Application Information Mapping Test: An Efficient Content-Level Semantic Equivalence Test Procedure for B2B Integration

Junho Shin*, Jaewook Kim, Nenad Ivezic

Manufacturing Engineering Laboratory, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, USA;
Tel: +001 301 975 3654, Fax: +001 301 975 4482, Email:{ junho.shin, jaewook, nenad.ivezic } @nist.gov

Abstract. This paper describes the Application Information Mapping Test (AIMT) – a type of content-level test procedure for Business-to-Business (B2B) integration across enterprises. The goal of AIMT is to assure that the semantics of a B2B message element are interpreted by the system under test (SUT) according to the element's intended meaning. AIMT provides this assurance by verifying that the element values are processed, stored, and utilized correctly by the SUT to reflect that intended meaning. This paper discusses the AIMT methodology for test development and test execution, which is based on our experience to provide a self-testing capability in a B2B-interoperability project for automotive industry. In particular, we propose an efficient and robust test procedure for AIMT that reduces the time-consuming manual steps and prevents mapping errors. This procedure also proposes a minimum set of test trials by considering practical message constraints of industrial environments. A simulation experiment is presented and results are analysed with respect to the effectiveness of the approach.

Keywords: *B2B interoperability; content-level test; application information mapping test; efficient and robust test procedure*

* Corresponding author. Email: junho.shin@nist.gov

1. Introduction

Business-to-Business (B2B) integration across a supply chain enables a company to focus on its core competencies and offload other services to partners in order to gain efficiencies and reduce costs. An essential ingredient for successful B2B integration is the ability to share information among suppliers, customers, and trading partners. One way of sharing data is to utilize an adapter that creates data maps between two specific application systems (Succi *et al*. 1998). This approach, though it could be useful for small groups of applications, is not efficient for supply chain networks that involve large number of participants where n^2 adapters (i.e., one for each directed connection) need to be developed. Another approach to information sharing is to utilize standard messages that can be commonly understood by all the participants across the supply chain (Succi *et al*. 1998). In this case, each participant who has information in its proprietary representation format transforms its data into a standard document in order to communicate with others.

The approach of utilizing standard messages, however, may incur semantic mapping errors in the message transfer process. The meaning of the elements in the standard document can be interpreted in different ways. For example, an application may interpret ‘Customer’ as a manufacturing plant, while another application may interpret the ‘Customer’ as an OEM that has multiple manufacturing plants. Such inconsistent interpretations of the business data can cause execution of inappropriate business actions.

To reduce semantic ambiguities and prevent semantic mapping errors, message implementation guidelines are specified that describe the meaning of elements in the standard documents. The developers can create the concrete maps between local data and the elements in the standard document based on such guidelines. Nevertheless, it is very hard to produce non-ambiguous specifications due to the lack of reliable and cost-efficient methods to encode data elements and message semantics. Consequently, testing is a crucial activity to interactively resolve any semantic ambiguities in mappings from proprietary to standard messages (Ray 2002). For that reason, we develop methods to facilitate content-level testing for B2B integration.

We explored four types of content-level tests to date: document verification test, application information mapping test, transaction behaviour test, and scenario-based test (Kulvatunyou *et al.* 2005). This paper focuses on the application information mapping test (AIMT) with the objectives to propose a robust and efficient test procedure for AIMT and to recommend a minimum set of test cases sufficient to guarantee that there are no semantic mapping errors. The proposed procedure is a sequence of systematic testing activities that reduces the time-consuming manual steps of previous test procedures and, consequently, increases the practicality of the test. Moreover, the investigation of the minimum set of test cases reduces the number of required trials during the procedure.

In Chapter 2, we first give an overview of content-level tests relevant for a B2B integration project and the types of AIMT with their general procedures. Then, we provide the robust and efficient test execution procedure for AIMT in Chapter 3, where more steps are introduced to prevent the information mapping errors of the fully automated test. In Chapter 4, we address the various types of practical message restrictions of industrial environments, from which a minimum set of test cases can be calculated. Chapter 5 illustrates with an example how test cases are derived based on message restrictions. To analyze the effectiveness of the proposed procedure, we run an AIMT simulation and discuss the results. Finally, we provide concluding remarks and discuss future work in Chapter 6.

2. Overview of Content-level Testing

2.1 Context for Content-level Testing

A notable use of content-level testing occurred in support of conformance and interoperability testing within the STEP (Standard for the Exchange of Product model data) effort. STEP content standards allow the companies involved in the same supply chain to communicate product descriptions electronically and their evolving changes over the product life cycle (Morris *et al.* 1993, ISO 1994, Kindrick *et al.* 1996). While conformance testing determines whether an implementation behaves in a manner defined by a standard, interoperability testing is concerned whether two systems can actually exchange data that supports intended behavior (Kindrick *et al.* 1996).

The STEP testing framework was developed to examine if the system under test (SUT) both (1) correctly processes an input file compliant with the STEP application information model into the SUT's proprietary format and (2) generates an output file compliant with the STEP application information model based on a provided input test file. Essential to these transformations from or to the STEP application information model is the preservation of the semantic content captured in the information model (Kemmerer 1999). The AIMT for general Business Object Documents (BOD) proposed in this paper is a testing approach inspired by and generalized from the STEP testing approach. The AIMT approach was used in supply chain integration projects such as B2B Interoperability Testbed at the National Institute of Standard and Technology (NIST) and Inventory Visibility and Interoperability project initiated by Automotive Industry Action Group (AIAG) (Kulvatunyou *et al.* 2003, Ivezic *et al.* 2004, AIAG 2008, OAG 2008, NIST 2008).

2.2 Application Information Mapping Test

The approach of utilizing standard messages for B2B integration is enabled by the implementation of application interfaces that can process the contents of standard messages. In real situations, it is probable that some message elements are not processed and utilized by the application in accordance with their intended meanings. AIMT assures that the semantics of any message element are interpreted and implemented by a SUT according to the intended meaning of that element. The practical method selected for this verification is to check the mapping between the message element and its corresponding concept in the SUT, defined as data element. AIMT consists of two tests: input test and output test. The input test verifies that the application interface successfully reads and maps the content from the standard input representation into its local representation. Conversely, the output test verifies that the application interface successfully reads and maps the content from the local representation, which was created from an input test file, into the standard output representation. Depending on the role an application plays in the integration scenario, it may require one or both types of tests. For example, only the output test is necessary when a purchasing system generates purchase orders but never consumes them.

(Figure 1) shows the general test procedures for input and output tests under the assumption that the SUT has implemented successfully the interface for generating and consuming the data according to the target exchange specification. For the input test, the input interface receives the standard message and transfers the contents of the message into local data in the SUT. In order to compare the standard message to the local data, the local data are transformed back to the standard message. During the transformation, the test framework provides the *Narrative and questionnaire* interface, where business analysts, acting as testers, answer the questions that assess values of business domain entities and their elements (rather than to request values of the

standardized document elements). On the other hand, the output interface takes the local data and transforms them to a standard message. The local data delivered to the output interface are obtained by transforming the standard input test message for two reasons: it is difficult to generate test instances using the application format and original standard message is necessary for the comparison. During the transformation, the test framework also provides the business analysts with the *Narrative and value* interface (1) to capture the semantics of the business domain entities and their elements and (2) to correctly map the message elements into the data elements in the SUT without the requirement to understand the standard message format. The business analysts then capture the semantics of each element and update the value of the matching element within the local system.

<Insert figure 1 about here>

3. Robust and Efficient AIMT Procedure

Fully automated test procedures that eliminate the manual steps have been developed as shown in (Figure 2). The automated test procedures perform the input and output tests simultaneously under the precondition that a SUT has both input and output interfaces. A standard message instance is transformed into the local data by the input interface and then transformed back into a standard message by the output interface. We can conclude that the SUT passes both the input and output tests if the result of the comparison of two standard messages is “equivalent”. The proposed automated procedures, consequently, enhance the testing efficiency and capability by trying out sufficient number of experiments within a reasonable time period.

<Insert figure 2 about here>

Regardless of its efficiency, in a large number of executions, the automated test also has some limitations of its own. First, the test cannot be employed when the SUT only requires one type of interface, i.e. an input interface or an output interface. Second, the test cannot detect certain types of information mapping errors.

(Figure 3) shows the input and output information mapping errors that may occur. They include (a) the input interface consumes an input element incorrectly while the output interface generates the output element correctly, (b) the input interface consumes an input element correctly while the output interface generates an output element incorrectly, and (c) both the input interface consumes an input element incorrectly and the output interface generates an output element incorrectly. The proposed automated test should be able to detect these types of mapping errors (called “General Type error”). However, the automated test cannot detect mapping errors when there is a coincidental correctness resulting from a symmetrically erroneous mapping as shown in (d) in (Figure 3). Here, we define this type of error as “Type II error” as the test fails to detect an erroneous information mapping (where our nomenclature is only inspired by the hypothesis testing nomenclature).

<Insert figure 3 about here>

In order to prevent Type II errors, the automated test should be accompanied by the input test or the output test. In other words, the automated test should be performed only after one of the input or the output interface is verified. This combination of the automated test with either the input or the output test is still more efficient than conducting the input test and the output test together. One of the practical issues here is how to

minimize the number of test cases whenever we cannot eliminate manual transformations. We will investigate the problem in further detail in Chapter 5. Another practical issue is that a human's misunderstanding of a message element meaning in the input test or the output test can also lead to an incorrect mapping. The errors cannot be eliminated but only be reduced by employing tools such as the narratives and questionnaires that help a tester grasp the meaning of the element. It is assumed in this paper that the incorrect mappings due to human misunderstanding do not occur.

As opposed to recognizing an erroneous mapping as normal one, there can be some cases where the test infrastructure recognizes a normal mapping as erroneous one. Such cases arise when the syntactic representations of the data elements are changed for several reasons as shown in (Figure 4), although the intended semantics is same. The first reason for the change is the data type conversion during the message transfer. An example is when a value of "Date" type is converted into "String" type. The second reason is the different coding schemes, which force the value of an element following one coding scheme to be changed during mapping into, say, a pair of elements following another coding scheme. The third reason is the use of different vocabularies for the same value. For example, a word "Maryland" (a state of U.S.A.) in one system can be represented as "MD" in the other system. The test infrastructure may not have a facility to detect the equality of these alternatively represented values. We call this type of error "Type I error" as the test fails to detect a correct information mapping (i.e., the test fails to accept the true hypothesis that the mapping is correct.)

<Insert figure 4 about here>

A Type I error screening test, thus, is necessary when either the input/output test or the automated test generates the result of "Fail". The screening will determine whether the failure is caused by imperfect test infrastructure (i.e. Type I mapping error) or it is a general mapping error. The screening test checks the meaning of the actual values of two target elements, one from the input standard message and the other from the output standard message. "Pass" in the result of the screening test means "no Type I error", whereas "Fail" means that Type I error occurred and consequently the result of the target test (input/output or automated) should be changed into "Pass". It is assumed in the paper that the analysis of semantic equivalence of target elements is performed manually by a human.

The number of automated tests only needs to be same as or larger than the number of the first conducted input tests (or output tests) in order to detect all the semantic mapping errors present in the mapping interfaces. However, if the purpose is to prevent as many Type I errors as possible and consequently improve the mapping interface, a sufficient number of automated tests for such purpose should be conducted. The increase in the number of test cases basically enhances the Type I error detectability by generating a variety of message instances within the allowable value range. With the detected cases that fall into three cases (See Figure 4), the system interface improvement that reduces the probability of Type I mapping errors can be done by developing an exception-handling module.

An AIMT test procedure designed to prevent Type I and Type II mapping errors and to improve mapping interfaces is shown in (Figure 5) and (Figure 6). The batch test procedure in (Figure 5) assumes that errors are fixed after all the analysis jobs are completed. The incremental test procedure in (Figure 6) assumes that errors found during the test execution are fixed immediately before further analysis. In the batch test procedure, the input/output test and the automated test are conducted in parallel, which results in one of the following conclusions: "No Mapping Errors (characterized by the "Pass" result from the input/output test and

the “Pass” result from the automated test)”, “Type II Detected (characterized by the “Pass” result from the input/output test and the “Fail” result from the automated test)” and, otherwise, “General Type Detected”. Before reaching the conclusion for each test, complimentary Type I screening test is done to check if there is any Type I error in case the test result is “Fail”. If Type I errors are detected (i.e. the result of Type I screening test is “Fail”), the test result should be changed into “Pass” for the case. The errors found in the tests (i.e. Type II errors and General Type errors) are fixed after all the analysis jobs are completed. To illustrate the test procedure, let’s suppose the results of both the input test and the automated test are “fail”. Then, Type I screening test should be conducted for both tests to check if the failures were caused by some changes in representation although their results should have been “pass”. If only the automated test does not pass the screening test (which means “Type I detected” for the automated test), then the result of the automated test should be regarded as “pass” and consequently the final result will be “Type II detected”. The table in (Figure 5) embodies all the possible cases that can occur in the batch test procedure.

<Insert figure 5 about here> <Insert figure 6 about here>

On the other hand, the incremental test procedure first conducts the input or output test and removes all the errors found (i.e. Type II errors and General Type errors) iteratively until no more errors exist. Once all the iterations for the input or output test are completed, the procedure then applies the same iterations to the automated test.

The incremental test procedure is more efficient in general without the chance of committing type II errors. In real practice, however, the batch test procedure would be practical since it requires only one-time correction whereas the incremental test procedure requires high frequency of corrections to fix every problem found at the moment. The communication cost between testing unit and development unit (responsible for fixing problems) is another disadvantage of the incremental test procedure.

4. Optimization of Test Cases for Manual Tests

As mentioned in Chapter 3, minimizing the number of test cases for the input/output test is important in real practice since the human involvement step may force the tester to spend much time on conducting the tests. The essential constraint for the problem, however, is that the test should detect Type II mapping errors. Hence, we can define the problem as follows: “What is the minimum number of input/output tests that guarantee that there is no Type II error?”

In order to detect all potential Type II errors, following test requirements are introduced.

“All message elements should be instantiated at least once in standard messages.”

The test requirement implies that each element of the standard message needs to be checked whether its semantic meaning is preserved in the applications or not. Repetitive verifications with different values may not be necessary since the test is only concerned with the semantic equivalence between message elements and data elements in the SUT.

Basically, one instance can contain all the message elements if there is no particular restriction in message specification. In such an ideal case, the input/output test with one test case is sufficient to ensure no Type II mapping errors. In real cases, however, multiple instances should be populated so as not to violate

message restrictions that exist in message specification. The various types of message restrictions (that prohibit all the message contents from being instantiated only in one instance) will be investigated first in the following paragraphs before summarizing how to obtain the minimum number of test cases.

The first type of message restriction can be defined as “mutual exclusiveness” where one message element should not be included when a particular associated element is instantiated in the message. For instance, message elements C_a and C_b in (Figure 7) cannot be populated simultaneously according to the “mutual exclusiveness”. As a result, one message instance needs to be populated to ensure the mapping between C_a and C, while the other message instance needs to be populated to ensure the mapping between C_b and C. An example of this restriction is the time period, which can be expressed by either the start time and the duration or the start time and the end time.

<Insert figure 7 about here>

The second type of message restriction can be defined as “disaggregation” where one message element can correspond to each of several data elements in a SUT under particular conditions. For instance, as shown in (a) of (Figure 8), the message element B is typically mapped to B_1 in the left case, whereas it is mapped to two elements B_1 and B_2 in the right case. As a result, another message instance needs to be populated to ensure the mapping between B and B_2 in addition to B and B_1. The contents of the message element B should be selected carefully to correctly affect which of the two mappings is used. One possible complex form of “disaggregation” is shown in (b) of (Figure 8), where Elements B and D in a standard message are typically mapped into the elements B and D in a SUT in case element D exists in a message. On the other hand, the element B is mapped into both element B and D in a SUT instead of only B in case an element D does not exist in the message. A real example of this complex “disaggregation” is the situation where “Customer Party” message element that normally describes customer party information is mapped into the “ShipTo Party” element in a SUT to describe ship-to party information omitted in the message. (It is assumed that customer party is same as ship-to party in this business case)

<Insert figure 8 about here>

The third type of message restriction can be defined as “aggregation” where, contrary to the case of “disaggregation”, several message elements can be interpreted as one element in a SUT under particular conditions. For instance, as shown in (a) of (Figure 9), the message element B_1 is typically mapped to B in the left case, whereas the message element B_1 and B_2 are mapped to B in the right case. As a result, another message instance needs to be populated to ensure the mapping between B_2 and B in addition to the mapping between B_1 and B. One possible complex form of “aggregation” is shown in (b) of (Figure 9), where Elements B_1 and B_2 in a standard message are typically mapped into B_1 and B_2 in SUT in case the element C does not exist in the message. On the other hand, all the elements B_1, B_2, and C are mapped into B in case the element C exists in the message. A real example of this complex “aggregation” is the case where an element itself serves as a specific location, but in a combination with another element, it could point to a more specific location.

<Insert figure 9 about here>

Every message restriction that falls into above categories requires multiple instances of messages for the input or output test. Let's define the number of test cases that should be executed by i^{th} restriction as n_i. Then, the minimum number of tests can be defined as,

$$\max\left(n_i \;\; for\ all\ restrictions\ i = 1\ to\ n\right)$$

based on the assumption that the interaction between restrictions does not exist. However, in the case interactions exist between certain restrictions, the number of tests should be the largest product of the number of occurrences n_i where i belongs to R_t, a set of restrictions that have interactions with each other:

$$\max\left(\prod_{i \subset R_t} n_i \;\; for\ all\ restriction\ sets\ R_t\right)$$

The number can be reduced according to the characteristics of interactions on a case-by-case basis, which will be delineated in further detail in Chapter 5.

5. Simulation Study

In this section, we illustrate the overall AIMT test procedure with the test environment. We use a scaled down message specification, *SyncShipmentSchedule*, which is shown in (Table 1), is used to test the input and output interface of a GM (General Motors) application. The message specification is a part of the standard B2B exchange specification developed for the purpose of inventory management among automotive manufacturers and their suppliers. The general meaning of the message is the authorization of the shipment of supplies. The first column in (Table 1) uses an XPATH expression to show field names (see the abbreviation legend at the bottom of the table to expand the field name) (W3C 2008). The second column (UO = Usage Occurrence) indicates the cardinality of each field. The cardinality *1* means the field must occur once and only once; *1+* means the field must occur and may occur more than one time; *0+* means the field may occur zero or more times; *1..n* means the field must occur and may occur up to *n* times; *C* means that the occurrence of the field is dependent upon other fields (other than its parent) and is governed by one or more restrictions indicated in the third column (Field Description). It should be noted that the cardinality of a child field is conditioned upon the cardinality of its parent. For example, although the cardinality of the field *SmSd/SmScHd/ShipTo/PId* is *1* (required), the cardinality of its parent *SmSd/SmScHd/ShipTo* is optional. This means that the field *PId* is required when the *ShipTo* is instantiated.

<Insert table 1 about here>

The first step for AIMT test is to conduct the input or output test together with the automated test since we apply the batch test procedure in Chapter 3 to this simulation experiment. The important factors that should be considered in the step are message restrictions in order to define the test cases and minimize the number of tests. The following four restrictions are employed for the test message.

- Restriction 1: Schedule period can be expressed by either the start time and the duration or the start time and the end time. [Mutual exclusiveness]

- Restriction 2: The *CustomerPId* can serve as the *ShipTo* party information if the *ShipTo* is omitted in the header section. [Disaggregation]
- Restriction 3: Two occurrences of the kanban number (*SmSd/SmScLn/Kanban/Num*) together with the *RangeCode* instantiated indicate that the kanban number is specified in a range. The kanban number, however, can occur two times individually without the *RangeCode* instantiated [Aggregation]
- Restriction 4: Multiple occurrences of the kanban location (*SmSd/SmScHd/ShipTo/Loc*) shall be interpreted as a key combination to identify a specific location, e.g., one location points to "Dock A" and another points to a location within the "Dock A" such as manufacturing "Line 12". [Aggregation]

The resultant minimum number of test cases (computed based on the restrictions) is three, as shown in (Table 2). The computation considering four restrictions and the interaction between restrictions 2 and 4 (the kanban location *SmSd/SmScHd/ShipTo/Loc* is the child element of the ship to party *SmSd/SmScHd/ShipTo*) results in four test cases. The test cases are further reduced to three cases by the characteristics of the child-parent relationship, based on which the child element *Loc* should exist only if the parent element *ShipTo* exists.

<Insert table 2 about here>

The input test was conducted for the three message instances based on the minimum test cases. The automated test was also conducted simultaneously with the input test. As a result of the two mapping tests, two cases (where at least one of the test results are "Fail") were found as shown in (Table 3).

<Insert table 3 about here>

The Type I screening tests were conducted for the cases of "Fail" and the failure of the automated test in the second case turned out to be "Type I error". The Type I error was caused by the representation change of the postprocessor, from "20080701T114718" to "2008-07-01T11:47:18", instead of transformation from "20080701T114718" to "20080701T114718", which results in the mismatch with the input value of "2008-07-01T11:47:18". Consequently, the test results of the second case, after the Type I screening test was complemented, were changed from "Fail/Fail" to "Fail/Pass". On the other hand, the results of the first case still remained the same. The final results for the two cases were all "Fail/Pass", which means Type II errors were detected in both cases. The reason for the Type II error in the first case was the incorrect mapping from *ShipTo/PId/DunsID* in the standard message to *sender/DUNS* in the GM application. The correct mapping was from the *SmSd/SmScHd/CustomerParty/PId/DunsID* to *sender/DUNS* and from *ShipTo/Pid/DunsID* to *plt* in GM application. The reason for the second case was the incorrect mapping from *SmSd/AppArea/CreationDateTime* to *CreationDateTime* element in the GM application. The semantic for C*reationDateTime* in the GM application is the creation time of the kanban, which is contained in the *Kanban/Status/EffectiveDateTime* field of the standard test message.

The interface errors found in the tests were fixed and tested again until no more errors (i.e., Type I errors, Type II errors, and General Type errors) occurred. The result showed no occurrence of General Type errors in this case, which coincides with our expectation that mapping errors usually occur in both input and output interface simultaneously due to the symmetric erroneous mapping. Thus, the proposed procedure for preventing Type I and Type II errors proved practical for the efficient and automated AIMT.

6. Conclusion and Future Work

This paper describes a content-level test method to resolve semantic errors committed during information mapping among applications in the B2B integration situations across supply chains. As one of the content-level tests, the AIMT is used to check that the semantics of message elements are interpreted according to the intended meaning.

The automated AIMT test procedure cannot detect Type I and Type II mapping errors. Thus, the paper suggests an augmented and more robust procedure for the purpose of detecting Type I and Type II errors in which the input test (or output test) and the Type I screening test are conducted in combination with the automated test. In a simulation experiment that applied the test *SyncShipmentSchedule* message to GM application, we showed that the proposed procedure not only elevated the efficiency of the test but also diminished the probability of incorrect decision-making. Moreover, an optimisation effort to minimize the number of manual tests was developed. We proposed an optimisation methodology based on message restrictions, which enabled us to minimize the number of test cases in the input or output test.

Type I errors currently can be detected only by a manual screening test. The module that checks the semantic equivalence between the values of two elements, however, will make the procedure much simpler by blocking the Type I errors beforehand. Further research should also be conducted to prevent incorrect mappings caused by human error in the input and output tests. Additional tools that help a tester grasp the meaning of the elements, such as narratives, should be developed to further reduce the possibility of errors.

Acknowledgments

The authors acknowledge contributions to the earlier versions of the paper by Dr. Serm Kulvatunyou.

Certain commercial software products are identified in this paper. These products were used only for demonstration purposes. This use does not imply approval or endorsement by NIST, nor does it imply these products are necessarily the best available for the purpose.

This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD). (KRF-2007-357-D00280)

References

AIAG, *Automotive Industry Action Group,* Available from: http://www.aiag.org [Accessed April 2008]

ISO 10303-1, 1994. *Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 1: Overview and Fundamental Principles*, Available from: http://www.iso.org/iso/catalogue_detail?csnumber=20579

Ivezic, N., Kulvatunyou, B.S., Frechette, S., Jones, A.T., Cho, H. and Jeong, B., 2004. An Interoperability Testing Study: Automotive Inventory Visibility and Interoperability, In *E-Challenges*, Vienna, Austria.

Kemmerer, S.J., 1999. *STEP the Grand Experience*, National Institute of Standards and Technology Special Publication 939.

Kindrick, J.D., Sauter, J.A., Matthews, R.S., 1996. Improving Conformance and Interoperability Testing, *StandardView*, 4 (1), 61-68.

Kulvatunyou, B.S., Ivezic, N., Martin, M.J. and Jones, A.T., 2003. A Business-to-Business Interoperability Testbed: An Overview, In *The 5th International Conference on Electronic Commerce (ICEC)*, Pittsburgh, PA.

Kulvatunyou, B.S., Ivezic, N. and Jones, A.T., 2005. Content-Level Conformance Testing: An Information Mapping Case Study, *Testing of Communication Systems. Lecture Notes in Computer Science*, 3502, 349-364.

Morris, K.C., Mitchell, M.J. and Barnard, A., 1991. *Validating STEP Application Models at the National PDES Testbed*, NISTIR 4735, National Institute of Standards and Technology.

NIST, *B2B Interoperability Testbed*, Available from: http://www.mel.nist.gov/msid/b2btestbed/ [Accessed April 2008],

OAG, Open Application Group Integration Specification (OAGIS) version 9.0, Available from: http://www.openapplications.org/downloads/oagis/oagis9.htm [Accessed August 2008]

Ray, S.R., 2002. Interoperability Standards in the Semantic Web, *Journal of Computing and Information Science in Engineering*, 2 (1), 65-69.

Succi, G., Valerio, A., Vernazza, T. and Succi, G., 1998. Compatibility, Standards, and Software Production, *Standard View*, 6 (4), 140-146.

World Wide Web Consortium: XML Path Language (XPath) 2.0, Available from:http://www.w3.org/TR/xpath20/ [Accessed January 2008]

Table 1. Message specification of *SyncShipmentSchedule*

Field (XPATH Expression)	UO	Field Description
SmSd/AppArea/CreationDateTime	1	Creation date and time for the message instance.
SmSd/SmScHd/DocumentDateTime	1	Actual date and time of the payload (document) instance or the physical document.
SmSd/SmScHd/CustomerParty/PId	1	Identifier of the customer organization. A customer refers to the party who uses the item. A customer can also serve as the *ShipTo* if the *ShipTo* is not specified.
SmSd/SmScHd/CustomerParty/PId/DunsID	C	Identifier of PId for business (provided by Dun and Bradstreet).
SmSd/SmScHd/ShipTo	C	The *ShipTo*, one of the primary keys of the kanban loop and kanban, includes the information about where the kanban item is used (typically kanbans are managed by location).
SmSd/SmScHd/ShipTo/PId	1	Identifier of the plant.
SmSd/SmScHd/ShipTo/PId/DunsID	C	Identifier of PId for business (provided by Dun and Bradstreet)
SmSd/SmScHd/ShipTo/Loc	0+	Element capturing locations within the parent object such as the Dock, the location (on the manufacturing line) where items are used, a storage location before the items are used, etc. Multiple occurrences of this element do not mean multiple locations but their *Id* fields indicate a key combination that points to a more specific location (i.e., location within another location based on its *Type*).
SmSd/SmScHd/ShipTo/Loc/Type	1	Location function.
SmSd/SmScHd/ShipTo/Loc/Id	1	Identifier of the location.
SmSd/SmScLn/CustomerItemId	1	Identifier provided by the customer for the item in the kanban container.
SmSd/SmScLn/ScPeriod	1	Time window that the truck will arrive for picking up the items to be shipped.
SmSd/SmScLn/ScPeriod/Start	1	Start time of the time window.
SmSd/SmScLn/ScPeriod/End	C	End time of the time window.
SmSd/SmScLn/ScPeriod/Duration	C	Duration of the time window.
SmSd/SmScLn/Kanban	1+	Element capturing information about each kanban.
SmSd/SmScLn/Kanban/Num	1..2	Number of the kanban. The kanban number can occur at most two times, when the user wants to specify a range of kanban numbers using the range code attribute. If it occurs one time, the kanban is specified individually.
SmSd/SmScLn/Kanban/Num/RangeCode	C	Indicator whether the kanban number is the first or last, in case the kanban number is specified in a range.
SmSd/SmScLn/Kanban/Status	1	Status of the kanban, e.g., Empty, Full, Authorized, Shipped.
SmSd/SmScLn/Kanban/Status/EffectiveDateTime	1	The date/time at which the Kanban status was effective.

Field Abbreviations: Sm = Shipment, Sd = Schedule, Hd = Header, Ln = Line, App = Application, Id = Identifier, PId = Party Identifier, Loc = Location, Num = Number

Table 2. Minimum test cases for input or output test

	Restriction 1 (Mutual exclusiveness)	**Restriction 2 (Disaggregation)**	**Restriction 3 (Aggregation)**	**Restriction 4 (Aggregation)**
Test case I	Have *SmSd/SmScLn/ScPeriod/Start* & *SmSd/SmScLn/ScPeriod/End* (No .../*Duration*)	No *ShipTo* field	One occurrence of the Kanban (*Num* field occurs twice with its *RangeCode* field instantiated)	No occurrence of the *Loc* field
Test case II	Have *SmSd/SmScLn/ScPeriod/Start* & *SmSd/SmScLn/ScPeriod/Duration* (No .../*End*),	Have *ShipTo* field	Two occurrences of the Kanban (no *RangeCode*)	One occurrence of the *Loc* field
Test case III	Any of the above	Have *ShipTo* field	Any of the above	Two occurrences of the *Loc* field

Table 3. Two error cases in the simulation experiment

Error cases		Input test / Automated test	Type I screening Test (Input / Auto)	Updated results after Type I scr. (Input / Auto)	Conclusion
Case I	Mapping from *ShipTo/PId/DunsID* to *sender/DUNS*	FAIL / PASS	Pass / -	FAIL / PASS	Type II detected
Case II	Mapping from *SmSd/AppArea/CreationDateTime* to *CreationDateTime*	FAIL / FAIL	Pass / Fail (Type I detected)	FAIL / PASS	Type II detected