Testability of Product Data Management Interfaces

David Flater and KC Morris

National Institute of Standards and Technology

U.S.A.

1999-12-16

Abstract

The Testability of Interaction-Driven Manufacturing Systems project seeks to enhance the design-for-testability of specifications for manufacturing software interfaces, derive a test method that is usable for interaction-driven manufacturing systems in general, and foster the reuse of testing artifacts. For our first testability study we constructed some prototype conformance and interoperability tests for the Product Data Management Enablers standard from the Object Management Group. We reused test data developed for the Product Data Management Schema, a developing standard based on ISO 10303 (informally known as the Standard for Exchange of Product model data), and enumerated the lessons learned for testing and testability. We plan to reuse some of our new testing artifacts for testing an ISO 10303 Standard Data Access Interface to data based on the Product Data Management Schema.

(Keywords: interface, OMG, PDES, product data management, STEP, testability, testing)

1. Introduction

Integration technologies such as the Common Object Request Broker Architecture (CORBA)[1] and the Component Object Model[*] (COM)[2] have changed the way that software systems for manufacturing and other domains are built. Components that were originally deployed in different places and times are now being wrapped with standard interfaces and made to interact with one another. This practice has created a new category of problems for software testers, who must not only find component faults, but also integration faults such as unintended interactions between components and misunderstood interface semantics.

The Testability of Interaction-Driven Manufacturing Systems (TIMS) project[3] in the Manufacturing Engineering Laboratory of the National Institute of Standards and Technology is seeking to develop a capability for testing in this new environment and to collect requirements for testability that can be designed into future specifications. Product Data Management (PDM) is now a focus for standardization in the manufacturing domain and promises to present some of the most important challenges in testing and testability in years to come. For this reason we made it a primary target for TIMS attempts to "learn by doing" by developing test methods and extracting the requirements for testability.

Our testability study helped us to identify a variety of features that affect the testability of a specification. Those features are introduced in the next section to provide context for the specific examples that will be discussed later.

---

[*] Commercial equipment and materials are identified in order to describe certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

Our work focuses on two significantly different, yet conceptually related PDM standards: version 1.1 of the Object Management Group's (OMG)[4] PDM Enablers (PDME) specification,[5] and the PDM Schema[6] based on ISO 10303 the Standard for the Exchange of Product Model Data (STEP)[7] using STEP's Standard Data Access Interface (SDAI).[8] These are both specifications for application interfaces to PDM systems, but they are from different sources and designed for different purposes. Our goal is to find methods that exploit the conceptual commonality, yet remain adaptable to the very different environments in which the two specifications are defined. The sections following the inventory of testability features detail our approach, the specific testability issues that we encountered so far, and lessons learned for the testing approach itself.

2. Inventory of testability features

2.1. Application interface specification

An obvious prerequisite for creating a test suite to exercise a given system is the availability of a well-specified interface. Systems that implement a standard application interface are inherently more testable than systems that only make their state visible by graphical user interfaces (GUIs) and exchange files because the test client can interact more directly with the Implementation Under Test (IUT).

For conformance testing of a given interface, there is always the risk that correct behavior is faked somehow, or that the semantics of the interface are not correctly realized inside of the IUT even though the correct data are returned. This is a philosophical issue of when do we find a test result to be credible. If multiple application interfaces to a system are specified, we can increase our confidence by testing over all interfaces. On the other hand, if the alternate interface is a GUI, we must choose between a strict but subjective testing approach that looks at the GUI, or a quantitative but lax approach that does not. While purists would restrict the scope of testing to exactly those features that are included in the standard interface specification, pragmatists would balk at test results that disagree with the subjective evidence presented by the GUI.

2.2. Implementability of the specification

Implementability is the software analog of the concept of manufacturability.[*] It is possible to construct well-formed specifications that are not implementable; that is, some practical consideration, rather than internal inconsistency of the specification, prevents a conforming IUT from being built. When this happens, the IUT and the tester must adopt some nonstandard workaround. The validity of the tests remains questionable until the specification is fixed.

2.3. Functional specification

By "functional specification" we mean the specification of the results of functions and operations supplied by the IUT. Usually these results are dependent on the inputs and the current state, but not always.

If the functional specification is incomplete, then formal testing is impossible because we have no standard of correctness. The best we could do is rely on human judgment to decide whether the results look "reasonable."

2.4. Usage specification

A usage specification is just that – a specification of how to use something. It is accepted practice for all consumer goods to be sold with a user's guide that explains the usage, no matter how obvious it might seem. More technically oriented products not destined for the mass market frequently arrive with no user's guide at all. Producing a user's guide for a consensus standard might be seen as an unnecessary expense

[*] "A measure of the design of a product or process in terms of its ability to be produced easily, consistently, and with high quality." APICS Dictionary, Eighth Edition, 1995.

because all of the prospective users are involved in the standards process and are presumed to understand the intended usage. Unfortunately, it is very possible that the usage understood by one such user is not the same as the usage understood by another, even though both approaches make perfect sense in their own context.

There is often a business case for allowing the usage specification to remain ambiguous. A strong usage specification is necessary to achieve true, plug-compatible interoperability between independently developed client programs and vendors' products, but to comply with a standard that is this strong usually requires nontrivial changes to the product. This is compounded by the costs of either maintaining backward compatibility, i.e., supporting two different software modules when one would do, or of losing an established customer base because of incompatible changes.

For example, suppose that two pre-standard software products require a client to perform operations in different orders to accomplish the same goal. Real interoperability can be achieved either by specifying a normative ordering of operations or, less optimally, by specifying a set of permissible orderings and a normative way for the client to determine which one should be used at run time. A weak usage specification would leave the order of operations unspecified so that both implementations already conform.

With a weak standard, clients might need to be rewritten for each revision of a server or for each different product. From the user's perspective, this defeats the main purpose of standardization. However, if the apparent costs of complying with a standard exceed the expected benefits, then vendors will not comply, and the standard itself will be a failure. To stick with a weak usage specification avoids these risks as well as the expenses inherent in developing a stronger standard, but testability and interoperability both suffer as a result.

2.5. Completeness of functional specification with respect to usage

The functional specification must meet all expressed requirements to be considered complete. A subtler problem occurs when the functional specification is incomplete with respect to the proposed usage. In this case, the functional specification could meet all expressed requirements, yet it would still be impossible to execute a usage scenario without using some extension to the specification. An implementation that strictly conformed to the specification but failed to provide the needed extensions would be "broken as designed."[9]

Incompleteness with respect to requirements does not necessarily impact testability. Incompleteness with respect to usage does because it is impossible to execute a test scenario without relying on unspecified behaviors.

2.6. Interaction specification

Even if we have complete and consistent specifications for the functions provided by two components in a system, these do not necessarily define how the components would work together to achieve a specific goal. If interoperability and substitutability of components is a goal of the specification, then the interactions must be completely specified.

When component A expects some function to be supplied by component B, the interaction between these components that results in this function being provided to component A by component B must be specified and supported by both components. The mode of failure when a necessary component is unavailable for some operation should also be specified.

2.7. Specification of precision and limits

Manufacturers specify tolerances and limits on physical parts as a matter of course, but this vital information is frequently omitted from information specifications. Without it, our test verdicts remain ambiguous. We do not know how close to the expected result is close enough, and we do not know on what scale the tests should be run.

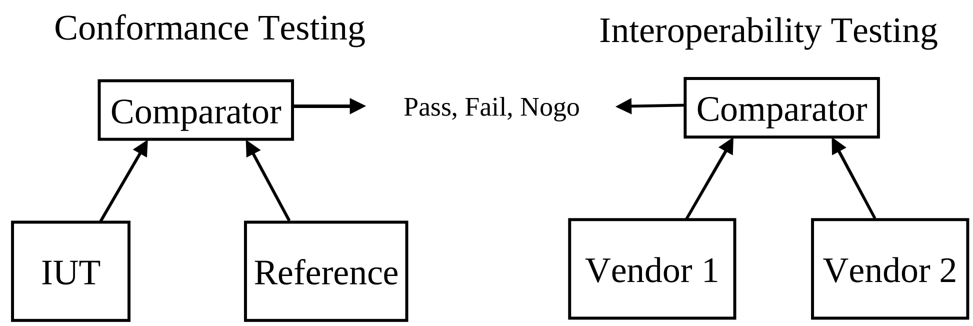

Figure 1: Conformance vs. interoperability testing with the Comparator

3. General testing approach

A testing approach can be coverage-based or scenario-based. We elected to follow a scenario-based approach partly because a plausible usage scenario helps to justify a conformance test. If a given behavior is the unique conforming solution to support an intended usage of the standard, then a test that exercises that behavior is valid against the intent of the standard, if not its letter. But more important was the value of scenarios in providing focus for the very small number of tests that we would be able to develop. Coverage-based approaches are meaningful if a test suite is complete enough to cover a significant portion of the standard, but if this is not the case, then it is better to cover the "hot spots," those parts of the standard that are likely to be used most often or most likely to cause problems.

We constructed test clients according to the following outline:

1. Print out boilerplate, normative references, and assumptions.
2. Initialize and set up.
3. Execute test scenario.
4. Check resulting state for correctness.

The test begins with the implementation under test (IUT) in a known state. Step 3 exercises behaviors that alter that state, and the observed resulting state is compared against the expected resulting state in step 4. The test verdict is one of "pass," "fail," or "nogo" (nogo indicates that an operational problem prevented the test from executing successfully). Although the most rigorous checking is done in step 4, an IUT can also fail if incorrect behavior occurs during step 2 or 3. There may be interactions with the IUT even during initialization, such as to open a session. An inappropriate response to one of these operations would register as a failure.

Step 4 is performed by a component known as the Comparator. It compares the states of different servers, produces a verbose log of the states, and flags any differences. To use the Comparator in conformance testing, we must supply a reference server containing the expected result state. In interoperability testing we would simply execute the test scenario against both servers and then verify that their resulting states were equivalent (see Fig. 1). By comparing the states using the on-line interfaces of the servers, we avoid an export of the states to exchange files, which could potentially lose information in the translation.

The Comparator is conceptually simple and generic. In our idealized, object-oriented system, objects have state, behavior, identity,[10] and relationships to one another. The Comparator needs to traverse a graph of related objects, comparing the objects' state and identity attributes but not exercising their behaviors. With appropriate bindings, this procedure is applicable to all object-oriented databases and to most databases having even a vaguely object-oriented flavor, including, for example, a database that runs on EXPRESS.[11]

If the implementation context is CORBA and Interface Definition Language (IDL),[12] then state is implemented with attributes, behavior with operations, and relationships with the standard CORBA Relationships Service.[13] If the implementation context is SDAI/EXPRESS, then state is implemented with entity attributes, behaviors are non-existent, and relationships are implemented with entity attributes of type entity,[14] i.e., pointers.

Unfortunately, what was simple in theory had complications in practice that we describe later in this paper.

4. PDM Enablers

PDM Enablers is a specification of the Manufacturing Domain Task Force (MfgDTF)[15] within OMG. The purpose of PDME is to "provide the standards needed to support a distributed product data management environment as well as providing standard interfaces to differing PDM systems."[16]

PDME defines twelve modules. Three of these – PdmResponsibility, PdmFoundation, and PdmFramework – are "basic" modules, providing common infrastructure for the others: PdmBaseline, PdmViews, PdmDocumentManagement, PdmProductStructureDefinition, PdmEffectivity, PdmChangeManagement, PdmManufacturingImplementation, PdmConfigurationManagement, and PdmSTEP. Testing the entire specification would not have been feasible with our resources, so we limited our scope to PdmDocumentManagement, PdmProductStructureDefinition, and portions of the other modules on which they depend.

Section 1.16.2, "Proposed Compliance Points," casts doubt on PDME's testability. It contains the following statements:

> Compliance to this specification is to be judged against the IDL definitions of the interfaces and their attributes and operations.
>
> The UML object model diagrams are not to be considered as part of the specification for the purposes of judging compliance. They are provided to illustrate and describe the behavior of the IDL interfaces.

The PDME specification consists of IDL, Unified Modeling Language (UML),[17] and informal descriptions. The language above disclaims the UML and informal descriptions, leaving only the IDL as a possible target for formal testing. IDL defines the signatures of operations but it does not define what they do or how they work. A completely formal approach to conformance testing of PDME would therefore be useless.

Fortunately, the PDME specification itself supplies usage scenarios in Section 1.12, "Mapping the Product Development Process to the PDM Enablers." These are the scenarios that were used to define the expected functionality of the PDM Enablers,[18] so they are presumed valid. We had to simplify the scenarios to achieve a manageable scope for testing, but we did not need to invent our own.

Our first two scenarios were trivial uses of DocumentManagement that did not involve product structure. We built tests for them using Interface Testing Language (ITL), one of the input languages for the Manufacturer's CORBA Interface Testing Toolkit (MCITT),[19,20] which is described in more detail below.

The remaining scenarios involved the less trivial ProductStructureDefinition, and we had need of some valid test data. No "blessed" data was available in the PDME context, but data was available for the STEP PDM Schema and for AP203[21] (AP is "Application Protocol). PDME supplied scenarios but no data; STEP supplied data but no scenarios (or, at best, only exchange-based ones). This was a manifestation of the interface-centric nature of the PDME standard and the data-centric nature of the STEP standard.

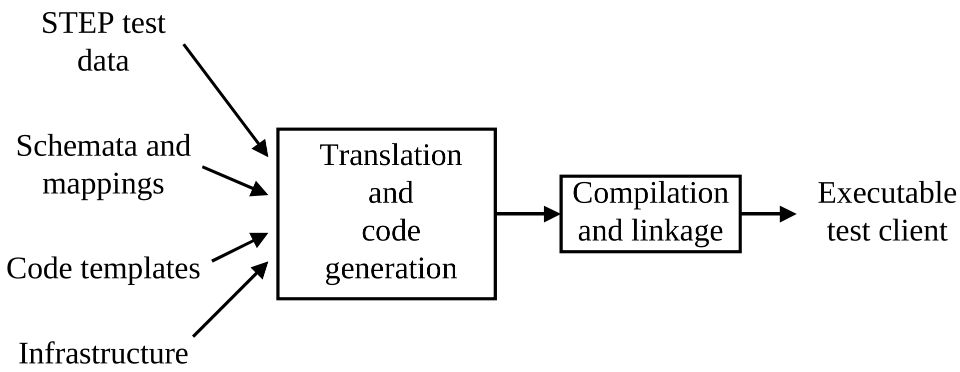

Figure 2: Test client generation

4.1. Test client generation

We partially automated the construction of PDME tests from STEP data using the process shown in Fig. 2. The relevant software components are described below. For a more detailed understanding of the steps in the process, the reader is encouraged to review the sample data that was distributed to the MfgDTF.[22]

4.1.1. PDM Schema to PDM Enablers translator

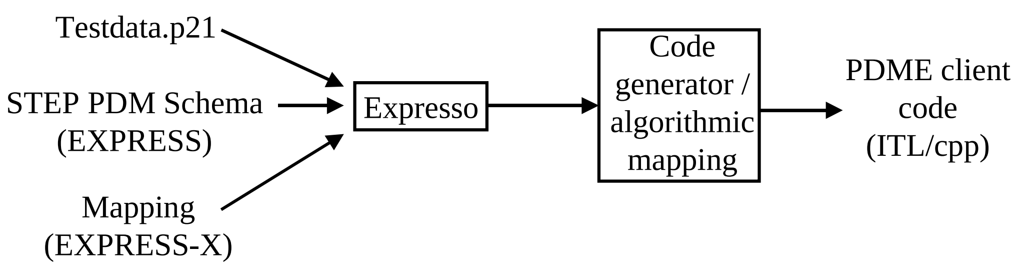

Figure 3: PDM Schema to PDM Enablers translator

The PDM Schema to PDM Enablers translator is based on EXPRESS-X[23] and uses the NIST Expresso[24] development environment (see Fig. 3). Expresso processes EXPRESS, STEP's Part 21[25] file format for instance data, and EXPRESS-X. The EXPRESS-X schema describes the information needed for creating calls to the PDM Enablers using a data set described by the PDM Schema. The translator application uses that data to generate the code that calls the PDM Enablers and outputs code in a "macroized" version of the Interface Testing Language used by MCITT.

The macros play the role of PDME convenience functions[26] for the test effort (see Fig. 4). They are expanded using the standard C preprocessor, which also expands directives to include common test boilerplate. A subsequent pass is made with TEd, the scripted test editing tool provided with MCITT, to work around the line-breaking limitation of the C preprocessor.

In addition to making the tests more readable, the macros help to increase the flexibility of the test system. The PDME specification often specifies no normative ordering for the operations that are needed to accomplish a given convenience function, yet implementations in practice do require particular orderings. Macros help to accommodate the needs of these different implementations. Instead of changing many operations in many tests, we just change the macro, and the changes become effective in every instance.

The mapping of STEP PDM Schema data into the PDM Enablers is imperfect in that we limited the scope of the data mapped in order to get a prototype working but also in that the feature spaces of the PDM Schema and PDM Enablers are not identical. The PDM Schema includes some features, such as approvals, that are omitted from PDME, while PDME includes other features, such as extra mechanisms for relating documents to parts, that the PDM Schema does not have. Testing of some features therefore required manual coding. Nevertheless, generating a test case using STEP test data covered most of the relevant PDME features and was clearly preferable to building an entire test case from scratch.

4.1.2. Manufacturer's CORBA Interface Testing Toolkit

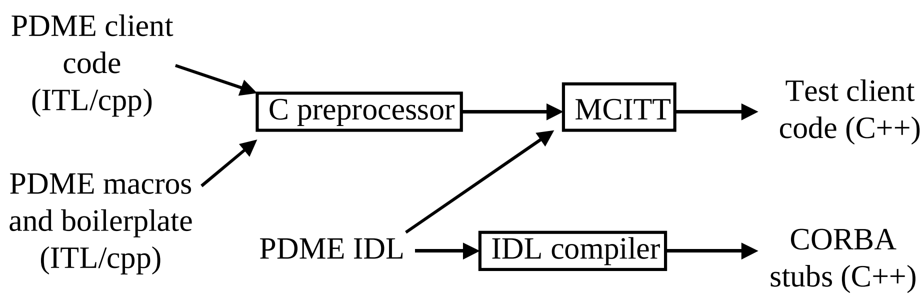

Figure 4: Macro expansion and code generation

MCITT, a CORBA testing toolkit with a variety of useful functions, was originally produced in support of the Advanced Process Control Framework Initiative[27] and the National Advanced Manufacturing Testbed Framework Project.[28] In the PDME test building process, MCITT is used to translate Interface Testing Language into C++ bound to a particular CORBA product (see Fig. 4).

One could view the usage of MCITT in the test generation process as a second layer of macro processing after the expansion of the PDME macros mentioned above. MCITT includes yet more boilerplate and expands some ITL commands to large blocks of CORBA C++.

MCITT's Interface Testing Language (ITL) and Component Interaction Specifications (CIS) are also used to specify and generate the reference servers needed for determination of the test verdict as described below. For details of MCITT and its specification languages, please see the cited references.

4.1.3. Comparator

The previously described Comparator is linked into the final executable test from a separately maintained object code library (see Fig. 5). This is a trivial step; the interesting issues related to the Comparator occur earlier, in the design stage, while defining a binding between it and the PDM Enablers.

PDME does not fit the idealized Comparator view of object-oriented systems; some important state information is only accessible through a series of operations, and some attributes are not significant. For example, consider the SecuredFile interface:

Version 1.1 IDL, abridged (with exceptions removed):

```
interface SecuredFile: File
{
      attribute long size;
      attribute MimeType type;
      string get_pathname();
      string get_url();
      long begin_put_buffer(in long bufsize, in long filesize,
            in MimeType type, in string transfer_encoding);
      void put_buffer(in buffer buf);
```

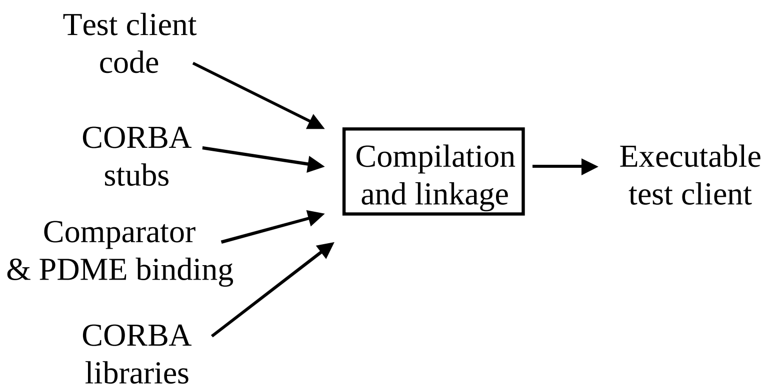

Figure 5: Compilation and linkage

```
        void end_put_buffer();
        long begin_get_buffer(in long bufsize,
              in string transfer_encoding);
        buffer get_buffer();
        void end_get_buffer();
  };
```

Inherited attributes: name, created_date, last_modified_date, short_description, long_description, related_object, roles_of_node, constant_random_id

Inherited operations: get_id, get_id_seq, bind, change_id, is_locked, lock, unlock, get_info, set_info, get_viewable_info, get_updatable_info, copy, move, remove, copy_node, move_node, remove_node, get_life_cycle_object, roles_of_type, add_role, remove_role, is_identical

In this example, PDME practice diverged from Comparator theory in the following ways:

1. Some pieces of information that contain state, such as pathname and URL (Uniform Resource Locator),[29] are instead represented as operations.
2. The most important state information for a SecuredFile – its content – is not an attribute. To access it, the client must execute a nontrivial data transfer protocol.
3. The created_date, last_modified_date, and constant_random_id attributes are not comparable between the IUT and the reference.
4. The related_object attribute is there even though it makes no sense in this context.[30]

Fortunately, there is a practical solution. The Comparator already requires a "binding" for each context (i.e., PDME or STEP) to map the Comparator's generic node (object), edge (relationship), and attribute concepts onto the features that realize these concepts in the specific context. The problem was easily solved by extending the binding to do arbitrary *projections*[31] instead of just mappings (see Fig. 6).

Within the binding, attributes are flagged for one of three kinds of treatment: print values and compare, print values only, or skip. "Good" attributes are both printed and compared. Volatile attributes that cannot match the reference are printed but not compared; a failure in the attempt to retrieve the attribute's value can still impact the test verdict. Finally, attributes that are not really attributes at all, such as those that get projected as relationships, are skipped. (Relationships are checked by graph traversal, which is a different part of the process than this value-comparison.)

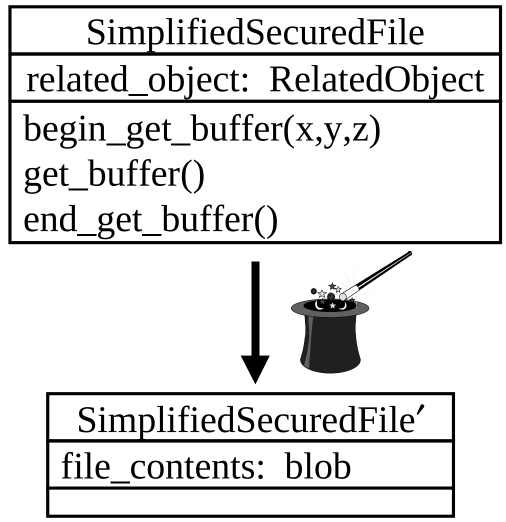

Figure 6: Projecting an object

4.2. Testability of PDM Enablers

4.2.1. Application interface specification

PDME is precisely an application interface specification for a PDM system, so no testability problems related to application access were found. However, there is not necessarily any alternative interface that could be used to corroborate the results of testing over the PDME interface or to verify that the mapping between PDME concepts and the internal business schema of the PDM is meaningful.

4.2.2. Implementability of the specification

PDME inherits Common Object Services in which some interfaces inherit other interfaces by multiple paths. This case requires special treatment in C++, and unfortunately it was not properly handled by our IDL compiler. To work around, we used a combination of IDL changes and edits to the output of the IDL compiler. Meanwhile, we submitted the problem as OMG Issue #2345 which has been assigned to the C++ Revision Task Force (RTF).[32]

Another implementability issue concerns the imperfect realization of the PDME model for Substitute by CosGraphs and CosRelationships. The concept of "substitute" is defined as follows:

> A Substitute is a component within an assembly whose form, fit, and function might be different from the component for which it is a replacement, but that can fulfill the requirements of another part within the context of the assembly. A substitute is specified by defining another usage relationship, the substitute, which can replace the original usage relationship, the base.[33]

Substitute was modeled as a relationship between two Usages, which themselves are relationships between Parts. But the resulting model cannot be well-realized using CosGraphs because it would require an edge that connects other edges, which is not supported. The realization used in PDME effectively creates two separate "graphspaces," one in which Usage appears as an edge and one in which Usage appears as a node. To navigate from a Part to a Substitute relationship, a client has to stop and cast the Usage object from edge to node and initiate a new traversal.

Handling this operation transparently in the Comparator binding would have been very messy; instead, we made two separate calls to the Comparator in the test client, one for each "graphspace."

4.2.3. Functional specification

The PdmStep module of PDME defines two operations, StepTranslator::import and StepTranslator::export, to support the use of STEP for PDM data exchange. The function of the import operation is to "instantiate new PDM entities corresponding to the entities contained in [a STEP Part 21 file]."

PDME Section 2.12.1, "PdmStep Overview," explains that the mapping of STEP data into the PDM Enablers "schema" is non-trivial:

> The PDM Enablers interface models are guided by STEP. However, the PDM models are not a representation of STEP Application Interpreted Models (AIM). Rather the PDM models represent user-level objects that are analogous (but not identical) to corresponding Application Resource Models (ARM) from various Application Protocols.

The problem is that neither PDME, nor STEP, nor any other formally recognized specification existing at this time defines a mapping in either direction. There is no specification for which PDME entities are to be instantiated as a result of importing any given Part 21 file.

The MfgDTF considers this out of scope for the PDME specification but acknowledges that such a mapping is necessary to achieve interoperability. However, there is disagreement over how specific such a mapping should be. The feasible mappings depend on the internal "business schema" of the PDM, and the MfgDTF will not invalidate any existing commercial product by constraining that schema. Nevertheless, several efforts are underway involving the MfgDTF and PDES Inc.[34] to reach some agreement. Ford, Boeing, ProSTEP,[35] and NIST have created their own mappings independently, but these have not achieved consensus approval or been widely published.

4.2.4. Usage specification

One of the most common usage scenarios for a PDM is checking in a new revision of a document. PDME defines lots of operations that might be used somewhere in this scenario, but these operations could be performed in the wrong order or invoked on the wrong object. For example, consider locking. In PDME, every ManagedEntity inherits lock() and unlock() operations from the PdmFoundation class Lockable. Almost everything is a ManagedEntity, so almost everything is Lockable. SecuredFile, DocumentMaster, DocumentRevision, and DocumentIteration are all Lockable. Which of these must be locked for a checkout, and unlocked upon checkin? Also, what is different if we are checking in a completely new document? Are new entities created in the locked or unlocked state?

PDME does contain an attempted usage specification for document checkin, but it proved to be problematic. We raised this as OMG Issue #2485, assigned to the PDM RTF.[36]

Another usage issue arises where PDME defers to a particular STEP AP without specifying what happens for other APs. It is unclear what to do about PartMaster::part_type and part_classification if a part has multiple classes or just doesn't mesh with AP203. We raised this as OMG Issue #2622.[37] We later learned that the AP203 references were a legacy from before when the MfgDTF shifted its focus from AP203 to the STEP PDM Schema, and should be removed.[38]

4.2.5. Completeness of functional specification with respect to usage

Identifications are needed for practically every PDM usage scenario. An IdentificationContext is a prerequisite for applying identifications to created PDME entities. Known IdentificationContexts can be retrieved using IdentificationContextFactory::find_id_context (in string the_context_name), but there is no IdentificationContext named in the PDME specification and no function to retrieve a default context. Similarly, IdentificationContextFactory::create takes a catch-all PropertySet as its only argument, and there is no normative usage that would guarantee a successful create.

From a usability standpoint, a normative context would be little value added. PDME does not specify a context because most companies need to define their own contexts anyway. Likewise, the relevant properties for creating a new context are company-specific. But the lack of a normative context hurts testability since the test suite must be customized to use whatever contexts happen to be available at each deployment.

For better testability, PDME could have defined a normative test context that all implementations would support. It did not need to be complicated; its only purpose would have been to support the testing of other features.

Identifications are also problematic from the perspective of database integrity. The method to create an Identifiable object with a given initial identifier is not clear. It would be harmful for unidentified entities to persist in the PDM, so one would expect to provide some form of identification when creating an entity. But PdmFoundation::IdentificationFactory::create requires the Identifiable to have been created already.

A rule of thumb being used by the PDME implementers is that the initial values of attributes (broadly interpreted) of entities that are being created may be supplied in the PropertySet given to a Factory::create method, setting property_name to the name of the attribute and property_value to the initial value. A client could use this method to supply an initial identification; however, this interpretation is not normative. There is nothing actually in the specification to prevent unidentified entities from persisting.

The functional specification is also incomplete with respect to the usage of Attributable to supply part data, such as classifications and cost estimates, which appear in the usage scenarios. Although the interface for supplying attributes is defined, the implementation is not actually required to support any particular set of attributes, and support for *ad hoc* attributes is optional:

> This technique allows access to all legal attributes for the managed item in the PDM system, even those that are not exposed to clients through the IDL definitions. PropertySets support the manipulation of attributes that are known to a customized PDM system but are not specifically exposed by IDL. And PropertySets defined and created by the client support the manipulation of ad hoc or loose attributes that are not known to the PDM system's schema (if the PDM system supports ad hoc attributes).[39]

4.2.6. Interaction specification

PDME relies on another OMG-specified component, Workflow,[40] to handle approvals. Approvals are delegated to Workflow, but Workflow is generic and does not define specific usage for approvals. The interactions between PDME and Workflow are not specified. The first draft of a white paper attacking this problem has just been written,[41] but there is still no agreement on which component, if either, is a client of the other. (Plausibly, both could be servers of a third component that completes the integration.)

4.2.7. Specification of precision and limits

Section 2.6.3.12 of PDME, "Data Transfer," rightly specifies a minimum limit on the negotiated buffer size (256 bytes). This avoids a potential challenge from the vendor of an implementation claiming a maximum buffer size of 1 byte. (If the product being wrapped does not already support buffered data transfer, the degenerate case effectively avoids the need to implement it.)

Most of the data in a PDM remains uninterpreted by the PDM system, so precision is not a major issue in this case.

5. Analysis of the test method

Since one of our goals is to derive a test method that is usable for interaction-driven manufacturing systems in general, we have analyzed the PDME testing approach for features that could impact its reusability in other contexts.

5.1. Features of testing approach

The testing approach that was used for PDME requires a reference for determination of the test verdict. With no trusted PDME implementations available, we must define the expected results for each scenario using an MCITT script (CIS). Although defining a reference emulation for any particular scenario is not difficult, the high degree of automation of the test creation process puts pressure on the relatively labor-intensive activity of defining the references, particularly with respect to the explosion of interactions performed by the Comparator.

To speed up the generation of server emulations, we created two more code generation tools: out2cis and makecomp. The scripted responses to the interactions performed in the test scenario are generated from the test client source code, while the long scripts needed to "feed" the Comparator are generated from much shorter Extensible Markup Language (XML)[42] descriptions of the expected PDME database contents. Unfortunately, the explosion of interactions from the Comparator combines with the linear code generation style of MCITT to produce an explosion in the size of the source code needed to emulate a reference. The resulting tests were built successfully on a heavy-duty desktop machine, but lessons learned include that our approach is not scalable to much larger test cases than our PdmProductStructureDefinition test.

Clearly there exists some number of tests beyond which it would be less effort to build a reference implementation than to create individual reference scripts for each test. However, any test suite faces a similar challenge. Completely automatic conformance testing is only possible if the specification defines all of the correct behaviors in a machine-processable way, but this is effectively the same as having built a reference implementation already. After all, that is what source code is – a machine-processable specification of behavior.

Finally, although we succeeded in mostly automating the generation of test code, the resulting test generation process is more complex than one would want for a non-experimental test suite.

5.2. Features of Comparator implementation

The most significant problem found in applying the Comparator approach to PDME was with the identification of PartStructures. PartStructure objects are not distinguishable from one another except by their associations with other objects or by implementation-dependent means. They have no persistent, unique, external identifiers; they have only their run-time object handles and the constant_random_id attribute inherited from CosObjectIdentity. This violates a critical assumption that was made in the Comparator approach, that nodes in the graph will be identifiable and distinguishable. Because of this, it is nontrivial to establish an association between PartStructures in the reference server and PartStructures in the IUT.

The existence of objects that are not distinguishable is a significant complication that we did not consider. Comparing state object-by-object was not a requirement for the Comparator. The system state could have been defined more generally, requiring a certain number of PartStructure nodes with certain relationships instead of designating which nodes have which relationships. With extensive reworking of the Comparator, it would be possible to relax the condition on distinguishability of nodes. To work around the problem less expensively, we synthesized identifiers for the PartStructures using the identifiers of the nodes to which they are related.

One of the services that is provided by a Comparator binding to a particular database is a function to find all relationships pertaining to a given object. In PDME this can be done using CosGraphs; in SDAI this can be done using the "Find entity instance users" operation.[43] But many object systems use one-directional pointers and do not supply any "find object users" operation. In these cases, the comparison of the databases would have to be initiated from a well-connected node from which all other nodes are reachable.

6. STEP SDAI / PDM Schema

SDAI is a standard for programmatically accessing data described by the information models in STEP. Unlike the PDME specification, which implicitly constrains operations on data by supplying relatively coarse-grained access, SDAI provides fine-grained, unconstrained access to data. This means, for instance, that SDAI does not impose an order on the creation of instance data – using SDAI, the entity corresponding to a version of a product might be created before the entity for the product itself. However, SDAI does provide a function, "check all global constraints," to check that existence constraints are met. On the surface, this approach is inferior to disallowing the behavior implicitly; however, there are times when such fine-grained control is actually a benefit. For example, if every instance of A is constrained to have a corresponding instance of B, and the reverse is also true, it is not possible to create any instances at all unless constraint checking can be deferred until both have been created.

The most significant impact on testing is that a system hosting SDAI is allowed to reach an inconsistent state. In other words, there may be points in the execution of a test case at which the data will visibly fail to conform to all of the constraints imposed by the standard, yet the IUT will still be considered conforming. It is therefore necessary for testing purposes to identify those points during a test case's execution at which all constraints should be satisfied. In database terminology, these points define transaction boundaries. All constraints must be satisfied at the end of the transaction, but while the transaction is in progress, the database may pass through an inconsistent state.

Given a particular application, the end of a transaction is evident from the semantics of the application. However, these semantics are not defined in a manner that can be used for an automated testing system. We will now describe two approaches to automating the testing process for SDAI having different transaction boundaries.

6.1. Testing approaches

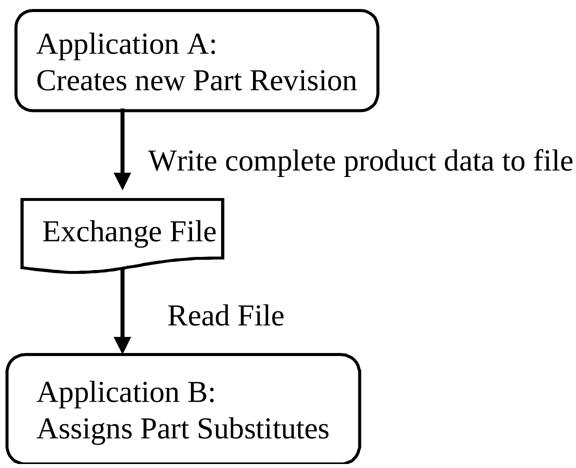

Figure 7: Operational model of file exchange

Since the information models in STEP are static and do not include operational models, they provide little insight into what would be appropriate bounds for transactions. The only real definition of transactions is in the context of the exchange of an entire product model. As depicted in Figure 7, this model consists of two transactions: the production of the exchange file and the consumption of the exchange file. The first transaction, typically read-only, starts when the file is produced. It is not very interesting from the perspective of interactive testing since it does not change the state of the underlying system. (It is interesting from the perspective of testing for compliance with STEP, since producing a conforming exchange file is a significant compliance point.) The second transaction begins when the exchange file is read into the receiving system and ends when the entire file has been processed and the state change is complete.

The operational model for SDAI (see Fig. 8) is somewhat different than that for data exchange since it contains a shared database; however, the only known uses of the interface thus far have been in the context of the static information models that were designed for file exchange. In the SDAI operational model, multiple applications operate against the same set of data. Both applications can see updates to the data set, and both applications can change the data set. For a more complete discussion of the differences between data exchange and data sharing, see "Chapter 6: Sharing versus Exchanging Data," of *STEP the Grand Experience*.[44]

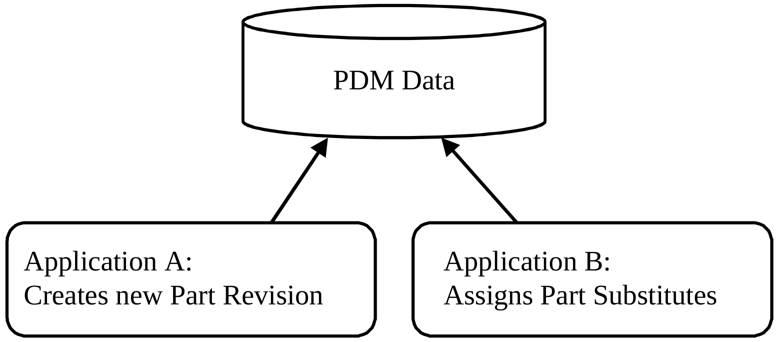

Figure 8: Operational model for data sharing

Given the prevailing operational model for STEP, the first approach to automate testing of SDAI is to transfer transaction boundaries as defined for file exchange into the SDAI context. In other words, a transaction would consist of the creation of a complete set of data for representing a product. This type of transaction would not address any incremental changes to the data set or any conflicts that could arise based on shared use of the data. As with file exchange, the entire data set would be validated at one time for conformance with the information model.

The second approach to automating the testing process looks beyond STEP for the definition of transactions. It is a scenario-based approach in which scenarios are defined for the application context. The definition of data sharing scenarios for SDAI access is analogous to the definition of abstract test suites for data exchange. To validate this approach, we look to apply it in the PDM context where we have suitable scenarios described within the OMG PDM Enablers specification. Fortunately, the PDM Enablers specification and STEP specifications in this area are very compatible.

Using a scenario-driven approach, transaction boundaries are defined based on the semantics of the scenarios that mimic the applications for which the interface is intended. These scenarios call for incremental changes to the PDM system, such as creating new versions of parts and establishing relationships amongst the data. In this situation, incremental changes should not require validation of the entire data set. One of the remaining challenges is to define a scope for the data that would need to be validated after an incremental change.

6.2. Test system design

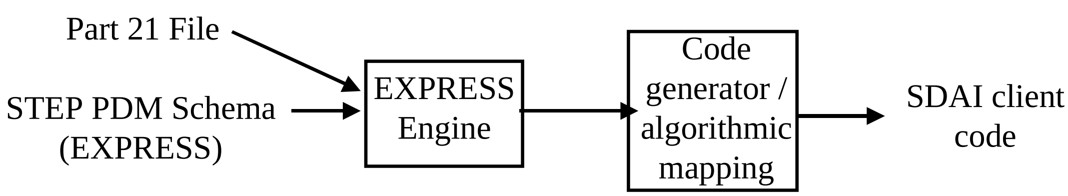

Figure 9: Exchange-driven approach

Figure 9 depicts a simple test system that uses the exchange model to define transaction boundaries. This system is relatively straightforward to produce but the tests would not be conceptually much different than those for file exchange. They would not really exercise the capabilities of SDAI.

The scenario-driven approach to testing implies a different test method. We plan to validate this approach through prototyping. The prototype test system will reuse some of the code generation technology developed for PDM Enablers testing, but in this system the test-client code will be a series of SDAI calls instead of PDME operations.

The prototype test system will test an SDAI interface to the PDM Schema. The scenarios will be derived from the PDM Enablers specification and populated with data corresponding to the PDM Schema. These scenarios will be executed against an existing PDM Enablers implementation that uses the PDM Schema as its internal data model.[45] The PDM Enablers implementation will translate calls to the Enablers to operations on the PDM Schema that we can easily transform into equivalent SDAI operations. We will then be able to reuse our existing PDME test cases to test SDAI/PDMS. The test system is depicted in Figure 10.

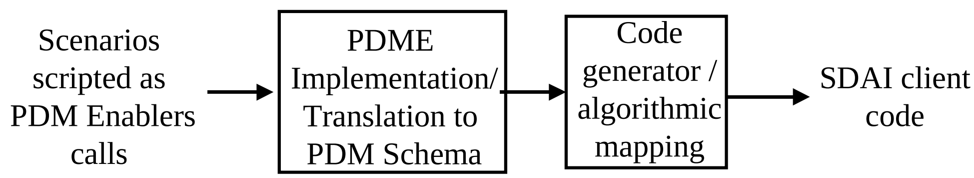

Figure 10: Scenario-driven approach

The test system will generate the basic data access calls as well as data validation calls to check that the system is in a consistent state. The Comparator will then perform an external check of the data values themselves.

6.3. Testability considerations

Four Parts of STEP are involved in testing an implementation of STEP using SDAI as its access mechanism: EXPRESS, SDAI, an SDAI language binding, and the Application Protocol (AP) itself. (For our study we have chosen to use the PDM Schema in the place of an AP since it is smaller and will be included in several APs.)

EXPRESS is used to represent an information model that captures many of the application requirements. SDAI is the functional specification of the data access mechanisms and hooks for checking data constraints at run-time. The SDAI language binding is the manifestation of those functions as operations in a particular programming language. Finally, the AP specifies the particular application requirements using the features available in both the EXPRESS and English languages. "Conformance to STEP" implies conformance to the relevant combination of these parts.

EXPRESS is a declarative language. It has no execution model – that is, no semantics associated with an ordering of events. It is used to describe the "world" as it should be rather than how the world is constructed. SDAI is the programmatic piece of the puzzle. SDAI specifies a generic operational model for how to access data and how to check constraints complete with error conditions. SDAI does not specify any of the semantics of an application and as such does not specify how errors should be treated. An AP specifies the semantics of an application area.

Testing of STEP exchange files is rather straightforward. They can be tested for conformance with an EXPRESS declaration; the test is whether the world is as it should be. To test an implementation based on SDAI one must test how the world is constructed. Testing an "implementation of STEP" using SDAI requires extensions to the AP to specify semantics for entity construction and transactional boundaries. The mechanisms provided by SDAI can assist in implementing those semantics, but neither part is complete without the other. The four testability issues detailed below particularly stand out as not being fully covered by the existing specifications.

6.3.1. Application-defined transactions

As discussed previously, a primary consideration for testability of SDAI implementations is the need to define meaningful transactions to test. In the exchange-based operational model the unit for testing is the entire product model as embodied in static exchange file. The file itself can be tested for conformance to the standard. Problems in the file indicate problems with the producing system. The other extreme, in the

SDAI operational model, would be to test the entire database after each operation. Not only is this approach inefficient since each SDAI operation affects only a small portion of the entire data set, it is also not realistic since many SDAI operations will leave the data set in an invalid state. A suitable median needs to be defined.

6.3.2. Application interface specification

In order to test whether a system is capable of producing valid data, one must have a means for it to first consume valid data. In existing, exchange-based, STEP conformance tests the initial data set is input into the system through a non-standard interface. The tests would not be credible if the system were allowed to read in an exchange file and then reproduce it because a null filter or word processor would pass.

In the SDAI operational model, it is not necessary to use any non-standard interfaces to input or retrieve data. Moreover, for applications like the PDM Schema, it is likely that implementations that supply no other interfaces will be built. Even the capability to load an exchange file is not required of SDAI implementations.

These facts lead to the conclusion that it is only logical to reverse the order of tests for SDAI (relative to the current testing practice for file exchange): first test whether the system consumes valid data and then test whether it produces valid data. This reality highlights the importance of a usage scenario in developing an abstract test suite for a standard. While the resulting scenario is similar to the testing that was done for PDME, STEP conformance testers are used to the higher level of assurance that is given by testing over multiple interfaces.

6.3.3. Usage specification: ordering of operations

Valid SDAI operations may exist that the hosting system is not able to support but that do not denigrate the service provided by the system. Consider the example of the operational constraint that a product master must exist before a product version can be created. A system that enforces this constraint in its internal business schema would not allow it to be violated through SDAI or otherwise. (This could be a legacy PDM system wrapped by an SDAI/PDM Schema interface, which is a usage of SDAI that was clearly intended.) Moreover, a user of the system might not want to allow the operational constraint to be violated. However, a strict interpretation of the existing specifications could label such a system as non-conforming.

6.3.4. Implementability of the specification: deferred constraint checking

SDAI provides flexibility in handling invalid data. The special SDAI operations to validate data do not fit the exchange file operational model, where invalid data are signaled when the file is loaded, but they present their own set of considerations. In particular, to test the SDAI validation operations one must be able to input invalid data and then test that the validation functions catch the errors. With error classes such as existence requirements this approach is logical. If an entity instance has a required attribute that has not been assigned a value, then the check for required attributes should produce an error code. With other error classes the approach does not work. For example, if an aggregate has a limitation on its bounds, would it really be non-conforming behavior to disallow the out-of-bounds insertions immediately instead of waiting for the "validate aggregate bounds" function to be called? In the extreme case, since EXPRESS allows for infinitely large aggregates, all implementations would be non-conforming.

This example highlights a possible problem with our conformance criteria. In most cases, an implementation that checked constraints on insert would simply be non-conforming; the SDAI standard clearly states "EXPRESS constraints are validated only at the request of the application."[46] But if EXPRESS constraints are used to define acceptable hard limits for the underlying database, as they were in this example, then the deferred validation requirement of SDAI might not be implementable.

Furthermore, the SDAI specification clearly allows a system to remain in an inconsistent state since transactions can be committed with invalid data. Application semantics must be defined to signal the logical end of a transaction – the point at which all data must be in a valid state. These transactions need not correspond one-to-one with SDAI transactions, resulting in an unfortunate name collision.

In light of this flexibility, interpretation of test results will be difficult. A strict interpretation of the SDAI specification would reveal the "correct" results; however, in reality, implementations may handle invalid data in different ways. A strict comparison of results with a reference implementation may not suffice, but it could be very difficult to develop a reference implementation that can mimic any "correct" interpretation.

We expect other testability considerations to arise as we gain more experience in implementing the prototype test system.

7. Conclusion

The work of the TIMS project will continue through SDAI/PDM Schema testing and on to testing of larger, interacting systems of components. Thus far, our testability work has produced the following guidelines for improving the testability of future specifications:

- First ensure that the specification is implementable. No one can test something that cannot be built.
- Define an application interface. It is very difficult to automate the testing of a system that only provides a nonstandard graphical user interface.
- Eliminate dependencies on unspecified features. If a particular implementation-defined facility is required to execute a usage scenario, specify an inexpensive, built-in testing version of the facility.
- Strive for completeness in the functional specification. Unanswered questions about what a given operation does harm testability, usability, and interoperability.
- Define a set of usage scenarios and ensure that all functions required to execute them are specified fully.
- Specify usage as well as functionality. Include the usage scenarios in the document as examples, and require the implementation to support them. If the system is transactional, define the transaction boundaries.
- If the component in question is not self-sufficient, fully specify all interactions with other components and validate these interactions against the other specifications. Supply sequence diagrams showing the interactions that occur for the usage scenarios.
- Leave no hard limits or precision unspecified.
- If objects of a particular class have so little value added that it is not worth distinguishing one instance from another, consider getting rid of them entirely.

The importance of usage scenarios cannot be overemphasized. STEP includes Abstract Test Suites for each Application Protocol. These test suites are usage scenarios for data exchange. The developers of STEP recognized the role of test suites in achieving the interoperability goal. In a data sharing environment such as that provided by SDAI, usage scenarios are even more important for defining bounds for transactions. Similarly, with PDME, usage scenarios are necessary guidelines for achieving interoperability.

Of foremost importance for manufacturing systems integration is that the interface specifications support a cohesive world view. A cohesive world view reflects design interoperability. This area has gone largely untouched in research on systems integration and tools to support system testing. A cohesive world view does not mean that components for which the interfaces are specified should share the same world view but rather that the world views need to come together in a meaningful way. The interfaces are the points of coherence. Thus, testing for compliance to an interface means testing whether the system supports the world view.

In our research, we are not asking the question how do we implement systems to support testability, but rather how do we define interfaces to support testing of the systems that implement those interfaces. Just as researchers of the former question conclude that the most complete form of testing that can be done uses formal methods, we must conclude that an interface can be most fully tested when it is completely specified. However, a complete specification cannot really be less than a complete implementation, often called a reference implementation.

Complete specification is in contention with the goal of an interface as a bridge between, often pre-existing, implementations. A reference implementation may be too costly to produce, may restrict the interface more than is necessary to support system interoperability, or conversely may allow for more freedom in the interface than is desirable. In short, there is no good way to verify that a reference implementation supports the interoperability requirements inspiring the interface specification. Furthermore, a reference implementation still does not provide usage guidelines such as are necessary for producing relevant and meaningful tests.

What is the solution to this dilemma? To enable testability, an interface specification must have a strong and agreed-upon set of requirements which can take the form of usage guidelines or scenarios. Usage guidelines or scenarios are the interpretive information necessary to explain an interface in a manner that conveys the world view that the interface supports. Working without such guidelines is the equivalent of trying to teach calculus without examples – a nearly impossible undertaking. An interface is only useful if people know how to use it.

8. Acknowledgements

Peter Denno was instrumental in the effort to develop a PDM Schema to PDM Enabler translator through his patient tutoring in the EXPRESS-X language and the Expresso software. We also thank Edward Barkmeyer, Jim Fowler, Simon Frechette, and the many reviewers of this paper for their contributions and general support of the PDM test effort.

References

"By selecting these links, you will be leaving NIST webspace. We have provided these links to other web sites because they may have information that would be of interest to you. No inferences should be drawn on account of other sites being referenced, or not, from this page. There may be other web sites that are more appropriate for your purpose. NIST does not necessarily endorse the views expressed, or concur with the facts presented on these sites. Further, NIST does not endorse any commercial products that may be mentioned on these sites."

[1] "What Is CORBA?" (tutorial), http://www.omg.org/corba/whatiscorba.html, Object Management Group, 1999.

[2] "About Microsoft COM," http://www.microsoft.com/com/about.asp, Microsoft, 1999.

[3] KC Morris, David Flater, Don Libes, and Al Jones, *Testing of Interaction-driven Manufacturing Systems*. NISTIR 6260, http://www.mel.nist.gov/msidlibrary/summary/9827.html, 1998.

[4] Object Management Group home page, http://www.omg.org/, 1999.

[5] PDM Enablers revised submission (including errata changes), http://www.omg.org/cgi-bin/doc?mfg/98-02-02, 1998.

[6] Version 1.1 of the PDM Schema is a pre-normative specification available from http://www.pdm-if.org/, 1999.

[7] ISO 10303:1994, *Industrial automation systems and integration — Product data representation and exchange.* Available from ISO, http://www.iso.ch/. See http://www.nist.gov/sc4/www/stepdocs.htm for an overview of the parts of the standard.

[8] ISO/DIS 10303-22:1996: *Industrial automation systems and integration — Product data representation and exchange — Part 22: Implementation methods: Standard data access interface.* Available from ISO, http://www.iso.ch/.

[9] "BAD," in Eric S. Raymond, *The Jargon File*, version 4.1.2, http://www.tuxedo.org/~esr/jargon/, 1999.

[10] Grady Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991.

[11] ISO 10303-11:1994: *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual.* Available from ISO, http://www.iso.ch/.

[12] "What Is CORBA?" (tutorial), http://www.omg.org/corba/whatiscorba.html, Object Management Group, 1999.

[13] CORBAservices: Common Object Services Specification, http://www.omg.org/cgi-bin/doc?formal/98-12-09, Chapter 9, "Relationship Service Specification," December 1998.

[14] ISO 10303-11:1994: *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual.* Available from ISO, http://www.iso.ch/.

[15] Manufacturing Domain Task Force home page, http://www.omg.org/homepages/mfg/, 1999.

[16] Manufacturing DTF RFP1, http://www.omg.org/cgi-bin/doc?mfg/96-06-01, 1996.

[17] OMG Unified Modeling Language Specification version 1.3, http://www.omg.org/cgi-bin/doc?ad/99-06-08, 1999.

[18] Manufacturing DTF RFP1, mfg/96-06-01, Section 6.5, "Manufacturing Enterprise Business Processes."

[19] Manufacturer's CORBA Interface Testing Toolkit home page, http://www.mel.nist.gov/msidstaff/flater/mcitt/, 1999.

[20] David Flater, "Manufacturer's CORBA Interface Testing Toolkit: Overview," *Journal of Research of the National Institute of Standards and Technology*, v. 104, n. 2, 1999. Available at http://www.mel.nist.gov/msidlibrary/summary/9904.html.

[21] ISO/DIS 10303-203:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 203: Configuration Controlled 3D Designs of Mechanical Parts and Assemblies*. Available from ISO, http://www.iso.ch/.

[22] TIMS test source code and output, http://www.omg.org/cgi-bin/doc?mfg/99-03-04, 1999.

[23] ISO TC184/SC4/WG11 N078, *EXPRESS-X Language Reference Manual,* http://www.steptools.com/library/express-x/n078.pdf, 1999.

[24] The Expresso Homepage, http://www.nist.gov/expresso, 1999.

[25] ISO 10303-21:1994: *Industrial automation systems and integration — Product data representation and exchange — Part 21: Implementation methods: Clear text encoding of the exchange structure.* Available from ISO, http://www.iso.ch/.

[26] 19990826 PDM Convenience Services RFP Working Draft, http://www.omg.org/cgi-bin/doc?mfg/99-09-01, 1999.

[27] Project Brief: Advanced Process Control Framework Initiative, http://jazz.nist.gov/atpcf/prjbriefs/prjbrief.cfm?ProjectNumber=95-12-0027, Advanced Technology Program, National Institute of Standards and Technology, 1996.

[28] Howard M. Bloom and Neil Christopher, "A Framework for Distributed and Virtual Discrete Part Manufacturing," in *Proceedings of the CALS EXPO '96*, Long Beach, CA, 1996.

[29] RFC2616, "Hypertext Transfer Protocol — HTTP/1.1," http://www.faqs.org/rfcs/rfc2616.html, 1999.

[30] CORBAservices, formal/98-12-09, Section 9.4.5, "The CosGraphs Module."

[31] Jeffrey D. Ullman, *Principles of Database and Knowledge Base Systems*, Vol.1, Section 2.4, "Operations in the Relational Data Model," 1988.

[32] OMG Issue #2345, http://www.omg.org/archives/issues/msg01158.html, 1999.

[33] PDM Enablers, mfg/98-02-02, Section 2.7.3.19, "Substitute."

[34] PDES = Product Data Exchange using STEP. PDES, Inc. home page, http://pdesinc.scra.org/, 1999.

[35] Christian Donges, Norbert Lotter, Lutz Laemmer, Andreas Schreiber, Max Ungerer, Anne Wasmer, "Comparison of the PDM Enablers Object Schema with the STEP PDM Schema," provided to the STEP/OMG PDM Harmonization Task Force, 1999.

[36] OMG Issue #2485, http://www.omg.org/archives/issues/msg01309.html, 1999.

[37] OMG Issue #2622, http://www.omg.org/archives/issues/msg01547.html, 1999.

[38] Larry Johnson, personal communication, September 1999.

[39] PDM Enablers, mfg/98-02-02, Section 2.3.3.2, "Attributable."

[40] Workflow Management Facility convenience document (specification with errata included), http://www.omg.org/cgi-bin/doc?bom/99-03-01, 1999.

[41] Dan Matheson, "The OMG PDM Enablers And The OMG Workflow Facility," white paper, Draft 1, http://www.omg.org/cgi-bin/doc?mfg/99-03-01, 1999.

[42] W3C Extensible Markup Language home page, http://www.w3c.org/XML/, 1999.

[43] ISO/DIS 10303-22:1996, Section 10.10.8, "Find entity instance users."

[44] Sharon J. Kemmerer, editor, *STEP the Grand Experience*, National Institute of Standards and Technology, Special Publication 939, June 1999.

[45] NASA STEP Testbed, http://step.nasa.gov.

[46] ISO/DIS 10303-22:1996, Section 5, "Fundamental principles."