Modern Fortran Transformation Rules for UML Sequence Diagrams

Aziz Nanthaamornphong and Anawat Leatongkam
Faculty of Technology and Environment, Prince of Songkla University, Phuket Campus, Thailand.
aziz.n@phuket.psu.ac.th

***Abstract*—Recently, reverse engineering has been widely adopted as a valuable process for extracting system abstractions and design information from existing software systems. The proposed research will focus on ForUML, a reverse engineering tool developed to extract UML diagrams from modern, object-oriented Fortran code, which are still used by scientists and engineering application developers. The first version of ForUML produces only UML class diagrams, which provide a useful window into the static structure of a program, including the make-up of each class and the relationships between classes. Rather than visualizing class diagrams, the developers need to understand class behavior and interactions between classes. UML sequence diagrams provide such important algorithmic information. Therefore, we proposed rules for transforming object-oriented Fortran into UML sequence diagrams with the goal to extend the ability of ForUML. The proposed rules were designed by Atlas Transformation Language. We believe that the contribution of this work would enhance the development, maintenance practices, decision processes, and communications in the scientific software community worldwide.**

***Index Terms*—Fortran; Reverse Engineering; Software Engineering; UML Sequence Diagram.**

I. INTRODUCTION

At present, reverse engineering becomes widely well known, especially for software developers. Reverse engineering for software engineering is about reviewing source codes to understand the software. In terms of software development, if a software is large or comprises of numerous lines of code, it will result in complexity, thereby being difficult in reviewing and understanding those source codes. Thus, reverse engineering will help developers understand an overall picture of the system easily in order to maintain and improve the software. However, the reverse engineering of large or complex software is painful and challenging [1]. One of reverse engineering process challenges is to build a point of view that represents the meaning of abstract or intangible of the complex system by visualizing the source code in a form of readable and understandable notations [2]; such as, Unified Modeling Language)UML(.

In the previous work, the second author developed a tool namely ForUML [3], which is capable of extracting UML class diagrams from object-oriented Fortran code. The UML class diagram is a diagram that represents classes' structure and relationship between other classes. A Fortran programming language was further developed to be an object-oriented programming language like Java or C++. It is still a popular programming language for scientific and engineering software development in various domains, such as weather forecast, astronomy, and mechanical engineering. However, software development for these fields still lacks of quality software development tools [4]. Furthermore, such software development is largely based on a trial and error method and self-studies, since developers in these fields are generally scientists and engineers who have only fundamental programming knowledge which limits them on advance coding [5].

The first version of ForUML has been adopting by multiple Fortran software development teams. However, the first version of ForUML has some limitations, because this tool can only represent codes in a class diagram, which represents a structural model of the system but that does not imply operational behaviors, procedures or sequences. Hence, the diagram is not enough for analyzing and understanding the system. In addition, a user of this tool suggests about adding new properties or capabilities, such as UML sequence diagram generation, since this behavioral diagram will describe a sequence in a system and that will not only show the overall system developed from Fortran language, but also help make better decisions about software development and maintenance.

With this regard, this study aimed to propose transformation rules to convert Fortran source codes to an UML sequence diagram. As far as we know, no one has created transformation rules for that case. We argue that this study will benefit to adding ForUML capabilities on creation of UML sequence diagram, and also having a variety of design documents will help developers better analyze and understand the software, as well as develop and maintain the system [6].

The remainder of this paper is organized as follows. Section II provides an overview of related work. Section III, the transformation rules are described. Section IV summaries the results. Finally, conclusions are drawn and future work is presented in Section V.

II. RELATED WORK

This section describes related theories and literature, including the Fortran programming language, reverse engineering, UML metamodel, and ForUML.

A. Fortran Programming Language

Currently, Fortran is developed to support an object-oriented concept, which is called Modern Fortran [7] to support complex software development. Modern Fortran also emphasizes on software engineering principles for better software performance and that leads to more interests and adoption of Modern Fortran for software development by many scientists and engineers [8,9]. Besides, at present, a lot of Fortran compiler makers have enhanced their compilers to support Modern Fortran; for example, Numerical Algorithm

Group)NAG(and Intel Fortran.

Modern Fortran has many important features of object-oriented language, including inheritance, polymorphism, dynamic type allocation, and type-bound procedures. Nonetheless, since Modern Fortran is relatively new in the world of object-oriented programming, so there are a few tools available and those do not really adopt a software engineering concept, compared to other object-oriented languages such as Java and C++, especially for program comprehension tools, which help software developers and designers understand source codes or the software easier.

B. Reverse Engineering

Reverse engineering for large software frequently relates to analyzing parts of source codes to understand the system. Generally, it is used for analyzing binary codes. An example of reverse engineering software that can decompile binary codes to get source codes is Jad [10], a software that can decompile binary codes of Java language, such as a file with .class extension, to get back source codes, which developers can review and understand.

Periklis Andritsos and Renee J. Miller [11] state that, in general, when a software get older, it is difficult to understand and maintain the software. Sometimes, this characteristic leads to an inefficient system and additional maintenance cost. Thus, the software engineering community pays attention to building tools to help software engineers understand a structure of the system.

However, there are a few existing reverse engineering tools designed for Modern Fortran. This challenge inspired us to work on the Fortran-related reverse engineering tool.

C. UML Metamodel

UML is a modeling language which is standardized for generating object-oriented models or visualizing a system's architectural blueprints. UML can be used to create system's point of views, define system specifications, and develop the system. In this research, XML Metadata Interchange (XMI) document was used to represent an UML sequence diagram. XMI is an open standard with which developers or software vendors can create, read, manage, and generate XMI tools. Transforming the model (Modern Fortran code) to XMI requires the Model Driven Architecture technology, which is a standard using modeling issued by the Object Management Group (OMG). The information in the XMI document can be used to develop their own applications among a set of tools to crate and exchange. The basic idea of using an XMI file is to maintain the metadata for UML diagrams, called UML metamodel, which is used to describe syntax definition and meaning for structures or components in an UML model. This metamodel helps developers get insights into the meaning of model in the same way and creates the model in accordance with the UML standard.

D. ForUML

ForUML [3] is a reverse engineering tool that can be used to extract UML class diagrams from Modern Fortran source code. This tool is available as free software [12]. The model for transforming source codes to UML diagrams is based on the schema for the static structure of source code, called Dagstuhl Middle Metamodel (DMM) [13], which is widely used to represent models extracted from source code written in most common object-oriented programming languages.

The transformation process of ForUML comprises of four steps with details as follows.

i. Parsing: The tool parses source codes into elements by using Open Fortran Parser (OFP) library. To do so, this process will use grammar files and Fortran syntax in the OFP library. This step will validate the correctness of codes that are supplied by a user to a system to prevent errors in the next step.
ii. Extraction: This step is to find relationships among the elements obtained from Step 1. Then, the extraction module maps each relationship to a specific relationship's type object.
iii. Generating: The tool will collect elements and their relationships, which are the output of Step 1 and 2 respectively to build a document in a form of XMI. This XMI document stores necessary data to form an UML class diagram.
iv. Importing: The generated XMI document will be imported into a UML modeling tool to display the resulting class diagram. Note that ForUML currently integrates ArgoUML for displaying the class diagram.

In this study, we proposed rules for reversing Fortran source codes into a UML sequence diagram. We designed rules by using a metamodel of UML sequence diagrams and Fortran source code files. The proposed rules will be developed to a new feature, which will be integrated into ForUML.

III. The Transformation Rules

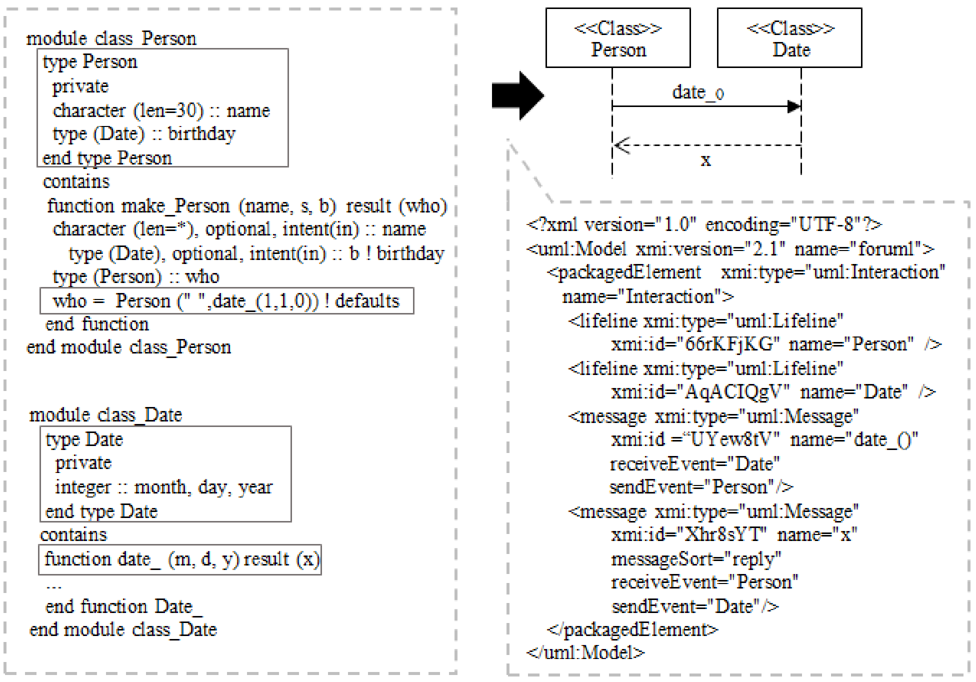
This research aimed at designing rules for transforming Fortran source codes to UML sequence diagrams. The transformation was based on applications of UML sequence diagram standards from UML specifications [14] and UML sequence diagram transformation rules from related literature [15-17] to create rules for transforming Fortran source codes to UML sequence diagrams.

Designing the rules for transforming Fortran source codes to UML sequence diagrams started from studying the specifications of XMI document, which are based on OMG. An example of XMI document embedding data of an UML sequence diagram for the Fortran code is shown in Figure 1.

Based on Figure 1, the XMI document consists of two main parts as follows.

i. xmi:type="uml:Lifeline" defines specifications of each lifeline, including xmi:id="66rKFjKG", which represents a lifeline ID and name="Person", which represents a lifeline name.
ii. xmi:type="uml:Message" defines message details of each message, including xmi:id="Xhr8sYT", which is a message ID, messageSort="reply", which represents a message type, name="Person", which represents a message name, receiveEvent="Person", which represents a lifeline that receives a message, and sendEvent="Date", which represents a lifeline that sends a message.

The main step of designing rules for transforming Modern Fortran codes to UML sequence diagrams is extracting for relationships between Abstract Syntax Tree (AST) metamodel of Fortran language and XMI document. The metamodel of both models will be a representative of the main model as shown in Figure 2.



```
module class Person
 type Person
  private
  character (len=30) :: name
  type (Date) :: birthday
 end type Person
 contains
  function make_Person (name, s, b) result (who)
   character (len=*), optional, intent(in) :: name
     type (Date), optional, intent(in) :: b ! birthday
   type (Person) :: who
   who = Person (" ",date_(1,1,0)) ! defaults
  end function
end module class_Person


module class_Date
 type Date
  private
  integer :: month, day, year
 end type Date
 contains
 function date_ (m, d, y) result (x)
  ...
  end function Date_
end module class_Date
```

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1" name="foruml">
  <packagedElement  xmi:type="uml:Interaction"
    name="Interaction">
     <lifeline xmi:type="uml:Lifeline"
           xmi:id="66rKFjKG" name="Person" />
     <lifeline xmi:type="uml:Lifeline"
           xmi:id="AqACIQgV"  name="Date"  />
     <message xmi:type="uml:Message"
           xmi:id ="UYew8tV"  name="date_()"
           receiveEvent="Date"
           sendEvent="Person"/>
     <message xmi:type="uml:Message"
           xmi:id="Xhr8sYT"  name="x"
           messageSort="reply"
           receiveEvent="Person"
           sendEvent="Date"/>
  </packagedElement>
</uml:Model>
```

Figure 1: An example of XMI document for an UML sequence diagram of Fortran based program

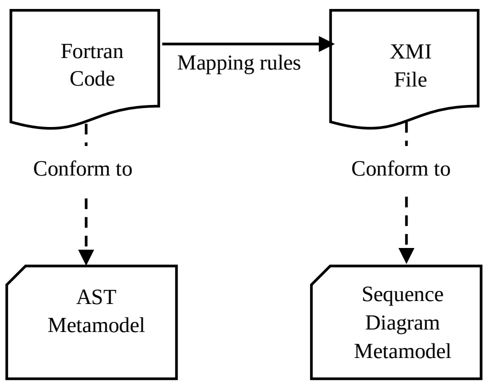
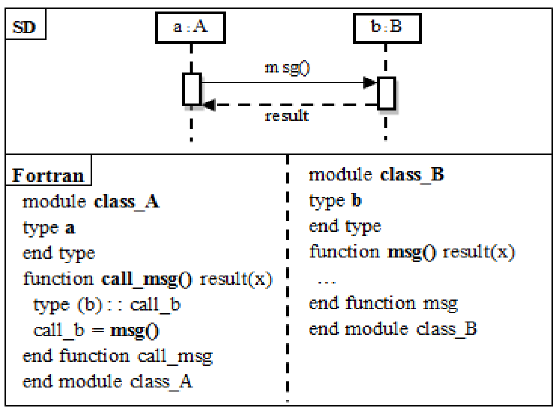


Figure 2: An overview of the transformation process

We will refer to Figure 3 to describe our developed rules for transforming Fortran source codes to UML sequence diagrams. To build transformation rules, we chose Atlas Transformation Language (ATL), a popular language for transforming models [18-20]. The important parts of transformation rules are listed as follows.



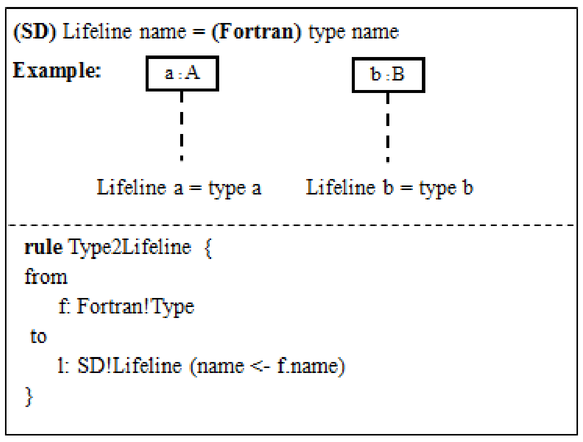
```
module class_A
type a
end type
function call_msg() result(x)
  type (b) : : call_b
  call_b = msg()
end function call_msg
end module class_A
```

```
module class_B
type b
end type
function msg() result(x)
 ...
end function msg
end module class_B
```

Figure 3: An example of rules for transforming Fortran source codes to UML sequence diagrams

1) Lifeline creation rules

These rules are used to bind between each lifeline in an UML sequence diagram and corresponding class name in Fortran source codes (as shown in Figure 4).



```
rule Type2Lifeline  {
from
    f: Fortran!Type
 to
    l: SD!Lifeline (name <- f.name)
}
```

Figure 4: Lifeline creation rules

2) Message creation rules

These rules link each message between lifelines in an UML sequence diagram and corresponding method name in Fortran source codes)as shown in Figure 5(.

3) Message sending and receiving rules

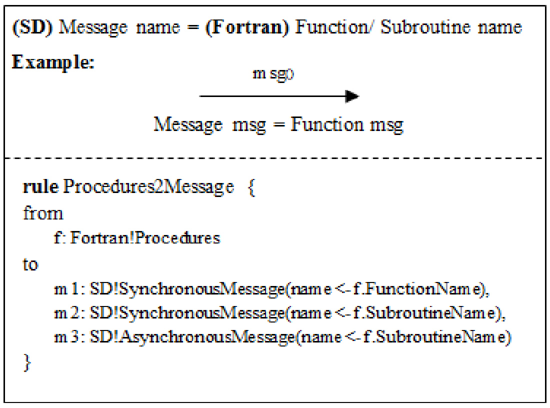
These rules are used to define how a lifeline sends and receives a message (as shown in Figure 6).

4) Rules for defining start and finish occurrences of message execution

These rules are to define start and finish occurrences of message execution on a lifeline (as shown in Figure 6).

5) Rules for specifying a message execution

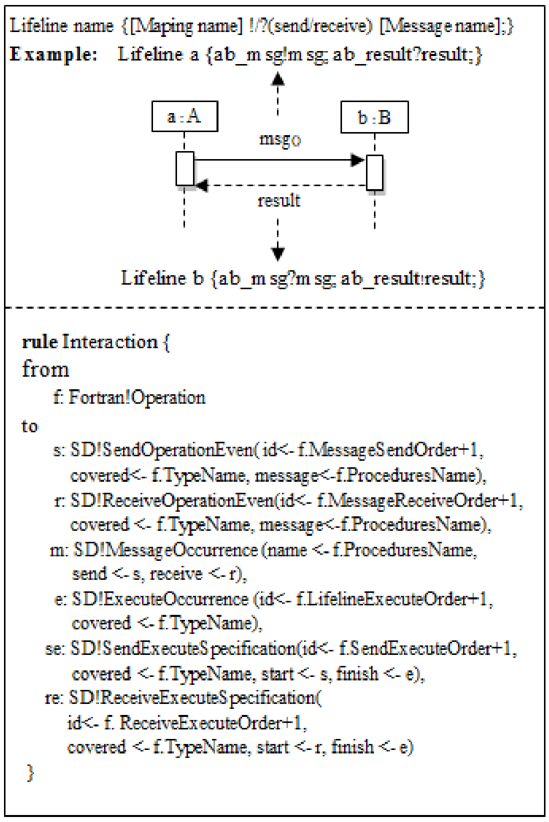
These rules specify how a message will execute on a lifeline (as shown in Figure 6).



```
(SD) Message name = (Fortran) Function/ Subroutine name
Example:
                    msg()
          Message msg = Function msg
```

```
rule Procedures2Message {
from
    f: Fortran!Procedures
to
    m1: SD!SynchronousMessage(name <- f.FunctionName),
    m2: SD!SynchronousMessage(name <- f.SubroutineName),
    m3: SD!AsynchronousMessage(name <- f.SubroutineName)
}
```

Figure 5: Rules for creating synchronous call and asynchronous call messages between lifelines



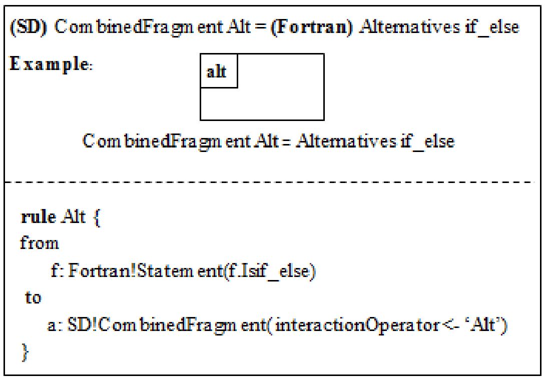
```
Lifeline name {[Maping name] !/?(send/receive) [Message name];}
Example:   Lifeline a {ab_msg!msg; ab_result?result;}
                a : A          msg()          b : B
                               result
       Lifeline b {ab_msg?msg; ab_result!result;}
```

```
rule Interaction {
from
    f: Fortran!Operation
to
    s: SD!SendOperationEven(id<- f.MessageSendOrder+1,
       covered<- f.TypeName, message<-f.ProceduresName),
    r: SD!ReceiveOperationEven(id<- f.MessageReceiveOrder+1,
       covered <- f.TypeName, message<-f.ProceduresName),
   m: SD!MessageOccurrence (name <- f.ProceduresName,
       send <- s, receive <-r),
    e: SD!ExecuteOccurrence (id<- f.LifelineExecuteOrder+1,
       covered <- f.TypeName),
  se: SD!SendExecuteSpecification(id<- f.SendExecuteOrder+1,
       covered <- f.TypeName, start <- s, finish <- e),
  re: SD!ReceiveExecuteSpecification(
      id<- f. ReceiveExecuteOrder+1,
      covered <- f.TypeName, start <- r, finish <- e)
}
```

Figure 6: Communication rules between messages and lifelines

6) Rules for creating a frame

The frame is for conditions, multi-conditional alternatives, and iterations (as shown in Figure 7).



```
(SD) CombinedFragmentAlt = (Fortran) Alternatives if_else
Example:     alt
   CombinedFragmentAlt = Alternatives if_else
```

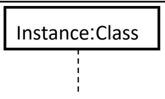
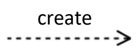
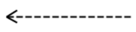
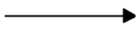
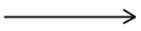
```
rule Alt {
from
    f: Fortran!Statement(f.Isif_else)
to
    a: SD!CombinedFragment(interactionOperator<- 'Alt')
}
```

Figure 7: Rules for creating a frame for a condition area

IV. RESULTS

From the design of rules for transforming Fortran source code to UML sequence diagrams, we presented study results to Modern Fortran experts, who are the founders of training center and advisors on Modern Fortran and scientific software development [12], which the organization is located at the United States of America, for validating correctness of transformation rules. Besides, we asked two experts to consider comparing syntax of object-oriented languages to find similarity in a representation of each notation of a UML sequence diagram. Each expert separately verified nine design rules based on his opinion and experience. The experts reported us that all rules are correct without any problems. In this study, we compared Fortran to Java, which is a popular object-oriented programming language, as shown in Table 1. The comparison splits into two perspectives in accordance with characteristics of notations as follows.

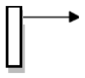
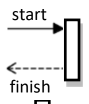
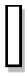
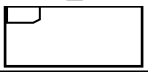
Table 1
A comparison of transformation from source codes to UML sequence diagrams between Java and Fortran

Rule	Java Syntax	Modern Fortran Syntax	Notations
Lifeline	public class MyClass	type MyClass	
Messages			
Create Message	MyClass my = new MyClass();	type(MyClass) ::my	
Reply Message	public int getID() { return id; }	function getID() result(id)	
Synchronous Message	my.getID();	call my%getID() my = getID();	
Asynchronous Message	my.getID();	call my%getID()	

i. Lifelines that are representatives of the class. These include an instance name and class name.
ii. Messages that sent between lifelines. These comprise of create message, reply message, synchronous call message, and asynchronous call message.

For interaction fragments, which represent a period in the instance's lifetime, including sending and receiving a message, start and finish occurrences of message execution on a lifeline, execution occurrence specifications on a lifeline and frame, the representation of corresponding notations is similar to that of Java as shown in Table 2.

Table 2
UML notations for interaction fragments

Rule	Fortran Semantics	Notations
Interaction Fragment		
Message Occurrence	Send and Receive Occurrence	
Execution Specification	Start and Finish Occurrence	
Execution Occurrence	Activation	
Combined Fragment	Loops, Branches, and Other Alternatives	

To verify whether the transformation rules could be practically applied to Fortran, we developed a software application for generating the XMI document. The transformation rules were employed in the developed application to generate a XMI document from the Fortran source codes brought from [21] (the code is available at http://research.te.psu.ac.th/aziz/FortranCode/page1.html).

Figure 8 presents an excerpt of XMI document obtained from the application of transformation rules. It consists of lifelines for a Main program, Person, Date, and Student. For interaction fragments, they can be categorized into two groups. The first category defines interactions between messages and lifelines, including message occurrence specifications, behavior execution specifications, and execution occurrence specifications, while the second category defines a frame for alternatives, options and loops (Note that the testing source codes did not have any combined fragments). The last messages, such as "create" in the example, is a create message, which, for instance, represents object instantiation of a class, defines a lifeline for sending and receiving messages, etc. Last but not least, we verified the results by manually comparing a XMI document from the testing application to Fortran source code. The verification results confirmed that transformation was correct.

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmlns:uml="http://schema.omg.org/spec/UML/2.1.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmi:version="2.1">
 <packagedElement xmi:type="uml:Interaction" xmi:id="_Q5lC9CwhEeeE65EyuZJOSA">
  <lifeline xmi:id="_Q5lC9iwhEeeE65EyuZJOSA" name="Program main" coveredBy="_Q5lC-iwhEeeE65EyuZJOSA _Q5lC_CwhEeeE65EyuZJOSA …"/>
  <lifeline xmi:id="_Q5lC9ywhEeeE65EyuZJOSA" name="Person" coveredBy="_Q5lC-ywhEeeE65EyuZJOSA _Q5lC_SwhEeeE65EyuZJOSA … "/>
  <lifeline xmi:id="_Q5lC-CwhEeeE65EyuZJOSA" name="Date" coveredBy="_Q5lDAiwhEeeE65EyuZJOSA _Q5lDFywhEeeE65EyuZJOSA …"/>
  <lifeline xmi:id="_Q5lC-SwhEeeE65EyuZJOSA" name="Student" coveredBy="_Q5lDDSwhEeeE65EyuZJOSA _Q5lDIywhEeeE65EyuZJOSA … "/>
  <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_Q5lC-iwhEeeE65EyuZJOSA" covered="_Q5lC9iwhEeeE65EyuZJOSA" message="_Q5lDOywhEeeE65EyuZJOSA"/>
  <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_Q5lC-ywhEeeE65EyuZJOSA" covered="_Q5lC9ywhEeeE65EyuZJOSA" message="_Q5lDOywhEeeE65EyuZJOSA"/>
  <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_Q5lC_CwhEeeE65EyuZJOSA" covered="_Q5lC9iwhEeeE65EyuZJOSA" message="_Q5lDPCwhEeeE65EyuZJOSA"/>
  <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_Q5lC_SwhEeeE65EyuZJOSA" covered="_Q5lC9ywhEeeE65EyuZJOSA" message="_Q5lDPCwhEeeE65EyuZJOSA"/>
  <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="_Q5lC_iwhEeeE65EyuZJOSA" covered="_Q5lC9ywhEeeE65EyuZJOSA" start="_Q5lC_SwhEeeE65EyuZJOSA"
     finish="_Q5lDBSwhEeeE65EyuZJOSA"/>
…
  <message xmi:type="uml:Message" xmi:id="_Q5lDOywhEeeE65EyuZJOSA" name="create" messageSort="createMessage" receiveEvent="_Q5lC-ywhEeeE65EyuZJOSA"
     sendEvent="_Q5lC-iwhEeeE65EyuZJOSA"/>
  <message xmi:type="uml:Message" xmi:id="_Q5lDPCwhEeeE65EyuZJOSA" name="make_Person" receiveEvent="_Q5lC_SwhEeeE65EyuZJOSA" sendEvent="_Q5lC_CwhEeeE65EyuZJOSA"/>
  <message xmi:type="uml:Message" xmi:id="_Q5lDPSwhEeeE65EyuZJOSA" name="Person_" receiveEvent="_Q5lDACwhEeeE65EyuZJOSA" sendEvent="_Q5lC_ywhEeeE65EyuZJOSA"/>
  <message xmi:type="uml:Message" xmi:id="_Q5lDPiwhEeeE65EyuZJOSA" messageSort="reply" receiveEvent="_Q5lDAywhEeeE65EyuZJOSA" sendEvent="_Q5lDAiwhEeeE65EyuZJOSA"/>
  <message xmi:type="uml:Message" xmi:id="_Q5lDPywhEeeE65EyuZJOSA" messageSort="reply" receiveEvent="_Q5lDBiwhEeeE65EyuZJOSA" sendEvent="_Q5lDBSwhEeeE65EyuZJOSA"/>
  <message xmi:type="uml:Message" xmi:id="_Q5lDQCwhEeeE65EyuZJOSA" name="set_DOB" receiveEvent="_Q5lDCCwhEeeE65EyuZJOSA" sendEvent="_Q5lDBywhEeeE65EyuZJOSA"/>
  <message xmi:type="uml:Message" xmi:id="_Q5lDQSwhEeeE65EyuZJOSA" messageSort="reply" receiveEvent="_Q5lDCywhEeeE65EyuZJOSA" sendEvent="_Q5lDCiwhEeeE65EyuZJOSA"/>
  <message xmi:type="uml:Message" xmi:id="_Q5lDQiwhEeeE65EyuZJOSA" name="create" messageSort="createMessage" receiveEvent="_Q5lDDSwhEeeE65EyuZJOSA"
     sendEvent="_Q5lDDCwhEeeE65EyuZJOSA"/>
...
  <message  xmi:type="uml:Message"  xmi:id="_Q5lDVCwhEeeE65EyuZJOSA"  messageSort="reply"  receiveEvent="_Q5lDOiwhEeeE65EyuZJOSA"  sendEvent="_Q5lDOSwhEeeE65EyuZJOSA"/>
</packagedElement>
</uml:Model>
```

Figure 8: XMI documents obtained from applying the rules

V. CONCLUSION

This study proposed a design concept of rules for transforming Modern Fortran source code to UML sequence diagrams with the aim of applying the rules for development of transformation tool to convert Modern Fortran source code to UML sequence diagrams. From the design of transformation rules, we compared those rules to Java, which is purely an object-oriented language, while Fortran is developed to be an object-oriented language later. When compared to each other, both languages had the same features, albeit different representations such as a class name and method name.

In the future, we will apply transformation rules presented in this study to enhance capabilities of ForUML on sequence diagram generation.

REFERENCES

[1] M. Lanza and S. Ducasse, "Polymetric views-a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–795, Sep. 2003.

[2] T. Systa, "On the relationships between static and dynamic models in reverse engineering java software," in *Proceedings of the IEEE 6th Working Conference on Reverse Engineering*, 1999, pp. 304–313.

[3] A. Nanthaamornphong, J. Carver, K. Morris, and S. Filippone, "Extracting uml class diagrams from object-oriented fortran: ForUML," *Scientific Programming*, vol. 2015, pp. 1–15, Jan. 2015.

[4] J.C. Carver, R.P. Kendall, S.E. Squires, and D.E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *Proceedings of the IEEE 29th International Conference on Software Engineering (ICSE)*, 2007, pp. 550–559.

[5] J.C. Carver, "Report: the second international workshop on software engineering for CSE," *Computing in Science & Engineering*, vol. 11, no. 6, pp. 14–19, Nov. 2009,.

[6] B. Dobing and J. Parsons, "How UML is used," *Communications of the ACM*, vol. 49, no. 5, pp. 109–113, May 2006.

[7] N.S. Clerman and W. Spector, *Modern Fortran: Style and Usage*, Cambridge University Press, 2011.

[8] D. Barbieri, V. Cardellini, S. Filippone, and D. Rouson, "Design patterns for scientific computations on sparse matrices," in *Proceedings of the European Conference on Parallel*, Springer, 2011, pp. 367–376.

[9] K. Morris, D.W. Rouson, M.N. Lemaster, and S. Filippone, "Exploring capabilities within ForTrilinos by solving the 3D burgers equation," *Scientific Programming*, vol. 20, no. 3, pp. 275–292, Jul. 2012.

[10] "A. Rukin - Java decompilers." Available at http://www.javadecompilers.com/. [Accessed: 12-June-2017].

[11] P. Andritsos and R.J. Miller, "Reverse engineering meets data analysis," in *Proceedings of the IEEE 9th International Workshop on Program Comprehension*, 2001, pp. 157–166.

[12] "D. Rouson - Sourcery Institute." Available at http://www.sourceryinstitute.org/. [Accessed: 12-June-2017].

[13] T.C. Lethbridge, S. Tichelaar, and E. Plödereder, "The dagstuhl middle metamodel: a schema for reverse engineering," *Electronic Notes in Theoretical Computer Science*, vol. 94, pp. 7–18, May 2004.

[14] "OMG - UML specification v2.5" Available at http://www.omg.org/spec/UML/2.5/. [Accessed: 12-June-2017].

[15] P. Sawprakhon and Y. Limpiyakorn, "Sequence diagram generation with model transformation technology," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 2014, pp. 12–14.

[16] C. Li, L. Dou, and Z. Yang, "A metamodeling level transformation from UML sequence diagrams to Coq.," in *Proceedings of International Conference on Information and Communication Technology for Competitive Strategies*, 2014, pp. 147–157.

[17] E. Merah, "Design of ATL rules for transforming UML 2 sequence diagrams into petri nets," *International Journal of Computer Science and Business Informatics*, vol. 8, no. 1, pp. 1–21, Jan. 2014.

[18] S. Buckl, A.M. Ernst, J. Lankes, F. Matthes, C.M. Schweda, and A. Wittenburg, "Generating visualizations of enterprise architectures using model transformations," *Enterprise Modelling and Information Systems Architectures*, vol. 2, no. 2, pp. 3–13, Dec. 2015.

[19] Y. Rhazali, Y. Hadi, and A. Mouloudi, "Model transformation with ATL into MDA from CIM to PIM structured through MVC," *Procedia Computer Science*, vol. 83, pp. 1096–1101, Dec. 2016.

[20] J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer, "Towards systematic mutations for and with ATL model transformations," in *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[21] J. Akin, "Object oriented programming via Fortran 90," *Engineering Computations*, vol. 16, no.1, pp. 26–48, Feb. 1999.