Hindawi
Scientific Programming
Volume 2019, Article ID 2542686, 22 pages
https://doi.org/10.1155/2019/2542686

Research Article

Extended ForUML for Automatic Generation of UML Sequence Diagrams from Object-Oriented Fortran

Aziz Nanthaamornphong and Anawat Leatongkam

College of Computing, Prince of Songkla University, Phuket Campus, Phuket Province, Thailand

Correspondence should be addressed to Aziz Nanthaamornphong; aziz.n@phuket.psu.ac.th

Received 4 September 2018; Revised 18 December 2018; Accepted 3 January 2019; Published 5 February 2019

Academic Editor: Cristian Mateos

Recently, reverse engineering has become widely recognized as a valuable process for extracting system abstractions and design information from existing software. This study focuses on ForUML, a reverse engineering tool developed to extract UML diagrams from modern object-oriented Fortran programs. Generally, Fortran is used to implement scientific and engineering software in various domains, such as weather forecasting, astrophysics, and engineering design. However, methods for visualizing the existing design of object-oriented Fortran software are lacking. UML diagrams of Fortran software would be beneficial to scientists and engineers in explaining the structure and behavior of their programs at a higher level of abstraction than the source code itself. UML diagrams can enhance discussions within development teams and with the broader scientific community. The first version of ForUML produces only UML class diagrams. Class diagrams provide a useful window into the static structure of a program, including the structure and components of each class and the relationships between classes. However, class diagrams lack the temporal information required to understand class behavior and interactions between classes. UML sequence diagrams provide this important algorithmic information. Therefore, herein, an extension for ForUML to extract UML sequence diagrams from the Fortran code is proposed, and this capability is provided using a widely used open-source platform. This study argues that the proposed extension will enable the visualization of object-oriented Fortran software behavior and algorithmic structure and thereby enhance the development, maintenance practices, decision processes, and communications in scientific and engineering software communities worldwide.

1. Introduction

Reverse engineering is a well-known process, especially among software developers. Reverse engineering is the process of abstracting the structural and functional information of a particular system or program by investigating system components. For example, one type of system abstraction involves analyzing source code to recreate the overall system structure, while another is intended to recover a design document or another document type that describes system operations. Such documents can be used to analyze a structure and understand how each component of the system works. These components are later used to build a similar system or program that may not have exactly the same structure as the reverse-engineered system [1].

In terms of software engineering, reverse engineering involves reading and understanding lines of source code to understand how the software system works. When the software system is large or has huge numbers of code lines, the system is likely complex; hence, it is difficult to read and complex to obtain insights from the source code alone. Reverse engineering can help developers visualize the overall system more easily to achieve their maintenance and future improvement goals. Nevertheless, reverse engineering large or complex software systems is difficult [2]. One challenge in reverse engineering is to create views that can capture the intent of source code in a format such as Unified Modeling Language (UML), making it possible for developers to understand the complexities in the code [3].

One common problem in software engineering involves legacy code that was initially written for early software versions and later enhanced when the system was updated to a more current version [4]. It is difficult for developers to modify or change the source code when they do not

understand the original system. In software engineering, developers generally base code development on design documents to build software that matches the design requirements. Over time, as developers modify or change the source code, structures or functions may be altered from the original design; however, the original design documents are often not revised to reflect such code changes. Therefore, developers often cannot rely on the original design documents when performing software maintenance. In addition, as the source code becomes more complex, it becomes more difficult to understand the system; hence, software reuse becomes challenging [5]. In addition, developers who do not understand the system may be unable to improve the system's structure or add new features.

Currently, many reverse engineering tools exist that support software engineers during analysis and understanding of complex software systems. The capabilities of these tools vary, depending on the underlying programming language. These tools support software maintenance processes by reducing the time required to analyze and understand the source code. Our previous work [6] proposed a tool called ForUML, which can transform Fortran-based source code [7] into UML class diagrams. UML is a language for describing and representing the definitions and relationships of a system or program, and it is written in the form of UML diagrams, a format that is well known and widely adopted in the software engineering field. Fortran is a well-known language that has been widely used to build various scientific and engineering software applications, including weather forecasting, astronomy, and medication. However, there is a lack of software engineering tools for use during the development of such applications [8].

Currently, ForUML is the only available tool for transforming Fortran source code (which later evolved into an object-oriented programming language similar to Java and C++) into UML class diagrams. The diagrams reveal the class structure of code, describe how the classes in an application are related to each other, and simplify the process of understanding the system. In addition, ForUML is used in teaching Fortran-based software design to help students grasp system structures better than they can by simply reading the source code.

Nevertheless, the first version of ForUML has limited capabilities because the software only supports class diagrams. In addition, the diagrams represent only a structural view—a static system structure with no interaction time sequence. Class diagrams depict only static information. They do not explain the sequences of events that happen at runtime. Hence, ForUML is insufficient for fully analyzing and understanding a particular system. To address this problem, ForUML users have proposed adding new features or capabilities (such as UML sequence diagrams, which represent a behavioral view of the interactions within the system) to gain insight into an overall system developed in Fortran and be able to make better decisions about the software development processes involved in software maintenance.

In this study, we developed the idea of extending the ForUML tool by adding the capability to generate sequence diagrams, which represent interactions between objects (class instantiations) arranged in a time sequence. Enhancing design diagrams with dynamic concepts imparts a richer understanding of the software systems. For example, software engineers use sequence diagrams to understand how objects interact with each other in a specific use case. A sequence diagram shows such interactions in the order of occurrence. These diagrams can also support the transition from requirements that are described as use cases to a subsequent and more formal level of refinement. This capability benefits software developers working with systems developed in Fortran. The authors strongly believe that having design documents that reflect different points of view, especially UML diagrams, will help developers better analyze and understand software systems and assist in software development and maintenance processes [9] because UML class diagrams alone may be insufficient to thoroughly analyze and understand large software systems. In summary, this study extends ForUML by adding the ability to generate sequence diagrams to help software developers make better decisions during their software development processes.

2. Related Works

In this section, we describe the concepts related to our proposed tool, including reverse engineering, modern Fortran, and related works.

2.1. Reverse Engineering. Reverse engineering is a system analysis process intended to identify system components and their relationships by representing the system model from intangible items. The resulting model is used to analyze the structure and understand how each component works. Reverse engineering is not intended to change how a particular system works or to produce new system functionality [10].

Reverse engineering for software is primarily related to the source code; therefore, it is sometimes called "reverse code engineering," which is the process of analyzing various parts of the source code to understand them. Reverse code engineering is frequently adopted for binary code analysis [11]. For example, the binary code of a software program currently in use is in the form of unreadable or uninterpretable binary code or machine language. Jad is one example of reverse engineering software that can decompile the code from binary to the source code form [12]. Jad transforms Java binary code, such as a file with a .class extension, back to a source code form that software developers can read and understand. Other reverse engineering tools that can transform data to various formats exist; these include ArgoUML [13], Modelio [14], UML Designer [15], and Umbrello [16], among others. The main purpose of these application is to transform the source code to UML diagrams such as UML class or sequence diagrams, as illustrated in Figure 1.

Andritsos and Miller [17] state that, as the majority of large software systems get older, understanding and maintaining them becomes increasingly difficult. Sometimes, this lack of understanding leads to inefficiencies and higher costs. Therefore, software engineering communities

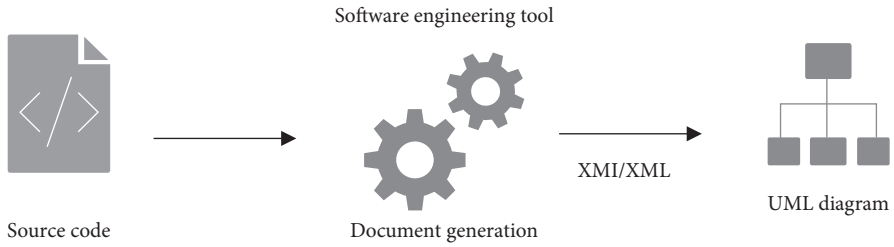


Figure 1: The example of a transformation process.

have paid increasing attention to creating tools that help system analysts gain insights into system structure. The important roles of reverse engineering tools in software engineering are as follows:

(1) *Program Analysis* aims at analyzing source code and extract related information, including classes and methods.

(2) *Plan Recognition* identifies patterns that may reflect either behaviors or system structures.

(3) *Concept Assignment* is designed to search for system patterns and source code structures and to identify the relationships between system components.

(4) *Redocumentation* creates system documentation for systems where no documentation (or only obsolete documentation) exists.

Reverse engineering helps developers gain insight into the structure of large software systems and is especially useful for complex legacy systems that may not have design documents or whose documentation has been lost or not kept up-to-date and does not reflect the current system. Thus, reverse engineering can recover design documents that developers can subsequently use for comparison to the current system structure, aiding in further analysis, understanding, and improvement of the software development process.

2.2. Modern Fortran. Fortran was originally designed for mathematically focused application development in [18–20], for instance, in scientific and engineering fields. Over the years, Fortran has evolved to include object-oriented concepts similar to those in Java and C++; this updated version is called Modern Fortran [21]. Modern Fortran is intended to support complex software development efforts and to increase productivity, especially for scientific and engineering software, through the application of software engineering principles. Clearly, many scientists and software engineering researchers are interested in and use modern Fortran for building software [22–26]. Many current Fortran compiler vendors such as the Numerical Algorithm Group (NAG) [27], GNU Fortran [28], IBM XL Fortran [29], Cary [30], and Intel Fortran [31] have enhanced their compilers to support Modern Fortran.

Table 1 shows a structural comparison of object-oriented concepts between Fortran and Java. Java is a well-known object-oriented programming language [32], while Fortran has added the main object-oriented programming language features, including Inheritance, polymorphism, dynamic type allocation, and type-bound procedures. When comparing Java and Fortran in terms of object-oriented structures, some similarities between the two are apparent. First, in Java, a *Class* specifies operational details, while Fortran includes a structure with the same responsibility, but called a *Type* instead. Next, for Java, a *Package* is defined to specify the purpose of classes contained in the package, while Fortran defines a *Module* that has the same responsibility. Next, in Java, a *Method* specifies a class operation. Fortran also includes a method concept, but methods in Fortran are categorized into 2 types: *Functions* and *Subroutines*. Java does not make a distinction between such operations.

Table 1: A structural comparison of object-oriented concepts between Fortran and Java [24].

Object-oriented equivalent	Fortran	Java
Abstract data type (ADT)	Derived type	Class
Attribute	Component	Property
Method	Type-bound procedure	Method
Parent class	Parent type	Base class
Child class	Extend type	Subclass
Package	Module	Package
Static polymorphism	Generic interface	Overloading
Abstract method	Deferred procedure binding	Abstract
Primitive type	Intrinsic type	Primitive type

Generally, object-based programming languages such as Java and C++ define that an *Instance* of a *Class* comprising data members and methods must be created before its data members can be set or call operations can be called. However, in Fortran, a *Module* can comprise a variety of data without the need for instantiation; instead, input variables are defined that are sent to methods in the module. The functions and subroutines in Fortran are collectively referred to as Methods, as presented in Figure 2.

Figure 2 shows an example of Fortran-based source code that comprises 2 classes, namely, circle_test and Circle. Circle_test is the main program responsible for calling the Circle operation in the class_Circle module. The operation calls the subroutine using the code call circle_print(c), which calls Circle, passing the variable *c*. The subroutine circle_print of Circle includes a function call, area = circle_area(this), which calls another function in the same class.

```
module class_Circle
  implicit none
  private
  public :: Circle, circle_area, circle_print

  real :: pi = 3.1415926535897931d0 ! Class-wide private constant

  type Circle
     real :: radius
  end type Circle
contains
  function circle_area(this) result(area)
    type(Circle), intent(in) :: this
    real :: area
    area = pi * this%radius**2
  end function circle_area

  subroutine circle_print(this)
    type(Circle), intent(in) :: this
    real :: area
    area = circle_area(this)  ! Call the circle_area function
    print *, 'Circle: r = ', this%radius, ' area = ', area
  end subroutine circle_print
end module class_Circle

program circle_test
  use class_Circle
  implicit none

  type(Circle) :: c     ! Declare a variable of type Circle.
  c = Circle(1.5)       ! Use the implicit constructor, radius = 1.5.
  call circle_print(c)  ! Call a class subroutine
end program circle_test
```

Figure 2: Fortran code example.

Because Fortran is newly extended to the object-oriented programming world, there are few tools available, and the concept of software engineering has been less adopted for software development. There are very fewer software engineering tools available for modern Fortran compared to any other object-oriented programming language—especially tools in the category of software analysis, which can help software developers and designers better understand the source code or software systems.

2.3. Literature Review. Table 2 shows a comparison between this study and the related works categorized into two groups as follows.

(1) Literature relating to transformation rule designs. The authors of [33, 34] discussed the design of transformation rules between source code and diagrams. Comparing those works to this study, the former applied a model transformation technique using ATLAS, while we applied this technique to build transformation rules to transform the source code in Fortran to UML sequence diagrams.

(2) Literature relating to the transformation process designs. The authors of [35–38] discussed process designs for transforming the source code to diagrams and vice versa using XMI as a data exchange medium. Comparing the aforementioned studies to this study, the prior research papers are related to object-based programming languages, while this study aims to transform Fortran, which is an object-oriented programming language, to UML sequence diagrams. Thus, we can apply concepts from those studies to create a transformation process.

3. Research Methodology

In this section, we describe our designed process and develop a new feature for ForUML.

3.1. Design of the Fortran-Based Source Code Transformation Process to UML Sequence Diagrams. This section describes a transformation process between Fortran-based source code and UML sequence diagrams that includes 4 steps, as shown in Figure 3.

Table 2: Summarization of related work.

Work	Transformation rules design	ATLAS	Transformation process design	UML diagrams					Programming languages				XMI exchange
				Class diagram	Sequence diagram	Use case diagram	Activity diagram	Petri nets diagram	Fortran	Java	C++	PHP	
Our work	✓	✓	✓	—	✓	—		—	✓	—	—	—	✓
Merah [33]	✓	✓	—	—	✓	—	—	✓	—	—	—	—	—
Sawprakhon and Limpiyakorn [34]	✓	✓	—	✓	✓	✓	—	—	—	—	—	—	✓
Briand et al. [35]	—	—	✓		✓	—	—	—	—	—	✓	—	✓
Alalfi et al. [36]	—	—	✓	—	✓	—	—	—	—	—	—	✓	✓
Korshunova et al. [37]	—	—	✓	✓	✓	—	✓	—	—	—	✓	—	✓
Parada et al. [38]	—	—	✓	✓	✓	—	—	—	—	✓	—	—	✓

Figure 3 presents a 4-step procedure to transform Fortran source code to a UML sequence diagram.

(1) *Parsing Source Code into Different Parts Using the Open Fortran Parser (OFP) Library.* This step uses Fortran grammars and syntaxes already developed in the OFP library (http://fortran-parser.sourceforge.net) and is done with ANTLR (https://www.antlr.org/), which can parse source code into smaller chunks for which relationships can be found in the next step.

(2) *Finding Relationships between Source Code Chunks.* This step applies designed transformation rules to convert source code in Fortran to UML sequence diagrams and then, as mentioned in Step 1, finds the relationships between the various source code chunks for each transformation rule.

(3) *Taking Derived Relationships to Create an XMI File.* This XMI file is stored in the form of Document Type Definitions (DTDs), which is a popular schema for building XMI files because it specifies the transformation patterns between the derived relationships and the XMI files that are used to represent a UML sequence diagram.

(4) *Import the XMI Files into Modelio.* Modelio is an open-source UML diagram creation tool; consequently, developers can alter the software. At present, Modelio is gaining developer interest and is under continuous development; thus, there is a data source for customizing the software functionality to meet various research needs. To open a UML diagram in another tool, a user can export an XMI file from ForUML and then import it to that tool.

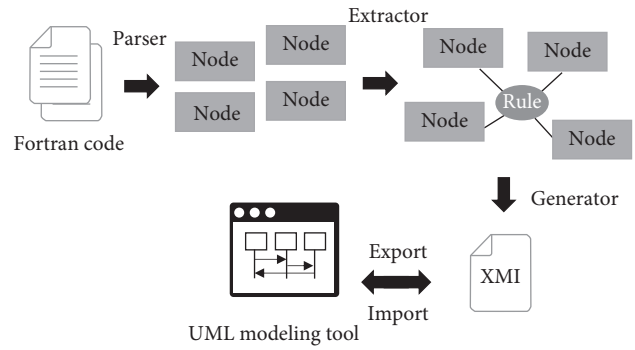


Figure 3: A transformation process.

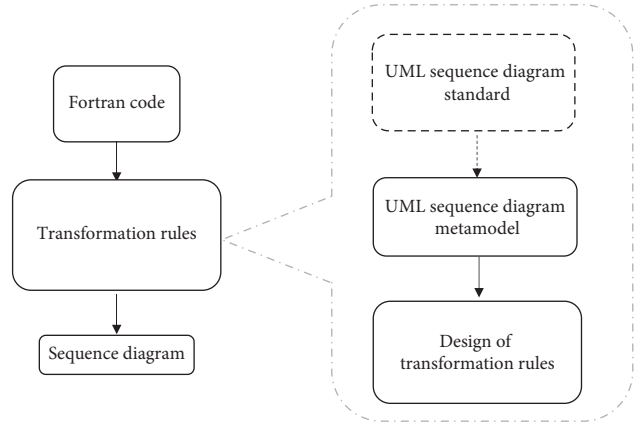


Figure 4: Concept of Fortran transformation rules.

Table 3: Mapping between the XMI and Fortran.

Fortran	XMI elements
Derived type	UML: class
Type-bound procedure	UML: operation
Dummy argument	UML: parameter
Component	UML: attribute
Intrinsic type	UML: datatype
Parent type	UML: generalization.parent
Extended type	UML: generalization.child
Composite	UML: association (the aggregation property as "composite")

3.2. Design of Fortran-Based Source Code Transformation Rules to UML Sequence Diagrams. In this section, the idea for designing Fortran-based source code transformation rules for UML sequence diagrams is discussed first. This idea is derived from UML sequence diagram standards and based on the UML specification [39] and the UML sequence diagram transformation rules from [33, 34, 40], and it is intended to build new Fortran-based source code transformation rules to create UML sequence diagrams. Finally, an explanation of the transformation rules is presented.

Figure 4 shows the basic idea for designing Fortran-based source code transformation rules for UML sequence diagrams. It starts by designing a UML sequence diagram metamodel for each transformation rule. This metamodel is represented as a class diagram, which is a model showing patterns in Fortran source code and UML sequence diagrams and represents UML sequence diagram transformation rules and standards for data exchange. The UML sequence diagram metamodel design considers the relationships between the source code and the XMI file regarding the generation of class diagrams from ForUML.

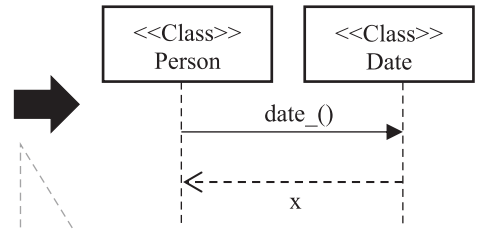
The XMI files relating to class diagram generation in ForUML are generated from source code transformation rules in Fortran as presented in Table 3. Here, derived types, type-bound procedures, dummy arguments, and components are applied to create rules for transforming the Fortran-based source code to UML sequence diagrams.

Regarding the design of Fortran-based source code transformation rules to UML sequence diagrams, the authors studied XMI files for generating sequence diagrams under the Object Management Group (OMG) standard. These XMI files describe the rules for transforming the source code into UML sequence diagrams and are also used as a data exchange standard. An example of an XMI document representing a UML sequence diagram is listed in Figure 5 with the following details.

(1) xmi:type = "uml:Lifeline" defines lifeline details, where xmi:id = "66rKFjKG" is a lifeline identification number and name = "Person" is a name in the lifeline

```
module class_Person
 public :: 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 5: Example of XMI for UML sequence diagram.

(2) xmi:type = “uml:Message” defines message details, where xmi:id = “Xhr8sYT” is a message identification number, messageSort = “reply” represents a message type, name = “Person” is a message name, receiveEvent = “Person” represents a lifeline receiving a message, and sendEvent = “Date” represents a lifeline sending a message

The main step in designing transformation rules for Fortran-based source code to UML sequence diagrams is to find the relationships between an AST metamodel of the Fortran language and an XMI file. These metamodels form the main models, as shown in Figure 6.

Figure 6 shows the transformation process to convert the Fortran source code to UML sequence diagrams. The process consists of an AST metamodel consistent with the source code and a sequence diagram metamodel consistent with an XMI file. Information concerning the AST metamodel is derived from the source code decomposition through an OFP library. These parts of the source code are then considered to specify the relationships for creating transformation rules to transform the source code into UML sequence diagrams and define the XMI file creation standards. The sequence diagram metamodel data conform to the UML version 2.1 specification managed by OMG.

After studying the UML sequence diagram standards based on the UML specification and the UML sequence diagram transformation rules from prior studies [33, 34, 40],

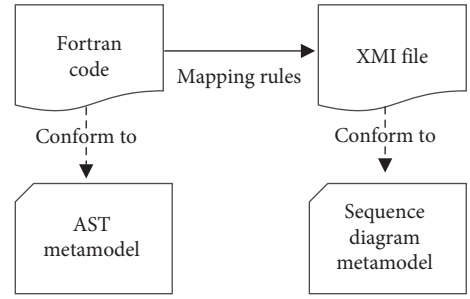


Figure 6: Fortran’s transformation process for UML sequence diagram.

we specified the Fortran-based rules to transform the source code into UML sequence diagrams as follows [41]:

(1) Lifeline creation rules
(2) Message creation rules
(3) Rules for sending and receiving messages
(4) Rules for defining the starts and ends of operations
(5) Rules for defining message operations on the lifeline
(6) Frame element creation rules

3.3. Software System Development. After designing the Fortran-based source code transformation process and the

rules for UML sequence diagrams, we decided to add the new feature to the current ForUML with a new GUI. Other development tools included NetBeans for coding Java applications and Modelio for rendering UML sequence diagrams from an XMI file.

3.4. Accuracy Evaluation of Fortran-Based Source Code Transformation to UML Sequence Diagrams. The evaluation of the accuracy of the results from transforming Fortran source code into UML sequence diagrams was based on comparing UML sequence diagrams generated by ForUML and the system's source code as evaluated by the authors of this study. We started the evaluation by verifying the total number of source code components in the system, followed by verifying a number of sequence diagram notations that are consistent with the system source code. Then, we compared a defined number of source code components to a number of items in the corresponding UML sequence diagram by creating a small program to count a number of source code components and reduce errors that may arise from manual counting. The numbers of source code components evaluated in the system are categorized as follows:

(1) The number of classes (Types) in the system; these must be located in a package called Module or Program Main in Fortran.

(2) The number of methods (Functions or Subroutines) called from other classes internally. These include calling functions and subroutines.

(3) The number of function calls in the system.

(4) The number of subroutine calls in the system.

(5) The number of statements that call internal functions or subroutines. These statements are conditions, multiple alternatives, and iterations with internal operations.

Information about the numbers of source code components in the tested system is crucial to UML sequence diagram generation, which aligns with the designed transformation rules as follows: (1) the number of Types must be consistent with the lifeline creation rules and the rules for defining message operations on the lifeline, (2) the number of functions or subroutines must accord with the message creation rules between lifelines, (3) the number of function and subroutine calls must be consistent with the rules for sending and receiving messages and the rules for defining the starts and ends of operations, and (4) the number of statements must accord with frame element creation rules. When the number of source code components in the testing system matches the number of notations in the UML sequence diagrams from the tool, that result is considered to be correct. A list of the software used for these tests is shown below:

(1) ForTrilinos [42] is open-source software with a user interface developed in Fortran for running Trilinos; the main program comprises a set of libraries for solving scientific and engineering problems.

(2) PSBLAS [43] is open-source software for solving a parallel sparse matrix developed based on Fortran 2003.

(3) MLD2P4 [44] is open-source software for solving a linear system developed in Fortran 2003.

The process for selecting tested software considered whether the software was developed from an object-oriented Fortran language, whether it was created for scientific and engineering purposes, and whether it is in active use.

4. Results

This study aimed to design and develop software to transform Fortran source code into UML sequence diagrams. The system is composed of 4 parts: (1) a Fortran-based source code management unit, (2) a unit for extracting relationships between the source code and UML sequence diagrams, (3) an XMI file generation unit, and (4) a unit to render UML sequence diagrams from XMI files. All four parts were developed for use with ForUML as follows.

4.1. Fortran-Based Source Code Management. Figure 7 presents a ForUML screenshot showing seven main buttons: (1) an "Add" button to add source code files, (2) a "Remove" button to remove the source code files, (3) a "Reset" button to reset the system when any error occurs, (4) "Class Diagram" and "Sequence Diagram" buttons to generate XMI files for the respective type of UML diagram, (5) a "View" button to view UML diagrams via Modelio, (6) a "Save as" button to select a location for saving XMI files, and (7) a "Status/Log" button to view messages when the system succeeds or an error occurs.

Fortran-based source code management starts by adding a file: the user clicks the "Add" button or drags a file to a program. Only files with an F90 extension containing the source code written in the object-oriented form of the programming language are applicable. After adding the file successfully, details of the added file are displayed on the system screen. The user can remove a document by selecting that file and then clicking the "Remove" button or reset the system by clicking the "Reset" button. When saving an XMI file, the user must specify a save location.

4.2. Extracting Relationships between Source Code in Fortran and UML Sequence Diagrams. After adding the Fortran-based source code file to the system, the relationships between the Fortran source code and the UML sequence diagrams are extracted. Using the OFP library, several parts of the source code can be extracted in the form of an AST structure. These parts are then used to retrieve the relationships and create an XMI file for the UML sequence diagram. According to the OMG standard, building an XMI file of the UML sequence diagram requires specifying five important elements: (1) lifeline, (2) message, (3) message occurrence specification, (4) execution occurrence specification, and (5) behavior execution specification. The authors developed a library to find the relationships between

Figure 7: Screenshot of ForUML.

the source code and the diagram. The library includes the classes presented in Figure 8, which have the following purposes:

(1) *Type.* The class name from the Fortran source code, which is used to specify details about the lifeline.

(2) *Function.* The method name from the source code, which is used to specify details about a message calling a particular function.

(3) *Subroutine.* The method name from the source code, which is used to specify details about a message calling a particular subroutine.

(4) *SD_Lifeline.* This is responsible for extracting a Type's name from the source code to specify lifeline details and the actions that belong to a particular lifeline. The SD_EOS class is linked to specify the lifeline's end of operation.

(5) *SD_Function.* This is responsible for extracting Function names from the source code to specify message details. The function call specification describes how to send or receive a message and the SD_MOS class specifies whether a message belongs to a particular lifeline.

(6) *SD_Subroutine.* This is responsible for extracting subroutine names from the source code to specify message details. A subroutine call specification describes how to send or receive a message, and the SD_MOS class is linked to specify whether a message belongs to a particular lifeline.

(7) *SD_EOS.* This specifies where the lifeline's end of operation is by verifying the function or subroutine call. When a function or subroutine includes no further calls to another type, it is assumed that the function or subroutine ends. Additionally, the SD_BES class is linked to specify message behaviors on a lifeline.

(8) *SD_MOS.* This specifies which message operations occur on which lifeline by verifying which functions or subroutines involve operations on which Types in the source code. Additionally, the SD_BES class is linked to specify message behaviors on a lifeline.

(9) *SD_BES.* This specifies operational behaviors between messages and lifelines by verifying function or subroutine calls between Types. When a call exists, the system specifies the beginning and end of the called lifeline by obtaining data from the SD_MOS class, and when a function or subroutine includes no other calls, the system specifies the end of the operation by receiving data from the SD_EOS class.

From the relationship structure between the source code and the XMI file of the UML sequence diagram, to aid in understanding, we created a UML sequence diagram of the structure, as shown in Figure 9, by importing some source code to the system, which then validates the source code. If validation fails, the system displays a message in the Status/Log window. After successfully validating the source code, the system parses the data, which consist of types, functions, and subroutines, and are stored in SD_Lifeline, SD_Function, and SD_Subroutine classes, to extract relationships. Classes specifying the relationships include SD_EOS, SD_MOS, and SD_BES. Finally, after successfully extracting the relationships, the system will generate an XMI file.

4.3. XMI File Generation. XMI file generation is the next step after extracting the relationships between the Fortran source code and the UML sequence diagram. These files conform to XMI file generation rules. We developed an XMI file generation library, as shown in Figure 10. The library has the following components:

(1) *ParserProcessor.* This component starts by checking the source code file, which must have an extension of F90, include a module component, and contain no source code errors. The validated file is then decomposed using the OFP library, and relationships are found as described in the relationship extraction step. The derived relationships are passed back to the Generator class, which generates an XMI file.

(2) *Generator.* This component generates an XMI file for a UML sequence diagram version 2.0 under the OMG XMI file generation rules.

To aid in understanding, we created a UML sequence diagram form an XMI file generation structure, as shown in Figure 11. The system begins by sending the data to be used for XMI file generation to the ParserProcessor class. The data resulting from the extracted relationships between the verified Fortran source code and the UML sequence diagram are passed to the Generator class, which generates an XMI file.

4.4. Rendering a UML Sequence Diagram from an XMI File. Rendering a UML sequence diagram from an XMI file is performed by Modelio, which is open-source software under the GPL license that generates or renders UML diagrams. The latest version is Modelio 3.7, which supports UML 2.0; we selected this version to render UML sequence diagrams.

To display a UML sequence diagram, the user clicks the "View" button. The system is integrated with Modelio, so it is

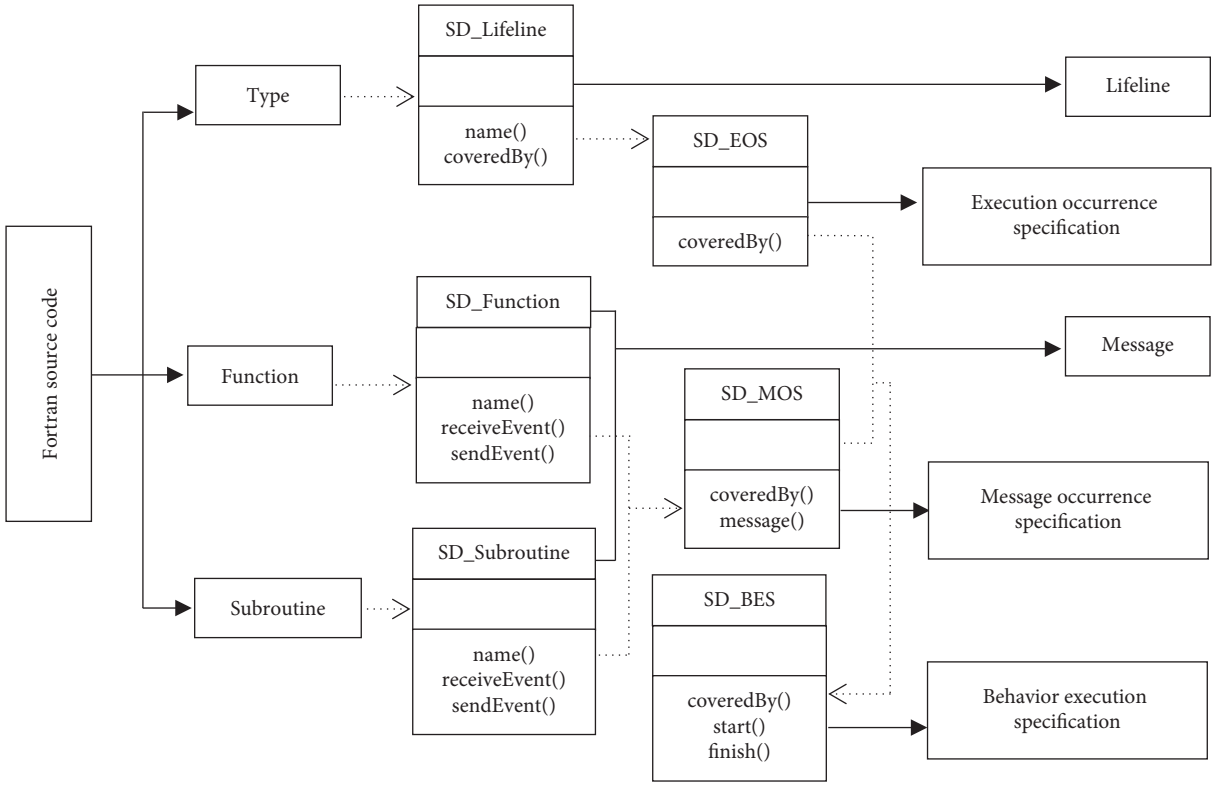


Figure 8: Relationships between source code in Fortran and UML sequence diagrams.

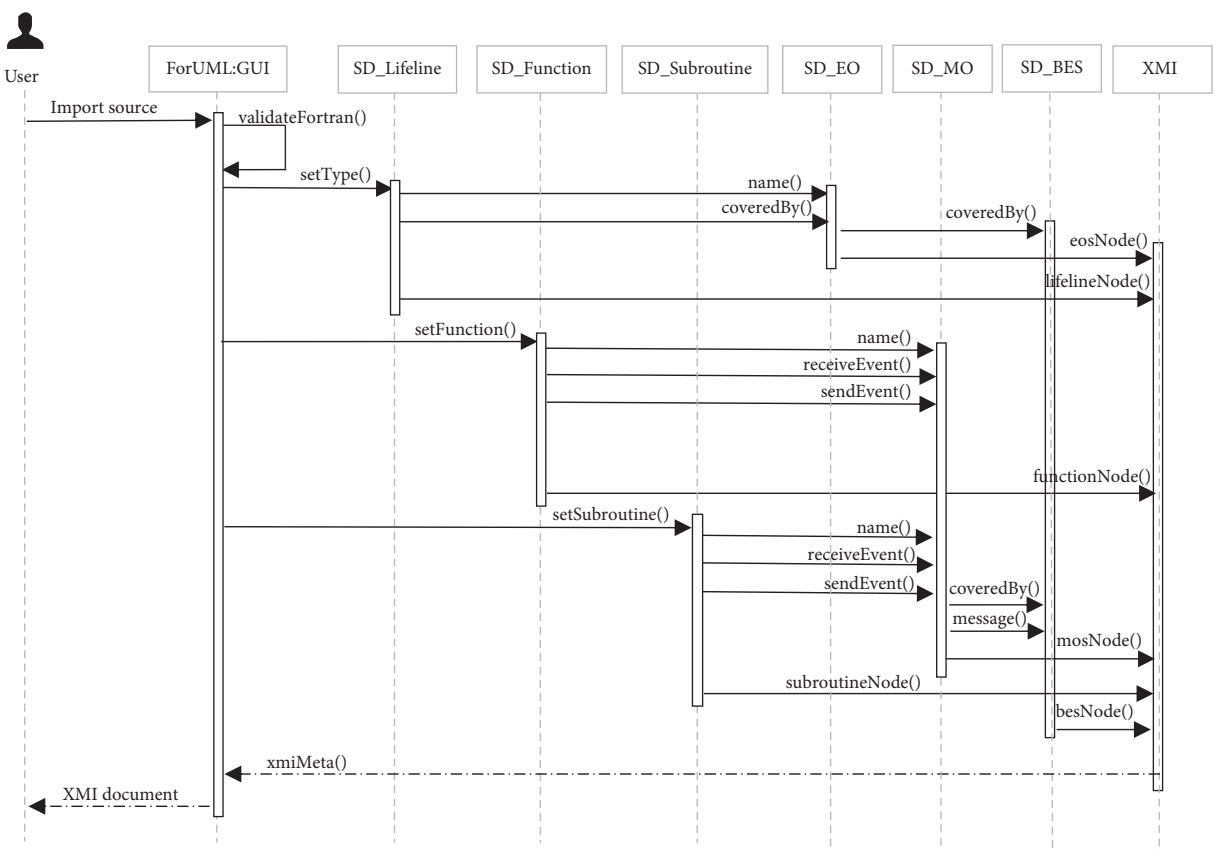


Figure 9: UML sequence diagram of relationships between Fortran code and XMI

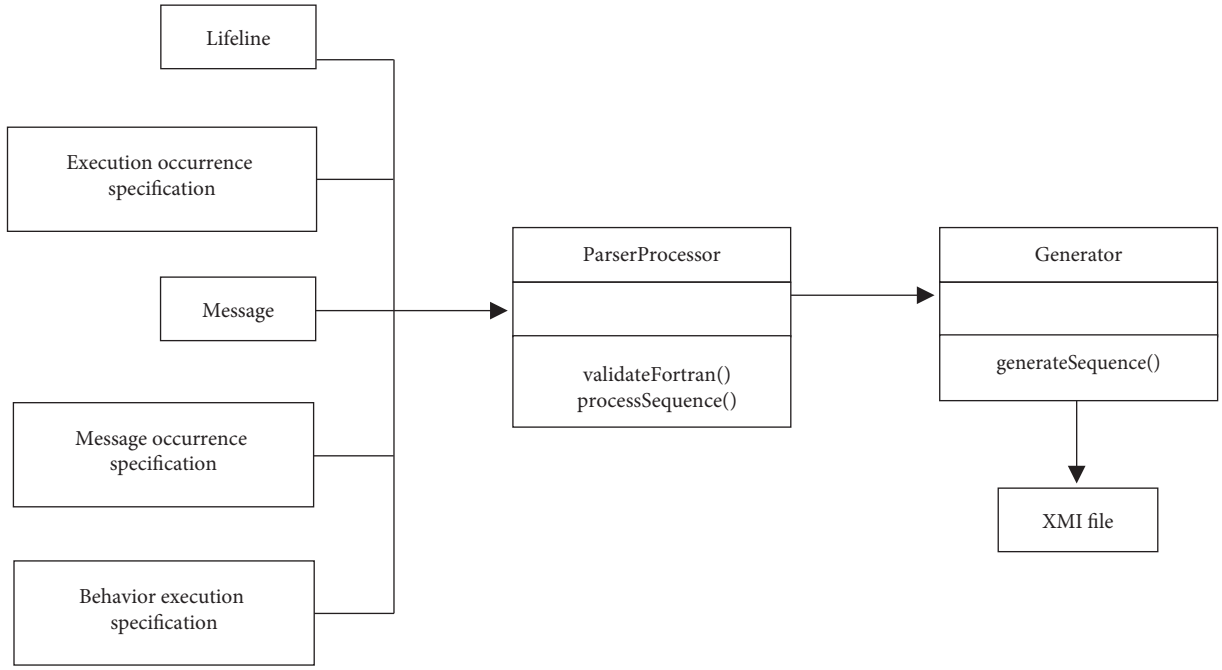


Figure 10: Relationships of classes for XMI generation.

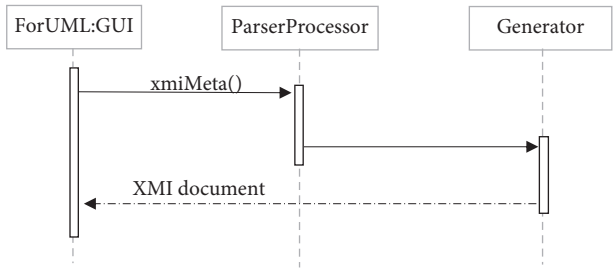


Figure 11: Relationships between Fortran code and XMI

unnecessary to install the software separately. An example of a UML sequence diagram rendered by Modelio is shown in Figure 12.

5. Evaluation

This section will describe the evaluation of the developed system by comparing the following results from ForUML in the form of UML sequence diagrams and Fortran-based source code from actual tests of the Fortran software packages: (1) the number of Types that represent all the classes involved, (2) the number of procedures, which represents all the methods called internally, (3) the number of function calls, which represents all function calls, (4) the number of subroutine calls, which represents all subroutine calls, and (5) the number of statements, which represents all the conditionals, multiple alternatives, and iterations involved in internal method calls. Three software packages, including PSBLAS, MLD2P4, and ForTrilinos, were used in this experiment.

For the experiment, we evaluated the comparison rules by validating a number of Fortran files based on evaluations by the authors and then asked experts to verify their accuracy. By analyzing the source code, we counted the number of classes in each tested software package starting from the main program and counting the called classes from the start to the finish of the source code. For the UML sequence diagram, we counted the following items to ensure that they matched the source code as follows:

(1) The number of lifelines is consistent with the number of Types, which represent all the classes used.

(2) The number of operations on a lifeline that send a message to another lifeline are consistent with the number of procedures that represent all the methods called internally.

(3) The number of messages on a lifeline sent to corresponding functions are consistent with the number of function calls representing all function calls.

(4) The number of messages on a lifeline sent to corresponding subroutines are consistent with the number of subroutine calls representing all subroutine calls.

(5) The number of frame elements on a lifeline are consistent with the number of statements relating to conditionals, multiple alternatives, and iterations that call methods internally.

The test results are determined by comparing the differences between the source code and the UML sequence diagrams by calculating the percentage of accuracy.

5.1. PSBLAS. The first software package contains 8 classes. We created a class for the main program, namely, hello, to call the software package. The hello class includes 5 subroutines: psb_init, psb_info, psb_rcv, psb_snd, and psb_exit. Each subroutine represents a function in the software

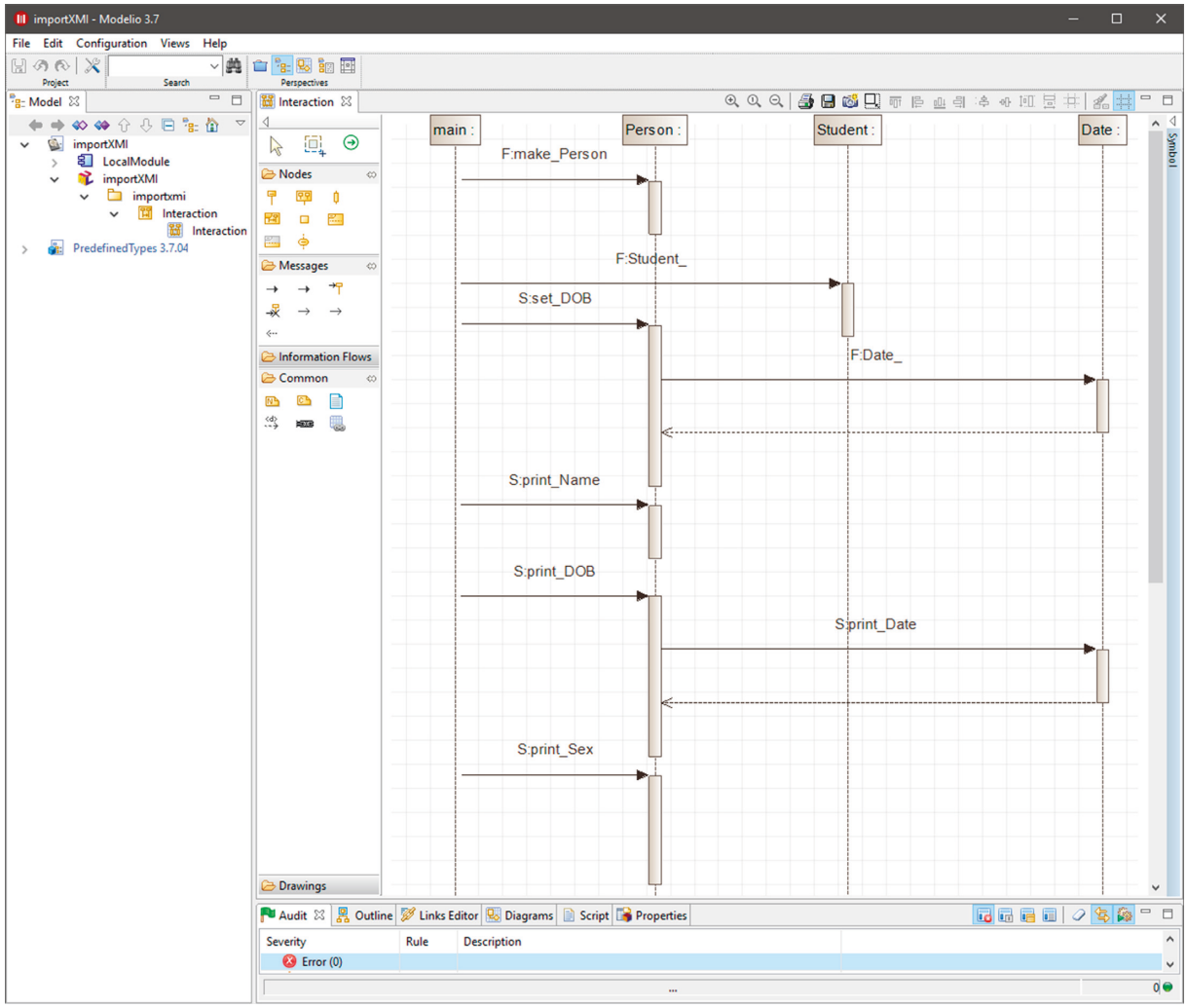


Figure 12: UML sequence diagram is shown in Modelio.

package. In other words, each subroutine in the software package is responsible for a corresponding interface, which then calls a subroutine, as shown in Figure 13.

Figure 13 shows an interface call under the main program. Because psb_snd and psb_rcv make several method calls, when counting messages in the UML sequence diagram, the number of messages is larger than the actual number of calls in the Fortran source code. This result occurs because the main program calls the interface, which then calls methods. However, a UML sequence diagram displays the result by replacing methods under the interface with existing methods, which can lead to excess data. To illustrate, as shown in Table 4, the hello class has 5 subroutine calls in the source code as opposed to the 100 messages shown in the UML sequence diagram. Thus, we rely on the source code counts when sending interface calls. If the number of UML sequence diagram notations is greater than or equal to the source code components, we interpret the results as 100% coverage.

Table 4 shows a difference comparison of the results between the source code and the UML sequence diagram for PSBLAS. The table shows the counted number of UML sequence diagram notations and the counted number of source code components comprising the 8 classes described below.

(1) *Hello*. This main program makes no method calls to the same program, no function calls, and makes subroutine calls to 5 subroutines; however, the UML sequence diagram shows 100 subroutines because the interface calls were mixed up and it has no conditional calls.

(2) *psb_init*. This called interface has 1 method that calls a corresponding method in another class under the same system, no function calls, makes subroutine calls to 3 subroutines, and has 1 conditional call.

(3) *psb_info*. This called interface makes no calls to other methods under the same interface, no function calls, no subroutine calls, and no conditional calls.

(4) *psb_rcv*. This called interface has 28 methods that make calls to corresponding methods in another class under the same system, no function calls, makes subroutine calls to 28 subroutines, and has no conditional calls.

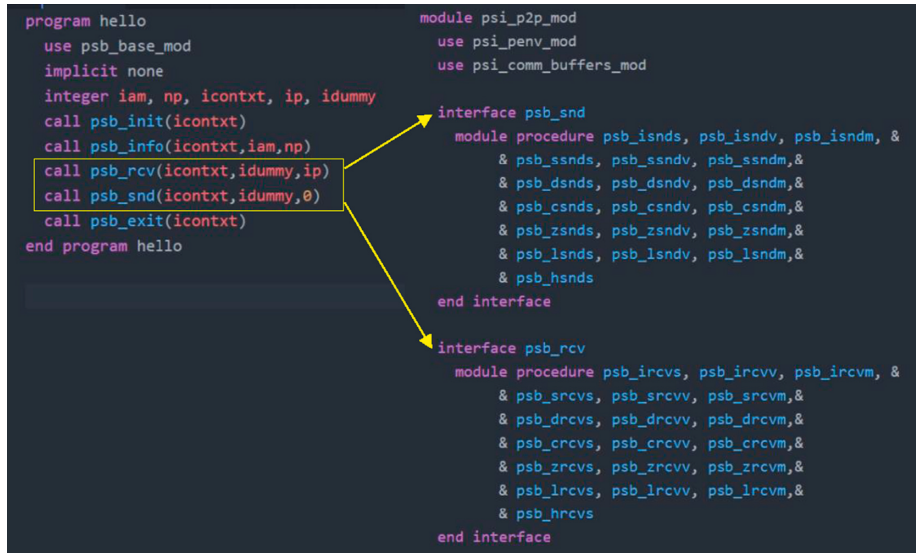


Figure 13: Calling interface.

Table 4: Evaluation results of PSBLAS.

Type	Procedure	Function call	Subroutine call	Statements
Hello	0/0	0/0	100/5	0/0
psb_init	1/1	0/0	3/3	1/1
psb_info	0/0	0/0	0/0	0/0
psb_rcv	28/28	0/0	28/28	0/0
psb_snd	0/0	0/0	0/0	0/0
psb_errstack	0/0	0/0	0/0	0/0
psb_exit	1/1	0/0	2/2	1/1
psb_buffer_node	0/0	0/0	0/0	0/0
Overall	30/30 100%	0/0 100%	133/38 100%	2/2 100%

(5) *psb_snd*. This called interface makes no calls to other methods in the same interface, no function calls, no subroutine calls, and no conditional calls.

(6) *psb_errstac*. This called class makes no calls to other methods in the same class, no function calls, no subroutine calls, and has no conditional calls.

(7) *psb_exit*. This called interface has 1 method that calls a corresponding method in another class under the same system, no function calls, makes subroutine calls to 2 subroutines, and has 1 conditional call.

(8) *psb_buffer_node*. This called class makes no calls to other methods in the same class, no function calls, no subroutine calls, and no conditional calls.

A UML sequence diagram of PSBLAS consists of 8 lifelines as presented in Figures 14–16. The hello lifeline includes 100 subroutine calls to the other lifelines. The psb_init lifeline shows 1 called method call and 3 subroutine calls to other lifelines. The psb_rcv lifeline shows 28 called methods and 28 subroutine calls to other lifelines. The psb_exit lifeline shows 1 called method and 2 subroutine calls to other lifelines. The psb_info, psb_snd, psb_errstack, and psb_buffer_node lifelines have no called methods; thus, they do not send messages to the other lifelines.

The comparison results between the source code and the UML sequence diagram show that PSBLAS's source code can be completely transformed into a UML sequence diagram. In other words, the UML sequence diagram's notations match the source code components. However, the hello class had additional counts due to interface calls.

5.2. MLD2P4. The second software package contains 10 classes. We created a class for the main program, namely, mld_dexample_1lev, to call the software package. In the mld_dexample_1lev class, there are 7 subroutine calls as follows: psb_init, psb_info, psb_barrier, psb_amx, and psb_exit are subroutines that call an interface, and psb_set_errverbosity and psb_errpush are subroutines that call a method. In addition, there are 2 function calls, namely, psb_genrm2 and psb_geamax, both of which are interface calls.

Table 5 shows the results of the difference comparison between the source code and the UML sequence diagram for MLD2P4. The table shows the relationship between the counts from the UML sequence diagram notations and the counted number of source code components for the 10 classes described below:

(1) *Mld_dexample_1lev*. This main program has no methods called in the same program and makes a function call to 2 functions in the Fortran source code; however, the UML sequence diagram shows 6 functions due to the interface calls. Additionally, the main program makes subroutine calls to 7 subroutines in the source code, but the UML sequence diagram shows 29 subroutines due to interface calls and includes 1 conditional call.

(2) *psb_init*. This called interface has 1 method that calls a corresponding method in another class under

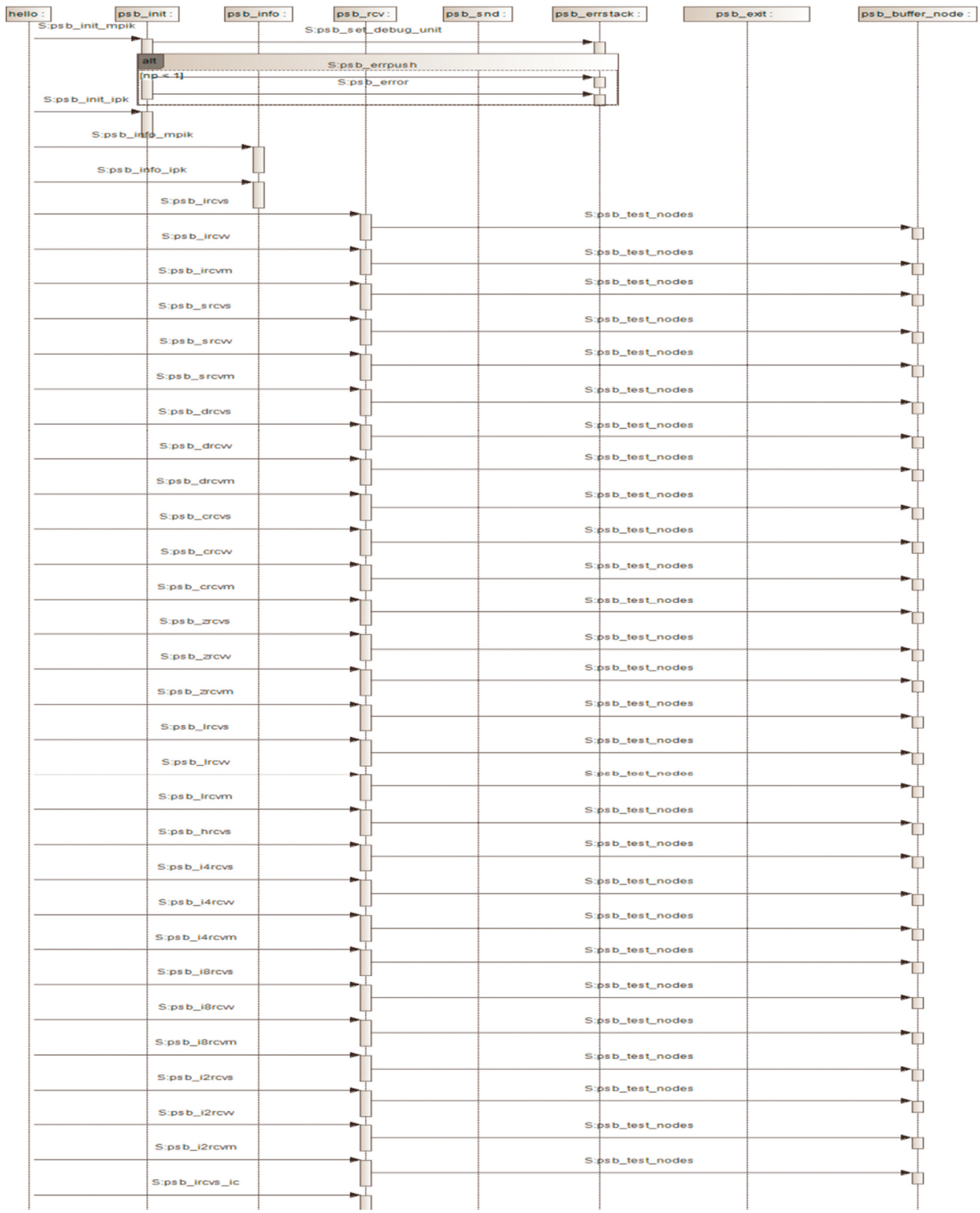


Figure 14: UML sequence diagram of PSBLAS (part I).

the same system, no function calls, makes subroutine calls to 3 subroutines, and has 1 conditional call.

(3) *psb_info*. This called interface makes no calls to other methods in the same interface, no function calls, no subroutine calls, and no conditional calls.

(4) *psb_barrier*. This called interface makes no calls to other methods in the same interface, no function calls, no subroutine calls, and no conditional calls.

(5) *psb_errstack*. This called class makes no calls to other methods in the same class, no function calls, no subroutine calls, and no conditional calls.

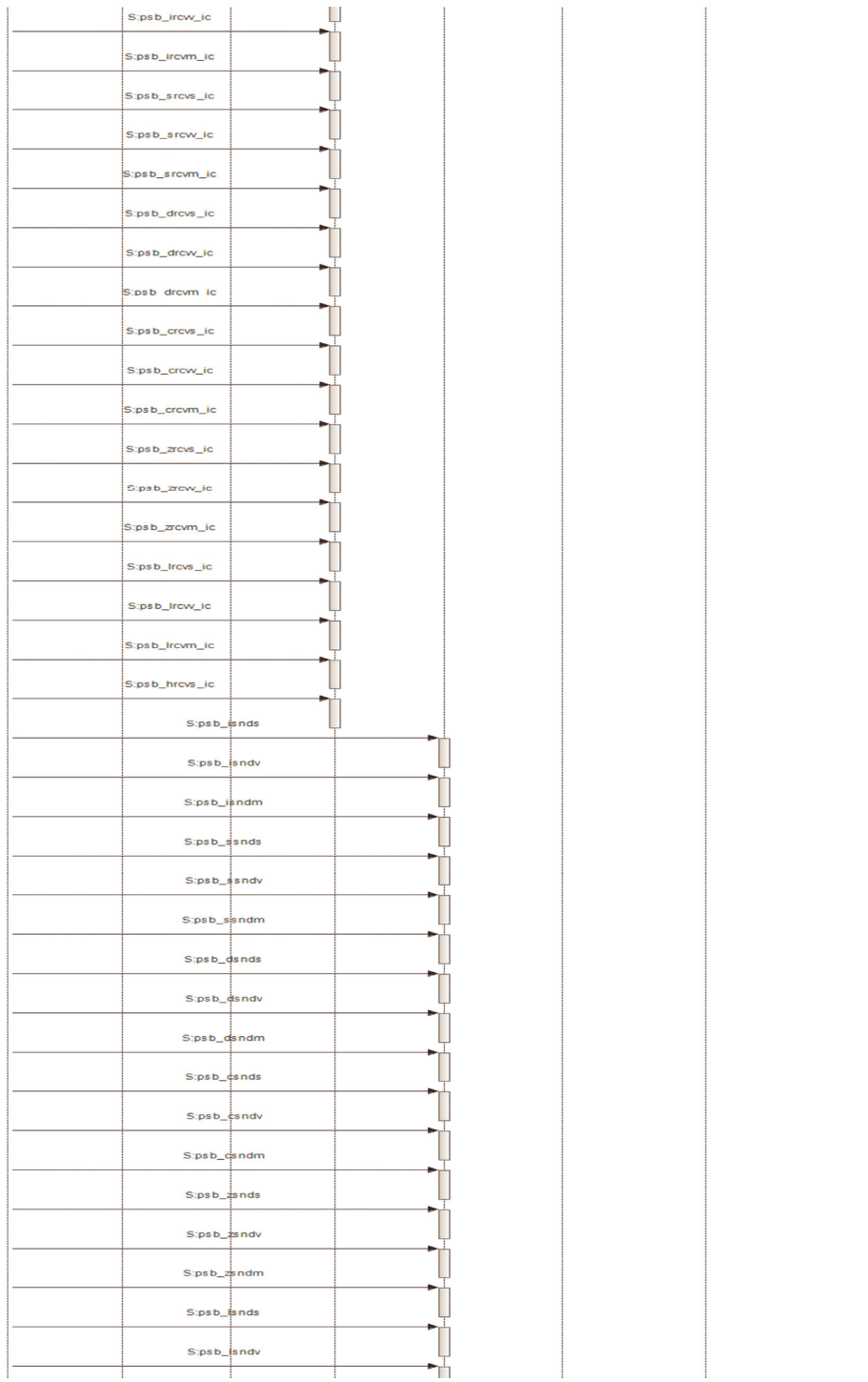


Figure 15: UML sequence diagram of PSBLAS (part II).

(6) *psb_amx*. This called interface makes no calls to another method under the same interface, has no function call, has no subroutine call, and has no conditional call.

(7) *psb_exit*. This called interface has 1 method that calls a corresponding method in another class in the same system, no function calls, makes subroutine calls to 2 subroutines, and has 1 conditional call.

(8) *psb_buffer_node*. This called class makes no calls to other methods in the same class, no function calls, no subroutine calls, and no conditional calls.

(9) *psb_genrm2*. This called interface makes no calls to other methods under the same interface, has no function calls, no subroutine calls, and no conditional calls.

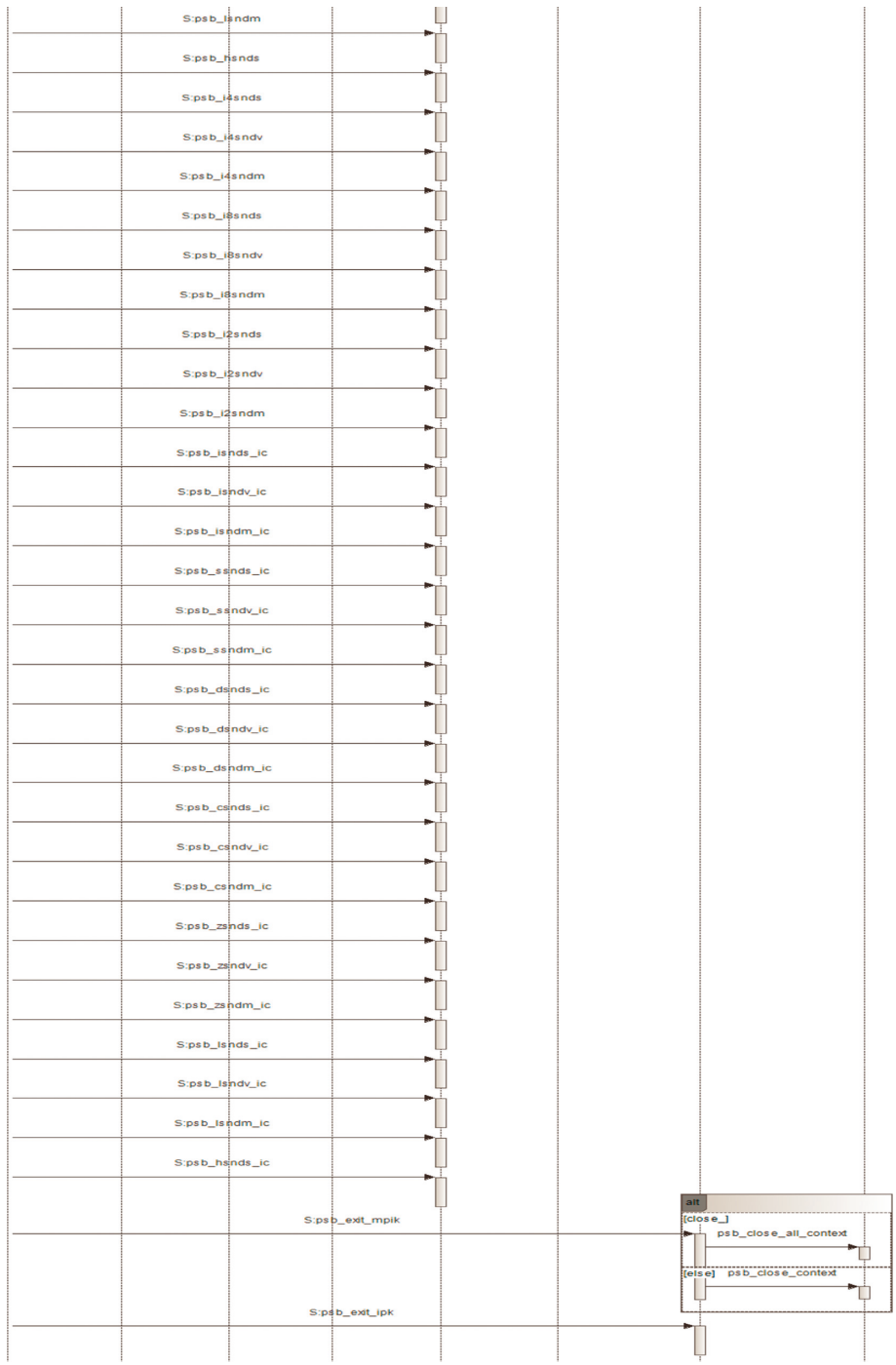


Figure 16: UML sequence diagram of PSBLAS (part III).

(10) *psb_geamax*. This interface has no call to another method under the same interface, has no function call, no subroutine calls, and no conditional calls.

The UML sequence diagram of MLD2P4 consists of 10 lifelines as presented in Figure 17. The mld_dexample_1lev lifeline has 6 function calls and 29 subroutine calls to another lifeline. The psb_init lifeline has 1 called method and makes 3 subroutine calls to another lifeline. The psb_exit lifeline has 1 method called and makes 2 subroutine calls to another lifeline. Additionally, psb_info, psb_barrier, psb_errstack, psb_amx, psb_buffer_node, psb_genrm2, and psb_geamax lifelines have no methods called; thus, they send no messages to another lifeline.

The result of the source code and UML sequence diagram comparison shows that the MLD2P4 source code can be completely transformed into a UML sequence diagram in which the sequence diagram's notations match the source code components. However, the mld_dexample_1lev class includes additional counts because of the interface call.

Table 5: Evaluation results of MLD2P4.

Type	Procedure	Function call	Subroutine call	Statements
mld_dexample_1lev	0/0	6/2	29/7	1/1
psb_init	1/1	0/0	3/3	1/1
psb_info	0/0	0/0	0/0	0/0
psb_barrier	0/0	0/0	0/0	0/0
psb_errstack	0/0	0/0	0/0	0/0
psb_amx	0/0	0/0	0/0	0/0
psb_exit	1/1	0/0	2/2	1/1
psb_buffer_node	0/0	0/0	0/0	0/0
psb_genrm2	0/0	0/0	0/0	0/0
psb_geamax	0/0	0/0	0/0	0/0
Overall	2/2 100%	6/2 100%	34/12 100%	3/3 100%

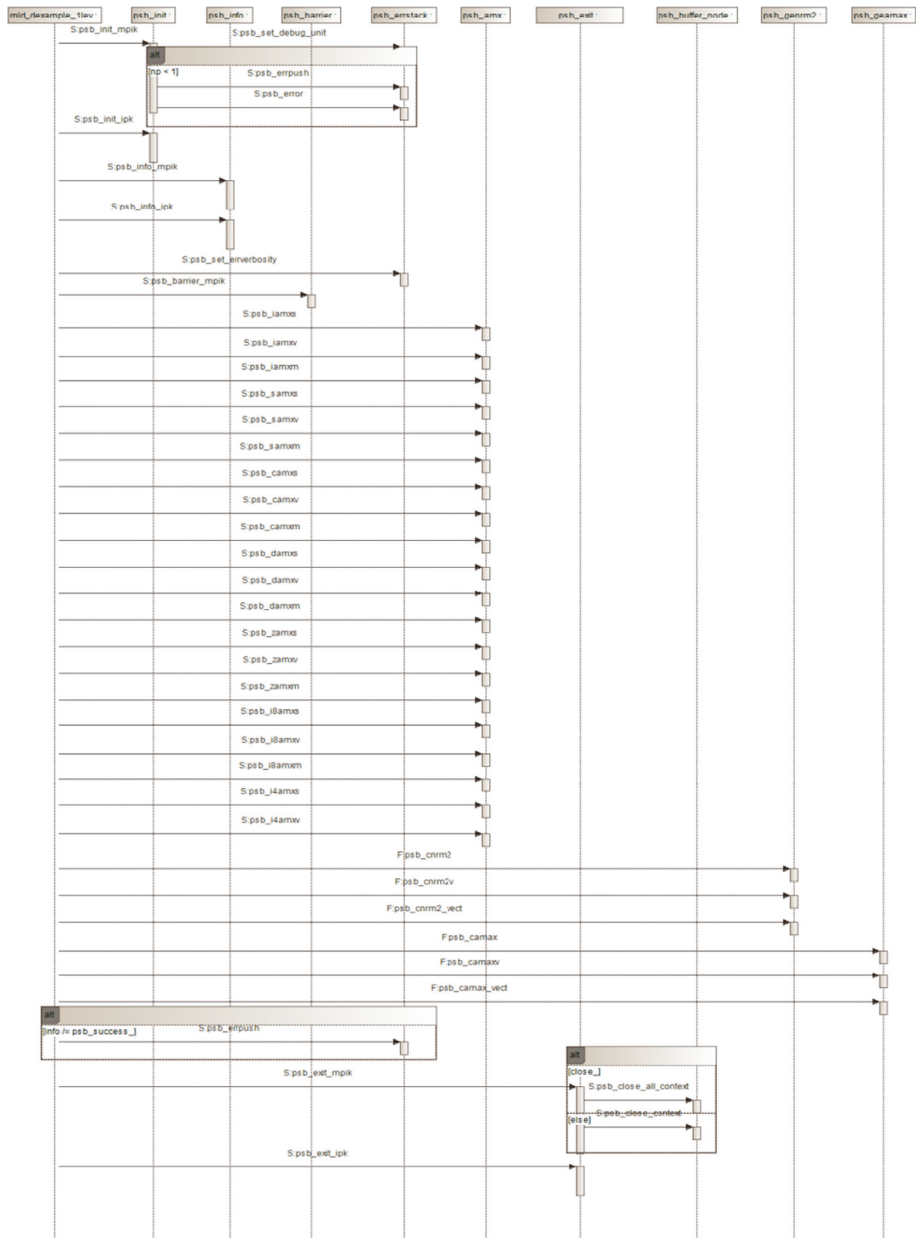


Figure 17: UML sequence diagram of MLD2P4.

5.3. ForTrilinos. The third software package contains 3 classes. We created a class for the main program, namely, the test_TpetraCrsGraph class, to call the software package. The test_TpetraCrsGraph class makes 3 subroutine calls, one each to createMPI, create, and release.

Table 6 shows the difference comparison results between the source code and the UML sequence diagram for ForTrilinos. The table shows the relationship between the counted number of UML sequence diagram notations and the counted number of source code components comprising the 3 classes described below:

(1) test_TpetraCrsGraph. This main program has no methods called in the same program, no function calls, makes subroutine calls to 3 subroutines in the source code (but the UML sequence diagram represents 5 subroutines due to the use of variables that refer to subroutines), and has 1 conditional call.

(2) TeuchosComm. This class has methods called via variables, has 5 methods called by the same class and makes calls to a corresponding method in another class under the same system. It makes 4 function calls, 1 subroutine call, and no conditional calls.

(3) C_PTR. This class makes no calls to other methods in the same class, no function calls, no subroutine calls, and no conditional calls.

The UML sequence diagram of ForTrilinos consists of 3 lifelines as presented in Figure 18. The test_TpetraCrsGraph lifeline makes 5 subroutine calls to another lifeline. The TeuchosComm lifeline includes 5 called methods and makes 4 function calls to another lifeline and 1 subroutine call to another lifeline. As per the C_PTR lifeline, no methods are called; thus, no messages are sent to other lifelines.

The comparison results between the source code and the UML sequence diagram show that the ForTrilinos source code can be completely transformed into a UML sequence diagram. In other words, the UML sequence diagram's notations match the source code components. However, the test_TpetraCrsGraph class included additional counts due to reference variables to the class's subroutines.

Table 6: Evaluation results of ForTrilinos.

Type	Procedure	Function call	Subroutine call	Statements
test_TpetraCrsGraph	0/0	0/0	5/3	1/1
TeuchosComm	5/5	4/4	1/1	0/0
C_PTR	0/0	0/0	0/0	0/0
Overall	5/5	4/4	6/4	1/1
	100%	100%	100%	100%

6. Case Study

In this section, we describe our experiences relating to the application of the proposed feature of ForUML on a real Fortran project. The groundwater flow simulation application used in this case study has been developed for use in an internal lab. The development team comprises three developers. Two of them hold an M.S. in physics and have worked on multiple object-oriented Fortran projects. The third developer holds an M.S. in computer science and is now pursuing his Ph.D. He has been working for a year and his objective is to gain experience in Fortran development. None of them have used the ForUML feature previously.

Expected users of this application include local scientists and graduate students. Many libraries in this application were developed by the students who had graduated, without any developers' documentation. Consequently, it is difficult for the current team to understand how the system works. Moreover, some open-source libraries were included in the application, such as PSBLAS-EXT (https://github.com/sfilippone/psblas3-ext), that contains an interface for handling GPUs. The development team asked us for approaches to reduce the code comprehension time. Thus, we introduced the ForUML feature to the team using a training session. During the development, we collected the data using two methods that include observation and interview. Observation was conducted by the second author for 4-5 hours per week for a month, while interviews were performed by the first author. The interviews aimed to assess the experience of the participants in relation to their use of the ForUML feature and the UML sequence diagrams. Questions were asked about the participants' opinions regarding this feature, including its benefits and problems.

As per the interview responses, all the participants felt that the sequence diagrams helped them better understand how objects interact in their code and how these interactions may change over time. They also used the tool to confirm the correct implementation of new features rather than reviewing the source code. One of the participants indicated that he used the sequence diagrams to identify the code smells and accordingly refactored that part of the code. More specifically, he observed many similar behaviors in the same class which he later removed and merged some functions. One participant mentioned as follows: "The UML diagrams are useful. In particular, the diagrams help us capture the same design patterns that exist in the libraries, such as the State pattern which allows us to have the same code running on normal CPUs as well as GPUs, even in heterogeneous mode."

However, we received the following feedback about the challenges in using ForUML:

(i) The process of showing the sequence diagram is slow when the library is large.

(ii) It would be useful if the sequence diagram can also show some control graph as a part of the diagram.

(iii) The tool should provide the descriptions of coarrays along with the number of dimensions and codimensions.

(iv) It would be great if ForUML can generate software design metrics and provide refactoring strategies based on such metrics.

Based on all these observations, we have found that the tool has been used effectively by the users to understand

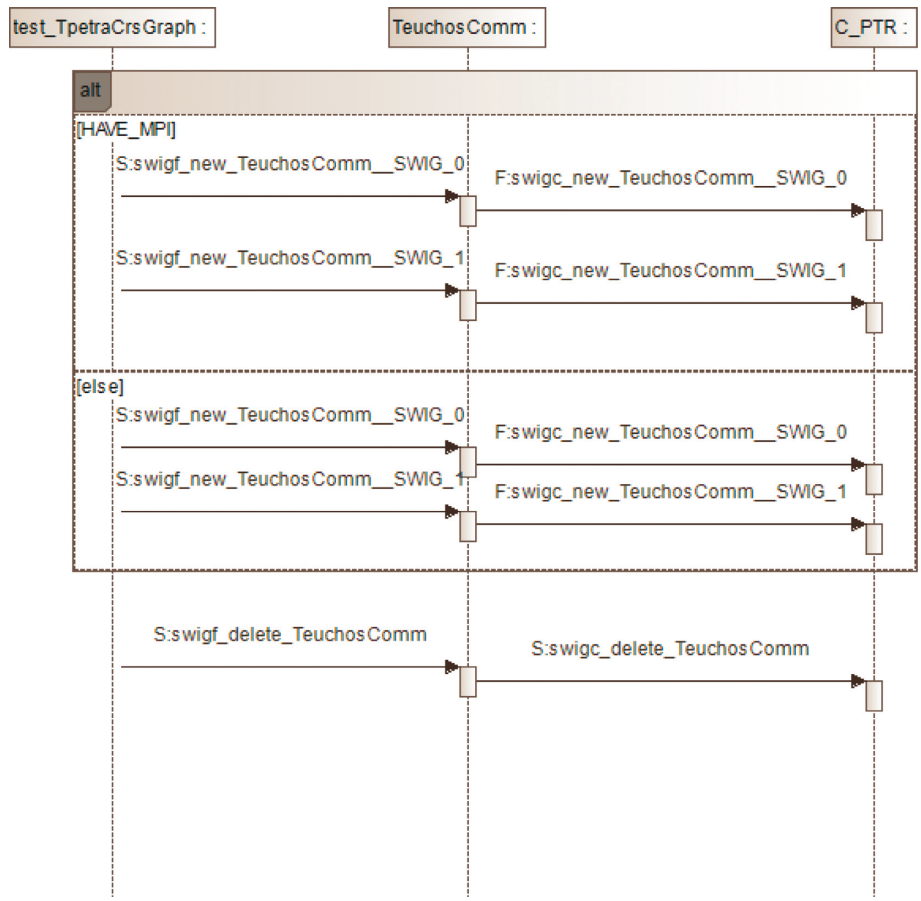


Figure 18: UML sequence diagram of ForTrilinos.

the flow of the programs, particularly those in the external libraries.

7. Discussion

The designed rules for transforming the Fortran-based source code into a UML sequence diagram cover all the notations in a UML sequence diagram. Fortran is an object-oriented programming language different from most other programming languages; for example, Fortran categorizes methods into functions and subroutines, whereas Java does not separate method functionalities. To consider such differences among object-oriented programming languages, we designed transformation rules to match the Fortran language. To illustrate, when rendering a UML sequence diagram, we name messages that involve function calls by prefixing an "F" to the message name, whereas messages to subroutines have an "S" prefixed to the message name.

In terms of testing three software packages, the result revealed that an interface call differs from a function call or subroutine call. Before developing the tool, we designed the system to support calls between classes only. However, we found that a majority of software packages create an interface specifying functionality to facilitate calls, and most interface calls occur from the main program. Hence, we built additional rules relating to interface calls to be able to generate more complete UML sequence diagrams. Nevertheless, during tool evaluation, we did not compare functionalities between the developed tool and other tools because our tool is the only one that can transform Fortran source code into a UML sequence diagram. The resulting UML sequence diagrams can be displayed via Modelio, helping users to understand and analyze the system more easily. The generated UML diagrams conform to the UML version 2.1 standard. Additionally, users can edit the diagram as desired, which benefits system analysis and supports future improvements.

For the tool accuracy evaluation, the authors first evaluated the system and then asked other experts to verify the accuracy. Those experts' opinions were similar to the authors concerning the evaluation results. However, the experts suggested that interface calls should also be rendered in the UML sequence diagrams; initially, we did not design the system to support interface calls. Based on the experts' recommendations, we added interface call support. Moreover, the experts provided advice about separating the types of messages represented in a UML sequence diagram; consequently, we adjusted the message presentation in the diagram through the "F" and "S" prefixes for function calls and subroutine calls, respectively.

In the current version, loops and conditions can be modeled in a sequence diagram by using combined fragments and interaction operands. Another alternative to analyze the code is to use a different type of behavior diagram such as an activity diagram, which represents specific

sequences of action or traces. We would like to add a new feature to show the control flow graphs of each and every function by applying an existing work of Kundu et al. [45], in which the information related to the start node, the end node, and the flows/arcs between the nodes is maintained in a sequence integration graph (SIG). More specifically, SIG is a directed graph, represented by (V, E), where V is a set of nodes and E is a set of edges. The nodes in V include a "control node" and a "message node." A "control node" represents a fragment start or a fragment end and a "message node" represents a message. In addition to supporting the users to view the logic of the function through a graphical representation, SIG can be used to automatically generate the Fortran code. Similarly, according to the work of Luengruengroj and Suwannasart [46], stubs and drivers used for testing each class can be generated from class diagrams and sequence diagrams. Thus, we also can apply this approach to create the stub and the driver for unit testing of Fortran.

This research has some limitations, and we make suggestions for future researchers or people interested in software engineering as follows:

(1) The system supports Fortran-based source code transformation to a UML sequence diagram under UML specification version 2.1 only. The system cannot open an XMI file generated from a UML specification below version 2.1.

(2) The system supports source code built only from object-oriented Fortran, language version 2003 or higher. If a user attempts to import the source code from a Fortran language version earlier than version 2003, the system will display an error message in the Status/Log window and will not process the source code file.Rendering a UML sequence diagram via Modelio is challenging and causes long delays when the system is large. Nevertheless, the diagram rendering is correct. When the diagram area exceeds the screen display area, the user can scroll the diagram to see portions not currently visible on the screen. We have a solution to reduce the amount of information generated, when the sequence diagram has many classes and messages. This involves splitting one XMI file into several files and refers to them as per the order of the diagram names, for example, diagram_1 and diagram_2, among others. Each sequence diagram is generated from the respective XMI file, which maintains the number of classes as per the user's preference. However, this feature will be implemented in the future version.

(3) In this study, we designed Fortran-based source code rules to transform a UML sequence diagram. The transformation mainly focused on a metamodel of source code in Fortran and a metamodel of a UML sequence diagram. Because both metamodels are key to the model transformation, we investigated a variety of tools that can design transformation rules and validate their accuracy. Eventually, we chose the ATL language to perform model transformation. The ATL language helps to verify the accuracy of the transformation rules and to align the rules with model transformation standards or specifications.

(4) The authors collected and self-analyzed a number of Fortran files prior to evaluation and comparison to a number of UML sequence diagram notations; however, self-verified data may be incorrect. Therefore, we attempted to reduce such errors by writing a program to measure the accuracy of the self-verification process. This program collected information about Fortran files as specified by the authors and presented it in a summarized form. This approach helps to reduce self-verification errors, thereby increasing the accuracy and reliability of the information. As per the verification of the UML sequence diagram notations, to reduce errors, we verified the data multiple times to ensure that collected data matched the UML sequence diagram.

8. Conclusions and Future Work

This study presented a transformation tool that converts the source code in object-oriented Fortran to a UML sequence diagram as an aid in analyzing and understanding systems developed in the Fortran language. When a user imports Fortran source code to the tool, the system generates an XMI file representing a UML sequence diagram and then renders the sequence diagram using Modelio. The tool resulting from this study worked as expected. Our experiments revealed another type of method call apart from function or subroutine calls; therefore, we enhanced the tool to present a comprehensive UML sequence diagram.

We believe that this tool would be more efficient and could be used in more scenarios if additional development and studies were carried out. In the future, we plan to develop the tool's support for more types of UML diagrams (beyond class and sequence diagrams) to the benefit of developers performing system analysis.

Data Availability

The ForUML and evaluation result data in this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors would like to thank Dr. Damain W. I. Rouson for his valuable discussion. This work was supported by the Research Funds for Graduate Study of College of Computing, Prince of Songkla University, Phuket Campus.

References

[1] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.

[2] 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, 2003.

[3] T. Systa, "On the relationships between static and dynamic models in reverse engineering java software," in *Proceedings of Reverse Engineering Sixth Working Conference*, pp. 304–313, Atlanta, GA, USA, October 1999.

[4] M. Feathers, *Working Effectively with Legacy Code*, Prentice Hall Professional, Upper Saddle River, NJ, USA, 2004.

[5] J. Q. Ning, A. Engberts, and W. V. Kozaczynski, "Automated support for legacy code understanding," *Communications of the ACM*, vol. 37, no. 5, pp. 50–57, 1994.

[6] A. Nanthaamornphong, J. Carver, K. Morris, and S. Filippone, "Extracting UML class diagrams from object-oriented fortran: ForUML," *Scientific Programming*, vol. 2015, Article ID 421816, 15 pages, 2015.

[7] V. K. Decyk, C. D. Norton, and H. J. Gardner, "Why Fortran?," *Computing in Science & Engineering*, vol. 9, no. 4, pp. 68–71, 2007.

[8] 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 29th International Conference on Software Engineering (ICSE'07)*, pp. 550–559, Minneapolis, MN, USA, May 2007.

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

[10] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, *Guide to the Software Engineering Body of Knowledge-SWEBOK*, IEEE Press, CA, USA, 2001.

[11] C. Willems and F. C. Freiling, "Reverse code engineering—state of the art and countermeasures," *IT-Information Technology*, vol. 54, no. 2, pp. 53–63, 2012.

[12] A. Rukin, "Java decompilers," 2019, http://www.javadecompilers.com/.

[13] P. Bogdan, T. Bob, and B. Harald, "ArgoUML," 2019, http://argouml.tigris.org/.

[14] Modeliosoft, "Modelio," 2019, https://www.modelio.org/.

[15] Obeo, "UML designer," 2019, http://www.umldesigner.org/.

[16] U. Team, "Umbrello," 2019, https://umbrello.kde.org/.

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

[18] J. Reid, "The future of Fortran," *Computing in Science & Engineering*, vol. 5, no. 4, pp. 59–67, 2003.

[19] M. Metcalf, "The seven ages of fortran," *Journal of Computer Science & Technology*, vol. 11, 2011.

[20] J. Reid, "The new features of Fortran 2008," *ACM SIGPLAN Fortran Forum*, vol. 27, no. 2, pp. 8–21, 2008.

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

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

[23] K. Morris, D. W. I. 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, 2012.

[24] D. W. I. Rouson, J. Xia, and X. Xu, "Object construction and destruction design patterns in fortran 2003," *Procedia Computer Science*, vol. 1, no. 1, pp. 1495–1504, 2010.

[25] D. W. I. Rouson, H. Adalsteinsson, and J. Xia, "Design patterns for multiphysics modeling in Fortran 2003 and C++," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, pp. 1–30, 2010.

[26] R. Budiardja, C. Cardall, E. Endeve, and A. Mezzacappa, "Poster: GenASiS: general astrophysics simulation system-object-oriented approach to high performance multiphysics code with Fortran 2003," in *Proceedings of High Performance Computing, Networking, Storage and Analysis (SCC)*, p. 1474, Salt Lake City, UT, USA, November 2012.

[27] F. Brian, "NAG," 2019, http://www.nag.com/.

[28] B. Paul, B. Steven, and D. Bud, "GNU," 2019, http://gcc.gnu.org/fortran/.

[29] IBM, *IBM Fortran Compiler Family*, IBM, New York, NY, USA, 2019, https://www.ibm.com/th-en/marketplace/ibm-fortran-compiler-family.

[30] Cray, "Cray," 2019, http://www.nersc.gov/users/software/compilers/cray-compilers/.

[31] Intel, "Intel Fortran," 2019, https://software.intel.com/en-us/fortran-compilers/.

[32] IEEE, "Top programming languages," 2019, https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages/.

[33] 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, pp. 1–21, 2014.

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

[35] L. C. Briand, Y. Labiche, and Y. Miao, "Towards the reverse engineering of UML sequence diagrams," in *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 57–66, Victoria, BC, Canada, November 2003.

[36] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Automated reverse engineering of UML sequence diagrams for dynamic web applications," in *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 287–294, Denver, CO, USA, April 2009.

[37] E. Korshunova, M. Petkovic, M. G. J. Van Den Brand, and M. R. Mousavi, "CPP2XMI: reverse engineering of UML class, sequence, and activity diagrams from C++ source code," in *Proceedings of the 13th Working Conference on Reverse Engineering*, pp. 297-298, Benevento, Italy, October 2006.

[38] A. G. Parada, E. Siegert, and L. B. de Brisolara, "Generating Java code from UML class and sequence diagrams," in *Proceedings of the Brazilian Symposium on Computing System Engineering*, pp. 99–101, Florianópolis, Brazil, April 2011.

[39] OMG, "UML specification v2.5," 2019, https://www.omg.org/spec/UML/2.5/About-UML/.

[40] 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*, pp. 147–157, Udaipur, India, March 2014.

[41] A. Nanthaamornphong and A. Leatongkum, "Modern Fortran transformation rules for UML sequence diagrams," *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, vol. 9, no. 3-4, pp. 131–136, 2017.
[42] ForTrilinos, "ForTrilinos," 2019, https://trilinos.org/packages/fortrilinos/.
[43] PSBLAS, "PSBLAS," 2019, http://people.uniroma2.it/salvatore.filippone/psblas/.
[44] MLD2P4, "MLD2P4," 2019, https://github.com/sfilippone/mld2p4-2.
[45] D. Kundu, R. Mall, and D. Samanta, "Automatic code generation from unified modelling language sequence diagrams," *IET Software*, vol. 7, no. 1, pp. 12–28, 2013.
[46] P. Luengruengroj and T. Suwannasart, "Stubs and drivers generator for object-oriented program testing using sequence and class diagrams," in *Proceedings of the 5th International Conference on Computational Science/Intelligence and Applied Informatics (CSII)*, pp. 32–36, Yonago, Japan, July 2018.