This paper begins with a brief description and comparison of existing techniques for describing and representing digital waveforms. We then introduce a new object-oriented data representation which allows timing, data, and format to be manipulated separately while providing the ability to reflect complex device cycle structures as well as sophisticated tester timing capabilities. We then present several examples illustrating transformations which can be performed on this representation while preserving the original waveforms. Next, we present several applications where this representation has been used. Finally, we provide a summary of our experiences with this representation including a short discussion of implementation experience and space requirements.

**WAVEFORM REPRESENTATIONS**

Descriptions of digital waveforms must incorporate timing and data. Some representations combine data and timing implicitly while others (historically because of data compaction requirements) require that timing and data be described separately. Often, for the sake of either compactness or clarity, the notion of format is introduced and made explicit as well. Most descriptions of waveforms fall into one of three categories: Stimulus Languages, Interchange Formats, and ATE Languages.

**Stimulus Languages**

These languages often evolved as a shorthand for specifying stimulus pattern data to simulators. Some permit the specification of expected response data as well. Many stimulus languages provide only low-level primitives such as "set node high at 10ns" while others have fairly sophisticated algorithmic capabilities and can be programmed.

Data and timing are seldom treated as equally visible and important aspects of device stimulus. That is, few languages permit the description of device cycle timing independent of the pattern data supplied during each device cycle. Some are capable of representing structure (e.g., buses, device cycles) but few have capabilities for representing tester resources.
Because these languages are intended to provide data only to a specific simulator, they are not generally useful as a means of either communicating or representing waveforms in general. Further, the languages often reflect features of the particular simulator and are thus not transportable either to other simulators or to the test domain.

**Interchange Formats**

These are typically tabular state vectors or event-oriented lists. They are generally ASCII text files and easily transportable. Because of their simplicity and lack of algorithmic constructs they are easily interpreted and translated into other representations. However, they do not provide algorithmic constructs or a means of representing test resources. Thus they cannot represent and communicate information about device cycles or serve as an intermediate representation between two different test systems. Interchange formats rarely separate format and data from timing issues and instead typically provide only a table of events or "print-on-change" data in which the time of every event change is given along with the state of every signal at that instant.

**ATE Languages**

These languages are targeted to specific testers and generally represent structure well. Because of memory limitations and tester architecture evolution, these languages tend to provide fairly compact representations which handle timing, data, and format independently.

However, tester programming languages necessarily reflect those hardware features that are available on a specific tester. Because they are meant to take full advantage of a specific tester architecture they are often idiosyncratic. Not only will one tester family not execute programs written for another, but even manually porting test programs is complicated by the semantic mismatch between tester architectures. A test program written for a target tester supporting multiplexed timing generators and pins as well as multiclock modes may be very difficult to port to a tester supporting only simpler timing capabilities.

**THE SEF DATA REPRESENTATION**

We have produced a new waveform representation that we believe has several advantages over each of the preceding formats, called "SEF" for "Standard Events Format". SEF is an object oriented, hierarchical waveform database that is tester and simulator independent. The SEF database described here is an enhanced version of an earlier form of our waveform database, which is now rich enough to represent directly most of the features and capabilities of devices and testers.

We have implemented SEF using object oriented techniques using the C programming language. Each object class has its own collection of methods, and no class is allowed to access directly the private data of any other.

Several of the waveform representations discussed above are "flat", in that they have no organization of events other than a monotonically increasing event time. Others, such as ATE languages, use cycle-based data. SEF is a hierarchical representation, the lowest level of which is a single cycle. This hierarchy is actually extra information that isn’t available with flat representations.

SEF is hierarchical in two senses -- there is both a hierarchy of objects that make up the data and a hierarchy of cycles and sequences of cycles. We'll refer to these as the "object hierarchy" and the "data hierarchy" and will describe each more fully below.

The main features that distinguish the SEF representation from other waveform representations are:

1. it is tester and simulator independent
2. it contains more information
3. it is usually more compact
4. it provides support for meaningful transformations

**The Data Hierarchy**

Waveform data in SEF may be partitioned into sub-sections of time, and each of those portions of time may be further subdivided. The lowest level of this hierarchy is the cycle, i.e. a section of time described by a period and zero or more events on one or more signals. This type of hierarchy allows data compression into subroutines, loops, and repeats.
Waveform data in SEF may also be partitioned into subsections of signals, and again subdivided as needed. This allows SEF to represent parameterized subroutines, "banks" of timing sets, interface circuitry (with both synchronous and asynchronous signals), and other device or tester characteristics.

Together, these two methods of dividing waveforms into sections provide what we call the "data hierarchy".

Waveforms may be partitioned by time

Each of the cycle-level objects (timing, format, and data) refers to a collection of signal-level objects that describe the timing, format, or data of each signal. The cycle-level objects are roughly analogous to the timing sets, format registers, and data vectors of a modern tester, although they are considerably more flexible than these tester structures. The signal-level objects are roughly analogous to the timing generators, signal formats, and data bits of a tester. (See figure 2.)

The Object Hierarchy

The top level of the object hierarchy in SEF is the basic building block of the data hierarchy -- the waveform object. A waveform object refers to either a collection of other waveform objects or a collection of cycle objects and a signal group object.

A cycle object refers to the cycle timing, cycle format, and cycle data objects that together describe the behavior of a collection of "columns" for a cycle. A signal group object gives signal names to those columns.

Each signal-level timing or format object may be arbitrarily complex. Any number of events may occur on any signal within a cycle, and the resulting waveform may be of any desired shape. The signal format is a list of shape symbols, each of which is either an explicit state (level and direction) or an indication to use the data symbol for that signal. Typical tester formats are easily described using this approach, but any waveform shape is possible. (See figure 3.)

Any object may have an arbitrary named property attached, which provides the flexibility to represent many other types of data.
Modelling Device Characteristics

SEF provides a natural method for describing device behavior. For example, it's very easy to describe a microprocessor "Read Cycle" or "Write Cycle". SEF provides a way to represent device behavior in a "meaningful" way due to the flexibility in organizing the data hierarchically.

Object properties allow other device specific data to be represented, such as edge relationships [1], Min/Max values, and stochastic information.

Modelling Tester Capabilities

As we've already mentioned, SEF allows a convenient representation of tester resources such as timing sets, format registers, and timing generators. SEF also provides a natural representation for:

1. MUX modes (TG or Pin MUXing)
2. Repeat modes
3. Pattern compression
4. Homing Loops/Loop-Until-Pass constructs

Most importantly, though, SEF is tester independent, which allows data to be moved easily among ATE and/or CAE environments.

TRANSFORMATIONS ON SEF

Most of the time, the transformations that one might wish to apply to SEF are those that add additional information to the data. For example, in some cases a single device cycle may require more than one tester cycle to reproduce, due to tester limitations. In these cases the information to be added is, indirectly, the tester limitation that applies. It's often possible to divide the device cycle into two or more "sub-cycles", that can be reproduced on the tester. Due to the way in which SEF is represented, this transformation need only be applied in one place -- to the timing and format for that cycle type. All cycles of that type are then automatically divided into testable fragments.

Other times the information to be added is the device cycle boundaries. For example, most logic simulators produce "flat" output waveforms without any cycle information. In order to represent that data in SEF, our default method is to partition the waveforms up into very short "cycles", so that no cycle has an event on any signal other than at the start of the cycle. Later, many small cycles may be merged into one larger "meaningful" cycle as information about device or tester cycle boundaries is added either automatically or by the user.
In the two cases above, the intent is to keep the flat representation of the waveforms the same but add information about where cycle boundaries are to go. In other cases you may want to add other sorts of information that will actually change the flat waveform representation.

For example, in order to change the data valid region for all cycles that have the same "shape" (timing and format), all you need to do is change the single cycle timing object to which they refer.

If you want to add cycles of the same shape but different data, you need never modify the timing or format objects at all. In this case, you are simply adding data vectors and using existing cycle timing and format objects to provide a "template".

The simulation information presented to PBRIDGE most closely resembles the interchange formats described in an earlier section. This flat representation contains no cycle information. However, in order to make intelligent use of a tester's resources and indeed, to program a tester at all, the unstructured event stream must be partitioned into cycles. As mentioned above, this initial transformation into the SEF representation may merely partition the flat waveforms into cycles of simple NRZ format. Once in this representation, trade-offs can be made between formats and timing constraints, for example by transforming the SEF representation from one "implementation" to an equivalent one. Transformations might change attempt to create more complex cycles, changing signal formats from NRZ to either DNRZ or RZ formats where possible.

**Applications -- generating test programs**

The SEF representation is being used in PBRIDGE, an automatic test program generation module which takes as input simulation log files and produces functional test programs. Using the results of a digital logic simulation, the PBRIDGE module automatically determines appropriate cycle times, assigns signal formats, and programs timing generators and timing sets. PBRIDGE identifies and exploits opportunities for pattern compression through the use of the target tester's algorithmic looping and subroutine constructs.

The simulation information presented to PBRIDGE most closely resembles the interchange formats described in an earlier section. This flat representation contains no cycle information. However, in order to make intelligent use of a tester's resources and indeed, to program a tester at all, the unstructured event stream must be partitioned into cycles. As mentioned above, this initial transformation into the SEF representation may merely partition the flat waveforms into cycles of simple NRZ format. Once in this representation, trade-offs can be made between formats and timing constraints, for example by transforming the SEF representation from one "implementation" to an equivalent one. Transformations might change attempt to create more complex cycles, changing signal formats from NRZ to either DNRZ or RZ formats where possible.

**Emulators -- emulating test programs**

Test program emulators provide a vehicle for "emulating" a test program written for a specific tester, applying timing, data, format, and algorithmic sequencing operations to produce a tester-independent representation of the waveforms. These waveforms can then be used to produce a test program for another tester.

For example, a test program written for a prototype verification system can be emulated and the resulting waveforms used in a PBRIDGE-generated test program for a production tester. Similarly, waveforms used in a production test program for an ASIC might be captured for use in an in-circuit model library element. Another application of emulators is in closing the loop between design and test: using the waveforms from a test program to provide stimulus and expected response for fault simulation to determine the actual fault coverage of a test program.

In designing and implementing emulators, one need only write a different parser and code to build the corresponding SEF representation for each tester. The resulting SEF representation can be interpreted independently of the tester to produce a "flat" (or interchange) waveform representation. Alternatively, through a series of transformations on the SEF, one can produce equivalent structures which correspond more closely to another tester and then generate a test program for that target tester.

**Applications**

Several applications have been written using the object-oriented representation described. They comprise several independent modules of TDS, a system for linking design and test. We describe three such applications here.

**PBRIDGE -- generating test programs**

The SEF representation is being used in PBRIDGE, an automatic test program generation module which takes as input simulation log files and produces functional test programs. Using the results of a digital logic simulation, the PBRIDGE module automatically determines appropriate cycle times, assigns signal formats, and programs timing generators and timing sets. PBRIDGE identifies and exploits opportunities for pattern compression through the use of the target tester's algorithmic looping and subroutine constructs.
**WaveMaker -- interactively manipulating waveforms**

WaveMaker is an interactive graphics environment for creating, viewing, manipulating, and searching waveforms. Its interface directly reflects the structure of SEF, permitting the user to independently create, view, and modify timing, data, and format. Separate but cooperating editors permit the simultaneous viewing of all three components. In addition, the user can view, search, and edit the flattened waveforms which appear with timing, data, and format combined. Designers change global system timing by simply changing the timing specification without needing to rearrange or resequence pattern data. Similarly, compare strobe windows can be placed and moved without the need to change expected pattern data. The effects of these changes are always visible in both the hierarchical and flattened forms.

**PRACTICAL EXPERIENCE**

We have been developing SEF since 1985, but this latest revision has been a major rewrite that we started in 1987. Referring only to this last phase of development:

1. The SEF database includes approximately 25,000 lines of C code. We didn't use C++ or another language with support for object oriented constructs due to the lack of a good symbolic debugger on all of the hardware platforms we support. The benefits of this object oriented programming style have included code reusability, enforced modularity, simplified support, and parallel development.

2. There are currently 6 applications that use the "enhanced" version of SEF, totalling 69,000 lines of C. (Not counting unit test programs.)

3. All code has been ported to Sun, Apollo, VAX (VMS), and HP9000 workstations.

4. We have spent a total of approximately 23 man months of full time development on SEF enhancements.

5. The average data sizes for SEF files are significantly smaller than state or event based representations, even without taking advantage of data compression using subroutines, loops, or repeats. Using our published ASCII state-based format as the basis for comparison, the relative sizes across a sample of testcases are:

   - 100% -- Flat, state based
   - 41% -- Flat, event based
   - 20% -- SEF (uncompressed)

At this time we are continuing to work on SEF. In particular, we are currently evaluating printed representations of SEF and adding further capability to support data transformations and operations.

It's not possible at this time to give experimental results comparing the execution speed of a product that uses SEF versus the same product that uses another format, as we have not implemented the same product using both methods.

**ACKNOWLEDGMENTS**

The authors wish to thank Bill Den Beste and Tom Hamilton for the original impetus for this project, and for many of the basic ideas behind this investigation. We also wish to thank Josi Althouse and Jeff Hahs for their assistance in writing the database code for SEF. Thanks also to Vance Noel for assisting with the illustrations and layout of this paper.

**REFERENCES**


