Harmonized Conformance Testing for Product Data Managers

David Flater, K. C. Morris
National Institute of Standards and Technology, U.S.A.
dflater@nist.gov, kc@nist.gov

Abstract

Product Data Management (PDM) systems are gaining importance as central databases used for enterprise integration in the manufacturing domain. Multiple standards and standards-track specifications already exist for PDM, including the PDM Enablers standard from the Object Management Group® and the "PDM Schema" derived from ISO 10303, the ISO Standard for Exchange of Product model data. In the interest of testability and standards harmonization, we developed a "harmonized" testing approach that allowed test scenarios defined for one specification to be combined with test data defined for the other. The resulting tests could be realized with approximate semantic equivalence in both contexts, and the resulting lessons about testability are now having an impact on standards development.

Note: All referenced figures appear at the end.

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 misunderstood interface semantics and unintended interactions.

In the manufacturing domain, an important part of the process of enterprise integration is Product Data Management. A Product Data Manager (PDM) is a specialized database designed to manage the many kinds of product-related information that are needed to support a manufacturing operation and to support the "business rules" that keep all of this data consistent with itself and with the material world.

Computer-Aided Design, Manufacturing, and Engineering (CAD/CAM/CAE), Manufacturing Resource Planning (MRP), Manufacturing Execution (MES), workflow, and other manufacturing functions are being integrated via PDM systems that play the role of central database. This central role multiplies the costs and benefits of PDM integration. To ensure that the benefits outweigh the costs, standards organizations have given PDM some serious attention.

The standards activity and the importance of PDM in the manufacturing enterprise led us to experiment with PDM in a research project about test methods and design-for-testability [3]. We used two significantly different, yet conceptually related PDM specifications: Version 1.1 of the Object Management Group's (OMG) [4] PDM Enablers (PDME) standard [5], and Version 1.1 of the pre-normative PDM Schema [6] that is 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 were both specifications for application interfaces to PDM systems, but they were from different sources and had different design goals. Our goal was to find methods that exploited the conceptual commonality, yet remained adaptable to the very different environments in which the two specifications were defined.

One outcome of this study was a "harmonized" test method that integrated test scenarios and test data from two different PDM specifications, yielding test cases that could be realized with approximate semantic equivalence in both contexts. That method is the primary topic of this paper. Another outcome, the documentation of some lessons learned about specification testability, will be discussed with respect to its current impact.

2. 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 conformance 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 focus on the "hot spots," those parts of the standard that are likely to be used most often or most likely to cause problems. Given very limited resources, scenario-based testing is better because it maximizes the return on a small investment. The sacrifice is that it provides no formal assurance that the resulting coverage will be sufficient for any particular purpose.

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. As in previous NIST conformance tests [9], 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 online 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™) [1], then state is implemented with attributes, behavior with operations, and relationships with the standard CORBA Relationships Service [12]. 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 [11], i.e., pointers.

3. PDM Enablers

PDM Enablers is a specification of the Manufacturing Domain Task Force (MfgDTF) [13] 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." [14]

PDME 1.1 defined twelve modules. Three of these – PdmResponsibility, PdmFoundation, and PdmFramework – were "basic" modules, providing common infrastructure for the others: PdmBaseline, PdmViews, PdmDocument Management, PdmProductStructureDefinition, PdmEffectivity, PdmChangeManagement, PdmManufacturing Implementation, PdmConfigurationManagement, and PdmSTEP. Testing the entire specification would not have been feasible with our resources, so we limited our scope to PdmDocumentManagement, PdmProduct StructureDefinition, and portions of the other modules on which they depend.

Section 1.16.2, "Proposed Compliance Points," cast doubt on PDME's testability. It contained 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 specification consisted of IDL, Unified Modeling Language (UML™) [15], and informal descriptions. The language above disclaimed 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 have been useless.
Fortunately, the PDME specification itself supplied usage scenarios in Section 1.12, "Mapping the Product Development Process to the PDM Enablers." These were the scenarios that were used to define the expected functionality of the PDM Enablers [16], so they were presumed to be valid and relevant to the "hot spots" of PDM usage. 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) [17][18], 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 [19] (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.

3.1. 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 [20].

3.2. PDM Schema to PDM Enablers translator

Put simply, the translator input STEP test data and output client-side code for testing a PDM Enablers interface.

The PDM Schema to PDM Enablers translator was based on EXPRESS-X [21] and used the NIST Expresso development environment [22] (see Fig. 3). Expresso processed EXPRESS, STEP's Part 21 file format [23] for instance data, and EXPRESS-X. The EXPRESS-X schema encoded most of the semantic mapping needed for creating calls to the PDM Enablers using a data set described by the PDM Schema, but some finishing touches were made algorithmically in the next stage. The final output of the translator application was PDM Enablers client code in a "macroized" version of the Interface Testing Language used by MCITT.

The macros were expanded using the standard C preprocessor, which also expanded directives to include common test boilerplate (see Fig. 4). A subsequent pass was 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 helped to increase the flexibility of the test system. The PDME specification often specified no normative ordering for the operations that were needed to accomplish a given sub-task, yet implementations in practice did require particular orderings. Macros helped to accommodate the needs of those different implementations. Instead of changing many operations in many tests, we would just change the macro, and the changes would become effective in every instance.

The mapping of STEP PDM Schema data into the PDM Enablers was 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 PDME Enablers were not identical. The PDM Schema included some features, such as approvals, that were omitted from PDME, while PDME included other features, such as extra mechanisms for relating documents to parts, that the PDM Schema did 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.

3.3. Manufacturer's CORBA Interface Testing Toolkit

MCITT, a CORBA testing toolkit with a variety of useful functions, was originally produced in support of the Advanced Process Control Framework Initiative [24] and the National Advanced Manufacturing Testbed Framework Project [25]. In the PDME test building process, MCITT was 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 included yet more boilerplate and expanded some ITL commands to large blocks of CORBA C++.

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

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

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

Version 1.1 IDL, abridged (with exceptions removed):

```idl
interface SecuredFile: File {
    attribute long size;
    attribute MimeTpe type;
    string get_pathname();
    string get_url();
    long begin_put_buffer(in long bufsize, in long filesize, in MimeTpe type, in string transfer_encoding);
    void put_buffer(in buffer buf);
    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) [26], 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.

Fortunately, there was a practical solution. The Comparator already required 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 realized these concepts in the specific context. The problem was easily solved by extending the binding to do arbitrary projections [27] instead of just mappings (see Fig. 6).

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

4. 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.

4.1. Potential testing approaches

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 [28].

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; however, it would be challenging to define a scope for the data that would need to be validated after an incremental change.

4.2. Test system design

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 or foster harmonization with PDME.

A more harmonized approach would be to recycle the test scenarios and even some of the code generation technology used for PDM Enablers testing, but generate test-client code as a series of SDAI calls instead of PDME operations. A prototype test system using this approach was partially implemented.

The prototype test system tested an SDAI interface to the PDM Schema. The scenarios were derived from the PDM Enablers specification and populated with data corresponding to the PDM Schema. These scenarios were executed against an existing, experimental PDM Enablers implementation that used the PDM Schema as its internal data model [29]. Using an embedded semantic mapping, the PDM Enablers implementation mapped calls to the Enablers into operations on the PDM Schema that we could easily translate into equivalent SDAI operations. Our existing PDME test cases were thus transformed into approximately equivalent test cases for SDAI + PDM Schema. The test system is depicted in Figure 10.

The test system generated the basic data access calls as well as data validation calls to check that the system is in a consistent state. Given a complete SDAI binding, the Comparator could then perform an external check of the data values themselves.

5. Improving testability

While constructing the experimental tests for PDM Enablers, we encountered several problematic areas in which the standard had insufficient "constraints on interfaces and implementation characteristics to ensure
that compliance [could] be unambiguously assessed" [30] or to guide the implementors of interoperable clients. After some analysis and discussion with the submitters we found that the problems belonged to only a few classes:

- Accidental ambiguities, where the standard was open to misinterpretation;
- Features whose behaviors were intended to be customized on a vendor- or user-specific basis;
- Interactions with external standards and interfaces that were considered out-of-scope.

Although it was easy to appreciate the need to accommodate both customization and reasonable scope of the standard, the dependency of standard features on features that were customized or out-of-scope damaged the feasibility of interoperable clients (including test clients).

To resolve the conflict over customized features, we proposed the concept of an explicit test mode in which an easily implemented, minimal behavior would be specified for customized features. This would resolve the incompleteness problem in the standard without placing undue burden on implementors.

This recommendation and four others designed to avoid or prevent the testability issues that we encountered have since been documented in a white paper of no official standing, "Improving Testability of Domain Standards," [31] hosted by the Test and Validation Special Interest Group of OMG [32]. The reader is encouraged to read the white paper for maximum clarity and completeness. However, by sacrificing technical precision, the five recommendations can be condensed to the following:

1. Specify an explicit test mode.
2. Do not make standard features dependent on non-standard "input" (properties, etc.).
3. Define minimal test mode behaviors for vendor- and user-specific features.
4. Provide code examples.
5. Completely specify interactions with other components.

Although the test mode behaviors remain to be fleshed out, the means to enable test mode is now included in the working draft of Version 2 of the PDM Enablers standard. Several code examples and a specification of the PDM-Workflow interactions for an engineering change order approval are also in process. A testability working group has been formed to continue this work [33].

6. Conclusion

We have presented a harmonized testing approach that permits the same test scenarios to be used in two different PDM contexts. A semantic mapping between the two contexts enabled testing artifacts developed for one specification to be reused on the other. By testing these specifications together, we supported their harmonization and the productive integration of conforming manufacturing systems.

In addition to demonstrating the value of harmonized testing, this experiment led to the first documented recommendations to help meet the testability evaluation criteria that apply to all OMG standards.

Harmonized testing could be further developed through increased formalization of the semantic map using ontologies. Our institution continues to investigate possible applications of ontologies in the quest to solve the broader semantic problems of interoperability for manufacturing systems.

7. References

8. Figures

![Diagram of Conformance vs. Interoperability Testing with the Comparator](null)

**Figure 1. Conformance vs. Interoperability testing with the Comparator**

![Diagram of Test Client Generation](null)

**Figure 2. Test client generation**

Figure 3. PDM Schema to PDM Enablers translator

Figure 4. Macro expansion and code generation

Figure 5. Compilation and linkage

Figure 6. Projecting an object
Figure 7. Operational model of file exchange

Figure 8. Operational model for data sharing

Figure 9. Exchange-driven approach

Figure 10. Scenario-driven approach