In addition, decomposing the system into modules provides the JavaTime compiler with powerful information that can be applied in optimizing the program. Because modules only communicate with other threads by passing immutable values across ports, all invocations of synchronized methods within a module can be converted into non-synchronized invocations.

Generalized modules can not be mapped directly into hardware. A restricted type of module that can be mapped into hardware, the synchronized module, is described below.

A module is a class that implements an asynchronous thread of control with its own internal state. Modules are implemented with the Java Thread class, but use the threads in a very restrictive way.

The best way to describe a module is by example. Here is an outline for a module which takes an image as input and applies JPEG compression.

```
finite immutable class Image { ... }

module CompressJPEG {
    public input Image in;
    public output Image out;

    always {
        Image source = in.receive();
        ...
        Image result = ... ;
        out.send(result);
    }
}
```

A module is a special kind of class. It must have an always method, which specifies the computation which is repeatedly performed by the module. The only public user-definable methods are constructors, which are subject to the same constraints as the constructors of immutable classes.

All modules must obey the following restrictions:
• no public data members except ports
• no public methods except constructors; the always method is protected and invoked by the run method, which is a public method generated by the JavaTime compiler.
• no calls to Thread.stop() or Thread.suspend()
• no calls to wait() or notify()

Modules may invoke synchronized methods, but since the only objects that modules can exchange with the outside world are values, those synchronization operations will always apply to module-local objects and will never block. Therefore, modules can make free use of classes from the Java libraries like Hashtable, Vector, and BitSet that are implemented with synchronized methods, and the JavaTime compiler can eliminate the synchronization operations.

The Java base class Module which is used to implement all JavaTime modules is defined as follows:

```
abstract class Module extends Thread {
    public void run() {
        while (true) {
            ...
        }
    }
}
```
try {
    always();
}
catch (Throwable t) {
}

abstract protected void always();

SYNCHRONIZED MODULES

In the hardware domain, modules correspond to functional blocks on a chip. To allow mapping into hardware, however, the dynamicity of modules must be somewhat restricted. The synchronized module is a restricted form of module that can be mapped into hardware that is synchronized with some sort of local clock.

Essentially, a synchronized module performs provably finite communication on each cycle, and provably does not deadlock.

Each invocation of the always method of a module is viewed as an abstract cycle; the actual cycle timings are implicit and are derived from the delay properties of the component modules and their methods. The always method must be finite, assuming that the receive operations will not block.

A synchronized module must satisfy the following requirements in addition to those required of all modules:

- the always() method and the constructors must be finite, assuming that the receive() method on input ports does not block
- ports may only me used to send copyable values, not ports
- module creation, port creation, port connection, and invocations of Thread.start() must be performed in initializers or constructors
- no threads may be created that are not synchronized modules
- no send() and receive() operations in constructors
- there must be a fixed, non-zero number of send()s and receive()s on each port during each iteration of the always() method
- it must be possible to divide the always() method of the module and its contained modules into subcycles such that no input port is received from or output port is sent to more than once per subcycle, and there are matching send() and receive() operations for every port that is used in a subcycle
- all values transmitted on ports must be of a fixed size
- the local state of the module must be allocated in the constructor and can not be subsequently expanded, for instance by inclusion of references to dynamically created objects.

The last condition is the most difficult to enforce. A crude way to enforce it is to disallow any object type that is reachable from the member data objects to be allocated outside of the constructor of the module. This condition is the “root” of the finiteness of storage for the entire language.
All methods of a synchronized module must be finite. All module creations, port creations, port connections, and invocations of `Thread.start()` must be performed in static initializers or constructors. All created modules must themselves be synchronized modules.

Each time its `always()` method is executed, a clocked module must perform a fixed number (at least one) of receive and send operations upon each of its ports. In the simplest case, a clocked module performs exactly one receive and one send upon each port each time the `always()` method is executed.

More complex send/receive patterns are permitted. However, it must be possible to divide the execution of the module and all of its contained modules into "subcycles" such that no port is read or written more than once per subcycle and the module does not deadlock.

**FINE-GRAINED PARALLELISM**

So far we have described mechanisms for coarse-grained parallelism (modules) and for helping the compiler identify potential medium-grain parallelism (copyable and immutable objects). However, a fundamental property of hardware is that fine-grained parallelism is cheap and plentiful.

Other hardware design languages like Esterel and Verilog have attempted to make fine-grained parallelism available via explicit fork-join operations. Unfortunately, unrestricted fork-join parallelism is quite costly in software. Including fork-join parallelism in the language forces programmers to choose between implementations that are efficient in either hardware or software, but not both.

Our approach is to include fine-grained parallel operators adapted from data-parallel languages like Fortran-90. These operators make it possible to express a lot of concurrency in a very concise fashion, and yet are also sufficiently high-level that they can be efficiently mapped into software constructs on a von Neumann architecture.

The simplest data-parallel statement in JavaTime is

```
a[0:10][0:10:2] = 0;
```

which can be implemented in standard Java as

```
for (int i = 0; i <= 10; i++)
    for (int j = 0; j <= 10; j += 2)
        a[i][j] = 0;
```

The expressions "0:10" and "0:10:2" are array section specifiers, and they must occur in the left-hand side of an assignment expression. When it is necessary to refer to corresponding members of multiple arrays, variables associated with each section can be specified, as in

```
a[int i=0:10][int j=0:10] = b[j][i]
```

which specifies a 10x10 matrix transpose.
The most general form is the *for each* statement:

```java
for each (int i = 0:10, int j = 0:10) {
    int t = a[i][j];
    if (t < 0)
        t = b[j][i];
    c[i][j] = t;
}
```

Data-parallel statements can be executed in any order, and arbitrarily interleaved. Assignments to any variables not locally declared must be provably unique. Invocation of `synchronized` methods, `synchronized()` statements, `wait()`, or `notify()` operations is forbidden.

### REDUCTION OPERATORS

The other form of parallel operation is reduction. The expression

```java
int x = +/a[0:10];
```

is equivalent to the standard Java

```java
int x = 0;
for (int i = 0; i <= 10; i++)
    x += a[i];
```

More generalized reductions can be performed using the *each* form from the *for each* statement:

```java
int x = +/each (int i = 0:10, int j = 0:10) a[i][j] * b[j][i];
```

Finally, the bit-wise boolean operators can be applied as reduction operators on integer values:

```java
boolean y = &/x;
```

### CONCLUSIONS

We have presented an initial design for JavaTime, a language designed to allow programs to be specified in such a way that they can be compiled into both hardware and software.

To compile a high-level program there are three fundamental issues that must be addressed:
- identification of multi-level parallelism, to make sufficient use of hardware;
- partitioning into units that can be mapped down to values transmitted over wires and functional units placed on the chip; and
- automatic determination of time and space consumption

We believe that our design for JavaTime does a very good job of solving the first two issues. Parallelism is manually exposed by the programmer in a natural way at a coarse-grain level by modules, and at a fine-grain level by data-parallel operators. Additional
medium-grain parallelism is relatively easy for the compiler to discover if the programmer makes use of immutable and pure objects.

Partitioning the program for hardware implementation is aided by the use of modules and pure objects, which specify functional blocks, and copyable and immutable objects, which specify types that can be passed between functional blocks over wires.

The area where the current JavaTime design is still weak is in automatic determination of resource requirements. This has to do with the fact that at compile-time, we are quite limited in our knowledge of what each object reference points to. Without knowing what a reference points to, it is difficult to know how much space the object consumes.

While we have made a considerable effort to avoid complicating the JavaTime compiler by requiring the use of complex alias analysis algorithms, it may be that such algorithms are indispensable for hardware compilers. One argument for requiring alias analysis is that every alias for an object will presumably have to be mapped into a wire connecting the object from the site of its use to its location on the chip. Clearly, reducing the number of such aliases will be absolutely critical to the production of reasonably efficient hardware implementations.

Finally, it is worth noting that specifications for low-level external timing constraints, as for instance in the UART chip design in Sternheim et al [1990], will not be made any less complex by using a high-level language. In fact, it may become even more cumbersome to express them. Using JavaTime to specify a UART chip would be like trying to write a device driver in Java.

While the implementation of external timing constraints will remain complex, JavaTime will allow each individual module to be vastly more functional. A UART is an extremely primitive device. JavaTime might be used to design a serial bus interface chip that also does automatic encryption, compression, and various other functions. In such a design, the advantages of JavaTime would be reaped because the description of the high-level functions would be vastly simpler than in Verilog and could probably use an existing software implementation with only minor changes.

REFERENCES


