JOURNAL OF ELECTRONIC TESTING: Theory and Applications 13, 299–314 (1998)

A Behavior Model for Next Generation Test Systems

LEE A. SHOMBERT
Intermetrics, Inc., 7918 Jones Branch Drive, Suite 710, McClean, VA 22102
las@wash.inmet.com

JOHN W. SHEPPARD
ARINC, 2551 Riva Road, Annapolis, MD 21401
jsheppar@arinc.com

Received June 20, 1997; Revised August 18, 1998

Editor: W. Needham

Abstract. Defining information required by automatic test systems frequently involves a description of system behavior. To facilitate capturing the required behavior information in the context of testing, a formal model of behavior was developed for use by test systems. The approach taken in defining the behavior model was based on information modeling and was derived from recent work in formal methods by the hardware and software design communities. Specifically, an information model was developed in EXPRESS capturing the relationships between essential entities characterizing behavior. In this paper, we provide a high level description of the behavior information model and several examples applying the model in a test environment.

Keywords: behavior modeling, automatic test systems, test requirements, test programming

1. Introduction

Defining information required by automatic test systems frequently involves a description of system behavior in one way or another. Behavior is a characteristic of an entity that describes how that entity acts or reacts within some context or environment. Within the context of test systems, behavior is defined by what is observed as a result of testing.

Recent work in defining an architecture for next generation test systems has determined that behavior descriptions are relevant in at least five contexts:

1. Characterizing expected behavior by the product
2. Defining test requirements
3. Defining resource capabilities and requirements
4. Defining behavior of test strategies
5. Guiding system diagnostics.

To date, behavior has been treated using ad hoc methods or relying on simulator-specific approaches to representation. Hardware description languages such as VHDL and Verilog have attempted to provide "standard" approaches to capturing behavior in digital systems but are still targeted at simulating a design. Further, existing HDLs are not specifically designed for addressing the needs of a test engineer.

To facilitate capturing the required behavior information in each of these contexts, we developed an information model, defining behavior. In this paper, we provide a high level description of the behavior information model and several examples applying the model in a test environment. Specifically, we focus on applying the model to capturing test requirements and resource capabilities.

The approach taken in defining the behavior model is derived from recent work in formal methods by the hardware and software design communities [1–3].

Formal methods are mathematically based languages used to capture essential attributes of a system being designed. These methods are used, typically, to guide the process of design verification and proof-of-correctness. As such, most formal methods apply a "declarative" approach to specifying systems. Declarative approaches specify systems by stating logical properties and relationships among entities within the system [4, 5]. The process of instantiating such a design corresponds to assigning legal values to parameters within the system description such that the properties and relationships defined by the description are consistent.

The approach taken in defining the behavior model was to develop an EXPRESS information model capturing the relationships between essential entities characterizing behavior [6, 7]. The approach is declarative in that, in the simplest case, the process of using the behavior model consists of matching an implementation to a model to verify that the two correspond to one another. The primary advantages to using a declarative approach come in the ability to "reason" about the performance of the system under test and evaluate test requirements versus resource capabilities. Further, such reasoning can be completed without the need to simulate either the system or the resources explicitly. Thus, in the case of selecting an appropriate resource to perform a test, a model of the required behavior is instantiated and compared with a model of the possible behaviors provided by each available resource. The resource is selected that best matches the required behavior.

In the following sections, we discuss a simplified view of the EXPRESS behavior model as well as a slightly more detailed view. This discussion is put in the context of a trivial test program to highlight and describe the major entities of the model. Following this introduction, we provide several examples using the behavior model in the context of test requirements, test program synthesis, resource behaviors, and triggers or events [8–10].

2. The Behavior Model

A simplified view of the *behavior model* is shown in Fig. 1. This simplified model will be expanded later, but it is sufficient to explain the basic concepts. There are five entities in this model.

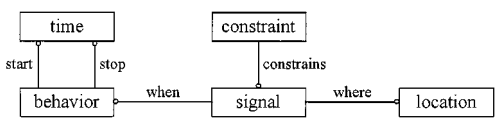

Fig. 1. Simplified behavior model.

1. `location` captures where something happens. In the current model, `location` can be ports on or in a unit under test (UUT) or cells in a UUT.
2. `behavior` captures when something happens. This entity is distinct from the whole behavior model in that it is limited to identifying a span of time over which something occurs. This entity is defined by its `start` and `stop` attributes.
3. `signal` captures what happens. Types of `signals` include the `DC_SIGNAL` and `AC_SIGNAL`, as well as standard programming data types, e.g., integer, real, or Boolean.
4. `constraint` defines rules constraining or restricting which values may appear on `signals`.
5. `time` exists in the model solely to support the definition of the `behavior` entity. It is a subtype of the `property` entity (defined below) and is used to "type" the `start` and `stop` attributes of `behavior`.

The behavior model can be used to describe requirements on a test program and capabilities of a test resource.

A more detailed view of the behavior information model is shown in Fig. 2. In this figure, the five entities from Fig. 1 are shown plus several additional entities capturing constraint information. Significant in this figure is the observation that a `behavior` can be a composite of lower level `behaviors`, ultimately containing zero or more `signals`. Both `behavior` and `signal` are characterized by a set of `properties`, each of which can be constrained in some way. Further, in addition to `constraints` being applied to `signals`, `constraints` can also be applied to `behaviors`. Finally, the current version of the behavior model identifies four types of `constraints`:

- `range` constraints (limiting a property to lie within some range of legal values),
- `accuracy` constraints (identifying the acceptable variation of a property value from a specified reference),
- `timing` constraints (indicating legal ordering or timing relationships with respect to another variable),
- `value` constraints (indicating possible legal values, such as discrete values).

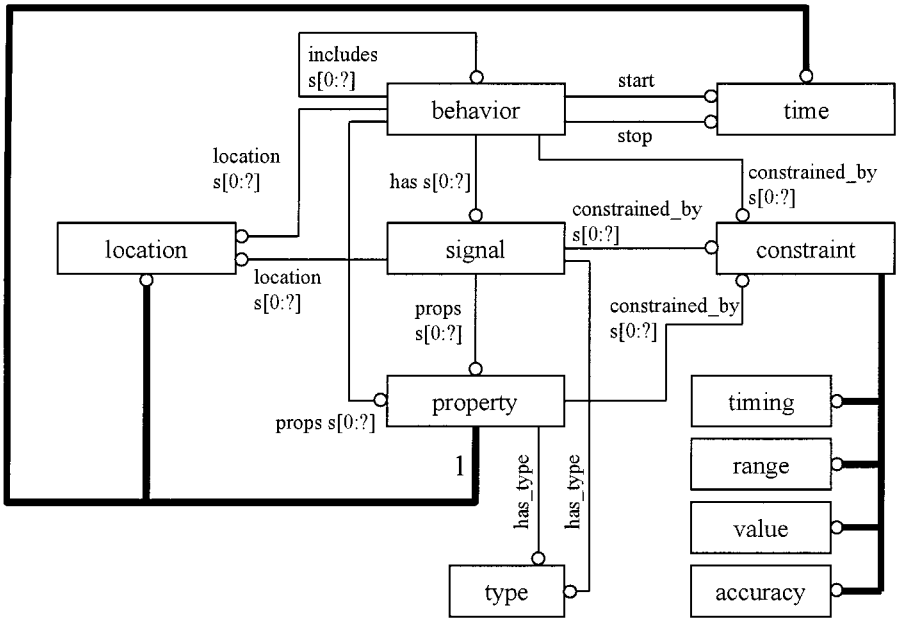

Fig. 2. Detailed behavior model.

Arbitrary relationship constraints can be defined in the `constraint` supertype without the need to instantiate one of the subtypes.

3. Interpreting Behavior

At first glance, it appears that the behavior model describes, for example, how a test program must execute. This is almost, but not quite, correct. The behavior really provides a set of criteria by which one can determine if a test program executed properly. To see why this distinction is important, consider a UUT with two power supplies that may be used to establish power once the ground has been established. No testing may be done on the UUT until after the power supplies have been established. This situation is shown in the flow chart in Fig. 3.

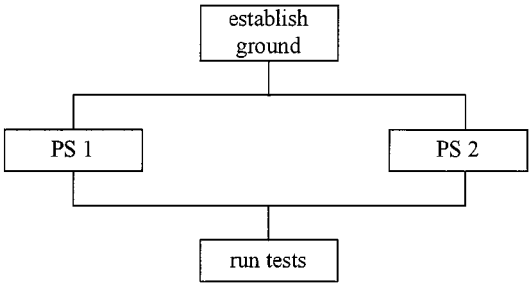

Fig. 3. Flow Chart for providing power for tests.

The branches labeled "PS 1" and "PS 2" can execute concurrently or sequentially—the only constraints are that the branches cannot begin until "establish ground" is complete, and that "run tests" cannot begin until both power supply branches are complete. Now consider a hypothetical test program. Because the test program is sequential, it might look something like the following:

```
establish ground
establish PS_1
establish PS_2
run tests
```

If the "`establish PS_x`" routines can be decomposed into a more basic set of statements (consisting, for example, of `setup` and `apply` statements), then the following version of the program is also legal.

```
establish ground
setup PS_1
setup PS_2
apply PS_1
apply PS_2
run tests
```

Many different legal execution paths may be taken by a test program. It would be inappropriate for a behavior specification to state that exactly one of those paths is correct and all others incorrect. Such an approach would improperly restrict other legal paths.[1]

The behavior model describes the desired result of a test program rather than the test program itself. The previous example provided pseudocode for a test whose objective was the verification of a specified behavior. If the example was executed, real signals would be generated by the test equipment and applied to the UUT, and real signals would be observed or measured. Ultimately, the behavior model specifies requirements on those signals.

Suppose that, following execution of a test program, a complete record of all signals at the interface to the UUT is available. For the sake of discussion, assume these signals have been recorded with infinite precision. This record is called an *execution trace* of the test program. For the example above, the execution trace would include the voltage and current at the ground and power supply pins of the UUT. After a test program has executed, the execution trace can be examined to see what the voltages were at any time during the test.

The behavior model only defines constraints on the execution traces. The constraints are not applied directly to the test program—they only influence decisions made in configuring and executing the test program. The model is a "declarative" representation of behavior and does not explicitly *prescribe* the behavior of the test program as might be expected from an imperative (e.g., procedural or functional) description. Instead, it *describes* the behavior that is expected from the program.

The following examples are provided using a language designed to implement the behavior model. This language, called the Test Requirements Model (TeRM) is defined in [11]. The TeRM language has been invented for prototype and illustration purposes and is expected to evolve further. The language is used to declare instances of entities from the behavior information model. The general form of a declaration in TeRM is:

```
(keyword name body)
```

The complexity of the body depends on the entity being defined. Behaviors, for example, contain declarations of behavior instances (i.e., sub-behaviors) as well as signals, constraints, and other attributes.

Note that the following examples are generally fragments of a complete TeRM description and depend on declarations of signals and behaviors whose characteristics should be clear in context. Ellipses are used to indicate the presence of bodies that have been left out for clarity.

Let us refine the elements of the behavior model for our example. First, the use of the phrase "establish ground" must be clarified. On a tester, this means that the ground pins of the UUT connect to the digital, analog, or system ground on the tester. Next, consider what it means to "establish PS_1." This means that there is a stable voltage of the proper value at the proper pins on the UUT. This can be broken into two pieces: the declaration of a voltage at the UUT pins and a constraint on the value of the voltage:

```
(behavior-def DC_SIGNAL ((pins VoltageLocation) (voltage V))
     ...)

(behavior-def establish_PS_1 ()
     (signal Vcc1 DC_SIGNAL (VoltageLocation Ps1_Pin Gnd_Pin))
     (and (> Vcc1.voltage (V 4.75)) (< Vcc1.voltage (V 5.25))))
```

At this point, four of the five entities in the simplified behavior model are evident:

- There is a signal named `Vcc1`. The type of the signal is `DC_SIGNAL`, where the type is an extension to the simplified model (as shown in Fig. 2).
- There are two locations, named `Ps1_Pin` and `Gnd_Pin`, where the signal Vcc1 occurs.
- There is a constraint that restricts the `voltage` attribute of `Vcc1` to lie between 4.75 and 5.25 V.
- There is a behavior named "`establish_PS_1`" that defines the period over which the property definitions are active.[2]

This behavior model states that there is a voltage between two pins on the UUT and that the voltage must lie within the range 4.75–5.25 V. The behavior does not provide the actual value for the voltage or even constrain the value to be constant for the duration of the behavior (however, constancy is implied by the type, DC-SIGNAL).

Given this model, a test program can be verified to satisfy the objectives and constraints of the model, thereby verifying that the test program is "behaving" as specified. Formally, the test program itself cannot be verified. Rather, the test program can be run and the resulting execution trace examined. From the execution trace, the voltage corresponding to `signal Vcc1` can be determined to ensure the voltage lies within the proper range for the duration of the `establish_PS_1` behavior.

Notice that no `time` entities are defined for the example. In a test program, the behavior `establish_PS_1` begins at some time and later ends. Note that no specific values have been assigned, nor should they be. For this example, the `start` and `stop` times are identified by examining the execution trace of the program rather than being included in the test program to prescribe specifically when the behavior occurs.

In considering the whole test program, however, one finds that the `start` and `stop` times *are* specified, but in terms of timing constraints. For example,

```
(behavior-def whole_test_program ()
     (behavior-ref PS1 establish_PS_1)
     (behavior-ref PS2 establish_PS_2)
     (behavior_ref RT run_tests)
     (and (<= PS1'stop RT'start) (<= PS2'stop RT'start)))
```

In this example, the behavior defined by `whole_test_program` contains three sub-behaviors (named `PS1, PS2,` and `RT`). Behavior `whole_test_program` also contains a constraint that relates the `start` and `stop` times of the three sub-behaviors such that the ordering of the behaviors conforms to the flow chart in Fig. 3. Notice that the `start` and `stop` times, while specified in the constraint, do not have values specified anywhere. As described above, the actual values are derived from the execution trace of the test program.

4. Applications of Behavior

In the following sections, we will provide several examples using the behavior model. These examples are simplified, but they serve to further illustrate the concepts introduced in the previous section.

The behavior model can be applied during test requirements specification, synthesis, and verification. A common thread through these applications is the use of the behavior model for test program specification. Test resource allocation is a fourth application that can occur during test program development (i.e., static allocation) or during test program execution (i.e., dynamic allocation). Other applications are expected as the model becomes more widely used.

4.1. Test Requirements Specification

The behavior model can be used to capture test requirements. Test requirements, in this context, are requirements on the test program itself, not requirements on the test development process [12, 13]. The behavior model addresses test requirements such as:

- A safe-to-turn-on test must be performed before any other test.
- Tests must be run over a range of ambient temperature.
- Frequency stability must be measured at 1.250000 MHz.

Using these requirements as examples, we will demonstrate how to represent requirements in the behavior model and how to use the requirements in test program synthesis and verification.

The above test requirements are provided in English and, although understandable to engineers, are difficult for a computer to understand. The first step for specifying requirements in a machine-understable way is to add detail to the above statements until they can be expressed formally.

4.1.1. Safe-to-Turn-On Test Requirement. A safe-to-turn-on (STTO) test is defined as a behavior that precedes all other behaviors in the test program. The test program itself represents an enclosing behavior that includes the STTO test and all other tests. The STTO behavior typically tests for shorts between power and ground with a direct resistance measurement and then tests for excessive current when the UUT is powered. The STTO "returns" a value that determines whether

any further tests may be run. Specifically, the STTO behavior can be represented as follows:[3]

```
(behavior-def safe_to_turn_on ((safe_to_test boolean))
     ...)

(behavior-def run_tests ()
     ...)

(behavior-def whole_test_program ()
     (behavior-ref ST safe_to_turn_on)
     (behavior-ref RT run_tests)
     (if RT'execute (>= RT'start ST'stop))
     (if RT'execute ST'safe_to_test))
```

In this model, a behavior is defined as `whole_test_program` that contains two sub-behaviors: `ST` and `RT`. Behavior `ST` is an instance of the `safe_to_turn_on` behavior (defined above). `ST` has a return parameter called `safe_to_test`. Behavior `RT` simply encapsulates all other tests in the test program.

The first constraint ensures that, if `RT` actually executes, then `RT` follows `ST`. The second constraint ensures that `RT` can only execute if behavior `ST` returns TRUE for the `safe_to_test` parameter. This constraint looks "backwards" because the behavior that is supposed to occur (`RT'execute`) appears as the test to the `IF` statement, and the behavior that is supposed to have occurred appears in the `THEN` clause of the `IF`. Behavior descriptions are declarative—they do not prescribe how to execute a test program; they describe what must be true after the test program executes.

After the test program executes, the execution trace will have a value for `RT'execute` and `ST'safe_to_test`. The only way the constraint can be satisfied is if both `ST'safe_to_test` and `RT'execute` are TRUE, or if `RT'execute` is FALSE. Notice that `RT` is not required to actually run. Instead, the requirement is that `RT` *not* run unless the UUT passes its safe-to-turn-on test.

4.1.2. Test Over Temperature Test Requirement. Products typically have a temperature range over which they are expected to operate. This normally implies that the product must be tested at more than one ambient temperature. Let us assume that our example product will be tested at three temperatures: room temperature, a low temperature, and a high temperature. This assumption can be captured by first expanding the definition of the `run_tests` behavior to include the notion of ambient temperature.

```
(behavior-def run_tests ((ambient temperature))
     ...)
```

Next the definition of `whole_test_program` is expanded to require that `run_tests` execute at least once with the temperature around room temperature, once with the temperature near the low end of the range, and once with the temperature near the high end of the range.

```
(function-def in_range () boolean
     ...)

(behavior-def whole_test_program ()
     (behavior-ref ST safe_to_turn_on)
```

```
(behavior-ref RT_room run_tests)
(if RT_room'execute (>= RT_room'start ST'stop))
(if RT_room'execute ST.safe_to_test)
(if RT_room'execute (in_range room_range RT_room.ambient))

(behavior-ref RT_low run_tests)
(if RT_low'execute (>= RT_low'start ST'stop))
(if RT_low'execute ST.safe_to_test)
(if RT_low'execute (in_range low_range RT_low.ambient))

(behavior-ref RT_high run_tests)
(if RT_high'execute (>= RT_high'start ST'stop))
(if RT_high'execute ST.safe_to_test)
(if RT_high'execute (in_range high_range RT_high.ambient)))
```

The whole_test_program description has been modified to include three copies of run_tests, each of which is constrained to run within a particular temperature range (denoted by names such as room_range, low_range, and high_range, that would be defined in a full model). Note there are no sequencing constraints between the various run_tests; an implementation is free to pick whatever temperature sequence it deems most appropriate.

Also, notice that none of the run_tests are required to execute. In this particular example, the decision about execution is left to the diagnostic controller. The diagnostic controller might decide, for instance, not to test at high and low temperature if the room temperature test fails. Such a decision is entirely consistent with the test requirements in the example [14].

4.1.3. Frequency Stability Test Requirement. A frequency stability test might verify that a UUT output signal has a constant frequency. The term "constant" must be qualified to mean "constant within some error." This test requirement can be captured by defining it in terms of a behavior. One approach to specifying the behavior is:

```
(behavior-def AC_SIGNAL
     ((location Hi) (location Gnd) (frequency Hz))
      ...)

(behavior-def frequency_stable ()
     (signal Osc AC_SIGNAL (VoltageLocation Osc_Pin Gnd_Pin))
     (< Freq_Error (abs (- Osc.frequency (MHz 1.25)))))
```

This model defines a behavior, frequency_stable, with one signal that is of type AC_SIGNAL. There is a single constraint that forces the frequency to be "close to" 1.25 MHz, where "close to" is defined as being within Freq_Error. Unfortunately, this description is incorrect because one does not know if the oscillator is really stable, nor can it be forced to be stable. The current description demands that the oscillator be stable and invalidates any observation of behavior for which the oscillator is not stable.

The description needs to report whether or not the oscillator is stable rather than forcing the stability of the oscillator. Again, the focus is on describing rather than prescribing behavior. For example,

```
(behavior-def frequency_stable ()
     (variable Osc_Ok boolean)
```

```
(signal Osc AC_SIGNAL (VoltageLocation Osc_Pin Gnd_Pin))
(== Osc_Ok
    (< Freq_Error (abs (- Osc.frequency (MHz 1.25))))))
```

This description says that the `Osc_Ok property` is TRUE if the oscillator frequency is stable and FALSE otherwise. Notice that `Osc_Ok` is not required to be TRUE. It will be TRUE for a fault-free UUT and may be FALSE for a faulty UUT. The behavior `frequency_stable` is permitted to take on either value, and it is reasonable to expect that diagnostics will use `Osc_Ok` (among others) to indicate whether the UUT is faulty.

4.2. Test Program Synthesis

Because it is declarative, a behavior description does not usually capture enough information to directly generate code for a test program. However, a behavior can be used to guide test program generation [15]. In general, the code synthesis process can follow a constraint satisfaction process coupled with a code generator as a side effect [16, 17]. For example, in determining the order in which to execute `RT` and `ST`, legal values (or ranges of values) need to be instantiated for the `RT'start` and `ST'stop` variables. As a side effect of instantiating these values, a code generator can determine that functions associated with `RT` and `ST` must be generated that are sequenced according to these legal values.

```
void whole_test_program ( void ) {
  boolean safe_to_test;
  -- Execute Safe_To_Turn_On,
  -- getting the safe_to_test parameter back.
  --
  safe_to_turn_on ( &safe_to_test );
  -- Execute Run_Tests only if safe_to_test is true. --
  if ( safe_to_test ) {
      run_tests();
  }
}
```

For illustration, consider the example from the previous section describing the STTO test. For this example, when synthesis begins, code must be generated to implement `whole_test_program`. This might involve including standard startup code and may include the generation of site-specific user interfaces. The test requirements in the behavior description are silent on such issues, and the synthesis program is expected to generate such details from other sources of information. The separation of test requirements that depend only on the UUT from information about the test equipment or local test procedures is deliberate and is a principal advantage to using the behavior model. The remaining discussion will assume that the synthesis program will be adding local information, and will talk only in terms of satisfying the test requirements.

Having begun `whole_test_program`, one must select a component of the test program to synthesize. There are four statements in `whole_test_program`, two `behaviors` and two `constraints`. Note that the first `constraint` prohibits running `RT` before running `ST`. Also note that the second constraint cannot be evaluated until `ST` executes. Therefore, `ST` will be synthesized first, which is exactly what common sense would dictate.

Now a branch that executes `RT` only if `ST` returned TRUE for its `safe_to_test` parameter will be synthesized. When using a programming language like C, the code for `whole_test_program` would resemble the following skeleton:

Note that the constraint that `ST` run before `RT` is ensured by the order in which the `safe_to_turn_on` and `run_tests` routines are called in the test program.

Also, note that the C program executes `run_tests` if `safe_to_test` is TRUE, even though the test requirements did not demand this. Often test programs are more constrained than the test requirements due to considerations such as operator convenience, test time minimization, or other concerns not directly related to the requirements.

In this case, `B2` is guaranteed to follow `B1` unless some catastrophic failure occurs in the compiler or in the host computer.

Other aspects of an execution trace can be predicted with less certainty. For example, consider a behavior that contains a `signal` whose voltage is constrained to a small range around 3.3 V:

```
(behavior-def some_voltage_test ()
     (signal Vcc DC_SIGNAL (VoltageLocation Ps1_Pin Gnd_Pin))
     (< (V 0.1) (abs (- (V 3.3) Vcc.voltage))))
```

4.3. Test Program Verification

Test program verification is the process of comparing an existing test program with a set of test requirements to ensure that the program satisfies the requirements [18]. As discussed earlier, the behavior description of the test program constrains its execution trace, and only the execution trace truly can be verified. Note, it is both impractical and inefficient to verify each execution trace; therefore, the alternative is to analyze the test program itself and predict whether or not all execution traces that can be produced by the test program will be correct.

Some aspects of an execution trace can be predicted with high confidence. Gross timing relationships are a good example. In the following example, the behavior `some_test` has two sub-behaviors (`B1` and `B2`), and `B2` must occur after `B1`.

```
(behavior-def some_test ()
     (behavior-ref B1 Test_A)
     (behavior-ref B2 Test_B)
     (>= B2'start B1'stop))
```

If the corresponding test program is written in a sequential programming language such as C, and if `B1` and `B2` are implemented as subroutines, the code for `some_test` might look like:

```
some_test ( void ) {
  test_A();     -- behavior B1 Test_A
  test_B();     -- behavior B2 Test_B
}
```

This behavior might be implemented by code that programs a power supply to 3.3 V. Analysis of the test program code would show that the power supply programming is consistent with the behavior; however, the execution trace would be consistent with the behavior only if the selected power supply had adequate accuracy and precision and if the line loss between the supply and the UUT was negligible. These are concerns that test engineers deal with on a daily basis and that must be considered during the analysis.

Sometimes analysis of the test program code does not yield enough information to conclude anything about the expected execution trace (e.g., when interactions between the selected test resource and the interface test adapter cannot be predicted easily). In these cases simulation of the test program, also called "virtual tests," can generate predictions of execution traces for the test program [19]. These predicted execution traces can be verified with respect to some behavior, and with enough such simulations, the test program could be declared acceptable.

4.4. Test Resource Allocation

Test resource allocation identifies candidate resources and then determines if the candidates are suitable for the required task [20]. Test resource allocation is an important function in test development, and automation of this function is important when automating the test program generation process. Automated allocation also enables dynamic allocation of test resources, which in turn leads to more portable test programs. A test program will have one or more requirements that

must be satisfied by a test resource, and a test resource will have a set of capabilities. If the resource capabilities satisfy the requirements then the resource is functionally suitable.

The behavior model supports the test for functional suitability by acting as a specification for both the test requirements and the test resource capabilities. For example, supposed the following test requirement is applied to an amplifier with a gain of two.

```
(behavior-def some_test ()
     (signal V1 DC_SIGNAL)
     (signal Vin DC_SIGNAL (VoltageLocation In_Pin Gnd_Pin))
     (signal Vout DC_SIGNAL (VoltageLocation Out_Pin Gnd_Pin))
     (signal V2 DC_SIGNAL)
     (variable Amp_Ok boolean)
     (< (V 0.5) (abs (- (V 4) V1.voltage)))
     (< (mV 2)  (abs (- Vin.voltage V1.voltage)))
     (< (mV 2)  (abs (- V2.voltage Vout.voltage)))
     (== Amp_Ok
         (<= 0.001 (abs (- (/ V2.voltage V1.voltage) 2)))))
```

There are four `signals` in this behavior:

1. `V1` is the input voltage to the amplifier program, as programmed
2. `V2` is the output voltage, as reported by some instrument
3. `Vin` is the actual input voltage to the amplifier
4. `Vout` is the actual output voltage of the amplifier

There are also four `constraints` in this behavior.

1. The programmed input voltage can be any value in the range 3.5–4.5 V.
2. The actual input voltage must be within 2 mV of the programmed voltage.
3. The reported output voltage must be within 2 mV of the actual output voltage.
4. The Boolean `property Amp_Ok` is true if the amplifier gain, calculated with the programmed and reported voltages, is two, plus or minus one-tenth of 1%.

From this description, three things about `some_test` can be observed. First, the use of `signals` for the programmed and reported voltages allows the behavior model to capture accuracy requirements. The relationship between `Vin` and `V1` is one accuracy constraint, and the relationship between `V2` and `Vout` is the other. Second, while the behavior model allows the test program to test the amplifier at any voltage in a one-volt range (e.g., 3.5–4.5 V), the accuracy of the overall test is required to be relatively high. This requirement cannot be specified with a simple range on `Vin`. Third, there is an implicit constraint between `Vin` and `Vout` since `Vin` is constrained relative to `V1`, `Vout` is constrained relative to `V2`, and an explicit constraint exists between `V1` and `V2`. This constraint is similar in form to the constraint between `V1` and `V2` and might be classified as a product requirement.

A test resource is required to implement the behavior specified by this model. To obtain the required behavior, two resources will probably be required—one to apply the input voltage and one to measure the output voltage. For the sake of discussion, consider only the measurement resource. The input resource would be treated similarly. A possible behavior model describing the measurement resource is:

```
(behavior-def measure_spec ()
     (variable Offset Voltage)
     (variable Tolerance Real)

     (signal Vmeas DC_SIGNAL (VoltageLocation Input_Pin Gnd_Pin))
     (signal Vread DC_SIGNAL)
```

```
(and (>= Vmeas.voltage (V -10)) (<= Vmeas.voltage (V 10)))
(< (abs (- Vmeas.voltage Vread.voltage))
     (+ Offset (*  Vmeas.voltage Tolerance)))
(and (>= Tolerance 0) (<= Tolerance 2e-4)))
```

This resource has two signals—the voltage that appears at the input and the voltage as reported by the resource. The behavior has four constraints. The first limits the range of the input voltage to ± 10 V. The last three constraints limit the measurement error. The first expresses the resource accuracy as a linear function of `Offset` and `Tolerance`, the second limits the legal range of `Offset`, and the third limits the legal range of `Tolerance`. Notice that the behavior does not state that the `Offset`, for example, is 100 μV. Instead, the behavior states that the `Offset` is no more than 100 μV.

When interpreting this model, the `measure_spec` behavior states the behavior of the resource. It gives no information about how to control the resource. One approach for controlling the resource is for the resource to have a separate set of methods that provide program control. If this was the approach taken, one would expect to find a method that returns `Vread.voltage`, but no methods for returning `Vmeas.voltage`, `Offset`, or `Tolerance`. Further, there would be no methods for setting any of the `properties` or `signals`. This set of methods is entirely consistent with the control of real resources.

Assume that some process has chosen the resource associated with `measure_spec` to implement some of the test requirements in `some_test`. If the resource is connected to the UUT, a new behavior will be specified that is the union of the original behaviors:

```
(behavior-def some_test ((vout V))
     ...
     (== vout Vout.voltage)
     (signal Vout DC_SIGNAL (VoltageLocation Out_Pin Gnd_Pin))
     ...)

(behavior-def measure_spec ((vmeas V))
     ...
     (== vmeas Vmeas.voltage)
     (signal Vmeas DC_SIGNAL (VoltageLocation Input_Pin Gnd_Pin))
     ...)

(behavior-def some_test_using_resource ()
     (behavior-ref X some_test)
     (behavior-ref Y measure_spec)
     (< (abs (- X.Vout.voltage Y.Vmeas.voltage) ITA_Loss)))
```

There is an instance of `some_test` and an instance of `measure_spec`. The sub-behaviors `X` and `Y` should be constrained to be simultaneous, but those constraints have been omitted for simplicity. There is an additional constraint that couples the UUT output voltage with the test resource input voltage, corresponding to wiring in the interface test adapter. The constraint does not say that the two voltages are equal. Rather, it states that the two voltages are closer than some factor called `ITA_Loss`.

If the test resource is compatible with the test requirements, there will be no conflicts in the constraints. The constraints (numbered for convenience) that are coupled by connecting the resource with the UUT are:

```
(signal V1 DC_SIGNAL)
     (signal Vin DC_SIGNAL (VoltageLocation In_Pin Gnd_Pin))
     (signal Vout_DC_SIGNAL (VoltageLocation Out_Pin Gnd_Pin))
     (signal V2 DC_SIGNAL)

     (variable Amp_Ok boolean)

1.   (< (V 0.5) (abs (- V1.voltage (V 4))))
2.   (< (mV 2) (abs (- Vin.voltage V1.voltage)))
3.   (< (mV 2) (abs (- V2.voltage Vout.voltage)))
4.   (== Amp_ok (<= 0.001 (abs (- (/ V2.voltage V1.voltage) 2))))

     (variable Offset V)
     (variable Tolerance Real)

     (signal Vmeas DC_SIGNAL (VoltageLocation Input_Pin Gnd_Pin))
     (signal Vread DC_SIGNAL)

5.   (and (>= Vmeas.voltage (V -10)) (<= Vmeas.voltage (V 10)))
6.   (< (abs (- Vmeas.voltage Vread.voltage))
          (+ Offset (* Vmeas.voltage Tolerance)))
7.   (and (>= Offset (V 0)) (<= Offset (uV 100)))
8.   ((and (>= Tolerance 0) (<= Tolerance 2e-4)))
9.   (< (abs (- Vout.voltage Vmeas.voltage)) ITA_Loss)
```

If the UUT is good, one can deduce from the model that the output voltage may range from approximately 7 to 8 V (the output may actually range from 6.992502 to 9.008502 V as derived from constraints 1, 2, and 4). This lies comfortably within the resource input range given by constraint 5. Note that the range of `Vout` was not constrained; therefore, one cannot be certain that `Vmeas` will lie within the 10 V range. In the present example, one could probably clamp `Vout` to protect the resource input without affecting the validity of the test program.

If the UUT is good, one can also deduce from the model that `Vread` is an adequate substitute for `V2`, provided the `ITA_Loss` is less than 100 μV. The maximum error of the resource (given by `Vread` with respect to `Vmeas`) occurs at 9 V and is 1.9 μV, as specified by constraints 6, 7, and 8. The maximum error permitted on `V2` (given by `V2` with respect to `Vout`) is 2 μV, as specified by constraint 3. The maximum error between `Vout` and `Vmeas` is `ITA_Loss`, as specified by constraint 9. Therefore, if `ITA_Loss` is no more than 100 μV, `Vread` from the resource can be substituted for `V2` in constraint 4.

This example illustrates that analyzing the behavior specifications of the test requirements and the resource capabilities "verifies" that the resource satisfies the requirements and can be used in the test program (provided the ITA loss is kept low enough). Different values in any of the constraints could lead to a different conclusion.

4.5. Triggers and Events

One reasonable issue for the behavior model to address is the representation of events, including various triggers and timers [8–10]. The behavior model quite readily represents these concepts but does not recognize them in any special way. A trigger defines a precise timing relationship between two actions. For example, a trigger can establish a relationship between the rising edge on a voltage signal and the start of an oscillator. This is shown in Fig. 4.

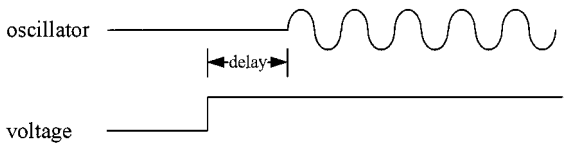

Fig. 4. Trigger initiating oscillator.

The behavior model captures the trigger as follows. Assume several sub-behaviors for oscillator_on, oscillator_off, DC_low, and DC_high have been defined.

```
(behavior-def oscillator_trigger ()
     (behavior-ref Osc_Off oscillator_off)
     (behavior-ref Osc_On oscillator_on)

     (== Osc_Off'start self'start)
     (== Osc_On'start Osc_Off'stop)
     (== Osc_On'stop self'stop)

     (behavior-ref Trig_Down DC_low)
     (behavior-ref Trig_Up DC_high)

     (== Trig_Down'start self'start)
     (== Trig_Up'start Trig_Down'stop)
     (> Trig_Up'stop  (+ Trig_Up'start Trig_Delay))
     (== Osc_On'start (+ Trig_Up'start Trig_Delay)))
```

There are four sub-behaviors of oscillator_trigger. Two govern the behavior of an oscillator and must occur sequentially (i.e., Osc_On starts as soon as Osc_Off stops). Together, these two sub-behaviors span the entire containing behavior. Two sub-behaviors govern the trigger signal itself. These two sub-behaviors are also sequential, but they need not run to the end of the containing behavior. In this example, once the oscillator is running, the trigger signal can change without affecting the oscillator. Finally, a constraint is imposed on the oscillator to turn on after some delay following the rising edge of the trigger signal. This delay is named Trig_Delay in the example and would normally be passed as a parameter to the behavior.

Notice that oscillator_trigger describes a simple timing relationship, but there is no indication of how the relationship is to be implemented. It could be implemented through a trigger, through software, or as a side effect of something else in the test program. The behavior model only describes the constraints—not how to satisfy the constraints.

In the real world, equality constraints are difficult to realize. For example, oscillator_trigger requires that the time difference between Trig_Up'start and Osc_On'start be *exactly* Trig_Delay. The time difference would never be exactly Trig_Delay in a real test program, nor does it matter. Instead, the actual time difference (as measured in the execution trace) should be Trig_Delay, within some error. This is captured in the following constraint:

```
(<(-(abs Osc_On'start Trig_Up'start)
         Trig_Delay) Trig_Error)
```

The new constraint says that the difference between the actual delay (Osc_On'start - Trig_Up'start) and the desired delay (Trig_Delay) must be less than some allowable error (Trig_Error). More complicated constraints can be used to capture asymmetric tolerances (i.e., tolerances not symmetrically distributed about the expected value).

Returning to the example, it is evident the behavior model adequately describes the relationships implemented by events and triggers. It is also evident that the behavior model does not distinguish between timing relationships that are implemented in hardware (e.g., triggers) or software (e.g., software delay loops). The primary difference between hardware triggers and software timing is in the potential error in the triggering delay. A test programmer selects a hardware implementation to get a precise delay but is then concerned with issues of accessing, allocating, and controlling the hardware to obtain the desired delay. On the other hand, the test programmer selects a software implementation when precision is less important than ease of programming.

A test program would not normally use a behavior such as oscillator_trigger. The trigger signal in the example is probably an implementation detail. A

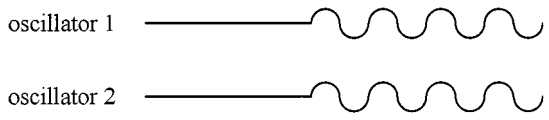

Fig. 5. Pairing two oscillators.

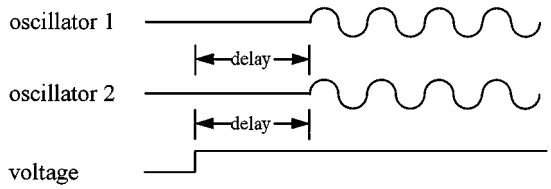

Fig. 6. Implementing two paired oscillators.

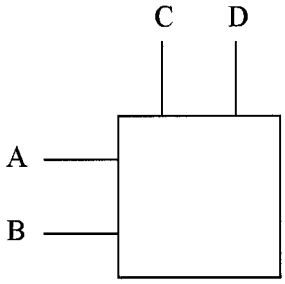

Fig. 7. Simple 2 × 2 matrix switch.

more likely scenario is a behavior in which two oscillators start within a fixed time of each other. This can be illustrated as in Fig. 5 and might be implemented as in Fig. 6. The behavior should properly describe the requirement that the oscillators begin at the same time and not the implementation detail of referencing both to a common rising edge.

4.6. Switches

Switches constitute an important class of test resource. Switches in automatic test equipment can be very complex, with many possible connections. Given two ports in a switch, they are either connected or unconnected. If they are connected, then the voltage difference between them is nominally zero, and the current flowing into one port is equal to the current flowing out of the other. This is expressed in the following behavior specification for `Connected_Behavior`.

```
(behavior-def Connected_Behavior ()
     (signal V1 Voltage (VoltageLocation ConnA ConnB))
     (signal I1 Current (VoltageLocation ConnA ConnB))
     (< (abs V1.voltage) (uV 100)))
```

If a port is unconnected, then we cannot make any statements about the voltage on the port, but we can assert that the current flowing through the port is zero. This property is expressed in the behavior specification for `Unconnected_Behavior`.

```
(behavior-def Unconnected_Behavior ()
     (signal I1 Current (VoltageLocation ConnA ConnB))
     (< (abs I1.current) (uA 1)))
```

`Connected_Behavior` is a behavior on two ports, but `Unconnected_Behavior` is a behavior on a single port.

Most of the complexity of ATE switches is a function of the possible connections that the switch can support. For example, assume that we have a 2 × 2 matrix switch as shown in Fig. 7. Ports A and B can be connected to ports C or D, and the switch can be in one of seven states: no connections, one of four possible single connections, and one of two possible pairs of simultaneous connections. We assume the states cannot occur simultaneously. A very simple description of the switch simply lists the possible states:

```
(behavior-def Switch_Behavior ()
     (behavior-ref S0 Connections_0)
     (behavior-ref S10 Connections_1 ConnA ConnC ConnB ConnD)
     (behavior-ref S11 Connections_1 ConnA ConnD ConnB ConnC)
     (behavior-ref S12 Connections_1 ConnB ConnC ConnA ConnD)
     (behavior-ref S13 Connections_1 ConnB ConnD ConnA ConnC)
     (behavior-ref S20 Connections_2 ConnA ConnC ConnB ConnD)
     (behavior-ref S21 Connections_2 ConnB ConnC ConnA ConnD)
     (NonOverlapping S0 S10 S11 S12 S13 S20 S21))

(behavior-def Connections_0 ()
     (behavior-ref UA Unconnected_Behavior ConnA)
     (behavior-ref UB Unconnected_Behavior ConnB)
     (behavior-ref UC Unconnected_Behavior ConnC)
     (behavior-ref UD Unconnected_Behavior ConnD))

(behavior-def Connections_1 ()
     (behavior-ref X Connected_Behavior ParamA ParamB)
     (behavior-ref UC Unconnected_Behavior ParamC)
     (behavior-ref UD Unconnected_Behavior ParamD))

(behavior-def Connections_2 ()
     (behavior-ref X Connected_Behavior ParamA ParamB)
     (behavior-ref Y Connected_Behavior ParamC ParamD))
```

The `NonOverlapping` constraint is not elaborated here, but it is implemented by relational constraints on the behavior start and stop times. For instance, if `S0` and `S10` are non-overlapping behaviors, then `S0` must end before `S10` starts, or `S10` must end before `S0` starts. Captured using a reusable function on two behaviors, we might have something like:

```
(function-def NonOverlapping_Pair
     ((B1 behavior) (B2 behavior)) boolean
          (or (<= B1'stop B2'start) (<= B2'stop B1'start)))
```

The `NonOverlapping` constraint, as it is used above, must deal with an arbitrarily long list of `behaviors`, but this is a language issue and not a model issue.

Any switch can be described by enumerating its possible configurations, but this becomes tedious. The model supports a variety of techniques in compressing the model by eliminating "redundant" configurations through a hierarchical arrangement or with reusable functions. As a language issue, it may also be possible to incorporate a construct similar to the `generate` statement in VHDL. Such "constructors" enable compact descriptions of models that can be elaborated to yield a full model at run time.

5. Conclusion

In this paper, we provide a description of the behavior information model proposed for next generation test system architectures. The focus of this work is on the development of a formal method for specifying UUT and resource behavior in electronic system test and is applicable to analog, digital, or mixed-signal testing. The information model facilitates the specification of behaviors related to test subjects, test requirements, test strategies, test resources, and product diagnostics using a declarative approach. As such, the behavior model provides a formal approach to specifying behaviors, thereby facilitating the specification and development of reusable and transportable test programs.

Acknowledgments

Support for this work was provided by the U.S. Navy through its Automatic Test System Research and Development Integrated Product Team (ARI). In particular, we would like to thank Harry McGuckin, Randy Simpson, Kirk Thompson, Steve Fortier, Walt Bailey, and all of the members of the ARI for their comments and suggestions. We also thank the anonymous reviewers whose comments led to a stronger presentation.

Notes

1. However, there may be compelling administrative reasons to insist upon a particular order, e.g., procedural reusability or test consistency.
2. In this example, no specific duration is specified.
3. In this example, we introduce the "`RT'execute`" notation. `RT'execute` is TRUE if the sub-behavior, `RT`, actually executes and is FALSE otherwise. However, just because a sub-behavior is declared inside a behavior does not mean that the sub-behavior actually executes. Additional constraints are required to ensure that the sub-behavior executes as many or as few times as necessary.

References

1. J. Cooke, "Formal Methods—Mathematics, Theory, Recipes or What?" *The Computer Journal*, Vol. 35, No. 5, pp. 419–423, 1992.
2. A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, pp. 11–19, Sept. 1990.
3. M. Thomas, "The Industrial Use of Formal Methods," *Microprocessors and Microsystems*, Vol. 17, No. 1, pp. 31–36, 1993.
4. S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, Chichester, UK, 1987.
5. C.J. Hogger, *Introduction to Logic Programming*, Academic Press, London, UK, 1984.
6. *ISO 10103-11:1994: Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 11: EXPRESS Language Reference Manual*, International Organization on Standardization, Geneva, Switzerland, 1994.
7. D. Schenk and P. Wilson, *Information Modeling: The EXPRESS Way*, Oxford University Press, New York, 1994.
8. *ARINC 626-3: Standard ATLAS for Modular Test*, Aeronautical Radio, Inc., Annapolis, Maryland, 1995.
9. *IEEE Std 716-1995: IEEE Standard Test Language for All Systems—Common/Abbreviated Test Language for All Systems (C/ATLAS)*, IEEE Standards Press, Piscataway, NJ, 1995.
10. *IEEE Std 771-1989: IEEE Guide to the Use of the ATLAS Specification*, IEEE Standards Press, Piscataway, NJ, 1989.
11. L. Shombert, *Test Requirements Model Language Reference Manual*, Draft 0.1, Technical Report CAE-1998-07-01, Intermetrics, Vienna, VA, 1998.
12. R. Atkins and D. Rolince, "TRSL Standard Supports Current and Future Test Processes," *Proceedings AUTOTESTCON '94*, 1994, pp. 271–279.
13. J. Nagy and J. Newberg, "Capturing Board-Level Test Requirements in Generic Formats," *Proceedings AUTOTESTCON '94*, IEEE Press, New York, 1994, pp. 61–69.
14. W.R. Simpson and J.W. Sheppard, *System Test and Diagnosis*, Kluwer Academic Publishers, Boston, 1994.
15. C. Papachristou and J. Carletta, "Test Synthesis in the Behavioral Domain," *Proceedings of the International Test Conference*, 1995, pp. 693–702.
16. R. Dechter, "Constraint Networks: A Survey," *The Encyclopedia of Artificial Intelligence*, S.C. Shapiro (Ed.), Wiley, New York, 1992.
17. P. Pearl, *Probabilistic Reasoning in Intelligent Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.
18. P. Caunegre and C. Abraham, "Achieving Simulation-Based Test Program Verification and Fault Simulation Capabilities for Mixed-Signal Systems," *Proceedings of the European Design & Test Conference*, 1995, pp. 469–477.
19. M. Miegler and W. Wolz, "Development of Test Programs in a Virtual Test Environment," *Proceedings of the 14th IEEE VLSI Test Symposium*, 1996, pp. 99–103.
20. G. Hardenburg and D. Nichols, "The IEEE ABBET Lower Layers Definition and Status," *Proceedings of AUTOTESTCON '95*, 1995, pp. 57–65.

Lee Shombert is a Principal Engineer at Intermetrics, Inc. He has a B.S. in Engineering and Applied Science from the California Institute of Technology and M.S. and Ph.D. in Electrical Engineering from Carnegie Mellon University. Dr. Shombert has worked on test information standards for the last ten years, primarily under the auspices of the IEEE SCC20. His interests include applying formal methods to the design process, to improve the flow of information from the designer to "downstream" activities.

John Sheppard is a Staff Principal Analyst for ARINC Incorporated. He holds a B.S. in Computer Science from Southern Methodist University and both an M.S. and Ph.D. in Computer Science from Johns Hopkins University. Dr. Sheppard is an internationally recognized expert in the area of system diagnostics and has developed model-based systems for diagnostics and diagnosability assessment. He has over 80 publications (including two books) in the area and holds US and European patents for an approach to diagnosis under uncertainty. Dr. Sheppard is actively involved in IEEE, IEC, and ARINC standards activities including the IEEE SCC20 on Test and Diagnosis for Electronic Systems and the AEEC/AMC Test Equipment Guidance Subcommittee. Currently, Dr. Sheppard chairs the AI-ESTATE subcommittee of SCC20 that has produced three standards on system diagnostics and is Secretary of IEC/TC93/WG7 on system test.