A Modeling Language Based on UML for Modeling Simulation Testing System of Avionic Software

WANG Lize, LIU Bin*, LU Minyan

Department of Project Systems Engineering, Beihang University, Beijing 100191, China

Received 24 April 2010; revised 27 July 2010; accepted 13 January 2011

Abstract

With direct expression of individual application domain patterns and ideas, domain-specific modeling language (DSML) is more and more frequently used to build models instead of using a combination of one or more general constructs. Based on the profile mechanism of unified modeling language (UML) 2.2, a kind of DSML is presented to model simulation testing systems of avionic software (STSAS). To define the syntax, semantics and notions of the DSML, the domain model of the STSAS from which we generalize the domain concepts and relationships among these concepts is given, and then, the domain model is mapped into a UML meta-model, named UML-STSAS profile. Assuming a flight control system (FCS) as system under test (SUT), we design the relevant STSAS. The results indicate that extending UML to the simulation testing domain can effectively and precisely model STSAS.

Keywords: avionics; hardware-in-the-loop; test facilities; meta-model; UML profile; domain-specific modeling language; abstract state machine

1. Introduction

Avionic systems are usually composed of software and embedded avionic devices. Embedded avionic devices are known as electronic control units (ECUs), which are complex proprietary hardware made up of a group of controllers, actuators and sensors. Building an efficient simulation testing system for hardware-in-the-loop (HIL) tests is important to each avionic software system.

In the area of avionics, several industrial applications of simulation testing systems are widely used[1-4]. They are proprietary which are designed based on platform-dependent technologies. They have concrete realizations and cannot reflect their essential characteristics. This means that software developers cannot develop models at a high level of abstraction in the design space. It is disadvantageous to improve productivity. So a kind of high-level modeling language is needed for HIL tests.

There are some kinds of high-level modeling languages, such as TestML[5], U2TP[6], SADL[7], etc. SADL is an architectural description language used in the area of simulation, but it aims to design simulation software. To build a simulation testing system, there should be more consideration for elements and relationships between simulation systems and testing systems. TestML is mainly used in automotive area as a test exchange language. It supports almost all kinds of test stages and test levels, and the abstract level is so high that some special execution characteristics, for example, the modeling abilities of simulation systems, are inevitably omitted. Although U2TP defines the common testing notions and profile elements of testing systems based on unified modeling language (UML), it cannot reflect the real-time, reactive, and continuous behavior characteristics of test modeling precisely and
the STSAS is divided into the following subsystems: According to its functional composition, it acts as test infrastructure to support model-based testing in avionics domain.

This paper presents a platform-independent modeling language UML-simulation testing systems of avionic software (STSAS) and gives an implementation scheme of the language, i.e., it gives a domain model and then defines the language. Compared with other testing languages, execution characteristics and patterns are well considered for the HIL tests in avionic area. The language also enables the designer to use domain elements to build test models and to improve efficiency and reusability by reducing the extra work caused by using general-purpose modeling languages.

2. Domain Analysis of STSAS

STSAS is used for prototyping, validating ECU, or acts as test infrastructure to support model-based testing of ECU. According to its functional composition, the STSAS is divided into the following subsystems:

— Test control and observation system controls or observes the running status of operating environment and system under test (SUT).

— Operating environment stimulates SUT, receives its reaction, and simulates the depending external environments of SUT. To improve the reusability of simulation, we differentiate it with test control and observation system. The former conforms to the logic of test, while the later, the logic of simulation.

— Test auxiliary system assists test control and observation system to graphically monitor or control test environment for man-machine interaction. This subsystem is necessary because it is not competent for the test component to fully complete a test purpose in some cases.

In order to achieve a test purpose, one or more test components which run simultaneously are necessary for controlling and observing the operating environment and SUT. For this purpose, test behavior is defined in the test component. Test behavior refers to a series of operation instructions, such as command, signal, formula and message. There exist five basic kinds of test behavior which cover the most possible spectrum of different behavioral descriptions in the field of avionics software testing: manipulating values, as well as text records.

SUT represents the system to be tested. From the viewpoint of testing, SUT is hidden behind its test interface, and is the completed product which does not need to be designed anymore. So we will mainly discuss the three systems mentioned above.

In order to achieve a test purpose, one or more test components which run simultaneously are necessary for controlling and observing the operating environment and SUT. For this purpose, test behavior is defined in the test component. Test behavior refers to a series of operation instructions, such as command, signal, formula and message. There exist five basic kinds of test behavior which cover the most possible spectrum of different behavioral descriptions in the field of avionics software testing: manipulating values of data, deciding what to do depending on the value, doing things repeatedly, doing several things simultaneously and doing with the time-related operations. According to its temporal phase, a set of test behavior is connected to the so-called STSAS automata, in which TestStates represent temporal phases of the set of test behavior and TestTransitions describe passages between the two TestStates (see Sections 3.1-3.5 for details).

The operating environment uses a set of software and hardware components to form the HIL circumstance on which SUT depends inevitably, stimulating and receiving the response of SUT. The simulation components exchange information with each other and communicate with SUT via standard or nonstandard connections, such as shared memory, Ethernet, ARINC 429, MIL1553 buses, etc. The operating environment is connected with the interface of SUT in order to exercise the SUT as if it was in a real flight environment. As the avionic system consists of a small amount of asynchronous tasks and a large number of periodic tasks, which have various execution rates, such as 40, 20, 10, 5, 1 Hz, etc., there must be a large number of periodic components and a small number of aperiodic ones. The periodic components are responsible for the mathematical calculations which represent the flight safety properties of the vehicle and must be executed in tight iteration loops in accordance with the implementation cycle to maintain a stable output. The aperiodic components correspond to aperiodic events, which can be either real-world events or flight operations during the simulation. The aperiodic components usually do not follow the regular time intervals and have soft real-time features.

Test auxiliary system uses graphical monitoring, data storage, fault injection and other technologies to monitor the status of the operating environment on-line and record test data for an off-line analysis. These technologies are encapsulated into test auxiliary components which can change status of operating environment for man-machine interaction. This subsystem is necessary because it is not competent for the test component to fully complete a test purpose in some cases.

The domain model of STSAS is shown in Fig.1. The relationship between any two components of the domain model is modeled as communication, which can be generalized to synchronous, asynchronous, and pure data transfer relationships. They draw and represent all possible execution characteristics of external behavior of components.

3. Definition of UML Profile: UML-STSAS

The domain model in Section 2 reflects the essential characteristics of solving the problem of modeling STSAS: how to build the complicated environment of SUT and control the execution of the environment according to the test purpose, making the SUT receive the stimulus and give response. Thus, an infrastructure of modeling language is needed, which inspires us to design the domain-specific modeling language (DSML).

Based on the lightweight profile mechanism of UML 2.2, the concepts and relationships in the domain model are mapped into a UML meta-model, which is defined using meta object facility (MOF).
under model driven architecture (MDA)\(^{[13]}\), and then a UML profile is defined. The profile takes a subset of UML, modifies UML and introduces new terminologies dependent on the domain model and gives the subset of UML new semantics and constraints. In this paper, we name the profile UML-STSAS, a kind of DSML.

We describe this language from various aspects, namely usage, syntax, semantics and constraint. Firstly, for each essential domain concept, a base UML metaclass whose semantics is the closest to it is selected, and the semantics and rationale of the metaclass as well as properties are specified. Then, the object constraint language (OCL) constraints define how to produce valid models with language constructs. The syntax of the language is represented by the language constructs together with a set of constraints. The semantics or meaning of the language is a specification of the relation between syntax elements and the corresponding domain concepts that the syntactical elements represent. The notations of the language constructs are expressed using standard notions which are symbols of UML profile’s stereotypes.

Fig.2 shows the stereotypes for the language constructs with a UML 2.2 package diagram. In our work,
Class is chosen as the stereotype of test component, test auxiliary component, aperiodic and periodic components. The standard class Device is extended to the field of avionic hardware. Since SUT is specified by its behavioral or structural characteristics, metaclass Property is selected as its stereotype. Association demonstrates the relation among component instances, namely synchronous, asynchronous and data transfer. Synchronous and data transfer stereotypes can be respectively extended to two sub-stereotypes. The elements of state and transition in UML 2.2 behavioral state machines are chosen as stereotypes of TestState and TestTransition for STSAS automata, respectively.

The execution model of the language is given before defining language constructs. It represents the execution control of components and the data processing of ports. The execution model has been deliberately kept simple to guarantee the real-time performance, using a fixed time increment \( G \) for calculation. The highly simplified execution model can be summarized as follows (see Fig.3).

**Step 1:** The component enters the initial phase and begins to initialize its data and time. The assignments of all ports for all \( t<0 \) are presumed to be undefined. The incremental count for sampling is set to zero. Proceed with Step 2.

**Step 2:** All components enter the running phase simultaneously at the point in time \( t=0 \). Afterwards, the calculation starts at the point in time \( t=0 \) and repeats for each increment \( G \). Although different types of components have their own characteristics and behavior, they are common in dealing with the discrete and continuous data value at each port of the component from the black-box perspective. The rule of processing port data obeys Step 3.

**Step 3:** The incremental count is increased by one \( \text{count} = \text{count} + 1 \) every time at sampling point, then the algorithm will perform Step 4 and calculate the input data. If the component needs to be informed by the system or other components, it blocks and waits for signal. Proceed with Step 6.

**Step 4:** The calculation time is \( t = \text{count} \times \delta \). According to the test scenario, the input port of the components is assigned for observation or capturing. For example, one output port of SUT is exactly the input port of a certain simulation or test component at the same time.

**Step 5:** The value of output port at time \( t \) is calculated based on the value of input port during the period \( t' \leq t \). As a result, each port is assigned determinated value till the end of test. Proceed with Step 7 after the calculation.

**Step 6:** In the blocking phase, components wait for activation signal. If the signal is received, then perform Step 2.

**Step 7:** If a component needs to control the other components, it will send a signal to the destination component. The calculation of Step 3 will be repeated if test scenario is not finished.

Abstract state machine (ASM)\(^{[14]} \) agents are used to model the semantics of control flow, thus the execution of the language system can be regarded as reading/writing actions on the system status by distributed agents. Multiple distributed agents form a group which can be regarded as a set of domain automata AuTomata Set (ATS). Each agent is controlled by an agent controller to implement a quasi-continuous execution style of sampling using a fixed time increment sampleTime. The controller determines the agent execution time \( t_{cur} \) which must not go beyond the setting duration.

\[
\begin{align*}
\text{if} \ t_{cur}(\text{Self}) \leq \text{duration}(\text{Self}) \text{ then} & \quad \text{count} = \text{count} + 1; \\
& \quad t_{cur}(\text{Self}) = \text{count} \times \text{sampleTime}(\text{Self}) \\
\text{else} & \quad \text{Mod}(\text{Self}) = \text{undef} \\
\end{align*}
\]

The types of entities in \( a \in \text{ATS} \) are modeled by type and the running phases by phase. For language elements, agents can be classified into \( \text{type}(a) \in \{\text{RealTime}, \text{softRealTime}\} \). When performing execution, each \( a \) can be in a phase(\( a \)) \( \in \{\text{init}, \text{running}, \text{switching}, \text{blocking}, \text{terminated}\} \). Each \( a \) begins to execute with the init as an initial phase, in which time and data are initialized. As an example, in its init phase, a test component initializes its local relative time \( t_{rel}(a) \), assigns the current state currentStep(\( a \)),
and then proceeds with running.

```plaintext
var a ranges over ATS
if phase(a) = init then
t(a) := 0.0;
currentStep(a) := initialStep(a);
phase(a) := running
endif
```

Each language construct will be individually defined from Section 3.1 to Section 3.12, and ASM is used to specify language semantics. Among the definitions, test relevant concepts are defined from Section 3.1 to Section 3.5, concepts on operating environment are introduced from Section 3.6 to Section 3.9, and relationships between components are given from Section 3.10 to Section 3.12.

3.1. Test component

Usage

A test component is used to describe the test control flow and model its communication relationships with the operating system or SUT.

Semantics and rationale

It is an instance of Class achieving to execute test behavior in the test sequence and control and observe the running process of simulation system or SUT. Sometimes, test sequence needs to be implemented by concurrent control flows, among which there are seldom reciprocal messages. Each control flow corresponds to one test component.

The test component is an active class with properties of CpuNum, priority, a set of ports, and ismain. The CpuNum property represents the actual processor number on the computer to which the component will be allocated for execution. Test components communicate with other components through ports. The isMain property means whether it is the duty of this test component to give the judgement of test verdict. In this paper, other components also contain the above properties except ismain, and these properties will not be repeated.

In order to express executable test sequence strictly and visually, STSAS automaton is introduced. An STSAS automaton consists of TestStates and TestTransitions.

Constraint

1) Only one ismain property can be true if several test components fulfill one test sequence, i.e.

```plaintext
self.isMain ->> forall (p,q:Test Component | p.isMain. value=true and q.isMain.value = true implies p=q) and self.isMain->>exists(i:Bool | i = true).
```

3.2. TestState

Usage

A TestState is the division of the execution of test sequence according to the temporal domain, and it represents exactly the state element of the STSAS automaton.

Semantics and rationale

TestStates are State instances of individual and temporal phases of the test sequence, subsuming test behavior into groups. Each STSAS automaton consists of TestStates and TestTransitions. Each TestState consists of either entities like one or more test behavior or an embedded STSAS automaton. Embedded STSAS automaton helps to establish hierarchal and complex scenarios, the execution rules on which are the same as STSAS automaton.

To equip an agent with the ability of modeling STSAS automata, a switching phase is added to phase(a). After agent a has reached the running phase, it begins to execute. The macro executeStep(a) immediately executes all entities contained in the current TestState currentStep(a), and the TestState will keep running until all its internal entities finish. The macro checkConditions(a) determines whether the current TestState should be changed. If type(a)=RealTime, the TestState runs along with checkconditions, meanwhile a state change occurs whenever the condition is met. If type(a)=softRealTime, the TestState and checkconditions go orderly, the condition will be evaluated after the TestState has completed, and the duration is platform-dependent. A comma is used to separate function updates which executes simultaneously in the same rule block.

```plaintext
var a ranges over ATS
if phase(a) = running then
  if type(a) = RealTime then
    executeStep(currentStep(a)),
    checkConditions(currentStep(a))
  elseif type(a) = softRealTime then
    executeStep(currentStep(a)),
    checkConditions(currentStep(a))
endif[15]
```

If entities in current TestState are test behavior, the execution of the TestState is exactly the execution of test behavior. If embedded STSAS automaton, it will be regarded as a new STSAS automaton and will be started at once.

```plaintext
executeStep(currentStep(a)) :=
  if type(currentStep(a)) = testbehavior then
    executeTestbehavior (currentStep(a))
  elseif type(currentStep(a)) = RealTimeATS then
    extend ATS with a
    type(a) := RealTime, phase(a) := running
  endextend
  elseif type(currentStep(a)) = softRealtimeATS then
    extend ATS with a
    type(a) := softRealTime, phase(a) := running
  endextend
endif[15]
```

Besides normal TestStates, two special kinds of
pseudo TestStates are designed to represent the initial TestState and final TestState of a STSAS automaton. Neither of them has an entity.

**Constraint**

1) An initial TestState is never the source of a transition, i.e.
   if self=init then self.incoming->isEmpty();
2) A final TestState is never the target of a transition, i.e.
   if self=final then self.outgoing->isEmpty();
3) A TestState has at least one incoming TestTransition except initial TestState, i.e.
   if self<> init then self.incoming->size() \geq 1;
4) A TestState has at least one outgoing TestTransition except final TestState, i.e.
   if self<> final then self.outgoing->size() \geq 1;
5) The same TestState does not belong to two STSAS automations, i.e.
   self.allInstances()->forAll(p | p.exists->(stateMachineA) and stateMachineA<>stateMachineB implies not p.exists->(stateMachineB));
6) One TestState does not allow to contain test behavior and embedded ATS at the same time, i.e.
   self.isEmbeddedMachine implies self.testbehavior.size()==0;
7) There is no entry and exit activity, i.e.
   self.entry->size==0 and self.exit->size==0.

### 3.3. TestTransition

**Usage**

A TestTransition marks passage between two TestStates of an STSAS automaton.

**Semantics and rationale**

A TestTransition is an instance of Transition. One TestTransition connects a target and a source TestStates. The TestTransition happens at the exact time of the termination of the source TestState. TestTransitions have no duration and may contain guarded conditions. In the running phase of an STSAS automaton, macro checkConditions implements the process of TestTransitions. Whether the execution time exceeds a maximum time of duration(currentStep(a)) will be firstly checked, and then whether any TestTransition in the set switches(currentStep(a)) is met will be checked. It is permitted for one TestState to have multiple outgoing TestTransitions, but only one can be met to switch to the next TestState. Several guard conditions can be assigned to the same TestTransition. There should be a default TestTransition if there would not exist any guarded condition. The default TestTransition is stimulated only if all entities in source TestState of the TestTransition have terminated.

```plaintext
if test(a) \geq duration(currentStep(a)) then
  if defaultSwitch(currentStep(a)) then
    phase(a) := terminated
  else phase(a) := switching
endif
endif
if evalRecoverSig(a)=true \land evalSwitch(currentStep(a))=false then phase(a) := blocking
if evalSwitches(switches(currentStep(a)))=true then
  phase(a) := switching
endif
evalSwitches(switches)\exists s \in switches:
eval(condition(s)) = true
If phase(a)==switching, agent a will switch to the next TestState. Take the new state as its current one, and start running it.
if phase(a) = switching then currentStep(a) := nextStep(switches(currentStep(a))),
phase(a) := running
endif
```

**Constraint**

1) Do not allow direct changes from the initial TestState to the final TestState, i.e.,
   stateMachine. TestState ->select(r | r <> initial and r <> final).notEmpty();
2) On each iteration of transitions there has to be at least one TestState, i.e.,
   let T=stateMachine.TestState.iterate(x:state;t: transition:S=Bag{} | t.including(x.outgoing)) in if T.forAll(t: TestTransition | count(t)>1) then t.state.outgoing<>t;
3) It must be deterministic that only one TestTransition is fulfilled when more than one outgoing TestTransitions of the current TestState are evaluated, i.e.,
   context TestState:: TestTransition():
   prec if TestState.outgoing->size() \geq 1 and then TestState.outgoing.guard->select(guard=true)->size()==1;
4) A TestTransition cannot connect TestStates belonging to different STSAS automata, i.e.,
   stateMachine.TestTransition.allInstances()->forAll (p,q|p<>q and p.TestTransition->excludeAll(q.TestTransition)).

### 3.4. Test behavior

**Usage**

This term is proposed to describe behavioral and operational characteristics of the process of defining test data and how these data are applied to the operating environment or SUT for the purpose of control or observation.

**Semantics and rationale**

It is applied as a stereotype of extended Behavior or Operation to describe how a test is performed in the course of test execution. Each behavior corresponds to one of the five kinds of behavior mentioned in Section 2. Equipped with operation code and corresponding parameters, each test behavior becomes statement,
such as command, signal primitive, arithmetic equation and message. At the execution moment, it is through test behavior that test components communicate with operating environment or SUT through communication relationships.

The temporal characteristics of test behavior can be defined as two kinds, namely type(currentBehavior((currentStep(a)))/{immediate, multiframe}). If type=immediate, the test behavior is executed immediately and deterministically, and the control flow is passed to next test behavior within the same sampling frame. If type=multiframe, the test behavior starts to calculate and then returns to the control. In next frames, the test behavior will be executed continuously, and control flow will not be passed until the test behavior terminates by expiration of a time, or matching a condition. This behavior usually refers to a reactive behavior or a behavior associated with continuous test data or time.

**Constraint**

1) When several automatons run simultaneously or hierarchically, the read/write operation may conflict with each other. The execution order is that the lower level goes first, and writer goes later than reader at the same level, i.e.

```java
def: lv:Integer = stateMachine.level; def: mode: {read,write}= operateType

def: pri:Integer = executeOrder; def: chl=ObjectTo-Operate

if self.forAll(p,q) p.time=q.time and p.chl= q.chl and exists(p.mode, q.mode) = write)
then p.lv>q.lv implies p.pri<q.pri p.lv= q.lv and p.mode = write implies p.pri > q.pri.
```

**3.5. Test auxiliary component**

**Usage**

It is used to record or monitor the status of operating environment and SUT.

**Semantics and rationale**

It is a kind of Class which encapsulates techniques like graphical user interface (GUI), data storage, error injection to monitor, record or even change the execution of the operating environment or SUT. By this component, events like data changes are subscribed, or can receive and process asynchronous events from test events during the simulation. It typically represents the simulation software having hard real-time requirements. Its periodic behavior is achieved when the component is invoked at regular intervals by a timer, or is periodically synchronized, so a series of a associated with the component is invoked at regular temporal intervals. The component is blocked to wait for an event to save computational resource of processor. It can receive and process asynchronous events from test components to modify its internal data, even in blocking phase. It must at once notify the component that has subscribed value- or status-dependent events associated with it.

```java
if ∃ s: existSubscribe(a, predecessor(a)) = true then
signal := waitSignal(a)
if type(signal) = frequencySync then 
if signalSource(signal) = timer \ signalSource(signal) \ predecessor(a) then
calculate(a), signalSuccessor(a) endif
elseif type(signal) = asynchronous then
if code(signal) = exit then phase(a):= terminated
endif
endif

data(a, signalSource(signal))
if phase(a) = running then signalSuccessor(a)
endif

There are several properties, frequency, deadline, and duration.

**Constraint**

1) All connected periodic components must have frequencies that are harmonics of one another, i.e.

self.allInstances().>forAll(p,q | p.exists(q.outgoing)
implies p.perd % q.perd =0 and p.perd % syncperd =0
and q.perd % syncperd =0);

2) Deadline must be less than or equal to the period strictly, i.e.
   self.deadline ≤ self.perd;

3) The frequency can be larger than zero or infinite, i.e.
   self.perd.oclIsTypeof(Real) and (self.perd >0 or self.perd = infinite);

4) Each periodic component can receive only one synchronization arc, i.e.
   self.allInstances()->forAll(p,q | p.exists(q.outgoing)
   implies q.allInstances().size()=1)).

3.7. Aperiodic component

Usage

It can express the operations on aircraft or external events from outside SUT.

Semantics and rationale

It is a kind of Class and an active class. The component represents aperiodic behavior of avionic systems, or handles the aperiodic events from components of simulation software. Some aperiodic events correspond to real-world events from flight operations within the digital portion of the simulated system, and others are side effects of the operation of simulation software itself. a, which is regarded as the agent of an aperiodic component, is blocked to wait for an event. But aperiodic components will not be invoked at regular time intervals, and there is no specific minimum time interval between invocations. They typically have soft real-time requirements.

if signalSource(signal) ≠ timer \ signalSource(signal)∈ predecessor(a) then
calculate(a); signalSuccessor(a)
endif

The aperiodic components also have the same characteristics as the periodic components to support test behavior. Except for frequency, they have the same properties as the periodic components.

Constraint

1) The case of an aperiodic component sending a synchronization mechanism to a periodic component is considered illegal and is not allowed, i.e.,

3.8. Hardware device

Usage

It represents real avionic device when the operating environment needs not to be simulated or is too hard to simulate.

Semantics and rationale

It is a kind of extended device class to represent many active or passive avionic input/output (I/O) hardware devices such as actuators, sensors, ECUs, etc., which can be used in an HIL simulation. Hardware Device is not the target device on which the test and simulation components run. Active devices can generate periodic or aperiodic interrupts while passive ones only receive resource access requests from others. The joint of the hardware devices improves the simulation fidelity and makes the simulation system look as if it were in a real flight environment. The internal structure of Hardware Device is regarded as black box, and the external behavior of Hardware Device can be substituted by simulation software.

3.9. SUT

Usage

SUT is used to represent features and behavior of a classifier.

Semantics and rationale

The SUT stereotype extends the Property metaclass. Since SUT is a black box combination of software and hardware, SUT constitutes test interface specifications and function specifications, and both of them form the behavioral model of the system under test. SUT can be looked as values of properties of system under test.

3.10. Synchronous relationship

Usage

This relationship represents a designation of synchronous communication relationship among components of test control and observation system, operating environment and test auxiliary system. According to its usage, it can be divided into frequency synchronization FreqSync and conditional synchronization CondSync.

Semantics and rationale

This stereotype defines a point-to-point synchronous relationship between two connected components. Each component may have multiple subcomponents, and the synchronization event happens among these subcomponents. When the relationship happens, the execution order of the two associated agent is that the source agent a (predecessor) will execute first, and provides a synchronization mechanism to the destination one a' (successor), and then a' begins to execute. The synchronization mechanism acts as the trigger for a' which blocks or polls while waiting for the synchronization. There can be a concomitant data transfer relationship.

waitSignal(a)=
   choose a' in a'∈ predecessor(a)
   if ∃:evalSyncConditions(a',a)=true then
      let x:=a in signal(a', x), phase(x):=running
   endif
   if ∃:evalDataTrans(a,a')=true then data(a,a')
   endif
   endchoose

Each predecessor must traverse all its successors to determine whether to release or issue a synchronous signal.
signalSuccessor(a) =
  \textbf{var } a' \textbf{ ranges over } a' \in \text{successor}(a)
  \textbf{if } 3\exists:\text{evalSyncConditions}(a,a')=\text{true } \textbf{then }
  \text{let } x:=a \text{ in } \text{signal}(a', x), \text{phase}(x)=\text{running }
  \textbf{endif}
  \textbf{if } 3\exists:\text{evalDataTrans}(a,a')=\text{true } \textbf{then } \text{data}(a,a') = \textbf{endif}

According to the usage, synchronization mechanisms can be divided into two categories, FreqSync and CondSync.

The FreqSync relationship is generally used among components in operating environment. It has two properties, frequency and releaseTime. The first property defines how often the source component sends the synchronization mechanism to the destination one. The synchronization happens periodically and circularly according to the specified frequency if it is greater than zero, or asynchronously and unconditionally if zero. The second property defines how long the source releases the synchronization since it starts the execution cycle. Releasing at the beginning of the execution cycle means the destination begins to execute at the same time as the source, and releasing at the end means the destination has to wait for the complete execution of the source. As the execution duration in every cycle is not very deterministic for the source, the release time could be changeable rather than fixed.

\textbf{Constraint}

1) The frequency is equal or greater than zero, i.e. self.value \geq 0;
2) The releaseTime is equal or greater than zero, equal or less than frequency, i.e. self.releaseTime \geq 0 \textbf{and} self.releaseTime \leq self.perd.

The CondSync relationship is often employed between test components, test auxiliary components and simulation ones. The destination must block or poll to suspend its execution and wait until a condition associated with the source evaluates true. It also has two properties, condExpression and overtime. The first property must be a predicate which is associated with the source component. For example, the data of the source component is greater than a certain value, or its value has been changed or written newly. The second property tells the destination how long to wait before the condition is met, and it is optional. A zero value means that the destination will not wait and continue to execute if the condition evaluates false. An infinite zero means that the destination will wait for the condition for ever.

\textbf{Constraint}

1) There can be one or more overtimes, i.e. self.condsync.size()>0 \textbf{implies} self.timeout.size()>0;
2. Overtime is equal, greater than zero or infinite, i.e.
   self.value \geq 0 \textbf{or} self.value=\text{infinite}.

3.11. Asynchronous relationship

\textbf{Usage}
It is used to denote event relationships between components of testing system and simulation ones of operating environment.

\textbf{Semantics and rationale}

The stereotype is extended from Association. To achieve a test object, components of testing system must have a mechanism to control the execution of simulation components at any appointed time through asynchronous events. In most cases, the mechanism aims to change the data value that simulation component holds and the effect will be indirectly transmitted to its successor probably including SUT. The mechanism needs real-time implementation, i.e., the destination component needs to respond immediately even if it has a synchronous relationship with other components and stays blocked or busy executing. The asynchronous relationship and FreqSync one with zero frequency value seem alike: they both depend upon some external event or condition that occurs asynchronously; but the former does not need to wait by blocking or polling, while the latter does.

3.12. Data relationship

\textbf{Usage}
It is used to express the pure data transfer relationship between two components, without considering synchronous or asynchronous mechanisms.

\textbf{Semantics and rationale}

The stereotype is extended from Association metaclass to define one kind of relationship. The relationship means that the source component produces some data which the destination component consumes, and it happens in the case that there either exists a pure data transfer relationship, or a synchronous or asynchronous relationship that occurs simultaneously with a data transfer activity at the same time between two components. There are no requests for implied timing relationship, order of execution and invocation frequencies for either component involved.

There are two kinds of transfer modes, sampling transfer and queuing transfer. Sampling transfer means that writing a value to the communication area overwrites the previous value and reading from the communication area always returns the last value written to it. It is assumed that the source and the destination have access to the data region, and use semaphore or spinlock to protect the shared data. Queuing transfer means that the source component produces data that the destination consumes using first in first out (FIFO) queue mechanism. Writing to the queue appends the new value to outgoing queue. Reading from the queue returns the oldest value of the incoming queue. The producer continues execution immediately after the output operation and the consumer is blocked on the input operation until it processes the received data.

data(a,a') =
  \textbf{if } \text{type} (\text{data}(a,a'))=\text{ sampling } \textbf{then } // a:\text{ writer},
  a':\text{ reader}
∀ s ∈ commarea(a,a'): mutemechism(s) = true → write(a,s) = true ∨ read(a',s) = true

elseif type(data(a,a')) = queuing then
    s := LIST(a,a')
    if isempty(s) then
        phase(a') := blocking, writetail(a,s), readhead(a',s)
    else
        writetail(a,s), readhead(a',s)
    endif
endif

Both modes have a property of data region using predefined data types and the queuing transfer has another parameter of length of FIFO queue.

Constraint
1) Length of FIFO queue is an integer equal or greater than 1, i.e. self.length() ≥ 1.

4. Case Study

This section provides an exemplary use of the UML-STSAS, assuming the flight control system (FCS) of some types of unmanned aerial vehicle (UAV) as SUT. To show the advantage that the proposed language brings for HIL tests in avionic applications, a test environment for system testing is designed. We build the operating environment according to the functions and interfacial descriptions of the SUT, take the function of take-off as an example, and then set up the corresponding test models.

FCS is the managing center of UAV, the flight course and pose of which are controlled by FCS. The working of FCS depends on the following operating environment:

1) Through ARINC422 bus, FCS is connected to engine controller (EC), remote control device (RCD), inertial navigation system (INS) and wireless height meter (WHM) and there exist frequency synchronization (the frequency is 50 Hz) and data transfer relationships.

2) Through analog to digital/digital to analog (AD/DA) bus, FCS is linked to steering gear actuator (SGA) with pure data transfer relationship. Here, SGA is hardware device.

3) Through digital in/digital out (DI/DO) bus, FCS is connected to wheel controller (WC) with the relationships of frequency synchronization (the frequency equals zero).

The model of operating environment is established and shown in package operating environment in Fig.4. Each component in the package stimulates and receives the reaction from the SUT in accordance with the established relationships. Each relationship between any two connected components represents a full duplex connection. For example, the SUT sends engine speed control requests to EC and EC replies with actual engine speed through ARINC422 bus with a frequency synchronization and a sampling data transfer relationships. ARINC422 bus, as well as AD/DA and DI/DO buses, is considered as low-level implementation of the relationships.

Operating environment subsystem conforms to the logic of simulation and provides an essential framework that the SUT regards as its circumstance. Usually the computations of the ECUs in the operating envi-

Fig.4 Class diagram of simulation testing system model of UAV FCS.
ronment involve solutions of differential equations, matrix transformations, and look-up-table operations, so usually the computations are too complex to be modified easily when test purpose changes. As a result, components in the operating environment are designed as reusable facilities for differential kinds of test purpose: they take charge of the communication duties with the SUT, while controlled or observed by test facilities.

It is the operating environment that makes test designers deal with affairs of the logic of test, so the proposed language can greatly improve the test reusability and efficiency for HIL tests. For example, EC keeps a communication pattern with SUT, and the behavior of EC is defined by the differential equation

\[ \dddot{x} + 10 \dot{x} + 25x = 25f(t), \]

where \( x \) represents the second-order derivative of actual engine speed and \( f(t) \) the speed request function on time \( t \). EC runs with a frequency of 500 Hz to solve the equation iteratively, while test component yields speed requests at any time via asynchronous relationship. In our example, the requests are transmitted from RCS to SUT and finally to EC. Specially, they can be sent to EC directly if necessary, e.g., fault injection.

In the following part, the test description ability of the UML-STSAS is shown and illustrated with two short examples taken from practice. Both examples examine the take-off test of UAV. The first example gives the test sequence of the take-off function in automatic mode, while the second emphasizes the test in manual mode.

The following steps are taken for the first test scenario:

1. UAV must be on the ground before take-off, namely the wheel load signal is ON, auto mode; start the engine (the default value of Engine after starting EC is 30%), wait for 10 s, maximize engine speed Engine to 100% to make UAV taxiing, and then wait for the occurrence of the condition speed>70 m/s.
2. When UAV speeds up to 70 m/s, send take-off command to the SUT through RCD, and then set the value of actuator at 15°.
3. When UAV starts climbing and reaches the specified height 50 m, use signal ramp to linearly lower the engine speed from 100% to 60% (the cruising level) within 30 s. At the same time, use Ramp to lower Actuator value from 15° to 0°, to enable UAV to fly horizontally.
4. Take-off phase is completed, and the vehicle begins its flight course.

The scenario above requires two control flows to operate EC and SGA simultaneously, so two test components, namely Actuator Control and Engine Control, are designed to send commands (see package test control and observation subsystem in Fig.4). For each test component, one STSAS automaton is defined to depict test behavior in a compact visualization fashion.

The two automatons respectively have three and four TestStates, and each TestState contains a name and one or more statements. Every TestState is named according to the number of the test step mentioned above. For example, the first TestState of STSAS automaton in Fig.5 is named “Step 1” and it corresponds to test Step (1). The statements that belong to the same TestState execute in order and apply the result to the port of the destination component through the interface variables in the statements.

![Fig.5 Specification of take-off test in auto mode with STSAS automata.](image)

The second test scenario is achieved by imitating the manipulation of the engine speed and actuator value manually rather than automatically. The following steps are used.

1. UAV must be on the ground before take-off; the wheel load signal is ON, manual mode; start the engine.
2. Repeat improving the value of Engine from 30% to 100%.
3. Wait for 5 s, then judge whether the height of the vehicle is greater than 50 m. If true, then set Actuator from 15° to 0°, and the command Takeoff, otherwise return to Step (3) until the vehicle takes off.
4. Wait for 5 s, then judge whether the height of the vehicle is greater than 50 m. If true, then repeat setting Actuator from 15° to 0° and Engine from 100% to 60%, otherwise return to Step (4);
5. The status of FCS is set to automatic mode.

Fig.6 shows the STSAS automata description of the above steps. TestStates “Step 1”, “Step 2” and “Step 5”
contain statements, while “Step 3” and “Step 4” are embedded STSAS automata, and each represents one step of the local test object.

Assuming that the SGA data needs to be recorded during the test, and the EC data and WHM data need to be online monitored, three test auxiliary components in the test auxiliary subsystem are designed to achieve the purpose. They communicate with the operating environment with conditional synchronous relationships (see package Test Auxiliary System in Fig.4).

The exemplary usage of the proposed language shows that it can be used to construct and set up the simulation test environment for avionic software user friendly. Once the operating environment has been completed, it will greatly reduce the workload of the test designers for its reusability. The test designer can concentrate on the design of tests. With a basic set of test behavior and their suitable combinations, the use of STSAS automata supports the definition of complex test scenarios.

5. Related Work

Actually, there are several testing systems being applied for HIL simulation environments[1-4,16-18], such as ADS[1], RT-Lab[2], RT-tester[3] and dSpace[4]. Almost each testing system relies on proprietary test languages and techniques and requires its own method, language and concept. These solutions lack portability, extension and integration, and they reduce productivity and automation.

Studies most related to our work are high-level test description or design languages such as TestML[5], U2TP[6], SADL[7], TTCN-3[19] etc. However, all those languages concentrate on solving certain concrete problems, and their concepts do not focus on the special patterns which reflect the essential characteristics for HIL tests in avionic area.

TestML is conceived as a powerful XML-denoted language for exchanging test descriptions in the context of model-based testing of embedded automotive software. As TestML is targeted for covering different test stages from the module to integration and system tests as well as test levels from model-in-the-loop (MIL) to HIL, a minimally necessary combination of test units which are considered is imported to operate heterogeneous test environments, such as stimulation units, capturing units, evaluation units, and so on.

Thus, all the elements in TestML must be elementary enough to directly stimulate SUT and receive the reaction, omitting the complicated operating environment of SUT in avionic area. If TestML is used for avionic software testing, operating environment must be developed together with test cases. The development process is mixed and is not transparent for different developers. The elements are so elementary that the workload of test modeling for avionic system is trivial and boring, which lowers the efficiency.

While UML-STSAS only concerns the execution characteristics of software and hardware in system testing phase, it separates the development of operating environment from the test case development, and defines the common concepts and patterns as first-order language constructs. The design principle is that the network of components in operating environment takes charge of stimulating SUT and receiving the reaction, while the execution of operating environment and SUT are strictly controlled and observed by test facilities. The interactions between components adopt the predefined communication relationships. Compared with UML-STSAS, TestML mainly focuses on test descriptions for multiple test stages and multiple test phases. But it inevitably omits some information of characteristics in system testing phase and loses the corresponding efficiency. So, TestML is short of modeling HIL tests in system testing phase for avionic software.

At present, several test languages are designed to standardize the black-box and conformance tests, such as U2TP[6], TTCN-3[19] and JUnit[20]. However, their concepts focus on general domains unilaterally, and they do not pay attention to testing of continuous domain models which are commonly applied in the avionic industry. Using black-box test design techniques, U2TP aims to deal with test specification and test design at an abstract level higher than test execution. TTCN-3 is mainly used to specify and execute tests for telecommunication applications. A new approach based on TTCN-3, integrating concepts of testing real-time control systems, is already taken into consideration but not yet available for industrial practice[21-22]. As TestML, the extension of TTCN-3 still acts as a kind of message or procedure-based test lan-
The simulation testing is a critical phase of testing avionic software. It takes domain concepts as first-order language constructs for test modeling, and describes solutions using specified execution model and rules in the domain. The language improves the fundamental productivity of developing STSAS by improving abstraction. Using the language, the STSAS could be effectively designed, extended and integrated with the existing software and hardware systems. Besides HIL tests in system testing phase, the language can also be used for prototyping validation, integrating and system testing phase, and infra-structuring in model-based testing of avionic software.

6. Conclusions

(1) A domain-specific modeling language UML-STSAS is presented for modeling STSAS in this paper. With UML-STSAS, software modelers and developers can directly express the constructs and ideas to build STSAS for HIL tests.

(2) UML-STSAS has several features. It has the ability to express different kinds of test activities covering the state-of-the-art avionic test infrastructures. It also can represent HIL simulation configurations including software components, and express characteristics of different communication relationships among those components. Additionally, the introduction of automaton notation can express a superset of the existing means of description of test control flow. The used automaton notation allows defining common classes of avionic test descriptions which include deterministic and reactive test behavior. All these features narrow down the design space and enhance the level of abstraction.

(3) The lightweight extended mechanism makes the profile fully compliant with the UML 2.2 meta-model defined in MOF. This results in reusability of the existing infrastructures under MDA.

References

[11] UML Object Management Group. UML 2.2 super-


Biographies:

WANG Lize  Born in 1979, he is a Ph.D. candidate in Beihang University. His main research interest is modeling the reliability simulation and testing system of avionic software. E-mail: wanglize@gmail.com

LIU Bin  Born in 1967, he is a professor and doctoral supervisor in Beihang University. His main research interest is software reliability testing. E-mail: liubin@buaa.edu.cn