Experimental Applications of Automatic Test Markup Language (ATML)

Chatwin A. Lansdowne, Member, IEEE
Chris Gorringe, Member, IEEE
Patrick McCartney

Abstract—The authors describe challenging use-cases for Automatic Test Markup Language (ATML), and evaluate solutions. The first case uses ATML Test Results to deliver active features to support test procedure development and test flow, and bridging mixed software development environments. The second case examines adding attributes to Systems Modelling Language (SysML) to create a linkage for deriving information from a model to fill in an ATML document set. Both cases are outside the original concept of operations for ATML but are typical for integrating large heterogeneous systems with modular contributions from multiple disciplines.

Index Terms— Software standards, Testing, Test equipment, Test facilities, Software management, Software reusability, Fault diagnosis, Sensor systems and applications, System-level Design

I. INTRODUCTION

Automatic Test Markup Language (ATML) is an emerging standard that offers sophisticated markup and a modular framework for representing information needed to determine what tests to run and how to run them. ATML was developed to support sophisticated box-level (Unit Under Test, UUT) maintenance testing in the field, depot and at the factory.

But the ATML Concept of Operations (CONOPS) is fundamentally built around transferable concepts that can be extended to other usages. The National Aeronautics and Space Administration (NASA) invests heavily in design, builds redundancy (the spare parts) in, but produces and maintains very small quantities. In this environment, the focus of testing is run-once, with some tests being repeated as the design is refined, or for record with production units. Tests can range from exploratory engineering evaluations to formal acceptances. Performance margins and anomalies discovered during development tests will eventually be discussed with review boards, and the test outline may be reprioritised or expanded by the test team based on test outcomes during the ephemeral test opportunity.

II. ATML COMES ALIVE

NASA has demonstrated that ATML can be used in live message-passing, to describe and manipulate state variables in software elements controlling the testbed. [1] As we began composing procedures, we found that we needed additional metadata for the state variables so that “arbitrary knowledge” about settings needn’t be learned by rote, and so that results can be used without assumptions or required conventions. In summary the ability to send enough information so that individual test results can be interpreted correctly and acted on without reference to any external context or knowledge.

A. Multiple Ranges: Sets, Pick-Lists, Health, and Safety

In the “pick-list” application, a script or procedure developer needs to discover the values to which a string variable can be set, and pick the appropriate value from the list. This means that the c:Parameter uses a c:Expected range associated with it, to carry this list, as the constraints of the value. As an example, we scripted a networked power controller which could respond to outlet states of {"OFF", "ON", "CYCLE"} (although it only reports {"OFF", "ON"}). We found that the ATML Datum type was not conceived to define multiple ranges to express a set, however a reasonable accommodation exists, wherein the string is described by a collection of strings (Figure 1).

![Figure 1. c:Range Expressing a Set](image)

In this example, we gave the range the name "valid", as it identifies the settings that are valid. Ranges can also be used to communicate how to interpret the setting or reading. Examples could be “high”, “mid”, “low”, “alarm”, “full”, “empty”, “degraded”, “default”, “nominal”, “completed”,

Manuscript received June 1, 2012. This work was performed in NASA Johnson Space Center’s Avionics Systems Division, collaboratively with Cassidian. Funding was provided by a NASA CIO Information Technology Labs Study.

C. A. Lansdowne is with the National Aeronautics and Space Administration, Houston, TX 77058 USA (phone: 281-483-1265; fax: 281-483-6297; e-mail: chatwin.lansdowne@nasa.gov).

C. Gorringe is with Cassidian Test Engineering Services Ltd. (phone: +44 1202 868585; e-mail: chris.gorringe@eadst.co).

P. McCartney is with METECS, Houston, TX, 77058 USA.
“safe”, “unsafe” or others.

Data that is harvested through a tightly-coupled software application program interface (API) will often use an internal representation that needs to be translated for a user or a script developer. In Figure 2, the switch states from Figure 1 are represented by an enumerated type. Here, the “name” attribute of the c:Item element is used to associate a functional meaning with these integer values.

Figure 2. c:Range Expressing a Set, with c:Item@name

In our scenario, each operator is controlling many software elements, and each of those software elements could be controlling hardware elements. It has been a long standing need of ours to inform the operator of elements that are missing or degraded. Last year we suggested enforcing conventions [2] for health and safety indicators. But constraints that are easy to meet for one development environment are challenging for another tool. Arbitrary constraints can be eliminated by instead taking a data-driven approach, providing markup to describe what “healthy” is or what “safe” is (observe, these ranges need not be orthogonal).

ATML attaches a named range to a Datum—but only one; consider the range name as the range classification to which the value belongs. It was not conceived that a “valid” range, a “safe” range, and a “healthy” range would all be provided; the multiplicity is one. Although awkward and verbose, a technique was identified that validates against the schema.

Figure 3. c:Datum with Multiple Ranges that are Sets

In Figure 3 the software has an error condition if the value is 0, is considered “healthy” if the value is -1, and the set (concept from Figure 1) of both 0 and -1 are “valid”. We also used the name attribute of the c:Item elements to communicate that the software environment uses an internal representation of 0 to mean “false” and an internal representation of -1 to mean “true”. The construct is curious, as nested empty c:Collection’s are used to carry the c:Range’s.

But this construct also allows c:SingleLimit and c:LimitPair ranges to describe “healthy”, “safe”, and “valid” conditions. In Figure 4 the same construct is applied to a floating-point data type. In this example, the thermocouple is reading 72.5°C. A “failed” thermocouple manifests by reporting a value of -20°C. The thermocouple can report readings in the (“valid”) range of -20 to 100, but readings outside the range of -10 to 80 are not “safe”. The “unsafe” range here illustrates using named sub-ranges (“unsafe.low” and “unsafe.high”) to not only classify a temperature reading as “unsafe” but provide further interpretation of what the unsafe condition is.

Figure 4. c:Datum with Multiple Ranges using Limits

B. tr:TestResults as a Status and Control Document

The tr:TestResults document construct can be used to describe a software configuration and status snapshot during a live operation (Figure 5). This enables a data-driven data harvest from source-points in ATML format, rather than collection in one format and later conversion to ATML.

In such a document, the tr:Parameters section is used to
describe the configurable (input) parameters, while tr:TestResult is used for measurements and status (output) parameters. tr:TestResult and tr:Parameter do not offer identical metadata. Internally tr:TestResult uses the tr TestData extension of tr:Data which has the sole variance of an added acquisitionTimeStamp attribute added to the c:Value type, while tr:Parameter itself has a timestamp attribute. Also, the ID attribute of tr:Parameter is a c:NonBlankString, whereas the ID attribute of tr:TestResult is a more restrictive xs:ID. And a tr:Parameter cannot have a tr:Transform or tr:Extension, which could be useful for converting between an internally useful representation and an externally useful representation. tr:TestResult also offers optional tr:Outcome, tr:Indictments, tr:TestLimits, and tr:Extension, differences we considered incidental.

Figure 5. tr:TestResults Document Structure

Placing a tr:Test construct inside of tr:ResultSet is optional. We have used this additional metadata to identify role (test-point) of the software in the context of the test.

One must eventually conclude that tr:Outcome must be "Aborted", as it is neither "Passed" nor "Failed".

The tr:Personnel construct allows the user of the software or host to be recorded with the data. It also could allow a user to take responsibility for manually inspecting the configuration, by creating a record using the tr:QualityAssurance element.

We also observed that in the dynamic environment of a development test, experimentation with algorithms can occur and the software version itself becomes a test variable. tr:TestProgram uses a c:SoftwareInstance, which has a c:ReleaseData of type xs:date, although in this situation, the timestamp on the software could change more than daily.

A self contained test results description is verbose, causing repetition of static metadata every time it is exchanged. The efficiency of the test result exchange can be dramatically improved by considering a test result as an instance of an ATML Test Description (1671.1), where the metadata is described in the test description. In this scenario the actual value construct remains the same, however the test result exchange needs only to identify the related test (and test metadata) and the specific value, the remaining parameter metadata can all be inherited from the test description.

III. DERIVING ATML FROM SYSML

In any multi-disciplinary high-performance design endeavour it is necessary to package information for subsystems while maintaining a clear depiction of the whole system and the environment it must operate in. In this context, ATML is part of a larger information ecosystem, and it needs to be exchanged between the tools of that ecosystem, not hand-generated. The information will have versions: original design, changed design, as-built, etc. Throughout the process, information must be traceable to the authoritative source, and copies of information must be maintained synchronously. Traditionally, this was enforced by manual reconciliation of branches. We explored the simplest automation, regenerating
information packages from single authoritative sources.

The ATML CONOPS employs standardized test sets. Test, maintenance, and diagnostic strategies must be considered during the design phase, and this means that information about test requirements must be reconciled with information about fielded test assets at Preliminary Design Review (PDR) [3].

Remember, ATML is an information framework. The philosophy is, describe the requirements, describe the capabilities of the test set, and describe the interconnections. This will allow different implementations and design abstractions to allocate resources, throw switches, run the test, and interpret the results. This data-driven approach does not require that every product use the same source code or development environment, only the same information format. The respective curators of information supply the information they curate. Capabilities can be changed without rewriting software, if only the affected information is maintained.

The Systems Engineering process has since the early 1980’s been portrayed using a “V” diagram (Figure 7). The process progressively decomposes a problem from studies and concepts into requirements, subsystems, and components. The design is executed, and then progressively integrated and tested from components to subsystems, to system verification (against requirements) and validation (against concepts of operations). Finally it is deployed, with continuing support for breakdowns, feature changes, and upgrades, ending with retirement or replacement at obsolescence.

To derive an ATML document from a SysML model, information that will be loaded into the document must be provided in the model. This requires creating libraries of blocks and stereotypes with inheritable attributes. An ontology, or at least a naming convention, is required so that the relevant attributes can be located and interpreted.

B. Literal or Abstract

Our first, direct approach was to represent ATML constructs literally in SysML. ATML attributes could be expressed as properties of a UML stereotype, or of a SysML block. Points of confusion include when to use a stereotype and when a block, where to put the value of the ATML element, and how to handle an xml “choice.” It did not look possible for SysML properties to themselves have properties (corresponding to XML attributes). Further, placing and changing default information in these structures in SysML without instantiating them requires frequent use of redefinition, which is clumsy in the MagicDraw 17.0 tool. The complex structures ATML supports for describing complex hardware could be daunting and unnatural for a SysML tool user. Also consider, both ATML and SysML are at an intermediate stage of development as neither presently achieves alignment with any ontology. The exercise was however useful for developing greater familiarity with both ATML and SysML.

Thus we fell back to creating library components with standardized ATML-mapped attributes that might work more naturally in the SysML editor but represent ATML concepts. This approach should allow us to export constructs from SysML, and map test results back to SysML.

C. Requirements

The requirements diagram in SysML is an extension to UML. As such it is a less mature component. Requirements from SysML do not appear useful for automatic testing, as they have been implemented as human-interpreted “dumb text,” and ATML needs formal testable requirements.

D. Quantities and Units

SysML ties units to QUDV, an ontology now used by OMG. Presently, only SI units are supported in QUDV, and MagicDraw only provides a subset of those. Units and Quantities in ATML are key concepts, however their validation and definition are relatively weak.

ATML “Values” have associated Quantities, Units identified through their “unitSymbol”. ATML inherits Quantities and unit Symbol definitions from the IEEE Std 1641 Standard, which codifies a normalized set of engineering Quantities and unitSymbols from the recommendations made in IEEE Std 260. A better approach, and one actively being pursued by the ATML working groups, is to extend the ATML Value model to base the units on a formally defined unit set, utilizing an ontology, enabling automated conversion between domain specific units without pre-build knowledge and extending the set of allowed Units.

ATML identifies some concepts that today’s unit ontologies overlook. These include the “unitQualifier”; although the usage of this field has not been standardized by IEEE, it is
intended to associate a statistical method with the measurement. This is important when comparing measurements (for example, 2 V p-p and 0.7 V rms). But it could also be the key to data-driven data aggregation. When aggregating “peak-peak” values, report the maximum value. When aggregating “rms” values, take the rms of the values. ATML also enables the capture of Resolution, Range, Confidence, and ErrorLimits which are needed for comparing a measurement against a test requirement.

E. Connectors and Wiring

SysML ports can be overloaded (Figure 8) to form a construct that resembles a connector with pins. In the model, we now represent an electrical circuit with a signal return. An Electrical Interface block can be added as an intermediate step; in the normal course of modeling, a high-level model would be generated first, and details would be added and defined progressively. Thus, the model now says that Power Controller has a powerCtrlr_EIF, but we haven’t yet specified how many connectors we’re using, how many pins, what kind they are, or what they’re called. This block however allows us to identify signals in the model that will be brought out.

If we use a SysML Internal Block Diagram (IBD) to connect the signals, it will create an instance of powerCtrlr_EIF which we can wire, but that doesn’t actually connect powerCtrlr_EIF. Alternatively, redefine the pins on Power Controller to connect them to powerController_EIF. Redefining pins in MagicDraw was tedious and error-prone.

Now we can add the electrical connectors. In Figure 9, the ConnectorACPowerPlug2 inherits from atmlConnectorElectrical, but its matingConnectorType and cost properties have been redefined. One method is to say powerCtrlr_EIF “is a” ConnectorACPowerPlug2 and also “is a” ConnectorACPowerSocket2, and the pins are connected through again by redefinition. We’ve iterated on this concept here, but haven’t concluded which answer is best. Again, on an IBD we could have directly created instances of ConnectorACPowerPlug2. But on the BDD we needed instead to create the J1 and J2 connectors and then type them from the ATML-derived connector library.

Figure 8. Adding an Electrical Interface to a SysML Model

Figure 9. Adding Electrical Connectors to a SysML Model

It might be desirable for SysML to report a type mismatch when connectors are mis-mated. Alternatively, users may prefer an iterative approach so that the model can be connected together and verified first, then come back and identify where mechanical adapters are needed.

F. Capabilities, Resources, Ports, and Signals

In the course of this work we came to understand that ATML does not begin test preparation by describing a configuration diagram. Instead it works from a list of requirements to be verified: the signal at certain pins is to meet some description with some tolerance. Then the test set peruses its inventory to find a capability to make such a measurement within the tolerance, examines wiring information, and configures the switch fabric to connect the signal to the instrument.

Figure 10. BDD and IBD Techniques for Representing Capabilities and Resources

The term “Port” has a specific meaning in SysML distinct from its specific meaning in ATML. In SysML, a port is a point on a boundary. In ATML, a “Port” is a logical entity, that represents an abstract interface through which signal(s) flow; where it also has a physical component these are defined by a nodal collection of pins and their connectors that need to be joined to perform a capability (generally, routing signals in hardware). SysML has UML standard ports and flow ports, and these ports are used to represent variable parameters.

An ATML Instrument has Resources which have Ports that are used to supply or measure signals. [5] The Instruments
themselves have Ports designating connectors and pins in the physical interface that Resources map to.

ATML resources have ATML ports, which must be “wired” to the instrument ATML ports; it is also possible to describe that a switch controls which instrument port is wired to the resource. [6] ATML resources can support multiple “capabilities,” which are either signal (stimulus) and/or measurement (response) descriptions, and these must also be mapped to the resource ports.

Switches were not represented in this iteration.

G. The Information Harvest

The Object Management Group (OMG) developed the Meta-Object Facility (MOF) as a means for expressing metadata from model languages such as SysML. The XML Metadata Interchange (XMI) is a means for exchanging this metadata using the eXtensible Markup Language (XML). Our approach to harvesting the model information from SysML was to process the XMI file exported from the SysML modeling tool.

The MOF uses the notion of MOF::Classes to define concepts (model elements) on a meta layer. The biggest challenge in parsing the XMI file is to map all of the classes with the relevant elements and so develop useful information about the described system. This is done using an extensive series of unique identifiers for each class and each element in the diagram. For example, the ACPowerPlug connector is defined as a packagedElement with a type of uml:Class and given a unique ID. The power supply itself (myAvBox) is also defined as a packagedElement with a type of uml:Class and given a unique ID. Within the power supply element, an ownedAttribute element is defined that is given a type ID that references the ACPowerPlug definition. This relationship is shown in Figure 11.

![Figure 11. XMI Snippet Showing Relationships between Block diagram elements and definitions](image)

The type of linkage shown in Figure 11 is propagated for every component of the SysML block diagram as well as all of the definitions and properties which are referenced within the diagram. The XMI cannot be processed as a stream since it is not known ahead of time which definitions will be referenced elsewhere in the file and how many times they will be used. Thus, an XMI parser must store every unique ID and all of the information associated with each uml:Class.

A simple PHP-based processor was written to prove that it is possible to trace through the XMI linkages and harvest all of the necessary information by mapping all of the unique identifiers. It is clear the information exists in the XMI output and it could be followed with a more robust processor. Once these linkages are traced, they can easily be mapped to another format such as ATML.

Using library support files to contain the ATML constructs was transparent, as MagicDraw duplicated the information in the project XMI file. The use of redefinitions was not observed to be a problem.

IV. CONCLUSION

ATML is not merely an information format, but an information framework designed to support a streamlined workflow. It was shown here how ATML documents might be generated programmatically, for automatic discovery and collection. ATML was applied to harvest not merely data, but also the metadata describing configuration and control variables in a heterogeneous software environment.

We also investigated how system models and ATML documents might be linked together. SysML models will need to be derived from libraries of ATML constructs so that information can be extracted by data-driven algorithm. The exchange of information between ATML and SysML cannot be performed by a data-driven XSLT translation, an intelligent application needed which knows the constructs on each side.

The techniques we explored here remain to be validated by the SysML user community [7], and it remains to make the actual conversion from SysML to an ATML document set and then actually use those documents to do useful work.

ACKNOWLEDGMENT

The authors would like to acknowledge the benefit we received from the superior experience of several SysML users at NASA’s Jet Propulsion Laboratory. These include Mike Seivers, who pointed out the UML/SysML Test Profile. Mark McKelvin, who identified our use of “Singleton instances” of classes. Mark also urged that libraries don’t own things, they own “characterizations” of the things. Marcus Wilkerson has already been able to generate wirelists from models. Marcus demonstrated for us how SysML ports can be overloaded, and how detail can be added incrementally to the model of the interface. We would also like to acknowledge Anand Jain, National Instruments, for identifying the distinction of carrying metadata in Test Description and data in Test Results.

REFERENCES