Generating a Domain Model from a Use Case Model*

Nayanamana Samarasinghe
SITE, University of Ottawa,
Ottawa, Ontario, K1N 6N5, Canada

Stéphane S. Somé
SITE, University of Ottawa
Ottawa, Ontario, K1N 6N5, Canada

Abstract

In this paper, we focus on the transition from requirements to subsequent phases of software development. We elaborate on creation of a Domain model from an existing Use Case model. A domain model can be seen as an interface or a pipe between the Requirement Engineering phase and the Design Phase of the software development life cycle. Our goal is to automate the extraction of domain information from requirements up to a feasible extent. The approach is integrated to UCEd, a tool for use cases based requirements analysis.

1 Introduction

The use case technique is an effective technique for user's requirements capture and representation. Use cases capture a system behavior as intended by stakeholders under various conditions. Several description formalisms including flow charts, sequence charts, Petri nets are possible for use cases. But because of the specific skills or training needed by these formalisms to be understandable by the common user, use cases are typically represented in a textual Natural Language form. There have been a number of successful attempts of expressing use cases using Natural Language [4, 5, 6]. However, automatic processing of use cases in full Natural Language is far from possible. Reasons for that include ambiguity, dependence on common sense knowledge and contextual nature of a natural language. We developed an approach for use cases based requirements engineering based on a controlled form of Natural Language. Our approach is implemented in a tool called UCEd (Use Case Editor) [1].

UCEd is used to define use cases in a controlled form of English. However, UCEd assumes an existing domain model (a high-level UML class diagram) for the syntactic analysis of use cases [1]. Domain models bridge the gap between the analysis of requirements and the production of design specifications. A domain model represents the common understanding of key concepts in an organization. It is important to note that users are usually unaware of the domain elements at the stage of use cases description.

In this paper, we describe an approach to generate a domain model from use cases defined in Natural Language. The rest of the paper is organized as follows. The next section describes the existing framework used for use case capture. In this section we focus on how the use cases and domain elements are captured using the UCEd tool, and elaborate the formal grammar used for modeling these elements. Subsequent section explains our work on extracting the domain elements from an existing use case model. In the latter part of the paper we discuss the related work that has been done in comparison to our current work. Finally the conclusion of this paper will consist of a summary of objectives achieved and any possible future work (based on our current work).

2 UCEd Process

Figure 1 describes our use cases based requirements process as supported by the UCEd.

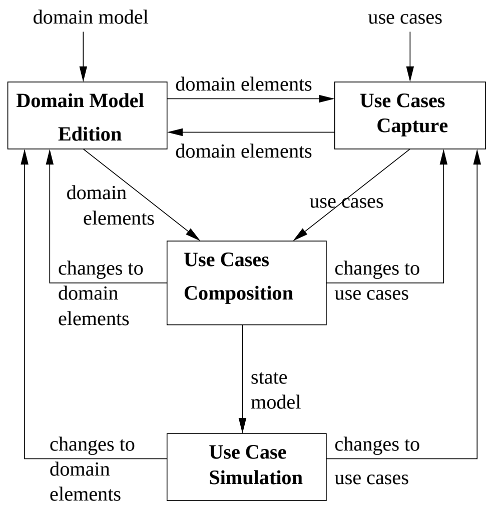
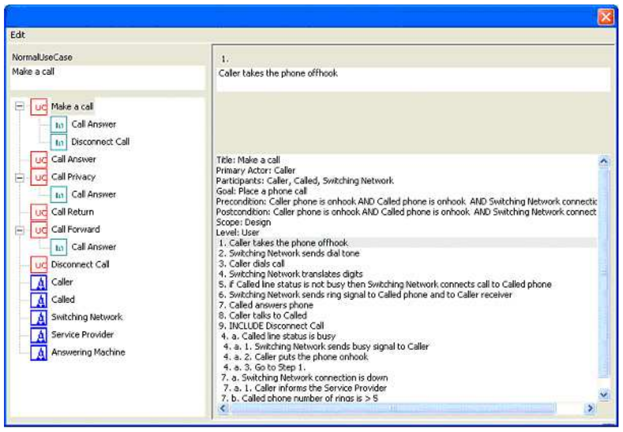


Figure 1: UCEd requirements engineering process.

*Work supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

Use Cases are captured in restricted form of natural language through a specific interface of UCEd. Figure 2 shows a use case being edited with UCEd. The grammar used for our restricted form of natu-



Edit

NormalUseCase
Make a call

Make a call
Call Answer
Disconnect Call
Call Answer
Call Privacy
Call Answer
Call Return
Call Forward
Call Answer
Disconnect Call
Caller
Called
Switching Network
Service Provider
Answering Machine

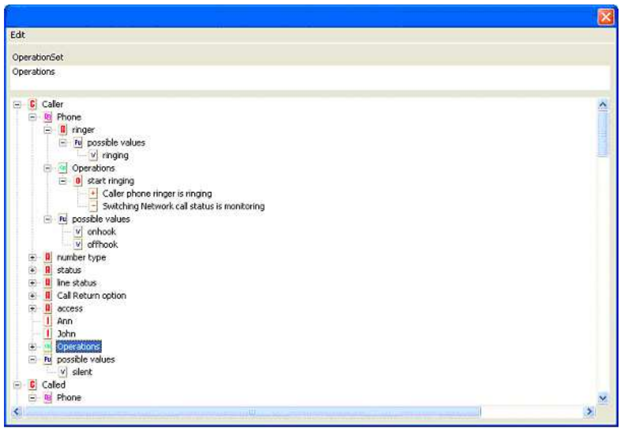
1.
Caller takes the phone offhook

Title: Make a call
Primary Actor: Caller
Participants: Caller, Called, Switching Network
Goal: Place a phone call
Precondition: Caller phone is onhook AND Called phone is onhook AND Switching Network connectic
Postcondition: Caller phone is onhook AND Called phone is onhook AND Switching Network connect
Scope: Design
Level: User
1. Caller takes the phone offhook
2. Switching Network sends dial tone
3. Caller dials call
4. Switching Network translates digits
5. if Called line status is not busy then Switching Network connects call to Called phone
6. Switching Network sends ring signal to Called phone and to Caller receiver
7. Called answers phone
8. Caller talks to Called
9. INCLUDE Disconnect Call
4. a. Called line status is busy
4. a. 1. Switching Network sends busy signal to Caller
4. a. 2. Caller puts the phone onhook
4. a. 3. Go to Step 1.
7. a. Switching Network connection is down
7. a. 1. Caller informs the Service Provider
7. b. Called phone number of rings is > 5

Figure 2: Extract of the Use Case Edition interface.

ral language is described in Section 4. UCEd checks that the corresponding domain entities and operations are defined in the domain model and reports inconsistencies as well as omissions. We introduce a typical telephone PABX system as a case study. The system includes the domain entities: Caller, Callee, Switching Network, Answering Machine and Service Provider. Use cases for this system can be defined for making a call, answering a call, returning a call, forwarding a call and disconnecting a call. The use case edition interface allows creation of a Use Case Model with a list of use cases and actors. A detailed description can also be added to each use case using the interface. In accordance with the UML recommendation [3], use cases can be linked by "include" and "extend" relations. As an example, in Figure 2 step 9 includes use case "Disconnect Call". A use case description is a sequence of steps. Each step involves an operation from the environment or the system and may have one or more extensions. As an example, step 4 involves operation "translate digits" from "Switching Network" and includes extension 4.a. Steps and extensions may have guards. A guard is some condition that must be satisfied prior to the execution of the use case step. In step 5, "Caller line is not busy" is a guard conditions. Each use case defines a part of the system behavior as sequence of events that may follow each other. We use the term "scenario" to refer to these sequences. A use case typically includes a "main scenario" and a set of "secondary scenarios".

A Domain Model is a high level UML diagram that captures the domain entities and their operations. Figure 3 shows an excerpt of the phone system domain model in UCEd Domain Edition Tool. We distinguish



Edit

OperationSet
Operations

Caller
Phone
ringer
possible values
ringing
Operations
start ringing
Caller phone ringer is ringing
Switching Network call status is monitoring
possible values
onhook
offhook
number type
status
line status
Call Return option
access
Ann
John
Operations
possible values
silent
Called
Phone

Figure 3: Extract of the Domain Edition Tool.

the following types of domain entities.

1. Concept: These are the most important type of entity in the context of the system. There could also be concepts defined under a parent concept (i.e. a sub-concept) to represent specialization or inheritance behaviour.

2. System Concept: The system under consideration that is represented as a black box.

3. Aggregation: Used to represent the "part-of" relationship with the participating parent concept or system concept.

4. Attribute: A feature of a concept, system concept or an aggregation.

5. Object: An instance of a concept or system concept.

It is important to note that Concepts, System Concepts, Aggregations and Attributes may have a set of possible values (state values) assigned to them. As an example, possible values of entity "Phone" are "onhook" and "offhook". Each concept, system concept or aggregation may also have a set of operations. Domain operations are declared in the format: "<action_verb> [action_object]" where <action_verb> is an infinitive verb. An example would be "start ringing".

The use case composition module deals with state model generation. It uses a state model generation algorithm described in [2]. The partial behavior of each use case is merged in an incremental manner to obtain a global state model. Use case composition is based on operation effects specified as added and withdrawn conditions. Generated state models are used as prototypes to validate the original use cases and to uncover any possible interactions between them [1].

3 Use Case Grammar

Domain elements extraction rely on a syntactical analysis of use cases. In this session, we present our restricted form of natural language grammar for use cases. The two parts of use cases that are expressed in natural language are conditions and operations.

"Conditions are predicative sentences describing situations prevailing within the system and its environment" [2]. Conditions are used to specify operations added and withdrawn conditions in the domain model. They are also used for preconditions, postconditions and guards. Figure 4 introduces a concrete syntax for conditions. An example of a condition is "Caller phone is onhook" which is a precondition in our phone system example.

```
<condition>   -> <acondition> "and" <condition>
  | <acondition> "or" <condition>
  | "("<condition>")"
  | <negation> <condition>
<acondition>  -> [<determinant>] <entity>
                 [<verb>] <value>
<determinant> -> "a" | "an" | "the"
<negation>    -> "not" | "no"
<entity>      ->  <concept> | <attribute>
<concept>     -> (<word>)+ {member of the model
                             concepts}
<attribute>   -> (<word>)+ {attribute of concept}
<verb>        -> {derived from to be in present
                  tense}
<value>       -> (<word>)+ {value of the entity}
 | <comparison> {entity is non-discrete ?}
<comparison>  -> <comparator> <word>
<comparator>  -> ">" | "<" | "=" | "<=" | ">="
 | "<>" | "greater than" | "less than"
 | "equal to" | "different to"
 | "greater or equal to" | "less or equal to"
```

Figure 4: Grammar for conditions.

Figure 5 describes use case operations syntax. A use case operation reference is the specification of an operation executed by a domain concept. The specification of a use case operation uses a conjugated form of the infinitive form of the operation declaration. For sake of simplicity, only the present form is used. As an example, "The Caller Phone starts ringing" is a use case operation statement where: the concept performing the operation is "Caller Phone", the operation is "starts ringing" a conjugated form derived from declaration "start ringing".

```
<operation_spec> -> <concept_operation>
  | <branching_statement> | <useCase_inclusion>
<concept_operation> -> [<before_delay>]
                       [<after_delay>]
                       [<condition_spec>]
                       <operation_reference>
<condition_spec> ->  "IF" <condition> "THEN"
<operation_reference> -> (<word>)+ {derived from
             an operation of the current entity}
<after_delay>  -> "AFTER" <duration_spec>
<before_delay> -> "BEFORE" <duration_spec>
<duration_value> -> <duration_value>
                    <duration_unit>
<branching_statement> ->
    "GO" "TO" "Step" <word> {corresponding to
                             a step label}
<useCase_inclusion> -> [<condition_spec>]
                       "INCLUDE" <use-case-name>
```

Figure 5: Grammar for use case operations.

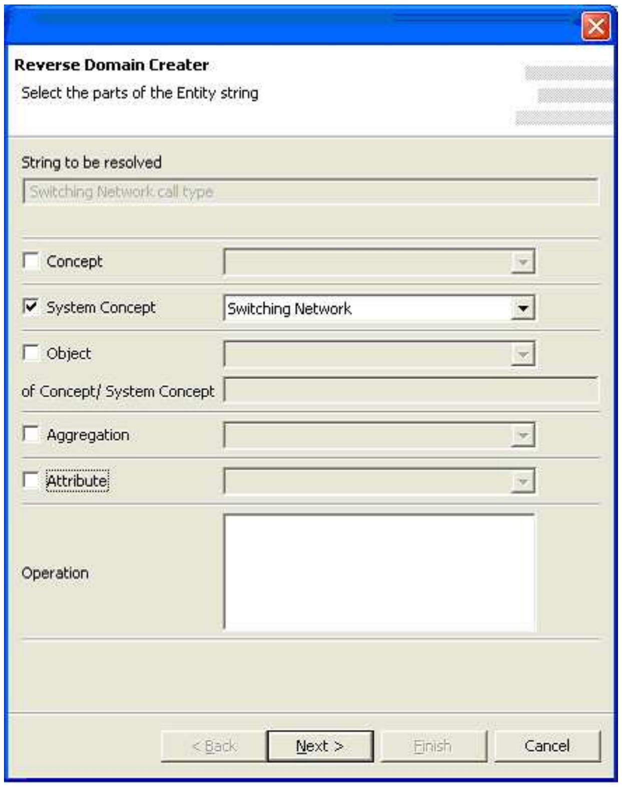
4 Domain Elements Extraction

The objective of domain extraction is to allow users to enter a set of requirements in the form of use cases and derive the corresponding domain entities. Notice that at the present time, domain specification is completely manual. Extraction of domain elements would ease UCEd requirements definition process by removing the necessity for users to go back and forth creating missing items in the domain during use case composition. We propose a semi-automated extraction mechanism. A fully automated approach comes up against the inherent ambiguity of natural language.

Figure 6 describes an algorithm for domain elements extraction from use cases. Let C, A and O be sets of non-existing Conditions, Actors and Operations respectively in the use case model. As explained in the domain grammar section a condition is in the form "<entity> <verb> <value>" if we omit the optional article for clarity. Also an operation reference can be defined as "<concept_name> <conjugated_action_verb> [action object]". Here too we omit [article] and [preposition action participant] for the sake of simplicity. According to step 2 the algorithm detects known sub-entities that are part of condition (if any) and prompts the user for any undefined sub-entities in order to ascertain its type. Figure 7 shows an example in this regard. In this example the system has detected that "Switching Network" already exist as a System Concept in the domain. As such it is now the user's turn to resolve the remaining part of the string: "call type". It could be or a part of an object, aggregation or an attribute. During this process user selections are limited to the possibilities,

```
1. For each use case $uc$,
    1.1 For each condition c $\in uc$
            If the entity of c, e exists and the value of
            c, v does not exist in the domain,
            Add a possible value v to e in the domain
            Else if e does not exists in the domain,
            $C = C \cup \{c\}$
    1.2 For each operation reference o in $uc$
        If o is not defined in the domain,
        $O = O \cup \{o\}$
    1.3 For each actor a in $uc$,
        If (a is not defined as a domain entity in the
        domain), $A = A \cup \{a\}$
2. For each element c in C,
   Let e be the entity part of c
    - Extract the part of e that has been already defined in the domain and prompt user to select
      the type of the sub entities for the rest.
    - Add the sub-entities to the domain
    - Insert v $\in$ condition value to the referring entity
3. For each element a in A that is still not in the domain,
    - Prompt the user for the type of entity
    - Add the entity to the domain
4. For each element o in O,
   For each token t in o,
    - If (t is a verb and a starting point of a logical
      operation)
            op = part of the phrase that starts from t
      Else
            Prompt the user to select the appropriate
            value for action specification, op
    - Process the infinitive form of op. Remaining
      part of o will contain the relevant entity for op,
      say e (i.e. first part after removing the action
      specification part)
    - Extract the part of e that has been already defined in the domain and prompt user to select
      the type of the sub entities for the rest.
    - Add the sub-entities to the domain
    - Insert the infinitive form of op to the referring
      entity of e as a domain operation
```

Figure 6: Algorithm for the domain elements extraction process.



Reverse Domain Creater

Select the parts of the Entity string

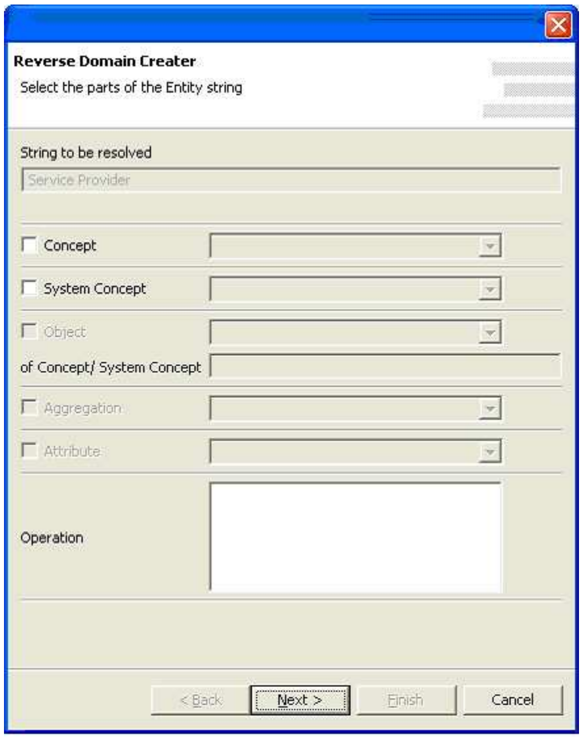
String to be resolved: Switching Network call type

- [ ] Concept
- [x] System Concept: Switching Network
- [ ] Object
- of Concept/ System Concept:
- [ ] Aggregation
- [ ] Attribute

Operation:

Figure 7: System prompt to resolve an entity in a condition.

Concept, System Concept, Object, Aggregation or an Attribute. In Step 3 of the algorithm, we prompt the user if the corresponding use case actor is not yet defined in the domain. An actor may be a Concept or a System Concept and will not be part of anything else. Finally, step 4 resolves use case operations. The algorithm considers the first word from the left of the string. If it is a verb, this word is marked as a start of an operation specification (automatic operation handling). In a case where the system does not find any verb in the operation string the user is prompted for the operation (manual operation handling). This way of automatic handling in order to ascertain the operation specification is rather tricky. The reason is that certain words appear in more than one form (i.e. noun, verb, adjective, etc.) Therefore what the system detects as the operation specification is not guaranteed to be accurate. In such a situation the user is given the freedom to make the correct choice. For example if the operation string given is “User Display blinks fast” where “User” is a concept, “Display” is a component of User and “blinks fast” is an operation, we will detect “Display blinks fast” as the operation since “Display” is a valid verb and is at the same time, the first verb encountered from the left of the string. We use the WordNet dictionary [7] to determine whether a word is a verb or not. Figure 8 shows an example for the system prompt for operation "Caller John places the phone offhook". In this case the system will detect


Reverse Domain Creater

Select the parts of the Entity string

String to be resolved

Service Provider

Figure 8: System prompt to resolve use case operations.

the action specification or the operation as "places the phone offhook". In the remaining part of the operation, "Caller" has already been identified as an existing Concept of the domain. But we might need to define what "John" is, which is an instance (or object) of concept "User". This is something similar to step 2 of the algorithm. Afterward the system attempts to insert the operation under the appropriate referring entity.

The most interesting part in the domain elements extraction process is deciding on a way to assign the relevant parts of a string representing an entity to respective entity types. For example consider string "Switching Network call type". We had to come up with a method that facilitates selection of any possible combination of words for a sub-entity type in this entity string. In this example the System Concept can be taken as "Switching" or "Switching Network" (which is the correct choice). The approach used to overcome this issue was to simply create all possible of combination of words from a given phrase. This will facilitate to select the appropriate sub-entity types from a given phrase. Given a string `w1 w2 w3 w4` $\cdots$ `wn` delimited by spaces where `w1`, `w2`, `w3`, `w4`, $\cdots$, `wn` are the words that consists in the string, we can derive all possible combination of words of the string as follows,

```
w1, w2, w3, w4, ···, wn
w1 w2, w2 w3, w3 w4, ···, wn-1 wn
w1 w2 w3, w2 w3 w4, ···, wn-2 wn-1 wn
·········
w1 w2 w3 w4 ··· wn
```

These combinations of words will have any pattern that we may need to choose for a particular selection of a sub-entity type of an entity string. As an example, suppose the entity string: "Switching Network call type", using the above notion we can derive the following word combinations:

Switching, Network, call, type
Switching Network, Network call, call type
Switching Network call, Network call type
Switching Network call type

But the correct sub-entity selections "Switching Network" (sub-entity type: System Concept) and "call type" (sub-entity type: Attribute) are actually a subset of the derived word combinations above. Thus this notion can be used to identify any sub-entity type of a given entity string.

As a case study, we created a use case model for our telephone PABX system example as mention in section 4 using the UCEd tool. The use case model that was built using the tool is shown in Figure 2. Initially there were no domain elements defined in the domain model. Afterward we ran the "Domain Extracting tool" on the existing use case model. After completion of this process we observed that a set of domain elements get added to the domain model as could be seen in Figure 3. In order to verify whether the created domain using this semi-automated process is valid, we validated both the domain and use case models and observed that the validation process for both these models is successful.

5 Related Work

One interesting research that has some similarity to our work is extraction of domain models out of existing code for generative reuse [8]. In this work, the authors discuss reverse engineering of a domain model from an existing code base. The recovered domain model is subsequently used to develop application generators.

Other related works are attempts to extract Sequence Diagrams [3] from use case models in order to validate use cases based requirements [9]. Sequence

Diagrams are merely representations of interactions between actors and a proposed system expressed as message exchanges. [9] states that these sequences are transformed in to comparison sequence diagrams, which in turn can be validated with a component based simulation model to verify the conformance of the system with its requirements. However this compliance checking process is not automated at the moment and the papers suggests that it should be possible to automate it in the future. Although this approach does not suggest a domain extraction from a use case model it highlights another possibility of an automated process that can be applied to a use case model, which would be much useful in the field of Requirement Engineering.

6 Conclusion

In this paper, we discussed domain model elements extraction from use cases. A domain model can be treated as a lexicon for use case description. A domain model also acts as an interface between the requirement analysis phase and the design phase. It can be used effectively as a basis for software design allowing thus a smooth transition from the Requirement analysis phase to Design phase.

We introduced and approach that automates a domain model creation given a use case model. This is an enhancement to a process where the insertion of the corresponding domain elements in the domain model was manual, and consequently error prone and time consuming. Because of the inherent ambiguity of Natural Language, complete automation of the extraction process is unfeasible. For instance, it is impossible to automatically ascertain the sub-entity type of an entity (i.e. whether it is a concept, system concept, aggregation, attribute, etc.). This paper however, suggests the feasibility of a semi-automated process. We also take the point that a right user interface can ease the domain extraction process.

Our approach of domain model extraction is specific to the UCEd framework. The approach could be generalized to use case models created from any tool with an exchange format such as XMI [10].

References

[1] Use Case Editor (UCEd) toolset, http://www.site.uottawa.ca/~ssome/Use_Case_Editor_UCEd.html.

[2] S. Somé, An approach for the synthesis of State transition graphs from use cases, Proceedings of the International Conference on Software Engineering Research and Practice (SERP), Volume. I, pp 456-462, 2003.

[3] OMG, OMG Unified Modeling Language Specification version 1.5 (2004) http://www.omg.org/technology/documents/formal/uml.htm.

[4] K. Boettger, R. Schwitter, D. Mollà, and D. Richards, Reconciling Use Cases via Controlled Language and Graphical Models, Web-Knowledge Management and Decision Support, Lecture Notes in Computer Science, Vol. 2543, pp. 115-128, Springer Verlag, Heidelberg, Germany, 2003.

[5] A. Fantechi, S. Gnesi, G. Lami, A. Maccari, Application of Linguistic Techniques for Use Case Analysis, Requirements Engineering Journal Vol. 8, Issue 3, pp 161-170, Springer-Verlag, 2003.

[6] Colette Rolland, Camille Ben Achour, Guiding the Construction of Textural Use Case Specification, Data & Knowledge Engineering Journal, Elsevier Science Publishers, Vol 25, N 1-2, pp 125-160, March 1998.

[7] WordNet lexical database for English Language http://www.cogsci.princeton.edu/~wn/.

[8] P. Devanbu and W. Frakes. Extracting formal domain models from existing code for generative reuse. ACM Applied Computing Review, 1997.

[9] Wolfgang Fleisch, Applying use cases for the requirements validation of component-based real-time software, Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, p. 75, Saint-Malo, France, May 02-05, 1999.

[10] An Overview to the XMI - XML Metadata Interchange Specification http://www.omg.org/news/pr99/xmi_overview.html.