

Experimental Applications of Automatic Test Markup Language (ATML)

Chatwin A. Lansdowne, *Member, IEEE*

Patrick McCartney

Chris Gorringer, *Member, IEEE*

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 when integrating large heterogeneous systems with modular contributions from multiple disciplines.

Index Terms— Software standards, Test equipment, Test facilities, Testing, 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

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. Gorringer is with EADS Test and Services (UK) Ltd. (phone: +44 1202 872800, e-mail: chris.gorringer@3eads-ts.com).

P. McCartney is with METECS, Houston, TX, 77058 USA.

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).¹

```
<c:Datum xsi:type="c:string">
  <c:Range name="valid">
    <c:Expected comparator="EQ">
      <c:Collection>
        <c:Item><c:Datum xsi:type="c:string"><c:Value>OFF</c:Value></c:Datum></c:Item>
        <c:Item><c:Datum xsi:type="c:string"><c:Value>ON</c:Value></c:Datum></c:Item>
        <c:Item><c:Datum xsi:type="c:string"><c:Value>CYCLE</c:Value></c:Datum></c:Item>
      </c:Collection>
    </c:Expected>
  </c:Range>
</c:Datum>
```

Figure 1. `c:Range` Expressing a Set

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”, “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.

```
<c:Datum xsi:type="c:integer" value="1">
  <c:Range name="valid">
    <c:Expected comparator="EQ">
      <c:Collection>
        <c:Item name="OFF"> <c:Datum xsi:type="c:integer" value="0"/></c:Datum></c:Item>
        <c:Item name="ON"> <c:Datum xsi:type="c:integer" value="1"/></c:Datum></c:Item>
        <c:Item name="CYCLE"> <c:Datum xsi:type="c:integer" value="2"/></c:Datum></c:Item>
      </c:Collection>
    </c:Expected>
  </c:Range>
</c:Datum>
```

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 (and 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 existing schema.

```
<c:Datum xsi:type="c:integer" value="-1">
  <c:Range>
    <c:Expected comparator="EQ">
      <c:Collection>
        <c:Item><c:Collection><c:Range name="error"><c:Expected comparator="EQ">
          <c:Datum xsi:type="c:integer" value="0"/>
        </c:Expected></c:Range></c:Collection></c:Item>
        <c:Item><c:Collection><c:Range name="healthy"><c:Expected comparator="EQ">
          <c:Datum xsi:type="c:integer" value="-1"/>
        </c:Expected></c:Range></c:Collection></c:Item>
        <c:Item><c:Collection><c:Range name="valid"><c:Expected comparator="EQ">
          <c:Collection>
            <c:Item name="false"><c:Datum xsi:type="c:integer" value="0"/></c:Item>
            <c:Item name="true"><c:Datum xsi:type="c:integer" value="1"/></c:Item>
          </c:Collection>
        </c:Expected></c:Range></c:Collection></c:Item>
      </c:Collection>
    </c:Expected>
  </c:Range>
</c:Datum>
```

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.](#)

```
<c:Datum xsi:type="c:double" value="72.5" standardUnit="°C">
  <c:Resolution>0.01</c:Resolution>
  <c:Range>
    <c:Expected comparator="EQ">
      <c:Collection>
        <c:Item><c:Collection><c:Range name="failed">
          <c:Expected comparator="EQ">
            <c:Datum xsi:type="c:double" value="-20"/>
          </c:Expected>
        </c:Range></c:Collection></c:Item>
        <c:Item><c:Collection><c:Range name="safe">
          <c:LimitPair operator="AND">
            <c:Limit comparator="GE">
              <c:Datum xsi:type="c:double" value="-10"/>
            </c:Limit>
            <c:Limit comparator="LE">
              <c:Datum xsi:type="c:double" value="80"/>
            </c:Limit>
          </c:LimitPair>
        </c:Range></c:Collection></c:Item>
        <c:Item><c:Collection><c:Range name="unsafe">
          <c:Expected comparator="EQ">
            <c:Collection>
              <c:Item><c:Collection><c:Range name="unsafe.low">
                <c:SingleLimit comparator="LT">
                  <c:Datum xsi:type="c:double" value="-10"/>
                </c:SingleLimit>
              </c:Range></c:Collection></c:Item>
              <c:Item><c:Collection><c:Range name="unsafe.high">
                <c:SingleLimit comparator="GT">
                  <c:Datum xsi:type="c:double" value="80"/>
                </c:SingleLimit>
              </c:Collection>
            </c:Collection>
          </c:Expected>
        </c:Range></c:Collection></c:Item>
      </c:Collection>
    </c:Expected>
  </c:Range></c:Collection></c:Item>
  <c:Item><c:Collection><c:Range name="valid">
    <c:LimitPair operator="AND">
      <c:Limit comparator="GE">
        <c:Datum xsi:type="c:double" value="-20"/>
      </c:Limit>
      <c:Limit comparator="LE">
        <c:Datum xsi:type="c:double" value="100"/>
      </c:Limit>
    </c:LimitPair>
  </c:Range></c:Collection></c:Item>
</c:Range>
</c:Expected>
</c:Datum>
```

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.



Figure 5. tr:TestResults Document Structure

In such a document, 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:Indigtments, tr:TestLimits, and tr:Extension, differences we considered incidental.

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. But in this situation, the timestamp on the software could change more than once per day.

C. Implementation

For demonstration, an implementation was developed in

LabVIEW which discovers the controls on a Virtual Instrument panel, then describes them in an ATML tr:TestResults document from properties configured by the software developer. This technique places minimal additional burden on the developer to produce a well-documented interface, and relieves the burden of maintaining the same software documentation in two formats (one in LabVIEW control properties, and one in ATML).

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... Throughout the process, information must be traceable to the authoritative source of information, and copies of information must be maintained synchronously. Traditionally, this was enforced by manual reconciliation of branches. Here we explore the simplest automation, regeneration of 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 project Preliminary Design Review (PDR) [3].

Remember, ATML is an information framework. The philosophy is, describe the requirements, describe the capabilities of the test set, 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. 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 6Figure-6). 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.

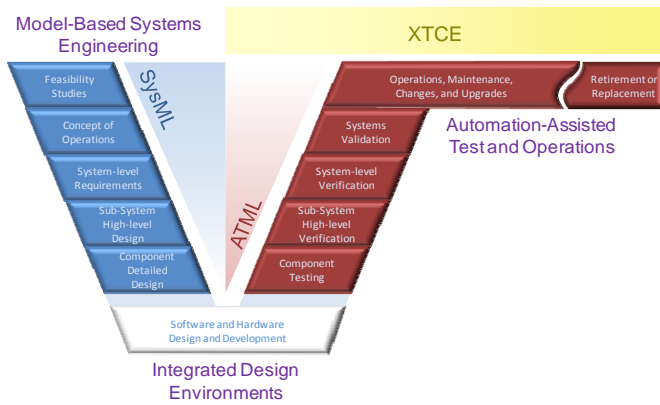


Figure 6. Systems Engineering “Vee” Diagram

As Figure 6 suggests, different kinds of tools support the work of different phases. Systems Modelling Language (SysML) is an Object Management Group (OMG) standard sponsored by the International Council on Systems Engineering (INCOSE) and gaining acceptance as a method of expressing, verifying, and validating a system design prior to execution in hardware and software. SysML is a subset of Unified Modelling Language (UML 2) with extensions.[4] The object of using SysML is that the design can be verified against requirements (close the “V”) before implementation begins. Following implementation, the unit, subsystem, and system need to be verified (ATML) against the model (SysML), and characterization (ATML) fed back into the model (SysML).

A. Implementation

To explore potential approaches, MagicDraw was used as a SysML editor. A test configuration block diagram for a power converter was modelled as a test case.

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. Further, ATML supports very complex structures for describing complex hardware, and these 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 trying to create library components with standardized ATML-mapped attributes that might work more naturally in the SysML editor but represent ATML concepts. This approach could 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.”

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 includes a subset of those. Units in ATML are relatively weak.

ATML identifies some concepts that today’s 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, a signal measuring 2 V p-p can’t be compared directly to a signal measuring 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. For example, it is intended that, given a requirement to measure the power converter output voltage with a passing result between 24V and 32V, a reasoner could find an instrument capability in its inventory that can make the measurement with the necessary resolution, and then compare the measured result with the requirement.

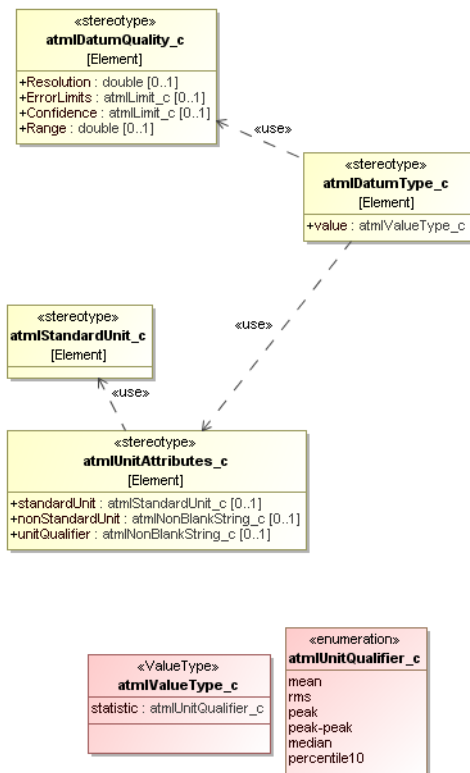


Figure 7. Extending SysML for ATML Quantities

The UML stereotypes shown in Figure 7 were not ultimately useful. The extended ValueType worked well in isolation, its value can be selected from the pick-list provided by the enumeration. It was not evaluated in a practical application.

E. Connectors and Wiring

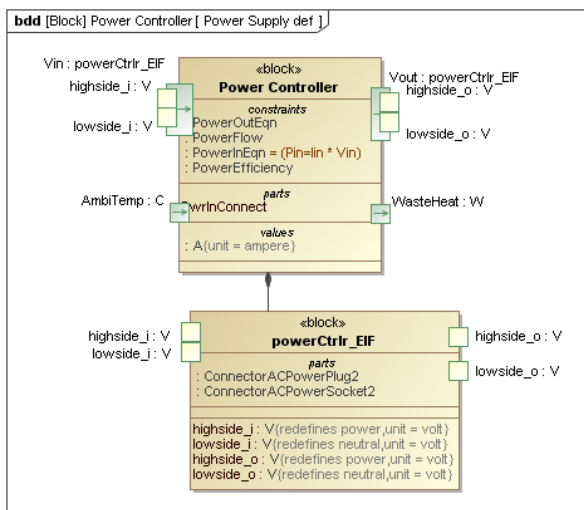


Figure 8. Adding an Electrical Interface to a SysML Model

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 does allow us to identify signals in the model that are going to be brought out.

Connecting the signals is tricky; if we use a SysML Internal Block Diagram (IBD), it will create an instance of powerCtrlr EIF which we can wire, but that doesn't actually connect powerCtrlr EIF. One approach is to 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.

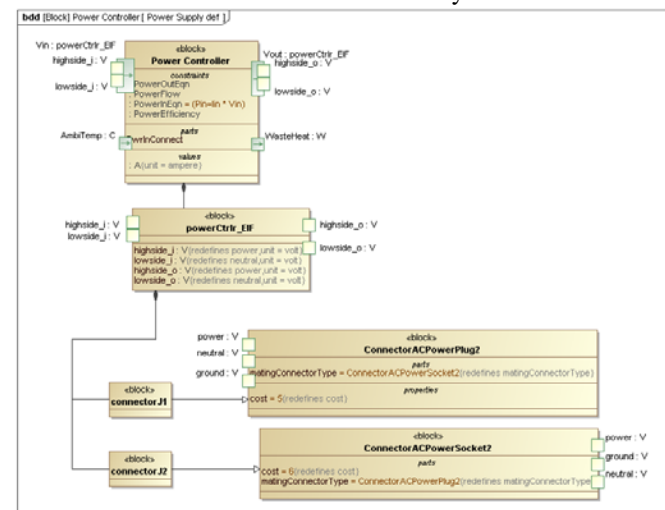


Figure 9. Adding Electrical Connectors to a SysML Model

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. It appears to stop short of running the test, and this is an area where the NASA Automation Hooks

Architecture can help by enabling discoverable parameters to be mapped to capabilities.

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 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 signals or measure signals. [5] The Instruments themselves have Ports designating connectors and pins in the physical interface that Resources map to.

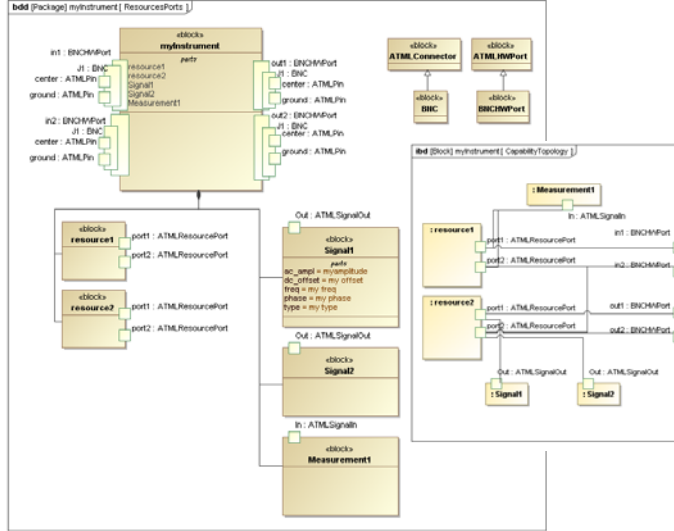


Figure 10. BDD and IBD Techniques for Representing Capabilities and Resources

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 several “capabilities,” which are either signal (stimulus) 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 appropriate elements and end up with 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

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. Tracing through these linkages can become quite complicated even for this simple model. 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` so that it can interpret the actual model of interest.

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. This processor would require additional development to work for a generic block diagram and also for other types of SysML models, but 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 rearranged and output in 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. A literal approach appears undesirable, as the ATML complex literal constructs are difficult to use in SysML, and ATML needs to transition to an ontology anyway. Thus, an abstract representation of ATML concepts is recommended even though this requires more work to strip. The exchange of information between ATML and SysML cannot be performed by a data-driven XSLT translation, an intelligent application is required which understands 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. We explored two techniques for “wiring” connector pins and ATML resources. One used instances in an internal block diagram (IBD) and the other used port redefinition in a block definition diagram (BDD). We would like further evaluation by SysML users to determine which is “best.” It also remains to be decided whether it should be possible or not to mis-mate connectors in the SysML editor.

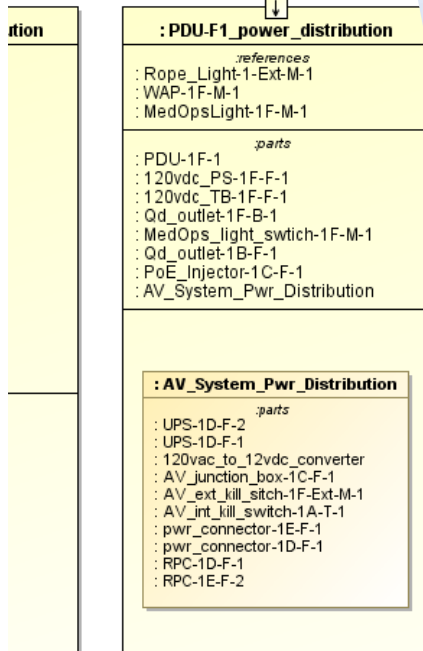
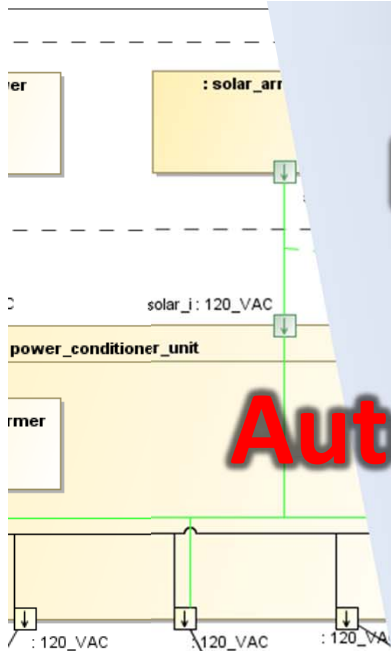
ACKNOWLEDGMENT

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.

REFERENCES

- [1] C. A. Lansdowne, J. R. MacLean, et. al., *Automation Hooks Architecture Trade Study for Flexible Test Orchestration*, ISBN 978-1-4244-7960-3, Autotestcon Proceedings, Sep. 2010.
- [2] C. A. Lansdowne, J. R. MacLean, et. al., *Automation Hooks Architecture—Concept Development and Validation*, EDAS 1569424063, Autotestcon Proceedings, Sep. 2011.
- [3] C. Gorringer, *The Use of ATML in Managing TPS Developments and Life Cycle Maintenance*, Autotestcon Proceedings, Sep. 2010
- [4] S. Friedenthal, A. Moore, R. Steiner, *A Practical Guide to SysML*, ISBN: 978-0-12-3786074-4, Elsevier, 2009
- [5] IEEE Std 1671-2010, DOI 10.1109/IEEESTD.2011.5706290, January 20, 2011, Appendix F
- [6] C. Gorringer, T. Lopes, D. Pleasant *ATML Capabilities Explained*, DOI [10.1109/AUTEST.2007.4374218](https://doi.org/10.1109/AUTEST.2007.4374218), Autotestcon Proceedings, Sep. 2007
- [7] C. Delp, L. Cooney, et al, *The Challenge of Model-based Systems Engineering for Space Systems, Year 2*, INCOSE INSIGHT, vol. 12, Issue 4, pp. 36-39, Dec. 2009

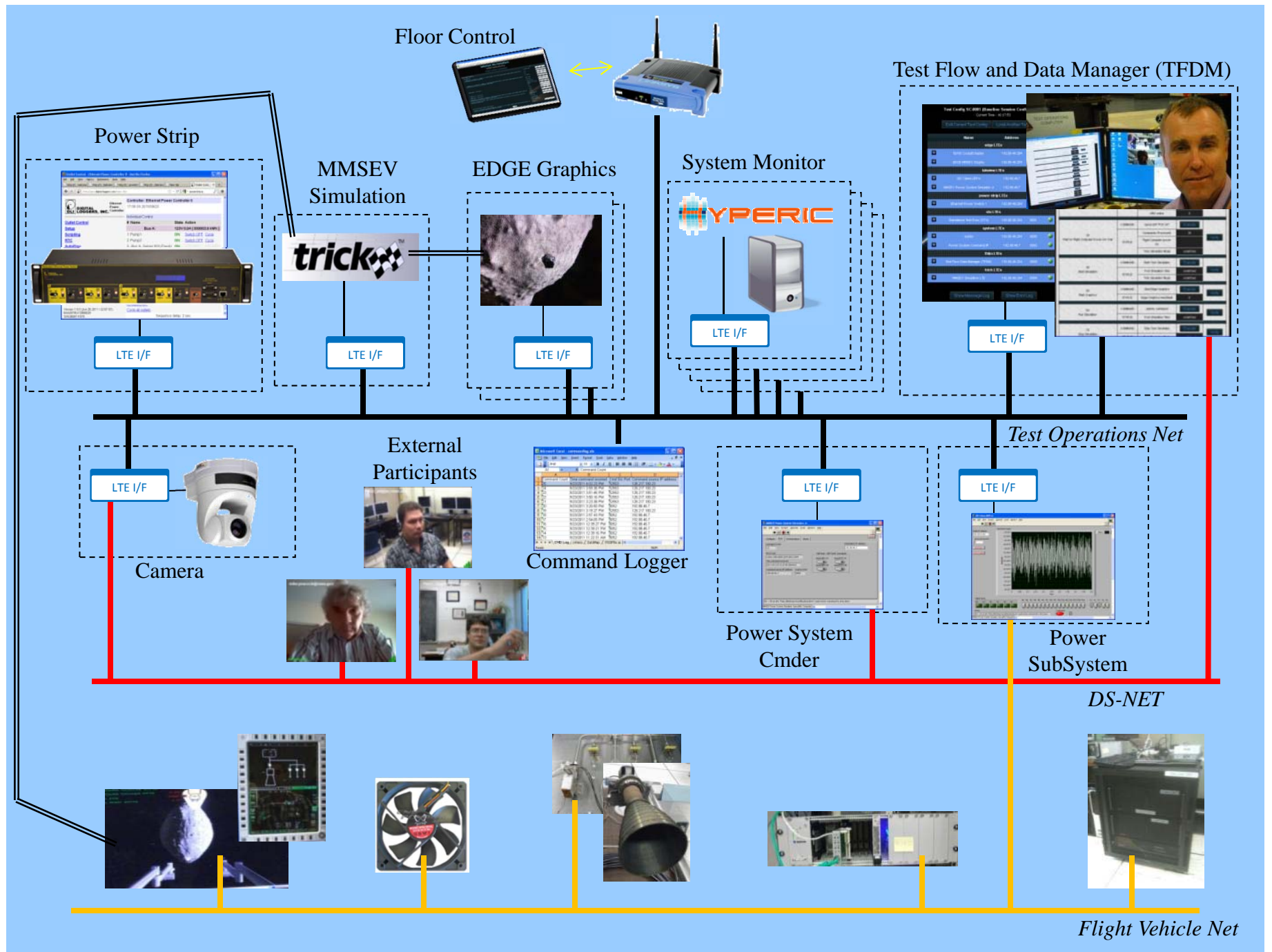
Experimental Applications of Automatic Test Markup Language (ATML)

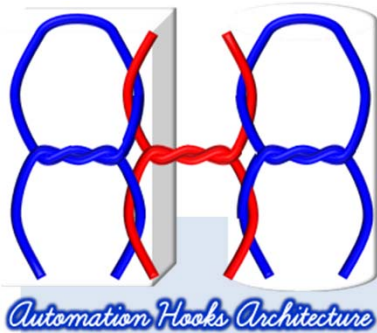


Chatwin Lansdowne
Chris Gorringer
Patrick McCartney

Sept. 13, 2012







Automation Hooks Architecture API

- **Advertised**
 - Automated Discovery: Dynamic “Plug-and-Play”
- **REST Architecture**
 - Two commands: GET and PUT
 - Versatile: co-host support files and hyperlinks– interface definitions, requirements, theory of operation, streaming data, GUI...
- **HTTP**
 - standard messaging, error messages, compression, security, caching

- **Xml**
 - Archive-quality
 - Enables Data-driven software architecture
 - Foundation of artificially intelligent data processing
 - Self-describing message format
 - Create database tables by script
- **hypermedia layout**
 - Insulates against layout changes
 - Coexistence of variations
 - Separate metadata for caching

- **xml:ATML (IEEE 1671)**
 - standardizes units, arrays, time zone
 - Scope includes signals, instrument capabilities, problem reporting
 - exciting opportunities for COTS tools and radically different engineering work flows

- **Orchestration features**
 - Health and Status Rollup
 - Synchronizing and Scheduling



mREST

Testing

Example: Controlling a Web Power Switch



- Switch vendor's interface requires screen-scrape to ATML
- But ATML can coexist with HTML

Outlet Control - Comfort Inn - ABCville - Mozilla Firefox

epcr.digital-loggers.com/index.htm

DIGITAL LOGGERS, INC. Ethernet Power Controller

[Outlet Control](#)
[Setup](#)
[Scripting](#)
[Date/Time](#)
[AutoPing](#)
[System Log](#)
[Logout](#)
[Help](#)

[Manual](#)
[FAQ](#)
[Product Information](#)
[Digital Logger Inc.](#)

Version 1.6.0 (Jun 22 2012 / 21:56:21)
8AA39795-EE789B21

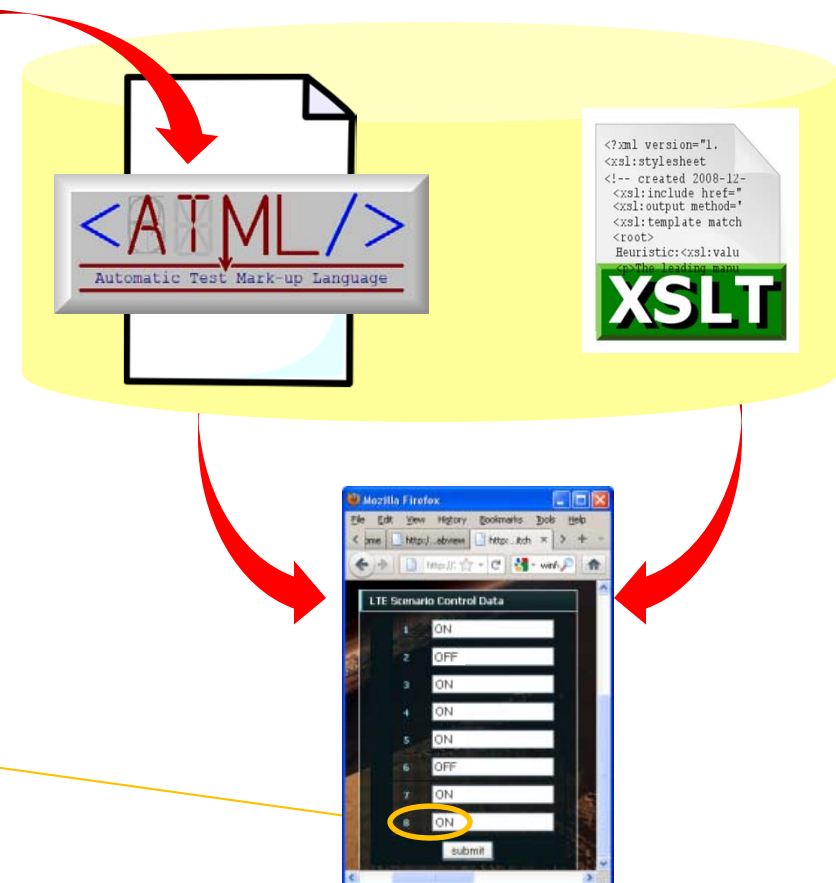
Controller: Comfort Inn - ABCville
Fri Jul 20 11:18:04 2012

Individual Control

#	Name	State	Action
Bus A: 121V 0.0A [000000.2 kWh]			
1	Nomadix 5600	ON	Switch OFF Cycle
2	E1 Switch	OFF	Switch ON
3	W1 Switch	ON	LOCKED LOCKED
4	E2 Switch	ON	Switch OFF Cycle
Bus B: 123V 0.0A [000003.1 kWh]			
5	Outlet 5	ON	Switch OFF Cycle
6	Outlet 6	OFF	LOCKED
7	Outlet 7	ON	Switch OFF Cycle
8	Life Support	ON	Switch OFF Cycle

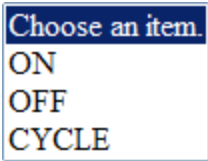
Master Control

[All Outlets OFF](#)
[All Outlets ON](#)
[Cycle all Outlets](#)



Parameter Pick-Listing using ATML c:Range (Expressing sets)


```
<c:Datum xsi:type="c:string">  
  <c:Range name="valid">  
    <c:Expected comparator="EQ">  
      <c:Collection>  
        <c:Item><c:Datum xsi:type="c:string"><c:Value>OFF</c:Value></c:Datum></c:Item>  
        <c:Item><c:Datum xsi:type="c:string"><c:Value>ON</c:Value></c:Datum></c:Item>  
        <c:Item><c:Datum xsi:type="c:string"><c:Value>CYCLE</c:Value></c:Datum></c:Item>  
      </c:Collection>  
    </c:Expected>  
  </c:Range>  
  <c:Value>ON</c:Value>  
</c:Datum>
```



- Associate a “valid” c:Range with the parameter
- Express the set as a c:Collection of c:Item
- *Issue:* requires “knowing how to” compare a string to a collection of strings

Parameter Pick-Listing using ATML c:Range

```
<c:Datum xsi:type="c:integer" value="1" >
  <c:Range name="valid">
    <c:Expected comparator="EQ">
      <c:Collection>
        <c:Item name="OFF">    <c:Datum xsi:type="c:integer" value="0"></c:Datum></c:Item>
        <c:Item name="ON">    <c:Datum xsi:type="c:integer" value="1"></c:Datum></c:Item>
        <c:Item name="CYCLE"><c:Datum xsi:type="c:integer" value="2"></c:Datum></c:Item>
      </c:Collection>
    </c:Expected>
  </c:Range>
</c:Datum>
```



Choose an item.	
ON	0
OFF	1
CYCLE	2

- Often, the internal representation of a parameter is not meaningful to the user
- Example: enumerated list
- ATML can carry both internal and user-oriented representations, with Item@name

“healthy”, “safe”, “valid”: Expressing Multiple Ranges

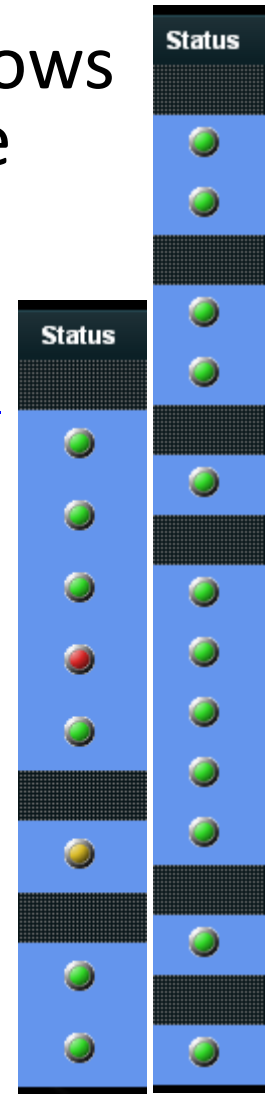
```

<c:Datum xsi:type="c:integer" value="-1">
  <c:Range>
    <c:Expected comparator="EQ">
      <c:Collection>
        <c:Item><c:Collection><c:Range name="error"><c:Expected comparator="EQ">
          <c:Datum xsi:type="c:integer" value="0"/>
        </c:Expected></c:Range></c:Collection></c:Item>
        <c:Item><c:Collection><c:Range name="healthy"><c:Expected comparator="EQ">
          <c:Datum xsi:type="c:integer" value="-1"/>
        </c:Expected></c:Range></c:Collection></c:Item>
        <c:Item><c:Collection><c:Range name="valid"><c:Expected comparator="EQ">
          <c:Collection>
            <c:Item name="false"><c:Datum xsi:type="c:integer" value="0"/></c:Item>
            <c:Item name="true"><c:Datum xsi:type="c:integer" value="-1"/></c:Item>
          </c:Collection>
        </c:Expected></c:Range></c:Collection></c:Item>
      </c:Collection>
    </c:Expected>
  </c:Range>
</c:Datum>

```

- ATML schema allows 0..1 named range

- Need a data-driven way to roll up health and safety status

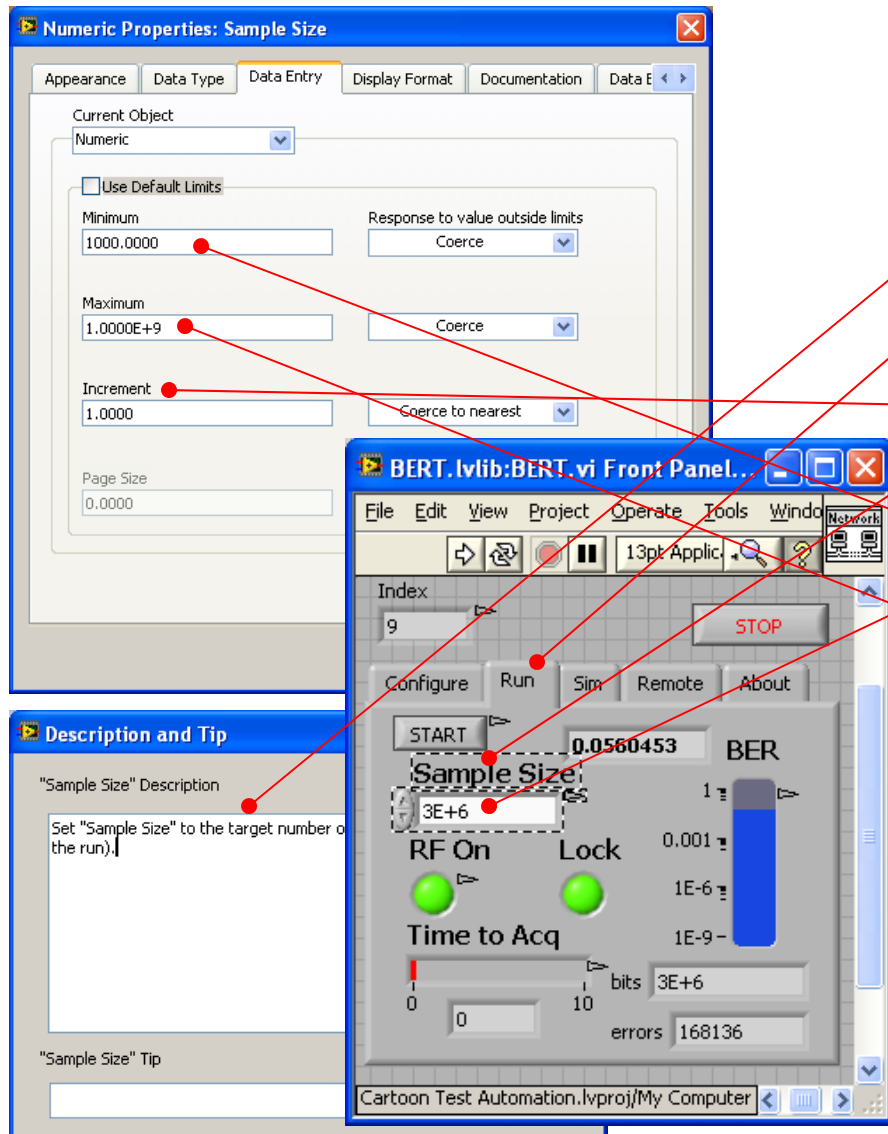


Using multiple ranges, with limits

```
<c:Datum xsi:type="c:double" value="-10"/>
</c:Limit>
<c:Limit comparator="LE">
  <c:Datum xsi:type="c:double" value="-10"/>
</c:Limit>
</c:LimitPair>
</c:Range></c:Collection></c:Item>
<c:Item><c:Collection><c:Range name="unsafe">
  <c:Expected comparator="EQ">
    <c:Collection>
      <c:Item><c:Collection><c:Range name="unsafe.low">
        <c:SingleLimit comparator="LT">
          <c:Datum xsi:type="c:double" value="-10"/>
        </c:SingleLimit>
      </c:Range></c:Collection></c:Item>
      <c:Item><c:Collection><c:Range name="unsafe.high">
        <c:SingleLimit comparator="GT">
          <c:Datum xsi:type="c:double" value="80"/>
        </c:SingleLimit>
      </c:Range></c:Collection></c:Item>
    </c:Collection>
  </c:Expected>
</c:Range></c:Collection></c:Item>
<c:Item><c:Collection><c:Range name="valid">
  <c:LimitPair operator="AND">
    <c:Limit comparator="GE">
      <c:Datum xsi:type="c:double" value="-20"/>
    </c:Limit>
    <c:Limit comparator="LE">
```

- Example demonstrates using sub-ranges to provide further interpretation for the operator

Example of Data-Driven Flow from LabVIEW to ATML



```
<tr:Parameter ID="Run.Sample Size" name="Sample Size">
  <!-- Parameter ID is a c:NonBlankString, so it could be traceable
  to the software variable name/-->
  <tr:Description>Set "Sample Size" to the target number of
  bits to test (duration of the run).</tr:Description>
  <tr:Data>
    <c:Datum xsi:type="c:double" value="3000000"
    nonStandardUnit="bits">
      <c:Resolution>1</c:Resolution>
      <c:Range name="valid">
        <c:LimitPair operator="AND" >
          <c:Limit comparator="GE"><c:Datum xsi:type="c:double"
          value="1000"></c:Datum></c:Limit>
          <c:Limit comparator="LE"><c:Datum xsi:type="c:double"
          value="1000000000"></c:Datum></c:Limit>
        </c:LimitPair>
      </c:Range>
    </c:Datum>
  </tr:Data>
</tr:Parameter>
```


The Test Results Document

```
<tr:TestResults xmlns:tr="urn:IEEE-1636.1:2011:01:TestResults" xmlns:c="urn:IEEE-1671:2010:Common" xmlns:sc="urn:IEEE-P1636.99:01:SimicaCommon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="urn:IEEE-1636.1:2011:01:TestResults TestResults.xsd" uuid="{589B634F-10F6-481e-AD22-7115020155BF}">
```

```
<tr:Personnel>
```

```
<tr:SystemOperator ID="jdoe1"/>
```

User

```
</tr:Personnel>
```

```
<tr:ResultSet ID="_f47ac10b-58cc-4372-a567-0e02b2c3d479" name="BERT.vi" startDateTime="2012-05-30T09:30:10">
```

```
<tr:Outcome value="Aborted"/>
```

```
<tr:Test ID="_6ba7b810-9dad-11d1-80b4-00c04fd430c8" name="my Space-to-Ground Link" startDateTime="
```

```
2012-05-30T09:30:18">
```

```
<tr:Parameters>
```

Read/write "configuration" variables

```
<tr:Outcome value="Aborted"/>
```

Outcome is always "Aborted"

```
<tr:TestResult ID="_1" name="bits">
```

```
<tr:TestResult ID="_2" name="errors">
```

```
<tr:TestResult ID="_3" name="BER">
```

Read-only "status" variables

```
<tr:TestResult ID="_4" name="Time to Acq">
```

```
<tr:TestResult ID="_5" name="Online">
```

```
<tr:TestResult ID="_6" name="Connected">
```

```
<tr:TestResult ID="_7" name="HW Status">
```

```
</tr:Test>
```

```
</tr:ResultSet>
```

```
<tr:TestProgram>
```

Software version

```
<c:Definition version="MyVersionOrRevisionNumber">
```

```
<c:Identification designator="My Operator- or Model-supplied identification of instance test-point, configuration
```

```
diagram designator, usage, or meaning-- assigned after start-up">
```

```
<c:Version>MyRevisionHistory</c:Version>
```

```
<c:ModelName>MyApplicationName</c:ModelName>
```

```
</c:Identification>
```

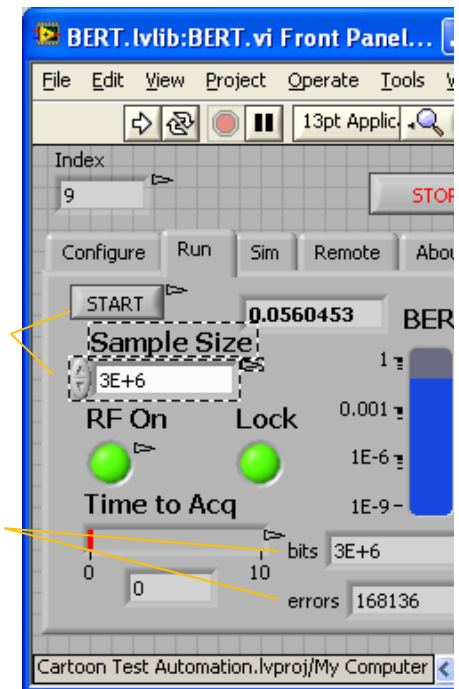
```
</c:Definition>
```

```
<c:SerialNumber>MyInstanceUUID</c:SerialNumber>
```

```
<c:ReleaseDate>2011-12-02</c:ReleaseDate>
```

```
</tr:TestProgram>
```

```
</tr:TestResults>
```



- Descriptions could be loaded into tr:TestResults

The Test Description Document

```
<?xml version="1.0" encoding="UTF-8"?>
<td:TestDescription uuid="f47ac10b-58cc-4372-a567-0e02b2c3d479" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xmlns:td="
urn:IEEE-1671.1:2009:TestDescription" xmlns:c="urn:IEEE-1671:2010:Common"
xsi:schemaLocation="urn:IEEE-1671.1:2009:TestDescription TestDescription.xsd">
  <td:UUT>
    <td:Description/>
  </td:UUT>
  <td:DetailedTestInformation>
    <td:EntryPoints/>
    <td:Actions>
    <td:TestGroups>
      <td:TestGroup xsi:type="td:TestGroupUnspecifiedOrder" name="none" ID="1">
        <td:Outcomes>
          <td:Outcome ID="1" value="Aborted"/>
        </td:Outcomes>
        <td:ParameterDescriptions>
        <td:TestResultDescriptions>
        <td>ActionReferences>
          <td>ActionReference actionID="0"/>
        </td>ActionReferences>
      </td:TestGroup>
    </td:TestGroups>
  </td:DetailedTestInformation>
</td:TestDescription>
```

- Static metadata is best loaded into tr:TestDescription

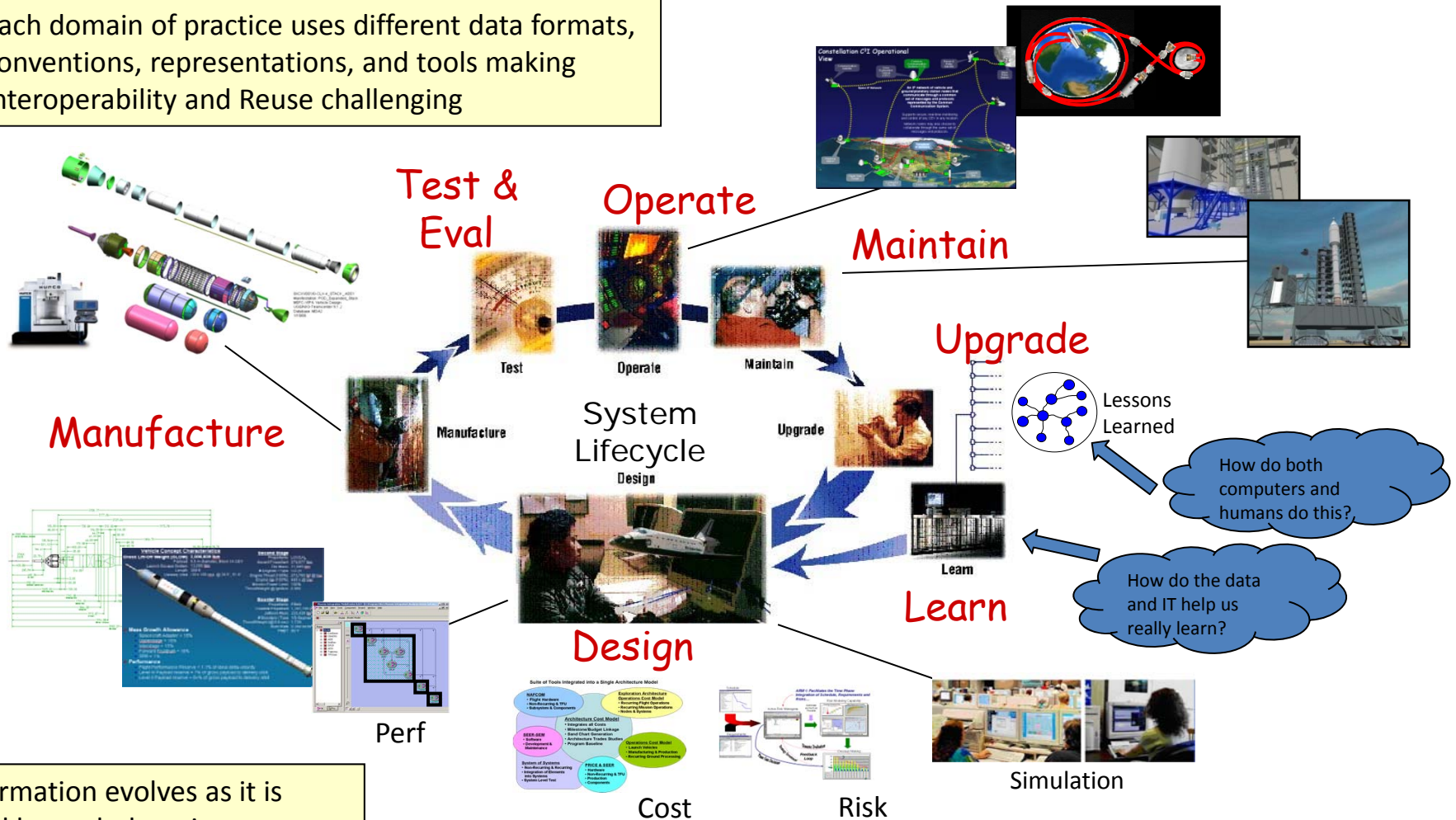
Future work: behaviors

Read/write “configuration” variables

Read-only “status” variables

The Mission Lifecycle

Each domain of practice uses different data formats, conventions, representations, and tools making Interoperability and Reuse challenging

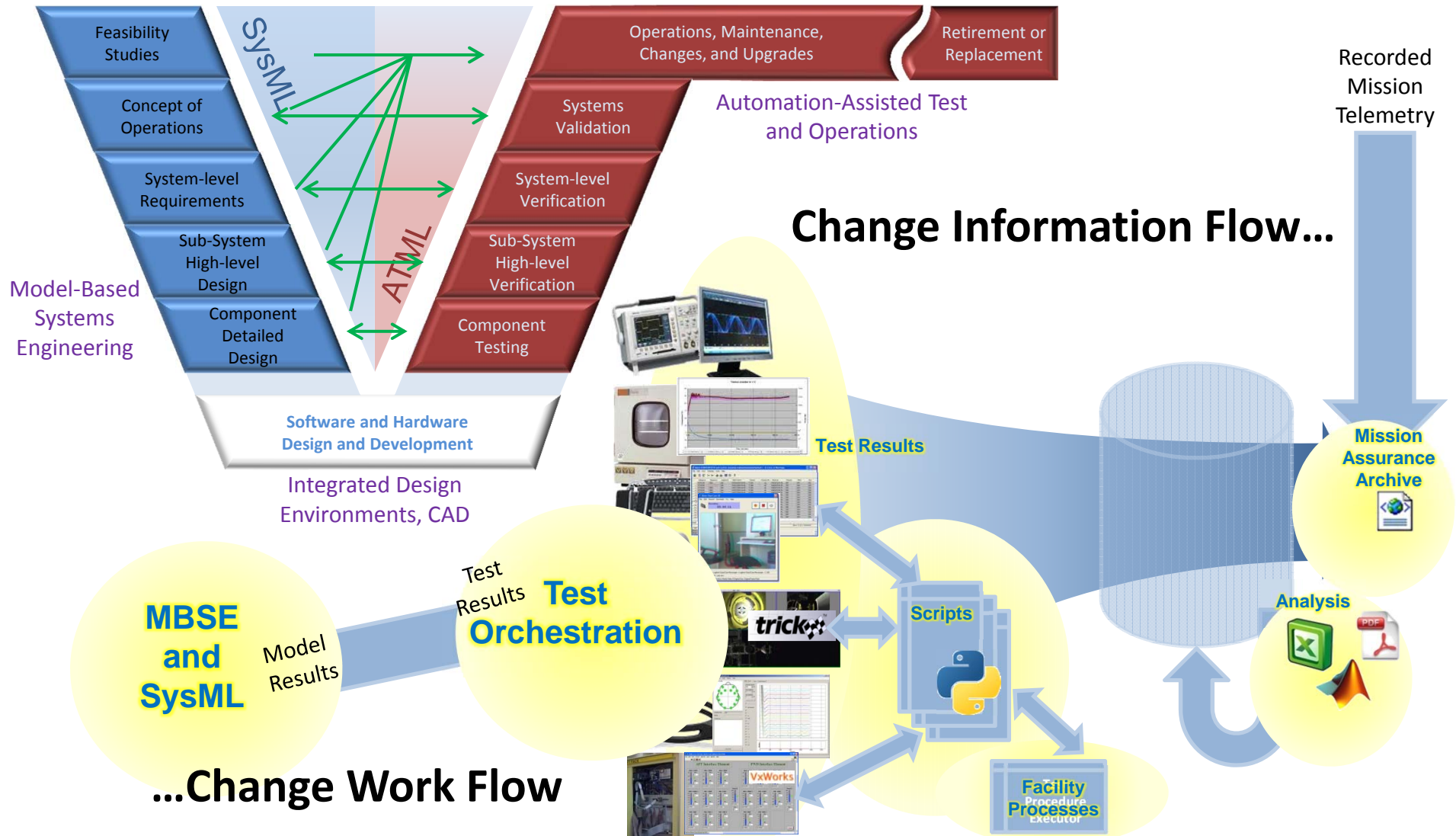


Information evolves as it is used by each domain

NASA Information Architecture

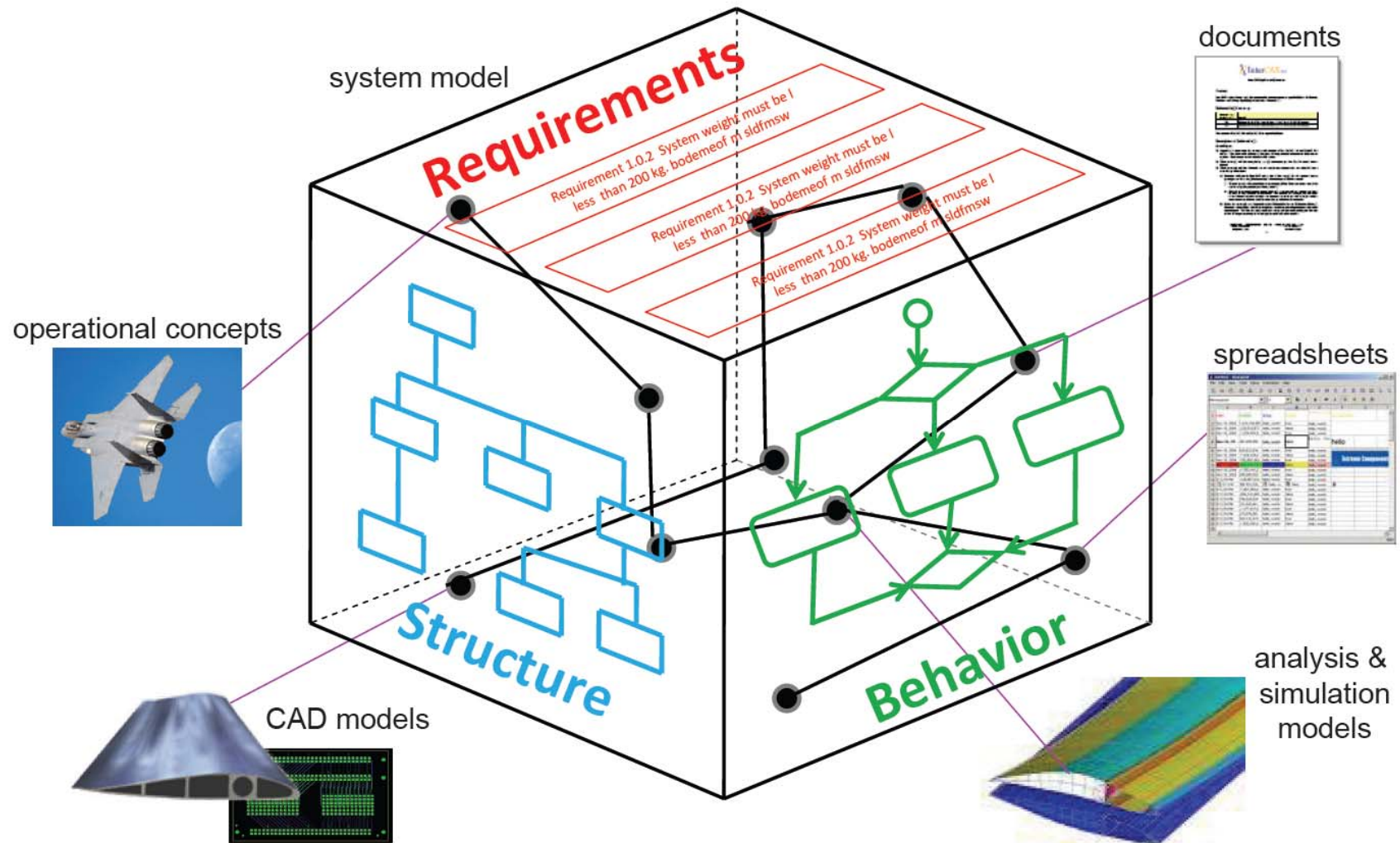


Test Orchestration and Data Harvest Complement MBSE

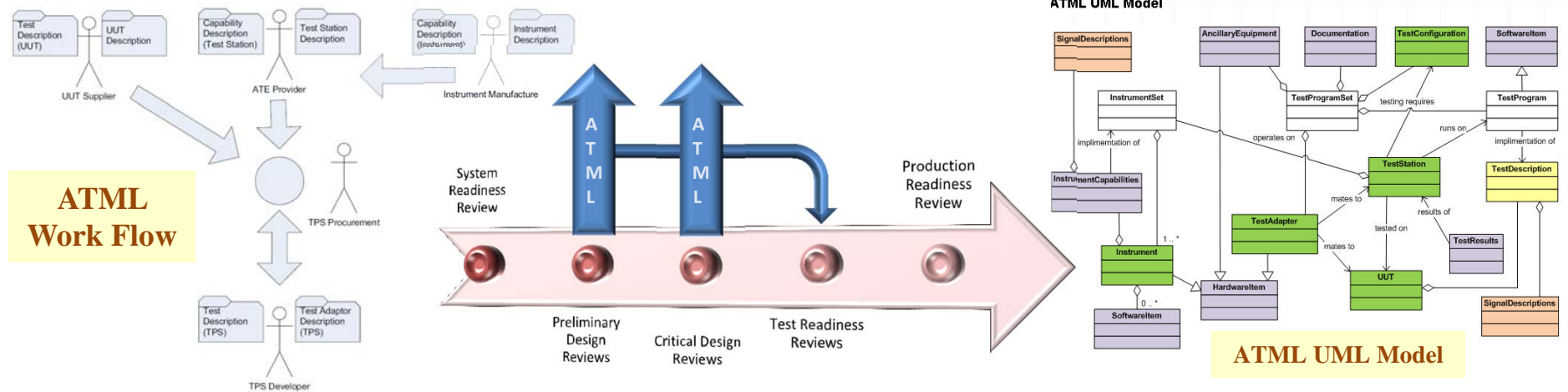


SysML: Maintaining Coherency

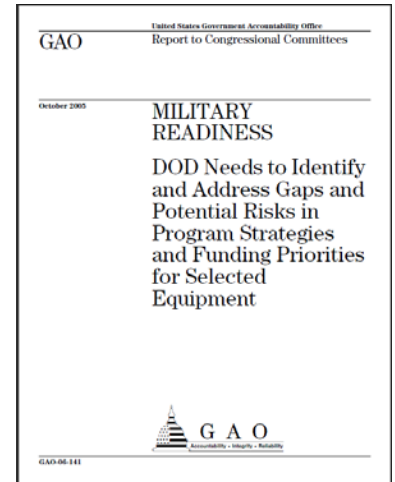
Fundamentally, a SysML model is used to generate a set of project documents that are maintained “in sync” with each other



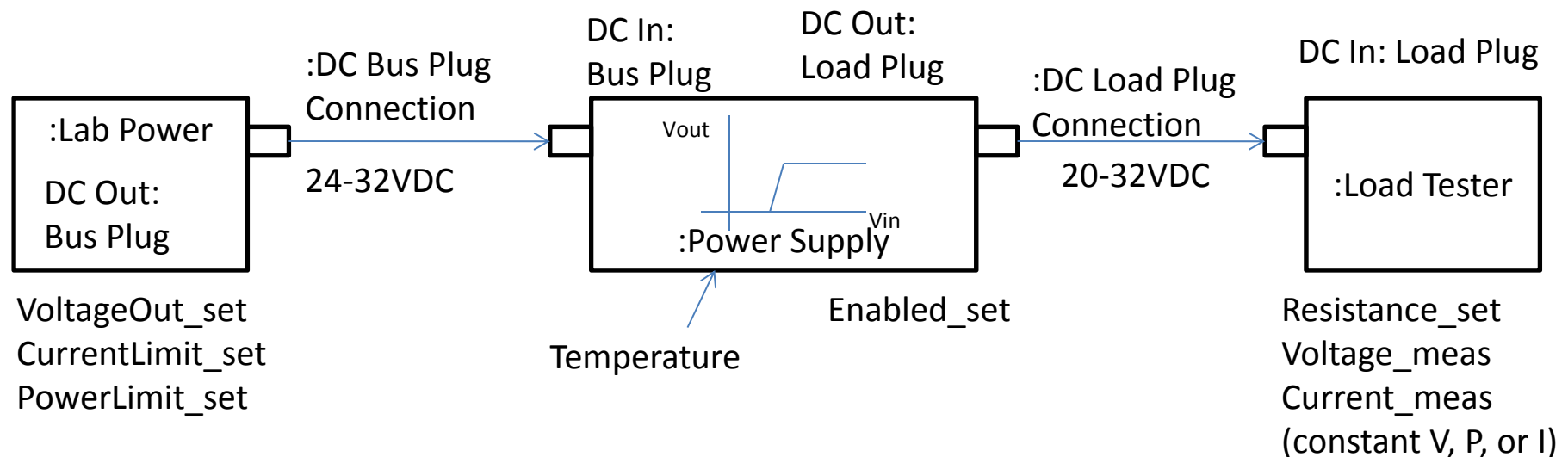
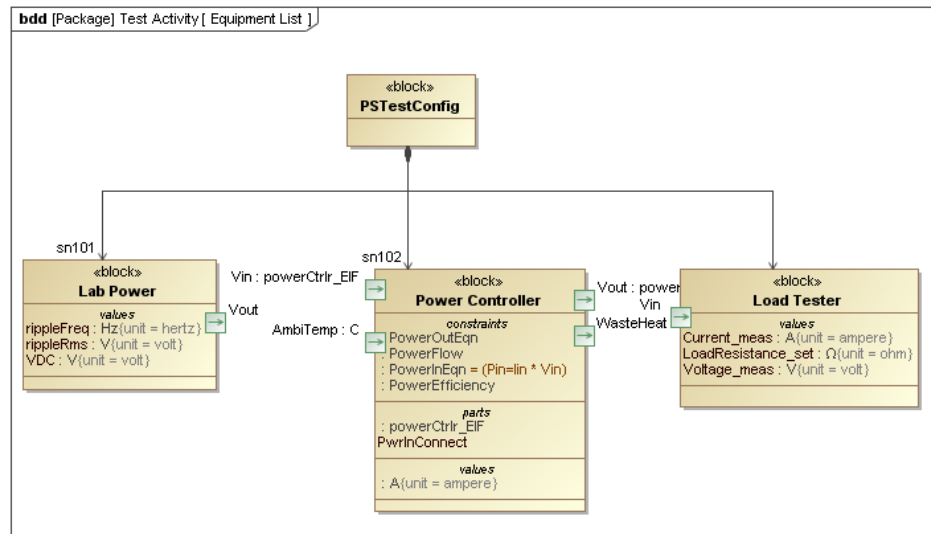
Define “Success” for ATML



- Test = Requirements + Capabilities + Wiring
- Maintain Documentation not Software
 - New Test Article, new Description
 - Replace an Instrument, Change a Description
 - Change Wiring, Update the Description
- At PDR, flag requirements that can't be tested on deployed test sets

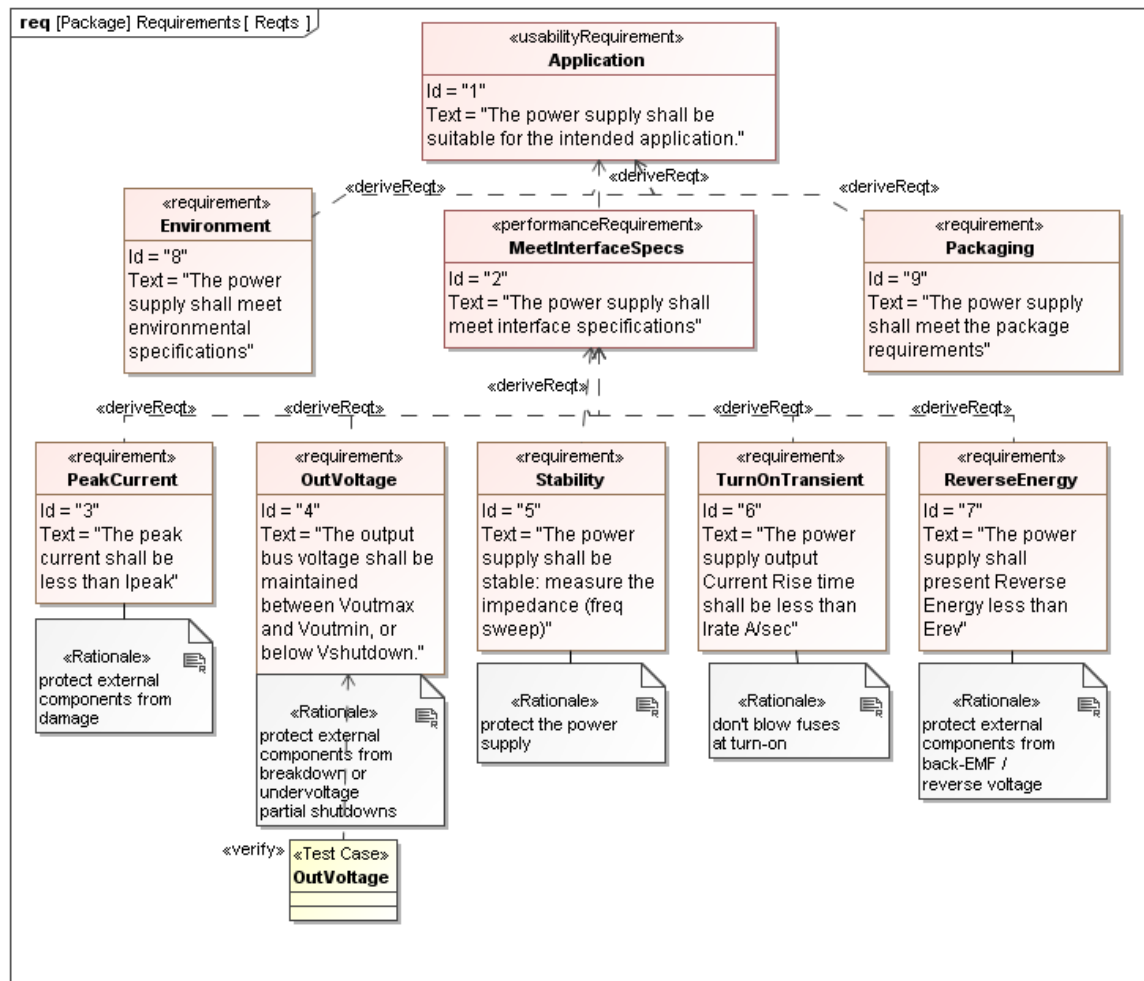


A Power Converter Test Configuration Interface Block Diagram



Questions: how to treat ports, vs. signals, vs. interfaces: which defines the connector type (dare I ask about gendered connectors), which the voltage. Want units attached where applicable so we can understand how they're being represented.

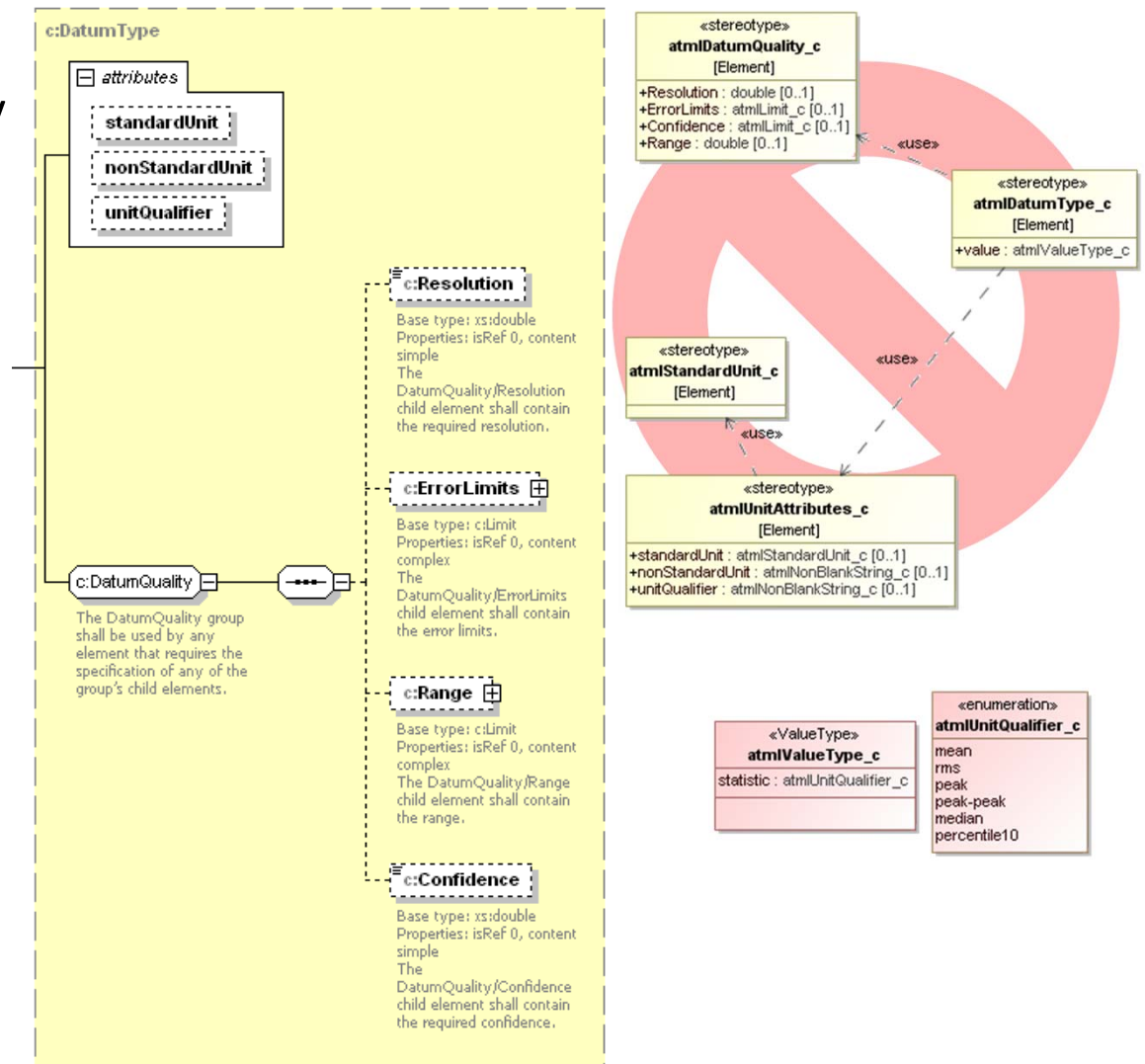
Requirements in SysML



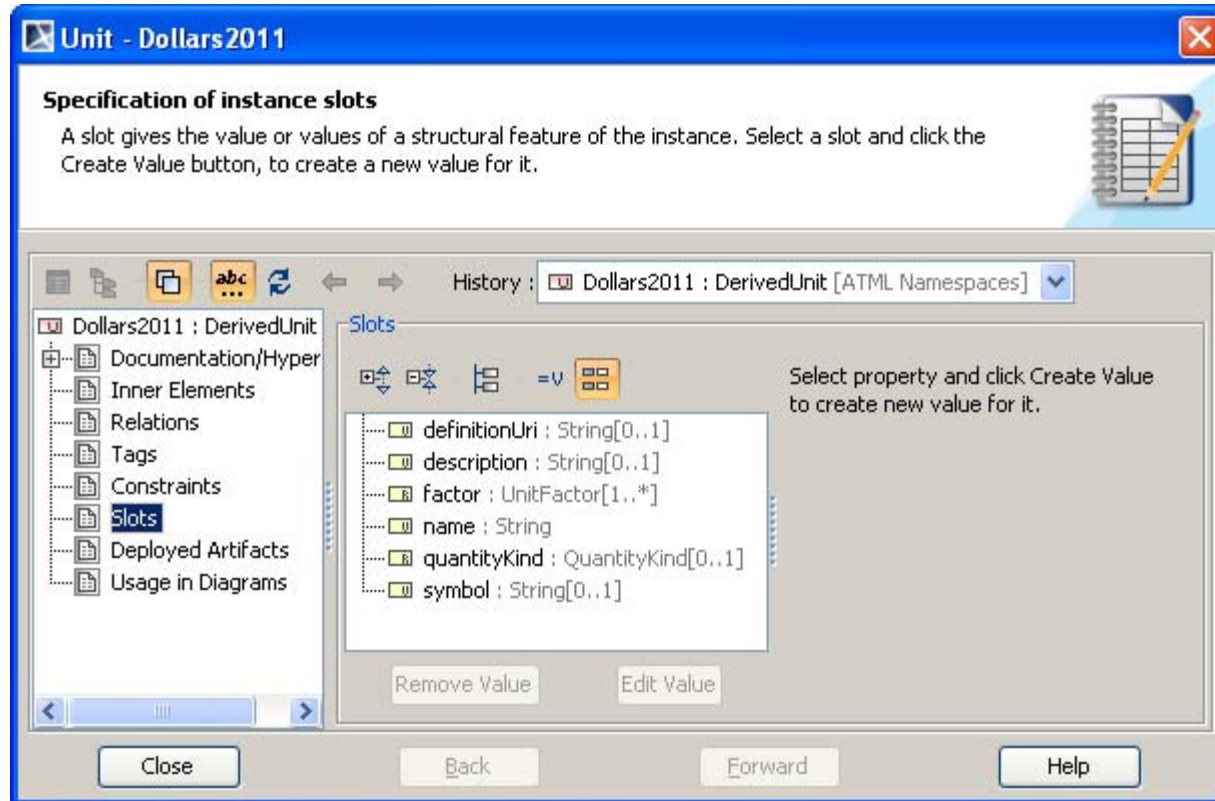
- Simplistic human-readable text representation
- Can be linked to the model
 - Model components can “satisfy” requirements
 - Test Cases can “verify” requirements
- MagicDraw plans to add support for SBVR

Quantities in ATML

- ATML provides an intriguingly rich mark-up for measured quantities
 - Standard units
 - Free-form units
 - Statistic (rms, peak...)
 - Resolution
 - Accuracy and confidence
 - Nominal Value
 - Acceptance Limits
 - Constraint Limits
- SysML uses OMG QUDV ontology
 - SI units only
 - Backed by an RDF/OWL knowledge model



QUDV representation for Units

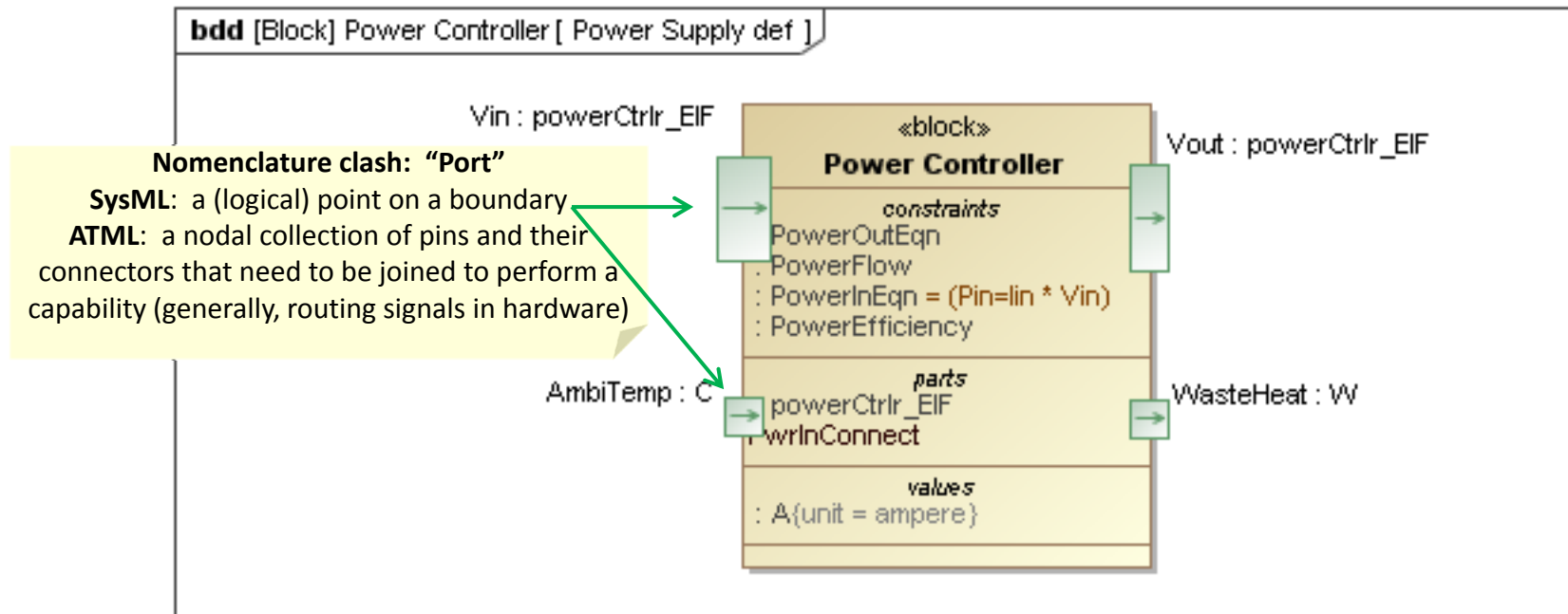


Defining a custom unit (\$US 2011) in MagicDraw

standardUnit filter in ATML

```
<xs:pattern value="(\+|\-)?\d+(\.\d*)?((E|e)(\+|\-)?\d+)? *((y|z|a|f|p|n|μ|u|m|c|d|h|k|M|G|T|P|E|Z|Y|
Ki|Mi|Gi|Ti|Pi|Ei)?(F|S|C|A|V|J|eV|T|N|Hz|lx|H|m|in|ft|mi|nmi|lm|cd|Wb|g|rad|deg|°|W|BW|Bm|P
a|bar|B(\.\d *m?W\))?)%|pc|decade|octave|Ohm|sr|kn|K|degC|°C|degF|°F|s|min|h|L|mol)\d*((\.\|/|)
(y|z|a|f|p|n|μ|u|m|c|d|h|k|M|G|T|P|E|Z|Y|Ki|Mi|Gi|Ti|Pi|Ei)?(F|S|C|A|V|J|eV|T|N|Hz|lx|H|m|in|ft
|mi|nmi|lm|cd|Wb|g|rad|deg|°|W|BW|Bm|Pa|bar|B|%|pc|decade|octave|Ohm|sr|kn|K|degC|°C|degF
|°F|s|min|h|L|mol)\d*)*)?" />
```

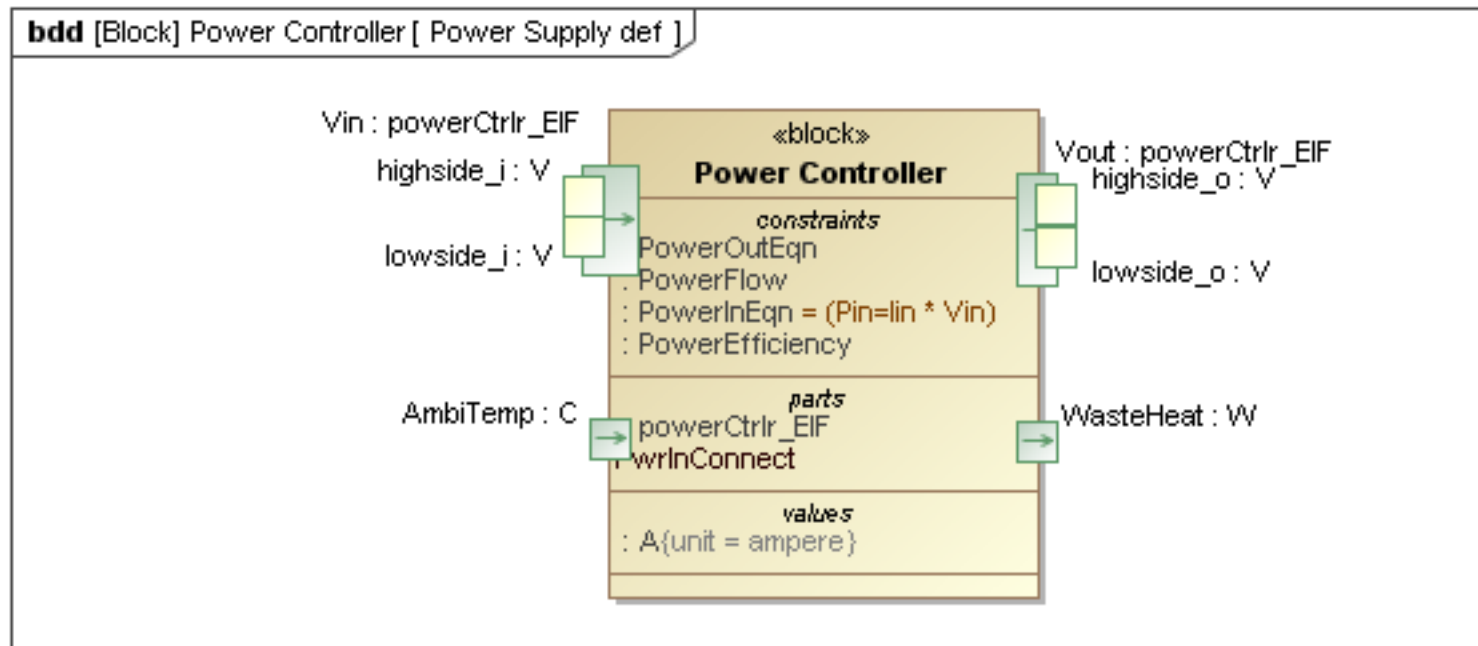
Pins, Ports, Connectors, Wiring



- Consider a power controller box...

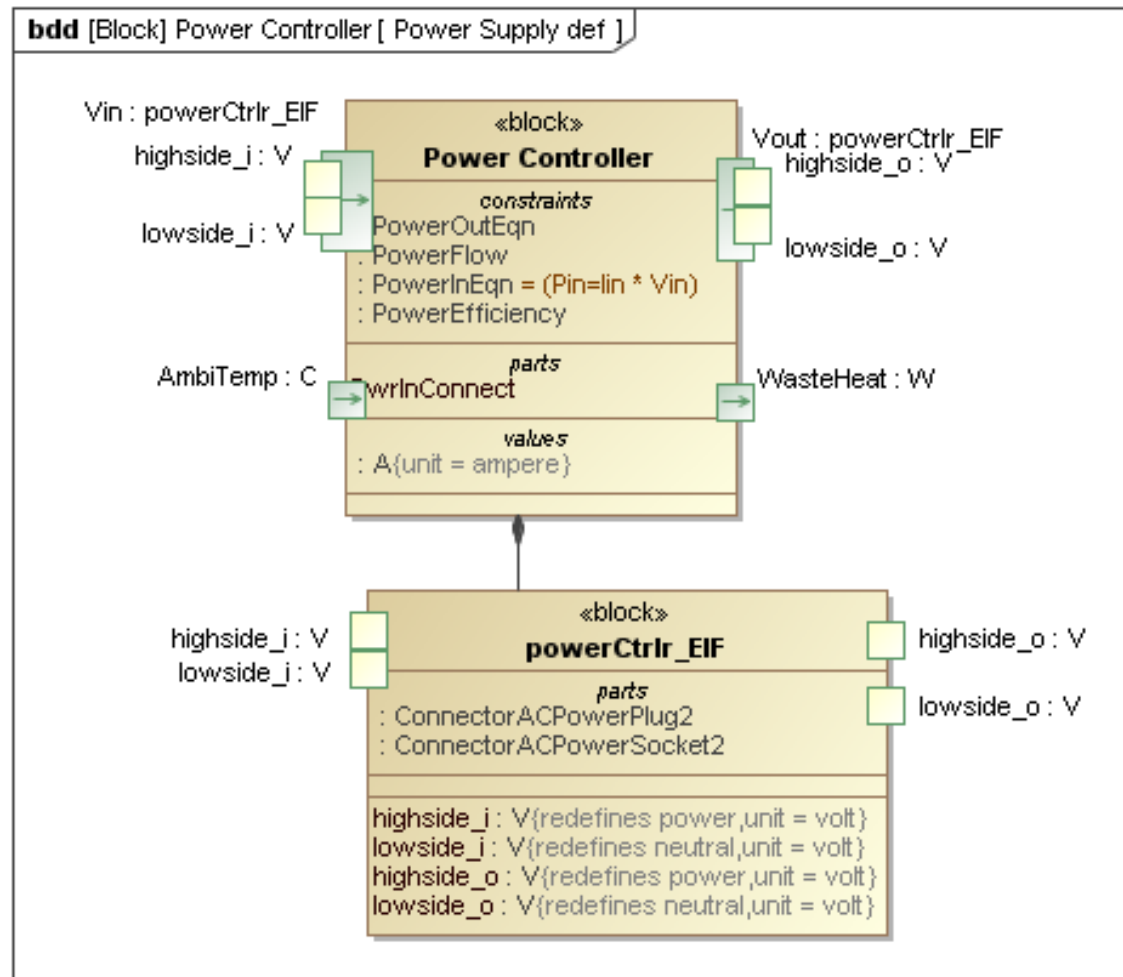
Note, **flow ports** may be deprecated in SysML

Pins, Ports, Connectors, Wiring



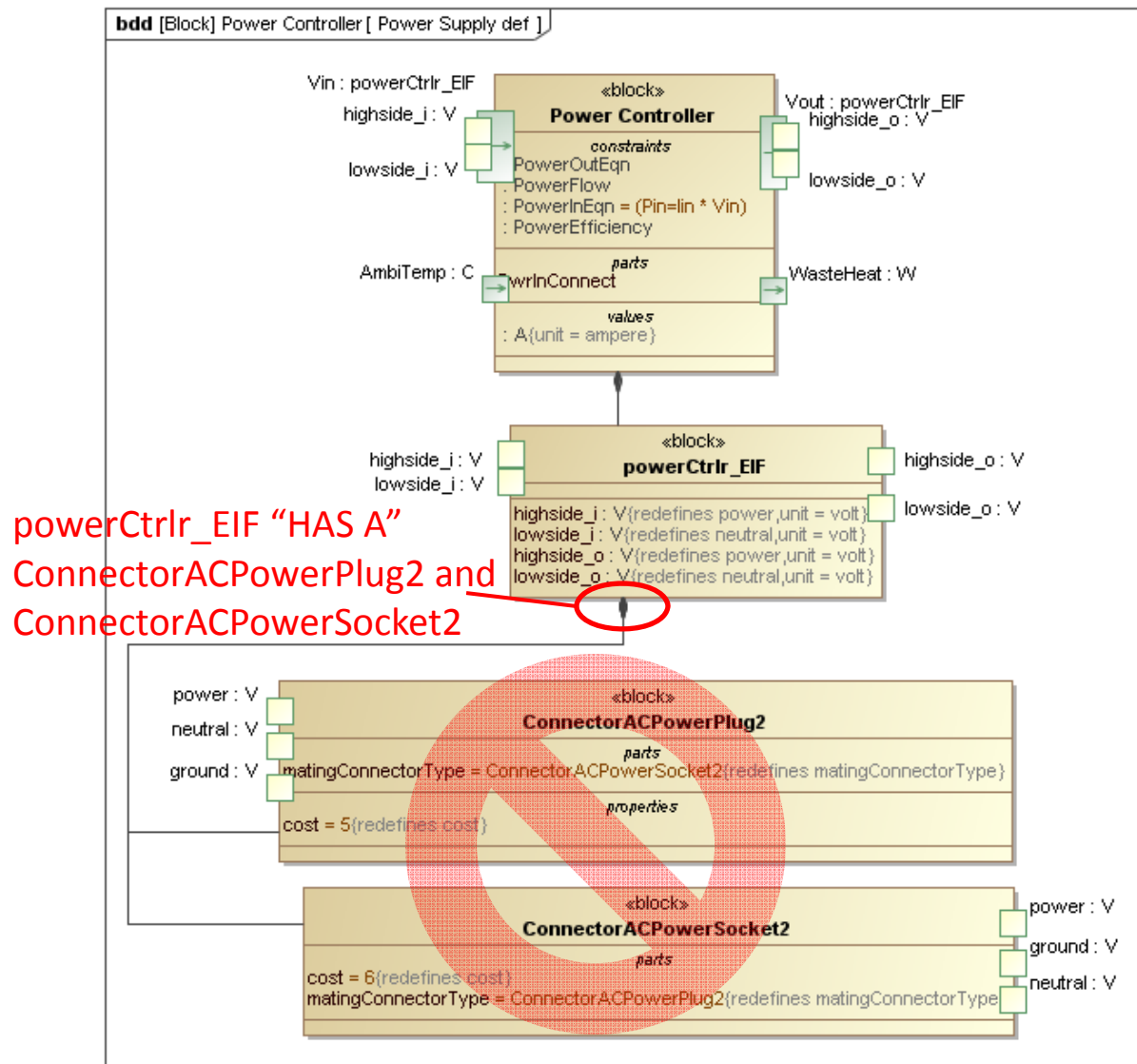
- Ports can be over-loaded to form something analogous to “pins.”

Pins, Ports, Connectors, Wiring



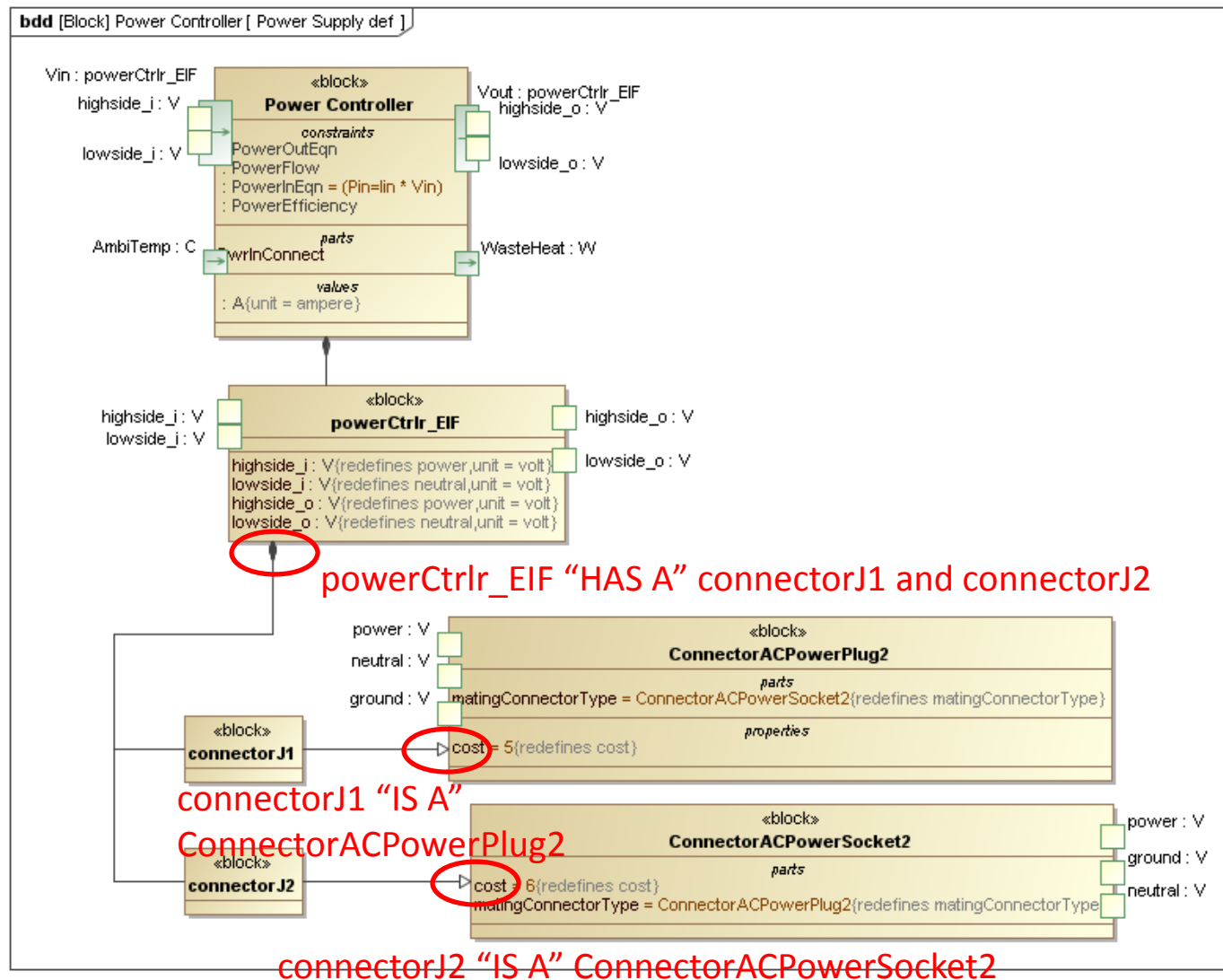
- JPL recommended adding an “Electrical Interface” component.
- The EIF allows the designer to include an electrical interface in the design, before connectors have been selected.

Pins, Ports, Connectors, Wiring



- Stock connectors can be pulled from a library, with inherited ATML metadata.
- Almost.
- The connectors can't be instances of the library parts, because the Power Controller isn't an instance.

Pins, Ports, Connectors, Wiring



- Instead, the power controller needs its own connectors, inheriting from the library parts.
- Property **redefinitions** are used liberally throughout.

A Better Place to Start

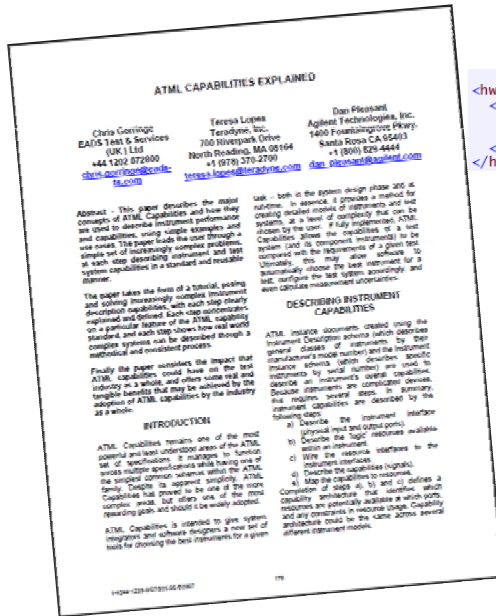


Figure 7 – Capability Description of a SineWave

```
<hw:Capability name="sinewave">
  <hw:Interface>
    <c:Ports>
      <c:Port name="sinewave" />
    </c:Ports>
  </hw:Interface>
  <hw:SignalDescription xmlns:std="http://www.ieee.org/1641">
    <std:Signal name="sinewavesignal" out="sinewave">
      <std:Sinusoid
        name="sinewave"
        frequency="10kHz range 1kHz to 10MHz err1mt 0.1Hz res 1Hz"
        amplitude="rms 1uV range 1uV to 1V err1mt 0.1% range 1V to 10V err1mt 1%"/>
    </std:Signal>
  </hw:SignalDescription>
</hw:Capability>
```

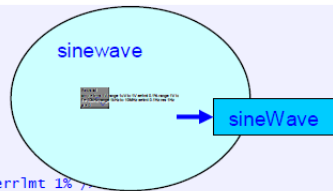


Figure 8 – Capability Description of an RMS Measurement

```
<hw:capability name="measRMS">
  <hw:Interface>
    <c:Ports>
      <c:Port name="Input" />
    </c:Ports>
  </hw:Interface>
  <hw:SignalDescription xmlns:std="http://www.ieee.org/1641">
    <std:Signal name="RMSsignal" in="Input" out="rmsMeas">
      <std:RMS
        name="rmsMeas"
        in="Input"
        nominal="rms range 1uV to 10V err1mt 0.1% range 10V to 150V err1mt 1%"/>
    </std:Signal>
  </hw:SignalDescription>
</hw:Capability>
```

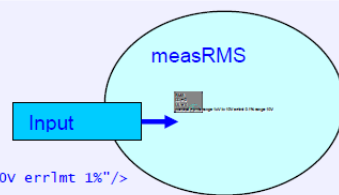


Figure 1 – Instrument Definition Interface

```
<hw:Interface>
  <c:Ports>
    <c:Port name="Front" />
    <c:Port name="Back" />
  </c:Ports>
</hw:Interface>
```

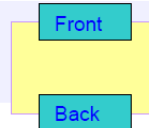


Figure 2 – Resource Definition

```
<Resources>
  <hw:Resource name="Resource_1">
    <hw:Interface>
      <c:Ports>
        <c:Port name="P1" />
        <c:Port name="P2" />
      </c:Ports>
    </hw:Interface>
  </hw:Resource>
  <hw:Resource name="Resource_2">
    <hw:Interface>
      <c:Ports>
        <c:Port name="P1" />
        <c:Port name="P2" />
      </c:Ports>
    </hw:Interface>
  </hw:Resource>
</Resources>
```

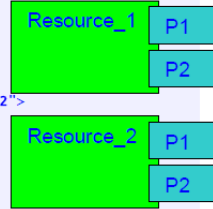


Figure 4 – Wire resources' ports to instrument ports (Diagram)

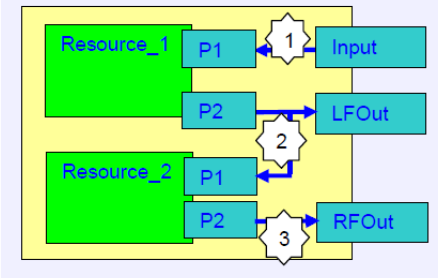
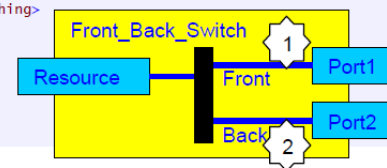


Figure 6 – Switch Definition

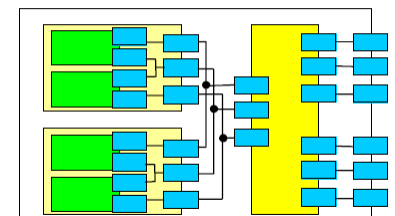
```
<Switching>
  <hw:Switch name="Front_Back_Switch">
    <hw:Interface>
      <c:Ports>
        <c:Port name="Port1" />
        <c:Port name="Port2" />
      </c:Ports>
    </hw:Interface>
    <hw:Connections>
      1 <hw:RelaySetting name="Front">
        <hw:RelayConnection from="Port1" to="Resource" />
      </hw:RelaySetting>
      2 <hw:RelaySetting name="Back">
        <hw:RelayConnection from="Port2" to="Resource" />
      </hw:RelaySetting>
    </hw:Connections>
  </hw:Switch>
</Switching>
```



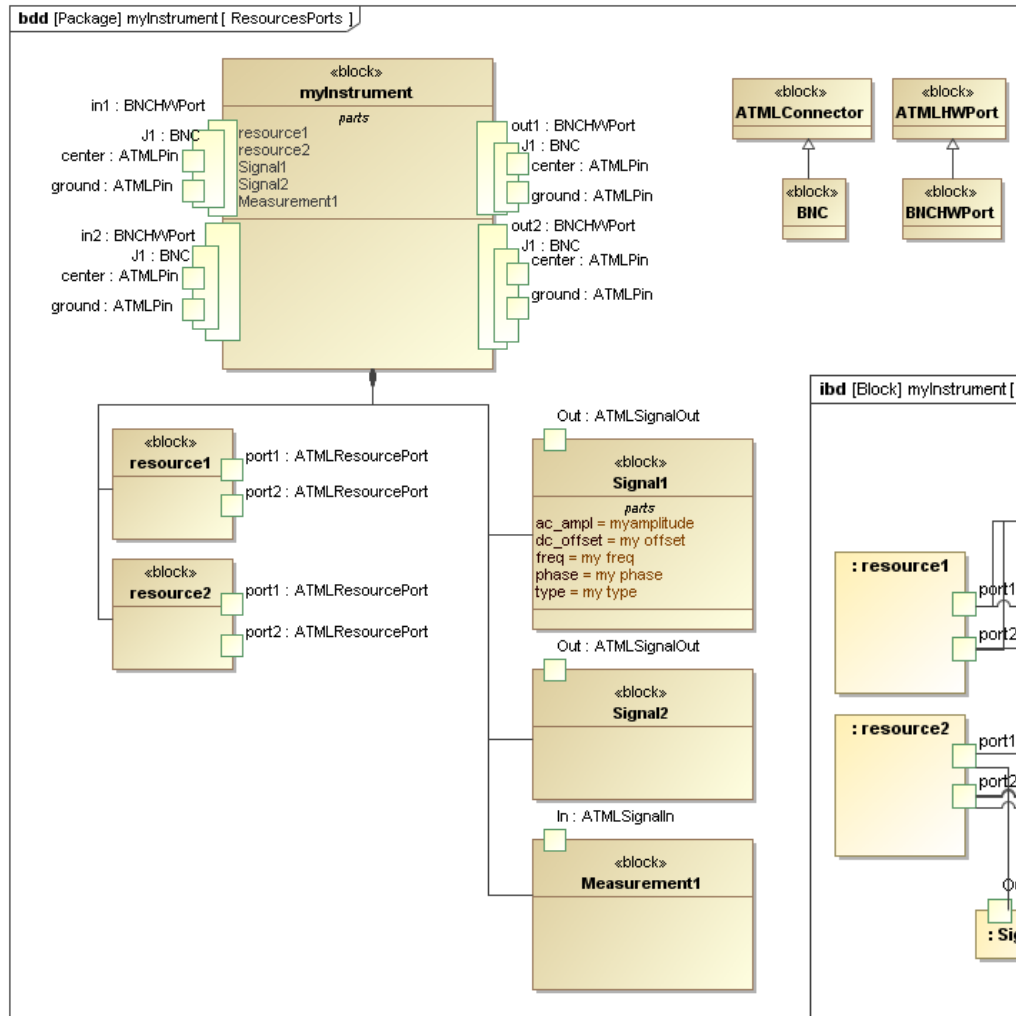
A Tutorial,
How to
model:

- Interfaces
- Ports
- Resources
- Switches
- Capabilities
- Signals

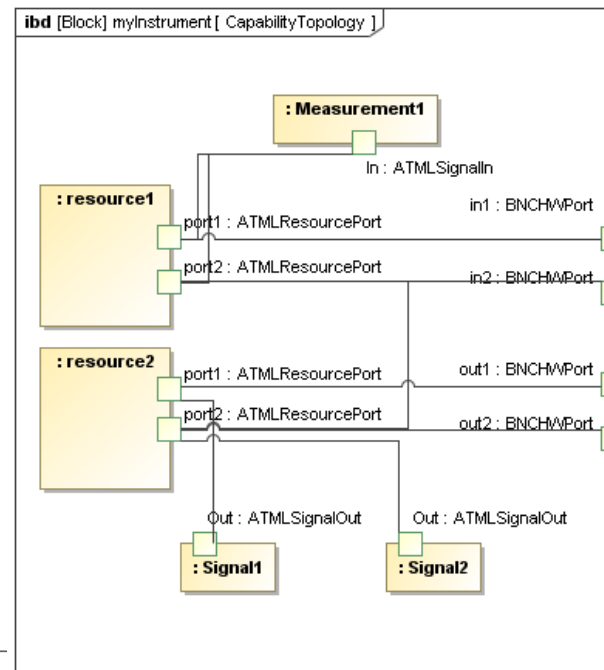
Figure 18 – Derived System Capabilities



ATML Ports, Resources, Capabilities...

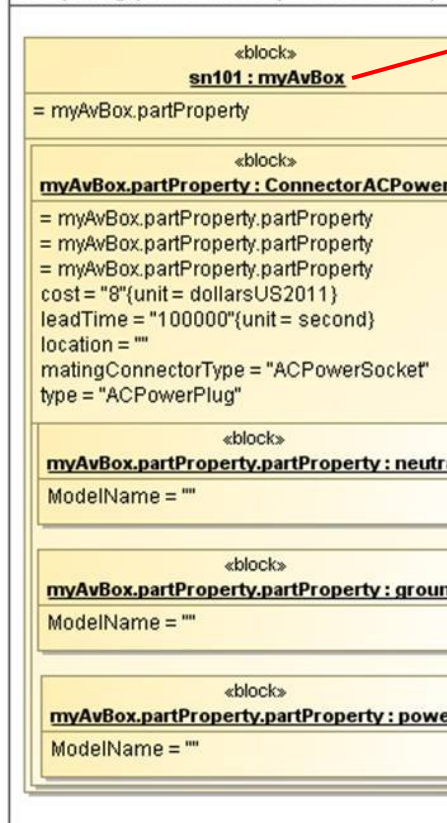


- Didn't try switches
- Two ways to "wire:" instances in an IBD, vs. port redefinition in a BDD



The Harvest

bdd [Package] sn101 Instance [Instance of the myA



```
<xmi:XMI>
<xmi:Documentation> ... </xmi:Documentation>
<uml:Model> ... </uml:Model>
<xmi:Extension> ... <xmi:Extension>
<xmi:Extension> ... <xmi:Extension>
<xmi:Extension> ... <xmi:Extension>
<xmi:Extension> ... <xmi:Extension>
<xmi:Extension> ... <xmi:Extension>
<sysml:Block> ... <sysml:Block>
<sysml:Block> ... <sysml:Block>
<sysml:Block> ... <sysml:Block>
<sysml:Block> ... <sysml:Block>
</xmi:XMI>
```

- Model saved in
OMG XML
Metadata
Interchange
(XMI)

```
<packagedElement xmi:type='uml:Class' xmi:id='_17_0_2_ecd035c_1314997189054_814634_12767'
name='ConnectorACPowerPlug'> .... </packagedElement>
:
<packagedElement xmi:type='uml:Class' xmi:id='_17_0_2_ecd035c_1314998427607_193279_13954'
name='myAvBox'>
  <ownedAttribute xmi:type='uml:Property'
xmi:id='_17_0_2_ecd035c_1314998668035_133257_14285'
  type='_17_0_2_ecd035c_1314997189054_814634_12767'/>
  :
</packagedElement>
```


Conclusions for SysML Interoperability

- SysML models will need to be derived from libraries of ATML constructs (abstract not literal representation) so that information can be extracted by data-driven algorithm.
- Neither SysML nor ATML has a knowledge model, although SysML Quantities do.
- Using redefinition to assign values to properties inherited from library “ATML” types was difficult.
- Two techniques for “wiring,” best not selected