RAI Systems Engineering/Analysis

Dr. Ronald C. Salley and Hugh A. Pritchett

Purpose:

This paper describes the ATS (Automatic Test System) systems analysis results that Dr. Ronald C. Salley and Hugh A. Pritchett, henceforth called the investigators, produced relating to the Resource Adapter Interface (RAI) ATS Framework element. The systems analysis was performed as a result of two Small Business Innovative Research (SBIR) grants, one from NAVAIR (N00-102) and one from the Air Force ATS program management office LEEA (AF04-272). Although this paper does not reflect all products of the SBIRs, it clearly and definitively exposes the information needed to attain the Holy Grail in test, *TEST PLATFORM INDEPENDENT TEST PROGRAMS*!

Foundation:

The investigators scientifically delineated and defined a platform independent RAI test paradigm by applying

- systems engineering processes,
- a proprietary underlying system theory, i.e., the *Theory of Real Systems*, and
- mathematical and computer science modeling techniques.

The RAI paradigm is based on a rigorously defined hierarchy of primitive data structures that allow a clear and complete visualization of

- the platform dependence of traditional test programs and
- the platform independence of RAI test programs.

The RAI paradigm

- unequivocally identifies all aspects of traditional test-program participation in platform dependence;
- represents all test requirements as (platform independent) data structures;
- enforces platform independence when used exclusively to express test requirements in test programs;
- allows visualization and containment of test requirements that are not available in traditional test programs; and
- allows test programmers a clear way to define what needs to occur without functional language calls and implied timing encumbrances that are demanded by traditional test programming.

RAI test programs

- define platform independent test-requirements,
- pass these requirements through an RAI to a resource manager, and otherwise
- provide only diagnostic guidance and sequencing.

See Figure 1.

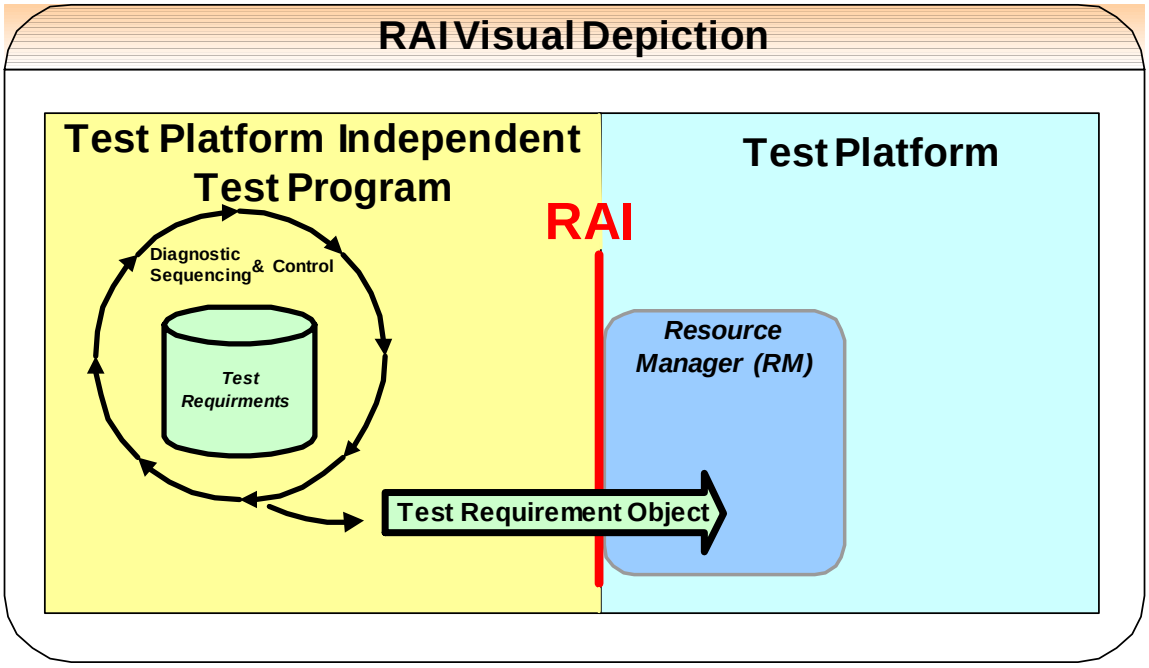

Figure 1

Definitions:

- A *capability* is a data object that is a *signal*, a *location*, and *timing*, where
  - a *signal* is an IEEE – 1641 signal,
  - a *location* is a point on the UUT, and
  - *timing* is a delay (with a precision) and a maximum lifetime of the signal.

  See Figure 2.
- A *test requirement* is a tree-structured data object whose nodes are capabilities that are related to their parents by the timing of their signals (which is the timing defined for the capability, above). See Figure 3.

  A test requirement is not a definition of a test. A test requirement is a definition of a requirement for a test. A test requirement is not executable. A test requirement is a data object. A test requirement is in a form that is directly definable by a data-definition language such as XML.
- A *Resource Adapter Interface* (***RAI***) is a boundary, through which test-requirement data objects pass, that provides platform independence for test application software.

Explanation:

Traditional ATS involves understanding UUT Test Requirements (UTRs) and manifesting them using test station assets and test languages. The problem has been that the concept of the UTR is nebulous in traditional test languages. Traditional languages disburse the UTR concept among language functionality and data definition making it a puzzle that must be discerned rather than an entity that is distinctly visible.

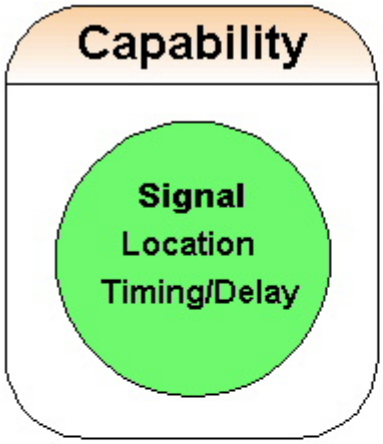

Figure 2

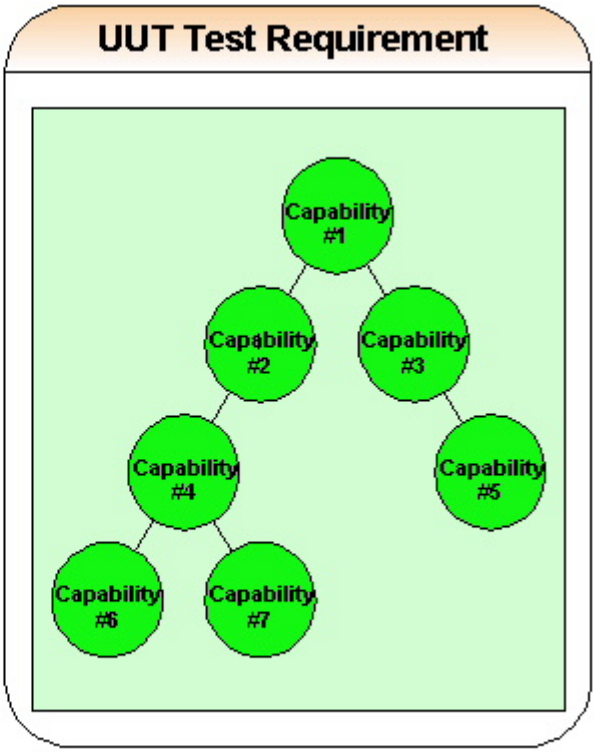

Figure 3

The ATLAS language attempts to provide test-platform independent UTRs using signals. The concept of signals is a good first step in that it showed a need and an approach. But traditional test languages do not achieved platform independent test programs. This white paper explains why traditional languages cannot be platform independent and provides a definition of how platform independence can be achieved.

UUT Test Requirements (UTRs) general discussion

UTRs are almost never single signals. UTRs are almost always multiple signals working in concert at different times and different locations (UUT connection points). The investigators reviewed several TPS migration projects in the course of their analysis and during this study found one pervasive aspect of traditional test languages, like ATLAS, that makes them unsuitable for providing complete platform independent test programs. Traditional languages define ***how*** signals work in concert by using executable language statements to explicitly sequence test platform assets (even though the assets might be unknown)… a major failing for test platform independence. Traditional languages may not, but often do, control ***how*** individual instruments on a test platform perform but traditional languages do control ***how*** test platforms perform in general.

Traditional languages may be somewhat instrument independent but are not platform independent. Traditional developers must understand ***how*** test platforms perform generically and, even then, their programs must still be modified when ported across test platforms.

The RAI research and analysis defines a cohesive scientific definition for a UTR, which is a definition for ***what*** the concert of signals is, not a definition of ***how*** the concert of signals is performed. No matter ***how*** test platforms perform, RAI test programs are completely portable…

as long as target test platforms can perform ***what*** is desired, i.e., as long as target test platforms can meet the UTR.

Capabilities

The term capability is traditionally used without formal definition. The RAI research and analysis lead to the formal definition, above, that says a capability is

- any service, i.e., signal, that a UUT might require,
- a location (usually a UUT pin) where the service is to be provided, and
- a time and maximum duration for which the service is to be provided.

See Figure 2.

Capability was defined with the realization that platform independent test programs would need to provide test platform capabilities in terms of UTRs and not in platform related ways. It is not coincidental that the ATLAS concept of signal is present in the RAI capability. The ATLAS concept of a signal is what gives ATLAS its attempted instrument independence but not platform independence. In furtherance of platform independence, a signal must be defined in concert with UUT locations and timing of other signals.

UUT Test Requirements (***UTRs***)

The definition for capability derived from the RAI research and analysis allows individual signals but a single signal alone is rarely sufficient for a complete coherent UTR. Capabilities must be packaged to act in concert with each other. Using systems engineering analysis, mathematical set theory and the *Theory of Real Systems*, the generic format of this package was found to be that defined for the UTR, see Figure 3. The format allows test programs to create UTRs for any test scenario and to present the UTRs to the RAI as an unambiguous platform independent data object.

Resource Adapter Interface (***RAI***)

The RAI itself is an almost innocuous entity with one method that has one argument, a UTR. Test programs call this one method to deliver the UTR to a resource manager. The resource manager sets up the test platform according to the UTR and causes the test platform to perform the test, without test program intervention. The method returns a test result which is no more than a copy of the executed UTR, with measured values fulfilled.

Various companies have referred to similar non-RAI methods by various names. A common name, represented here as an RAI method, is

```
Test_Result =  Get_Capability(UUT_Test_Requirement);
```

The investigators believe a more descriptive name would be

```
Test_Result =  Render_UTR(UUT_Test_Requirement);
```

Of course, if the assignment operator were overloaded, naming the method would be moot, e.g.,

```
Test_Result =  UUT_Test_Requirement
```

In any case, the questions is “can these scientifically based concepts be implemented”? The answer is, “yes.” The next section shows one possible implementation in C++. The C++ representation is a complete model in code of what was defined in words in sections above. Together they completely document the investigators’ findings. This implementation is provided only as a vehicle to easily convey the findings.

C++ Representations

Capability:

```
struct Capability
{
   1641_Signal*   m_pSignal;
   String         m_Location;
   Timing         m_Timing;
};
```

where `m_pSignal` is a IEEE-1641 signal with range, resolution and accuracy.

where `m_Location` is a string indicating UUT connection location as required of the UTR.

where `m_Timing` is:

```
struct Timing
{
     Double m_Delay ;
     double Life_Time;
};
```

where "`Double`" is a regular "`double`" with range, resolution and accuracy inherent.

Test_requirement:

```
struct Test_Requirement: Capability
{
   string             m_ID;
   Test_Requirement   m_Test_Req[];
};

typedef Test_Requirement UUT_Test_Requirement;
```

These clearly defined and now publicly documented definitions enable UTRs to be defined in clear, visualizeable, and most importantly hardware independent terms. These definitions are refined, complete, and conceptually simple. They are the only implementation known that exclusively uses data to express UTRs. No language function that imposes timing and general systems knowledge is required. Making this the only 100% RAI definition ever achieved. The constructs allow visualization of UTRs similar to that shown in Figure 3. The graphic depiction allows test programmers to clearly visualize UTRs and to better understand them as they are developed. The investigators have performed extensive use case application against the definitions and the definitions have been repeatedly proven valid.

The visualization (see Figures 2 and 3) afforded by these definitions of capability, test requirement, and UTR extends to parallel testing. A test requirement is a capability and an array

of test requirements. If a test requirement's capability is null, the test requirement's array of test requirements are requirements for parallel tests.

Conclusion:

This paper has presented the investigators' system analysis for RAI related research. The paper has supplied the only clear and unambiguous way of defining UTRs as data only, thereby enabling platform independent test programs. The scientifically based primitives provided in this paper can support any UTR and can therefore be used to implement any test requirement including parallel test requirements. The definitions are universally applicable, easily understood, and easily applied. The products are a result of a scientific analysis and engineering discipline that have been applied with a real world modeling paradigm and set of principals. The definitions support all test scenarios and solve the platform dependence problem of traditional test programs.

Challenge:

DoD and industry realize that platform dependent test programs are a problem that needs to be resolved. Traditional approaches to resolving the problem have been iterative, based on existing concepts/products, and produced incomplete solutions. Solving problems often means stepping outside the box. This paper documents the results of a systematic scientific approach that stepped outside the box to resolve the problem of platform dependent test programs.

Significant resistance to this solution has already come from some who are stakeholders in existing technologies. A common affront to the solution is that it is not proven and is not pragmatic. The testing community is challenged to analyze the solution and enumerate any perceived deficiencies and/or shortcomings that would make it fail as a basis for test platform independent test programs.