Transformation of UML Interaction Diagrams into Contract Specifications for Object-Oriented Testing

Huo Yan Chen, Chuang Li, and T. H. Tse, *Senior Member, IEEE*

***Abstract*—Testing is an important means to ensure the quality of software systems. Contract specification can be used to formally specify the cluster level of object-oriented software, which can then be tested using TACCLE, an advanced methodology for object-oriented testing. The use of formal specifications as a testing base has many advantages. However, such specifications are not easily understood and therefore not widely used in the software industry. On the other hand, UML, a semi-formal modeling language, is becoming increasingly popular and widely accepted. In particular, UML interaction diagrams specify the dynamic, interacting behavior among the objects of an object-oriented system. If the transformation of UML interaction diagrams into Contract specifications can be automated, the TACCLE methodology can be applied directly to test object-oriented software at the cluster level. In this paper, a method to transform UML interaction diagrams into Contract specifications is proposed based on the UML meta-model. A prototype has been developed.**

I. Introduction

Formal specifications are precise and unambiguous, and facilitate verifying, testing, deduction, and code automation. *Contract* is a formal specification language for defining interactions in object-oriented software. It "captures explicitly and abstractly the behavioral dependencies amongst cooperating objects" [1]. It consists of three parts: (a) a set of *communicating participants* with their *type obligations* and *causal obligations*, (b) *invariants* that participants must maintain via cooperation, and (c) *pre-conditions* and *operations* that instantiate the behavior. Contract has aroused a lot of attention by researchers. For example, Google Scholar reports 426 citations for [1].

In [2], we proposed a systematic methodology known as *TACCLE* for the testing of object-oriented software. The methodology was successfully applied to a technology-transfer project for ASM, the world's largest supplier of assembly and packaging equipment for the semiconductor industry [3]. In particular, Contract specifications are used in TACCLE for specifying cluster level behavior among interacting objects.

Manuscript received February 2007. This work was supported in part by the National Natural Science Foundation of China under Grant #60173038, the Guangdong Province Science Foundation under Grant #010421, and the Research Grants Council of Hong Kong under CERG Grant #714504.

Huo Yan Chen and Chuang Li are with the Department of Computer Science, Jinan University, Guangzhou 510632, China (phone and fax: +86 20 8522 0226, e-mail: tchy@jnu.edu.cn).

T. H. Tse is with the Department of Computer Science, The University of Hong Kong (e-mail: thtse@cs.hku.hk).

However, as a typical formal specification language, Contract is not easily understood by developers and not widely used in the software industry.

On the other hand, Unified Modeling Language (*UML*), a visual modeling language, is attracting more and more attention, and has become a de-facto standard in the industry. It has received wide acceptance because of its ease of understanding and ease of use. Since it was created as a semi-formal modeling language, UML does not include a formal semantics. This makes rigorous analyses difficult [4].

UML comprises various diagrams. Among them, sequence diagrams and communication diagrams [1], which specifying the interacting behavior among objects of a system, are collectively known as interaction diagrams. We propose to transform UML interaction diagrams into Contract specifications so that systems specified by UML is amenable to rigorous analyses, verification, and testing using established formal techniques such as TACCLE.

In general, there are two main approaches for diagram transformation, as highlighted by Solenon et al. [5]: a push approach where features of the source diagrams are considered in turn, and a pull approach where the features of the target diagrams are considered in turn. In our method, we use the push approach.

Several techniques have been proposed for transforming UML diagrams into formal specifications. Some of them, such as [6, 7, 8], are based on the UML meta-model; some methods, such as [9], are based on an extended UML meta-model; other methods such as [4] are not based on the meta-model.

UML has a layered architecture based on a four-tier structure. Its semantics is mainly specified in the meta-model layer. This layer is independent of implementation details. The UML meta-model constitutes the foundation of model interchange, reuse, and interoperability among tools. Hence, we have decided that our transformation technique should be based on the UML meta-model.

The UML meta-model consists of three main parts: semantics, notation, and standard profile [10]. *UML semantics* define the UML model elements, their relationships, and constrains. *UML notation* specifies the graphic syntax for expressing the semantics. UML standard elements and extension mechanism are explained in the *standard profile*. The UML notation comprises various diagrams. The diagram elements are visual representations of

[1] Formerly known as collaboration diagrams.

UML model elements. They must fulfill the well-formed rules defined in the semantics of UML before the diagrams can express the defined meanings. Hence, a single isolated diagram does not comply with UML.

In [11], Chen has proposed high-level guidelines to transform UML interaction diagrams into Contract specifications. Our present method conforms to these guidelines.

The rest of the paper is organized as follows: Background knowledge of Contract specifications and UML interaction diagrams are introduced in Section 2. Section 3 proposes a generic algorithm for transforming UML interaction diagrams into Contract specifications. Considering the fact that different UML tools implement different subsets of UML, a specialized algorithm is presented in Section 4 to fit a sample implementation. In Section 5, we discuss an implementation of our algorithm based on XMI. Section 6 concludes the paper.

II. BACKGROUND KNOWLEDGE

This section introduces the basic knowledge of Contract specifications and UML interaction diagrams.

A. Contract Specifications

The following is a sample Contract specification taken from [2]:

```
1   contract CustomerAccount
2   Customer supports
3   [
4   address : String
5   accounts : Accounts
6   Customer ← setAddress(S : String) =>
        @Customer.address; {Customer.address = S};
        Customer ← notify().
7   Customer ← getAddress() => return Customer.address.
8   Customer ← notify() =>
        (/Ac : Ac in accounts : Ac ← update()).
9   Customer ← openAccount(Ac : Account) =>
        {Ac in accounts}.
10  Customer ← closeAccount(Ac : Account) =>
        {Ac not_in accounts}
11  ]
12  Accounts : SetOf(Account) where each Account
        supports
13  [
14  customer : Customer
15  freeze : Boolean
16  Account ← setFreeze(B : Boolean) => @Account.freeze;
        {Account.freeze = B}.
17  Account ← update() =>
        if Account.freeze then return "Account is frozen"
        else Account ← changeAddress().
18  Account ← changeAddress() =>
        customer ← getAddress();
        {Account reflects customer.address}.
19  Account ← setCustomer(C : Customer) =>
        {customer = C}
20  ]
21  instantiation
        (/Ac : Ac in Accounts :
            (Customer ← openAccount(Ac) /
            Ac ← setCustomer(Customer)))
22  end contract
```

The example shows a cluster CustomerAccount which comprises a Customer class and an Accounts class. Each instance of Accounts is a collection of accounts that belong to the same customer (such as saving account, check account, and fixed deposit account).

For the easy of reference, we label each statement with a line number. Reserved words of the Contract language are typeset in bold. Line 1 indicates that the *name* of the Contract specification is CustomerAccount. Lines 2 and 12 show that there are two *communicating participants* in this Contract: Customer and Accounts. Their *type obligations* are shown in lines 4 to 5 and lines 14 to 15, respectively; and their *causal obligations* are shown in lines 6 to 10 and lines 16 to 19, respectively. Line 21 shows the *preconditions* and *operations* that instantiate the contact. There is no invariant in this contract.

In Contract specifications, the main part is the causal obligations. Each line of a causal obligation is a message-passing rule that explicitly expresses a message passed between participants as well as the post-conditions or related actions (such as the return of values) on acceptance of the message by the receiving object [2]. For example, line 6 means that, when an object of the Customer class receives a message setAddress(S : String), it will set the value of the attribute Customer.address to S, as indicated by the post-condition "{Customer.address = S}", and send a message to this object, as indicated by "Customer ← notify()". In this way, Contract specifications explicitly show the behavioral dependences between the participants specified. For more details, please refer to [1, 2, 11].

B. UML Interaction Diagrams

UML notation defines various diagrams to help specifying software systems. They provide multiple perspectives of systems under analysis or development [10]. These diagrams are based on the main modeling concepts of the language defined in UML semantics as model elements at meta-model level. There are mapping relationships between diagram elements and the model elements. The diagrams express the model elements and their relationships in a graphic manner.

However, the diagrams are not formally defined in UML. There are presentation options left for users or tool developers to choose from. This is one of the reasons why we base our transformation on the meta-model of UML rather than the diagrams themselves.

UML uses a UseCase to specify the scenarios of a system. The external environment interacting with the system (such as users or other systems) is expressed by an Actor. A UseCase can be further refined to a set of UseCases. The realization of a UseCase can be specified via the notion of Collaboration. The structure of the participants that play the roles in the performance of a specific task and their relationships is called a *Collaboration*. The roles are specified by ClassifierRoles. Their communication pattern is called *Interaction*. An Interaction is defined in the context of a Collaboration. Collaboration and Interaction then give out the two aspects of the description of a *behavior*.

In the graphic counterpart, *interaction diagrams* including sequence diagrams and communication diagrams express the behavior of a system. A *sequence diagram* shows the explicit sequence of interactions, while a *communication diagram* shows the participants of an interaction and their relationships. They share common meta-models with different emphases. Sequence diagrams emphasize the sequential order. Communication diagrams emphasize the structure of the collaborators and their associations. These two types of diagram specify behavior of objects in a system complementally.

Sequence diagrams and communication diagrams may be drawn in two forms: the specification level and an instance level. There is no difference, however, as far as our transformation is concerned.

Our transformation uses the following meta-models of UML: UseCase, Collaboration, Interaction, ClassifierRole, Classifier, Signal, Message, Procedure, Action, Parameter, Reception, and Feature. Formal definitions of the meta-models and detailed mappings of the diagram elements to the meta-models can be found in [10].

III. Generic Algorithm for the Transformation

For any UseCase of the UML model of a system, an Interaction represented by an interaction diagram (which may be a sequence diagram or a communication diagram) can be transformed into a Contract specification using the following transformation algorithm:

Algorithm 1

(1) Take the Interaction name as the Contract name.

(2) For every ClassifierRole in this Interaction that is not an Actor, take the name of its base Classifier as the name of the Contract participant.

(3) Suppose the name of the participant is NM. If the attribute Multiplicity of any ClassifierRole is not 1, then use the following string instead of the participant name:

NMs: **SetOf**(NM) **where each** NM **supports**;

(4) For every ClassifierRole of the Interaction, take the attributes of the Feature associated with it as the corresponding type obligations in Contract.

(5) For every Message received by every ClassifierRole of the Interaction, if it is a Signal, then the reaction of this Message is decided by the attribute Specification of the Reception associated with the Signal.

(6) Otherwise, if the Message is the invocation of a procedure, then the Messages within this Interaction whose Activator is the current Message form the "resulting messages" of the current Message, and the order of the "resulting messages" is decided by the Predecessor relationships among them. In other words, if Message *A* is Predecessor of Message *B*, then *A* is listed before *B*.

(7) For every Message in the "resulting messages" in (6), if the attribute Multiplicity of the Receiver is not 1, then when this Message is listed in "resulting messages", it should be transformed into a repeating form. Suppose the Message is $M_i()$ and its Receiver is ClassifierRole_*k* whose base Classifier is Classifier_*k*. Then the Message in "resulting messages" is changed to

(/V : V **in** Classifier_*k* : V $\leftarrow M_i()$);

(8) For every Message in the Interaction, if there are preconditions revealed by the attribute Body of its corresponding Procedure or by the corresponding detailed Action model, then add "**if** (preconditions) **then**" before this Message when it is included in "resulting messages". If post-conditions are revealed, then add "{post-conditions}" at the end of the "resulting messages" corresponding to this Message.

(9) For every Message in the Interaction, if it has Parameters and the attribute Kind of one Parameter is "return", suppose the attribute Name of the Parameter is "*pname*" and the base Classifier of the Receiver of this Message is Classifier_*j*. Then add the following clause at the end of "resulting messages" corresponding to this Message:

return Classifier_*j*.*pname*.

IV. Specialized Algorithm for Specific Modeling Tool

The algorithm given above is a generic algorithm. However, most of the existing UML modeling tools do not adhere to the original semantics of UML. When applied to a specific modeling tool, the algorithm should be adapted to follow the vendor-dependent semantics. We choose the UML case tool

IBM Rational Rose as an illustrative example owing to its widespread use in the software industry.

In Rational Rose, the label of a message only contains the sequence number and message name. The message name is mapped to the operation name of the class receiving the message. For consistency of modeling, we cannot add additional information on the label. A solution is to express the preconditions and post-conditions in the specification dialog box of the corresponding Operation. This is a use of the extension mechanism of UML. The preconditions and post-conditions are expressed in the attribute "DataValue" of TaggedValue at the meta-model level.

Thus, the actual transformation algorithm is summarized as follows.

Algorithm 2

This algorithm is the same as **algorithm 1** except step (8), which is replaced by the following, since only this step needs to be specialized:

(8) For every Message in the Interaction, if the TaggedValue associated with the Operation corresponding to the Message is not empty, and if the attribute "DataValue" of the TaggedValue is a logical expression (which signifies preconditions), then add "**if** (preconditions) **then**" before this Message when it is included in "resulting messages". If there are post-conditions in TaggedValue associated with the Operation corresponding to the Message, then add "{post-conditions}" at the end of the "resulting messages" corresponding to this Message.

V. A PROTOTYPE

We have developed a prototype of our algorithm based on XML Metadata Interchange (XMI).

A. Algorithm for the Prototype Based on XMI

As various UML case tools implement the language in different ways, problems may occur during model interchange between different modeling tools. OMG introduces XMI as a standard to ease this problem. XMI allows metadata to be interchanged as streams or files with a standard format based on XML [12, 13]. Its textual form further eases the process. Many UML modeling tools have plug-in programs to help convert models to XMI files. For Rational Rose Enterprise Edition, for instance, we can use Unisys Rose XML Tool 1.3.6.01 to export the XMI files from its models.

Our prototype takes the XMI file exported from the model in Rational Rose as input and produces Contract specifications of all the use cases is described by interaction diagrams.

The transformation algorithm is as follows:

Algorithm 3

(1) For every child element[2] of "UML : Interaction" in every "UML : UseCase" element of the XMI file, take the value of its attribute "name" as the name of the contract.

(2) Construct a tuple *M* consisting of all the child elements of "UML : Message". Arrange the sequence of elements in *M* as follows:

Suppose *A* and *B* are two elements in *M*.

If the value of attribute "activator" of *A* is equal to the value of attribute "xmi.id" of *B*, then

arrange *A* behind *B*.

If the value of attribute "predecessor" of *A* is equal to the value of attribute "xmi.id" of *B*, then

arrange *A* behind *B* and behind all other elements whose attributes "activator" have the same value as attribute "xmi.id" of *B*.

(3) Suppose there are *k* elements in *M*. For every $i = 1, 2, ..., k$, construct a tuple $ACTM_i$ consisting of all the elements whose attributes "activator" have the same value as attribute "xmi.id" of element *i*.

(4) Construct a tuple *OBID* consisting of all the values of attributes "sender" and "receiver" of all the child elements of "UML : Message", such that the sequence of *OBID* is not important. Construct a tuple *OB* consisting of all the "UML : ClassifierRole" elements whose attribute "xmi.id" has the same value as some element of *OBID*, such that the sequence of its elements corresponds to *OBID*. Construct a tuple *C* consisting of all the "UML : Class" elements whose attribute "xmi.id" has the same value as the attribute "base" of any element in *OB*, such that the sequence of its elements correspond to *OB*. Construct a set *MOB* consisting of all the elements of *OB* whose "UML : MultiplicityRange" has child elements with an attribute "upper" having a value not equal to "1".

(5) Suppose there are *n* elements in *OB*. For every $i = 1, 2, ..., n$, construct a tuple Rm_i consisting of all the elements of *M* whose attribute "receiver" has the same value as the attribute "xmi.id" of some element in *OB*.

(6) For every $i = 1, 2, ..., n$, if the value of the attribute "xmi.id" of the *i*-th element in *OB* is not equal to the value of attribute "xmi.id" for any "UML : Actor" element in the file, then

(a) Suppose the value of attribute "name" of the element in *C* corresponds to an element of some C_i in *OB*.

If C_i is not in *MOB*, then output: "C_i **supports [**".

[2] Each one of these child elements corresponds to a contract.

If C_i is in *MOB*, then output:

"C_i : **SetOf**(C_i) **where each** C_i **supports [**".

(b) For every element of Rm_i, if it is the j-th element of M and if $ACTM_j$ is not empty, then

(i) Suppose the value of attribute "name" of the element is M_j(). Output: "$C_i \leftarrow M_j() \Rightarrow$".

(ii) For every element of $ACTM_j$, perform **algorithm 3.1**.

(iii) If the value of attribute "xmi.id" of an element in Rm_i is equal to the value of attribute "action" of the j-th element of M, then:

Suppose the value of attribute "operation" of this element is Op_j.

If a "UML : Operation" element has an attribute "xmi.id" whose value is equal to Op_j, and a child element of "UML : Parameter" has an attribute "kind" whose value is "return" and an attribute "name" whose value is "*pname*", then

output: "**return** C_i.*pname*".

If a "UML : Operation" element has an attribute "modelElement" whose value is equal to Op_j, and the value of attribute "tag" of this element is "RationalRose : postconditions", then:

Suppose the value of attribute "value" is *r*_condition. Output: "**{***r*_condition**}**".

(c) Output: "**]**".

(7) Output: "**end contract**".

Algorithm 3.1

Suppose the input parameter is the x-th element of M with the following properties:

(a) the value of attribute "name" of this element is M_x(),

(b) the value of attribute "receiver" of this element is C_x, which corresponds (through *OB*) to the value of the attribute "name" of an element in C, and

(c) the value of attribute "action" of this element is equal to the value of attribute "xmi.id" of an element whose attribute "operation" has a value of Op_x.

This algorithm will output a "UML : Message" element as a "resulting message". The details of the algorithm are as follows:

(1) If the value of attribute "modelElement" of an element is equal to Op_x and the value of attribute "tag" of this element is "RationalRose : preconditions", then:

Suppose the value of attribute "value" is "condition".

If the element of *OB* corresponding to the value of attribute "receiver" of element x is not in *MOB*, then

output: "**if** condition **then** $C_x \leftarrow M_x()$"

else

output: "(/V : V in C_x : **if** condition **then** V $\leftarrow M_x()$)".

(2) Otherwise:

If the element of *OB* corresponding to the value of attribute "receiver" of element x is not in *MOB*, then

output: "$C_x \leftarrow M_x()$"

else

output: "(/V : V in C_x : V $\leftarrow M_x()$)".

B. Prototype Program

The prototype program consists of three parts:

(1) Definition of data structures.
(2) Parsing of the input file and construction of various data structures to obtain information required for transformation.
(3) Follow the main algorithms above to obtain the results.

The most important data structure in UML interaction diagrams are Messages and their relationships. We use linked lists to represent them. Each node of the linked list represents a message, which is defined thus:

```
typedef struct message {
    char  * id;
    char  * name;
    char  * activator;
    char  * prec;
    char  * postc;
    char  * receiver;
    char  * returnv;
    char  * sender;
    char  * predecessor;
    char  * action;
    char  * operation;
    struct mptr * activation;
    struct message * next;
} MG;
MG *mp;
```

The second part of the program parses the input file, extracts the information needed and fills in various structures such as messages. As the information in the file is not arranged

sequentially to meet our acquisition requirement, we need to parse the file several times to obtain the necessary information. After construction, these structures containing information of the diagram are arranged in proper order.

When we parse the XML file to find elements and values, we apply the "look ahead" technique commonly used in compilers. It means that the program can recognize the end of an element only after it has read one more character not belonging to the element. Hence, we need to return the last character back to the input stream.

The third part of the program implements the main algorithms above and outputs the contract specification of the diagram represented by the XML file. Hence, the UML specification can be used as input to the TACCLE testing method. In this way, the object-oriented software system can be tested at the cluster level.

The prototype program has only been implemented for demonstrating the feasibility of our approach. We concede that it has not been designed in the most efficient manner. More future work is required for time complexity improvements.

VI. Conclusion

In order to take advantage of the use of formal specifications for the testing of object-oriented software, we transform UML interaction diagrams into Contract specifications. The proposed transformation techniques and algorithms are presented in this paper. They are based on the UML meta-model. We give a generic transformation algorithm first. It is independent of implementation and can be used in various tools. Then we present its specialized form for use in one kind of UML implementation.

We have developed a prototype to evaluate the algorithm. We have also conducted a case study, which is not included in the present paper because of the page limit.

References

[1] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the 5th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA '90*), *ACM SIGPLAN Notices*, 25 (10): 169–180, 1990.

[2] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10 (1): 56–109, 2001.

[3] T. H. Tse, F. C. M. Lau, W. K. Chan, P. C. K. Liu, and C. K. F. Luk, "Testing of object-oriented industrial software without precise oracles or results", *Communications of the ACM*, 50 (8) (2007).

[4] J.-M. Bruel and R. B. France. Transforming UML models to formal specifications. In *Beyond the Notation: Proceedings of the 1st International Workshop on the Unified Modeling Language* (*UML '98*), volume 1618 of Lecture Notes in Computer Science. Springer, Berlin, 1999.

[5] P. Selonen, K. Koskimies, and M. Sakkinen. Transformations between UML diagrams. *Journal of Database Management*, 14 (3): 37–55, 2003.

[6] J. de Lara and H. Vangheluwe. AToM3: A Tool for multi-formalism and meta-modelling. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, volume 2306 of Lecture Notes in Computer Science, pages 174–188. Springer, London, 2002.

[7] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro. VIATRA: visual automated transformations for formal verification and validation of UML models. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering* (*ASE 2002*), pages 267–270. IEEE Computer Society Press, Los Alamitos, California, 2002.

[8] J. Saez, A. T. Alverez, and J. L. F. Aleman. Tool support for transforming UML models to a formal language. In *Proceedings of International Workshop on Transformations of UML Models* (*WTUML 2001*) in conjunction with *European Joint Conferences on Theory and Practice of Software* (*ETAPS 2001*) (Geneva, Italy), pages 111–115, 2001.

[9] S.-K. Kim and D. Carrington. A formal metamodeling approach to a transformation between visual and formal modeling techniques. Technical Report 02-23. Software Verification Research Centre, School of Information Technology, The University of Queensland, Queensland, 2002.

[10] Unified Modeling Language: Superstructure, version 2.1.1. Object Management Group, 2007. Available at http://www.omg.org/docs/formal/07-02-03.pdf".

[11] H. Y. Chen. An approach for object-oriented cluster-level tests based on UML. In *Proceedings of the 2003 IEEE International Conference on Systems, Man, and Cybernetics* (*SMC 2003*), volume 2, pages 1064–1068. IEEE Computer Society Press, Los Alamitos, California, 2003.

[12] MOF 2.0/XMI Mapping Specification, version 2.1. Object Management Group, 2005. Available at http://www.omg.org/docs/formal/05-09-01.pdf.

[13] Extensible Markup Language (XML). W3C. Available at http://www.w3.org/XML.