Automatic Generation of Valid Behavioral Scripts from UML Sequence Diagrams *

Paula Muñoz[1], Loli Burgueño[2,3], Antonio Vallecillo[1], and Martin Gogolla[4]

[1] University of Málaga, Spain (paula.munoz.ariza@hotmail.com, av@lcc.uma.es)
[2] Open University of Catalonia, Spain (lburguenoc@uoc.edu)
[3] Institut LIST, CEA, Université Paris-Saclay, France
[4] University of Bremen, Germany (gogolla@uni-bremen.de)

Abstract. This paper presents the extension of a UML and OCL tool that enables the textual specification of UML sequence diagrams, and the automated generation of all valid behaviors according to these sequence diagrams. Message Sequence Charts (MSC) are used as the textual notation to specify the UML sequence diagrams, and the USE high-level action language SOIL is used to specify behavior.

Keywords: Model-Based Software Engineering · UML Sequence Diagrams · Message Sequence Charts · UML and OCL tool.

1 Introduction

In UML [11], sequence diagrams (SDs) describe one type of interaction, which focuses on the partial order of message interchanges between objects. These diagrams enable rich interaction descriptions, with modularity mechanisms and combination operators such as parallel, alternative, optional, or repeated action or event occurrences (`par`, `alt`, `opt`, `loop`). The semantics of UML interactions, and in particular of SDs, is defined in terms of their valid and invalid traces [9,11]. In this context, traces refer to sequences of action or event occurrences.

Most UML modeling tools support the specification of SDs. In addition to checking that the names and types of the messages are valid, a few tools also provide analysis capabilities, such as checking whether the trace of a program or model execution is valid (w.r.t. a SD) [3], generating test cases [18] and even code from them [8]. Nevertheless, often, the analysis potential is not fully exploited at the modeling level.

One interesting alternative provided by some modeling tools with simulation capabilities is the use of sequence diagrams to represent execution traces, i.e., as *views* of the behavior of the system being modeled. In particular, this is the approach followed by the tool USE (UML-based Specification Environment) [5,6]. In USE, modelers textually specify the structure of their systems using standard UML class diagrams and their associated invariants using OCL [10]. For the

description of the behavioral aspects of the system, modellers can specify pre- and postconditions on the objects' operations, as well as and protocol state machines for the classes. A distinctive feature of USE is that it provides a high-level textual, OCL-like action language called SOIL [2] for specifying the behavior of operations, together with an engine able to create and delete object instances and links, assign values to attributes, and execute the SOIL implementations of the operations. Thus, sequences of operations (*scripts*) can be invoked by an external actor, enabling powerful simulations of the modeled systems, all at the same level of abstraction as the one used to specify the system. Sequence diagrams can be automatically produced from the execution traces of these scripts, allowing modelers to visualize the interactions that have happened between the system objects, including the option to link sequence diagrams to associated protocol state machines.

However, something that is missing in the tool USE is the possibility of specifying sequence diagrams that describe in general the valid execution traces of the system. In USE, the interaction diagrams are derived from the system executions, i.e., they are projections of an execution [1] and not general specifications on the system that impose restrictions on all its possible behaviors.

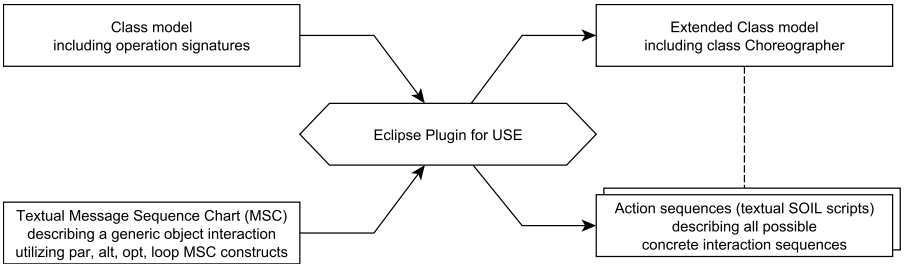

Fig. 1: Overview: Generating Action Sequences from MSCs.

In this paper, we propose an extension for UML and OCL tools, by means of (1) textual specifications of UML sequence diagrams using ITU-T Message Sequence Charts (MSC) [7], and (2) the automated generation of the set of valid behaviors from them. Thus, a modeller can specify a SD using MSC, the standard ITU-T textual notation—which is equivalent to the graphical notation provided by UML to describe sequence diagrams—and then generate the set of action language scripts (in our case SOIL scripts) whose behavior correspond to the valid traces of the system w.r.t. the given SD. The proposed extension has been implemented by means of an Eclipse plugin that allows modelers to write an MSC, type-check it, and automatically generate the set of corresponding sequences of action language commands (SOIL commands) that produce all its valid traces. Fig. 1 gives an overview on the approach.

The structure of this paper is as follows. After this introduction, Sect. 2 briefly describes the related technologies used in this work. Then, Sect. 3 presents our

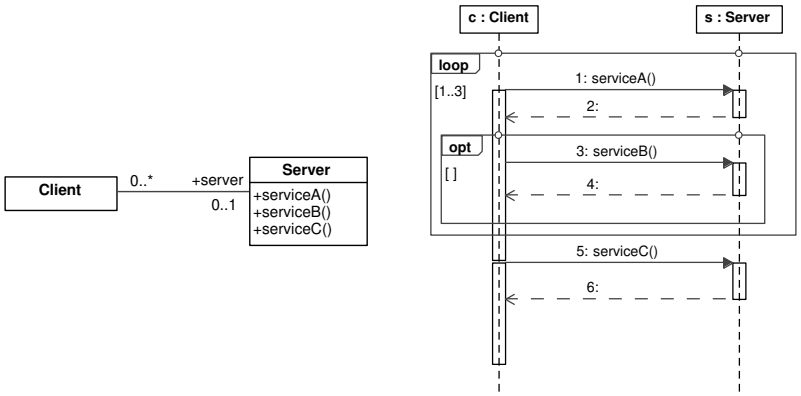

Fig. 2: An example of a UML Class diagram and associated Sequence diagram.

proposal and how it has been implemented. Finally, Sect. 4 discusses related work, and Sect. 5 concludes with an outline on future research lines.

2 Background

2.1 UML Sequence Diagrams

Sequence diagrams (SD) describe one type of interaction, which focuses on the sequences of message interchanges between a number of objects (represented by `Lifelines`). UML supports the specification of different kinds of messages and signals, both synchronous and asynchronous. Message exchanges can be specified to happen instantaneously, or have an associated duration. Different kinds of constraints (including time constraints) can impose further requirements on the SD elements. Several combination operators can be used for representing different kinds of interactions, such as parallel (`par`), loops (`loop`), alternative and optional interactions (`alt` and `opt`, resp.), negative traces (`neg`), critical regions (`critical`) and many others. SD *fragments* can also be used to encapsulate a set of interactions, which can then be reused in other SDs. This provides a very useful mechanism to achieve modularity in the specifications.

Despite its apparent simplicity, UML SDs are not easy to undertand [14], and, in fact, its semantics deserve a careful definition [9,11]. They are defined in terms of the valid and invalid traces specified by the SD.

Fig. 2 shows an example of a UML Class diagram (left) and one associated Sequence diagram (right) that describes the partial order in which the operations implemented by objects of class `Server` can be invoked by objects of class `Client`. A `loop` combination operator determines the minimum and maximum number of times that the interactions included in it may happen. The `opt` operator is used to indicate that the call to operation `serviceB()` is optional and could be omitted. Lines with solid arrows represent synchronous messages, and dashed lines represent the returns of the calls.

This specification defines 14 possible valid execution traces for the system, depending on the number of times that the loop is executed, and whether the optional operation is invoked or not:

- `serviceA()`, `serviceC()`
- `serviceA()`, `serviceB()`, `serviceC()`
- `serviceA()`, `serviceA()`, `serviceC()`
- `serviceA()`, `serviceB()` `serviceA()`, `serviceC()`
- `serviceA()`, `serviceA()`, `serviceB()`, `serviceC()`
- `serviceA()`, `serviceB()` `serviceA()`, `serviceB()`, `serviceC()`
- ...

In general, identifying and generating all the behaviors that produce the valid traces defined by a given SD is far from being a trivial task. However, this is relevant in several situations. For example, it helps identifying all possible behaviors of the system, as specified by the SD, and all the potential executions it could allow, which helps understanding the SD specifications. Another situation of interest happens in the realm of Model-Based Testing [16,17], when a SD can be used to generate the set of test suites that exercise all possible executions. In the following section, we show how our proposal achieves such goals and how we have implemented it for the USE tool.

2.2 ITU-T Message Sequence Charts (MSC)

To improve the UML 1.X notation used for specifying sequence diagrams, in UML 2.X sequence diagrams were defined based on the International Telecommunication Union's (ITU) Message Sequence Chart (MSC)—ITU-T Recommendation Z.120 [7]—, which is the standard notation commonly used to specify interaction protocols in the Telecommunications domain.

The ITU-T Recomendation defines two concrete syntaxes to represent MSCs, one textual and one graphical. The graphical notation was the one adopted by UML 2.X. The textual notation expresses the same information, using a well-defined grammar [7]. For example, Listing 1.1 shows the textual specification of the SD described in Fig. 2. Interestingly, in this notation each interaction is described both from the sender and from the receiver, since this enables the separation between the signals being sent and received that is required, e.g., when we need to deal with asynchronous interactions.

Listing 1.1: Textual MSC describing the sequence diagram shown in Fig. 2.

```
msc Example1;
    (inst c:Client, s:Server)
    c,s : loop<1,3> begin loop1;
        c : call serviceA() to s;
        s : receive serviceA() from c;
        s : replyout serviceA() to c;
        c : replyin serviceA() from s;
        c,s : opt begin opt1;
            c : call serviceB() to s;
            s : receive serviceB() from c;
            s : replyout serviceB() to c;
```

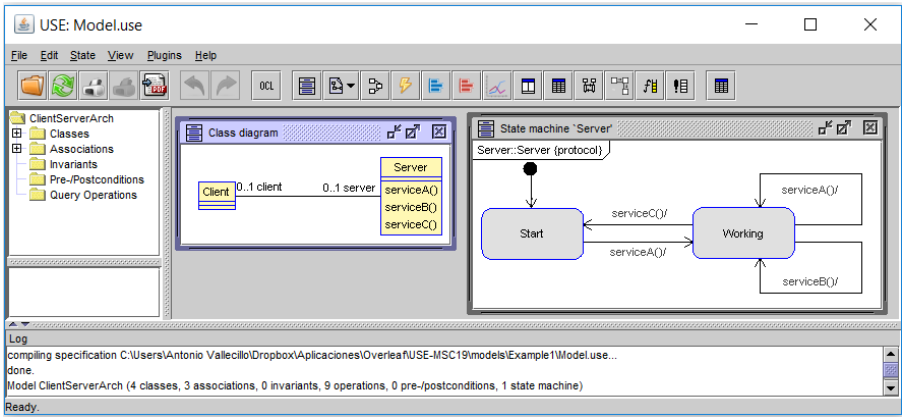

Fig. 3: A screenshot of the USE specification environment.

```
            c : replyin serviceB() from s;
        opt end;
    loop end;
    c : call serviceC() to s;
    s : receive serviceC() from c;
    s : replyout serviceC() to c;
    c : replyin serviceC() from s;
endmsc;
```

The expressiveness of ITU-T's MSCs is richer than that of UML SDs, which only implement a subset of all the numerous features and mechanisms of MSCs. In our proposal we are interested just in the interactions that happen between objects when they interact via method calls, and thus we have implemented a subset of MSCs. More precisely, we currently support instance specification (to identify the Lifelines of the objects engaged in the interaction and their types), method invocations and responses (using MSC constructs `call`, `receive`, `replyout` and `replyin`), and four types of combined fragments (`loop`, `alt`, `par` and `opt`), which can be arbitrarily composed.

2.3 USE

The UML-based Specification Environment (USE) [5] is a modeling tool that enables the specification and validation of UML and OCL models. The tool is open source and distributed under the GNU General Public License.

USE supports UML class diagrams for the specification of the structure of the modeled system, and provides full support for OCL. To specify the behavior of the system, operations can be enriched with pre and postconditions, and UML protocol state machines can be associated to the system objects.

Figure 3 shows a screenshot of the specification in USE of the client-server example. We can see how USE also allows modelers to specify protocol state machines for the objects of the system. The behavior of the system when being executed is checked against these state machines.

```
class Client
operations
test()
begin
 self.server.serviceA();
 self.server.serviceB();
 self.server.serviceC();
end
```

(a) USE specification.

```
!new Client ('c');
!new Server ('s');
!insert (c, s) into CS;
!c.test()
```

(b) SOIL script.

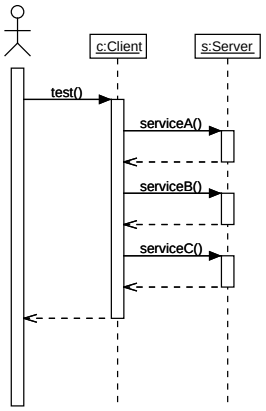

Fig. 4: An example of a SOIL script and the corresponding SD.

USE also provides an executable extension of OCL that enables quickly prototyping the system specifications, called SOIL (Simple OCL-like Imperative Language) [2]. Given a UML model, SOIL allows modelers to create instances and links of that model, assign values to the attributes of these objects, and invoke their operations.

For example, Fig. 4 (left) shows the specification of one `Client` operation called `test()` that calls the three server services in sequence; the SOIL script (center) that creates the instances of the client, the server, and the link between them and then calls method `test()` of the client; finally, the right part of Fig. 4 shows the SD obtained as a result of the execution of the SOIL script.

In USE, both sequence and communication diagrams can be automatically generated from a system execution, defined as a sequence of SOIL commands. However, USE modelers do not have any means for specifying UML sequence diagrams describing all the valid interactions between a set of objects.

Therefore, the goal of our proposal is, given the structural model of a system in USE, and the specification of a sequence diagram written using the MSC notation, automatically generate the set of SOIL scripts that produce all valid executions of the system, according to that SD.

For example, given the client-server model depicted in Fig. 2 and the corresponding MSC shown in Listing 1.1, our tool generates 14 SOIL scripts, each one producing a valid trace for the system. For illustration purposes, Fig. 5 shows the SDs generated for three of these 14 SOIL scripts.

The next section describes the way in which we generate the SOIL scripts from a UML class diagram as well as an MSC that specifies the possible interactions among the objects of that system, and the tool we have developed to achieve this.

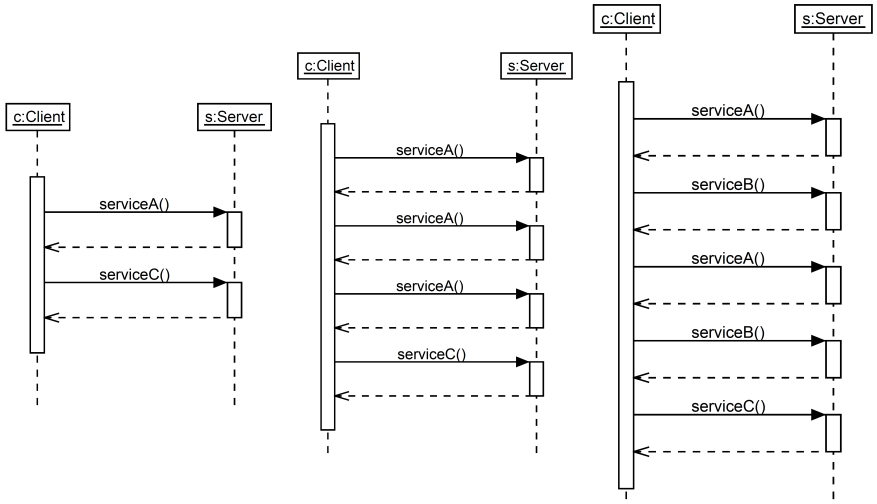

Fig. 5: Three of the 13 sequence diagrams produced for the client-server system.

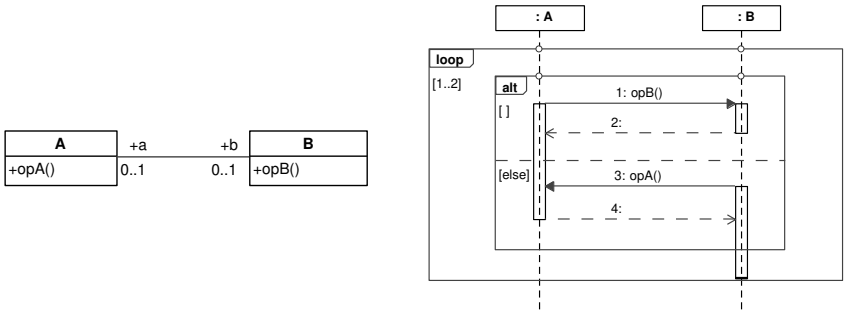

Fig. 6: A system with two objects that invoke each other's methods.

3 Our proposal

3.1 Specifying Sequence Diagrams in USE

To illustrate our proposal and how it works, we will use a slightly more complex system than the one employed until now. For example, let us consider the system shown in Fig. 6, which is composed of two types of objects, each one calling the other's method in the order specified by the SD shown in the figure, and whose equivalent MSC specification is described in the Listing in Fig. 7.

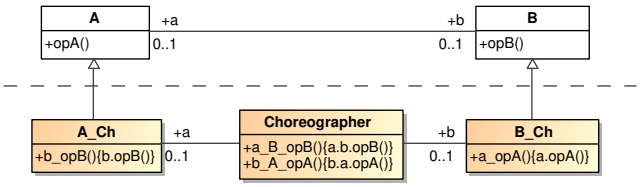

Fig. 8: Adding a choreographer and extending the interacting objects.

```
msc Example2;
  (inst a:A, b:B)
  a,b : loop<1,2> begin loop1;
    a,b : alt begin alt1;
        a : call b() to b;
        b : receive b() from a;
        b : replyout b() to a;
        a : replyin b() from b;
    alt;
        b : call a() to a;
        a : receive a() from b;
        a : replyout a() to b;
        b : replyin a() from a;
    alt end;
  loop end;
endmsc;
```

Fig. 7: MSC describing the SD of Fig. 6.

This system is not easy to simulate according to the behavior specified in the MSC, because we need to ask each object to individually perform the calls in different orders. This is why we need to extend the system with some essential information for implementing the executions.

More precisely, we need to (a) extend each object with the appropriate methods that invoke other objects' methods, and (b) add a new object to the system, which is in charge of calling the extended objects in the required order.

This new object is called the `Choreographer`, and the extended objects will be called after their original ones, suffixed by "`_Ch`" to indicate that they are their *choreographed* versions. Fig. 8 shows the resulting model after extending the classes of the system in Fig. 6 and adding the choreographer. We can see how the extended classes incorporate a method that carries out the required call, hence implementing the desired behavior.

Figure 9 shows the corresponding sequence diagram for the extended system, which is now able to be simulated by USE. All these extensions are automatically produced by our plugin from the UML specifications of the system (Fig. 6) and the corresponding MSC (Fig. 7).

In particular, our plugin is able to generate the set of sequences of SOIL commands whose executions correspond to the set of valid execution traces for that behavior, namely, `{opB()}`, `{opA()}`, `{opB();opA()}`, `{opB();opB()}`, `{opA();opB()}` and `{opA();opA()}`.

Figure 10 shows one of the possible sequences of SOIL commands and its related sequence diagram, as it is shown by the USE tool. It corresponds to one of the possible valid traces identified above, namely, `{opA();opB()}`. The left part of Fig. 10 displays the complete trace. In contrast, the right part shows the same sequence diagram where we have hidden the `Choreographer` object: it is one of the traces of the original UML SD previously shown in Fig. 6.

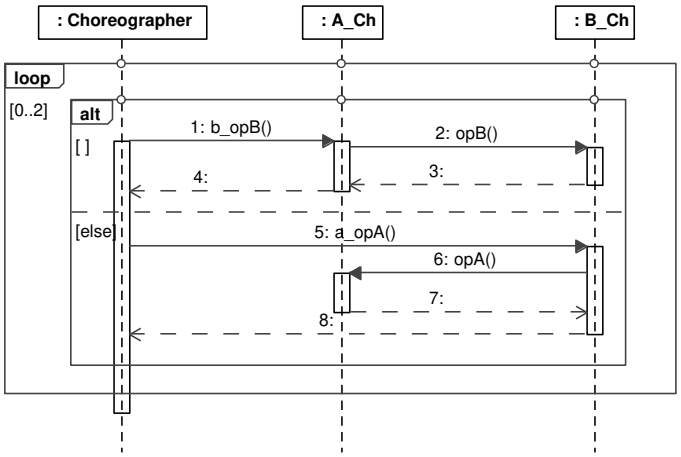

Fig. 9: The sequence diagram of the extended system.

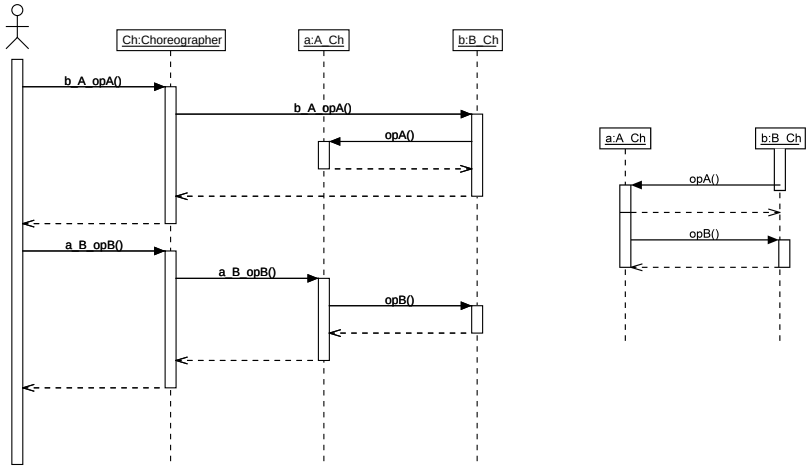

Fig. 10: One system trace showing (left) or hiding (right) the `Choreographer`.

The SOIL script whose execution generates the SD depicted in Fig. 10 is shown in Listing 1.2.

Listing 1.2: One of the generated SOIL scripts.

```
reset
!new Choreographer('Ch');
!new A_Ch('a');
!new B_Ch('b');
--
```

```
!insert (a,b) into RelAB;
--
!Ch.b_A_opA();
!Ch.a_B_opB();
```

3.2 Generating SOIL scripts from MSCs

This section describes how the SOIL scripts are generated from the USE model with the structure of the system, and a given MSC. The process has three steps.

a) Parsing the MSC. Using Xtext, we have defined a subset of the complete grammar of MSCs, as specified in the Z.120 Recommendation [7]. The textual file with the MSC specification is parsed by Xtext, which builds its Abstract Syntax Tree (AST) if the MSC specification is correct.

b) Extending the USE model. The AST generated by Xtext is then used by a program we have developed in Xtend to generate the extended USE model. Such an extension consists in adding to the initial USE model (a) the `Choreographer` class, (b) the new classes that extend the model classes involved in the MSC with the operations that the `Choreographer` will invoke, and (c) the relationships among them. An example of this extension of the USE model, with new classes `Choreographer`, `A_Ch` and `B_Ch`, was shown in Fig. 9.

c) Generating the SOIL scripts. Using the AST of the MSC and the extended USE model, we have developed an algorithm in Xtend that generates the SOIL scripts. The idea behind this algorithm is to traverse the control flow graph (CFG) [4, 18] that corresponds to the MSC.

The CFG is a graph whose nodes can be either actions (method invocations) or branching nodes: decision, merge, fork and join nodes. The arcs of a CFG represent the possible transitions between nodes. Thus, the CFG of an MSC is composed just of a sequence of method invocations, without any combined fragment (`loop`, `alt`, `par` or `opt`), corresponding to a list of action nodes. The presence of `opt` combined fragments in a MSC will make its CFG become a tree. Loops introduce cycles in the CFG. Fork and join nodes are introduced by `par` combined fragments, indicating that all the possible interleaving executions of the contents of the `par` fragment are possible.

Every `opt` fragment corresponds to two transitions, one with the CFG of the body of the option, and the other that jumps to the end of the body. Alternative nodes are treated similarly, with one transition to each of the possible alternative bodies. To deal with `par` fragments, we use the Heap algorithm to generate all valid permutations of the body of the `par` fragment. Loops have a lower (l) and upper (u) number of iterations, so we need to consider that they have to generate $u - l + 1$ alternative behaviors, each one containing a different number of repetitions of the loop body (ranging from l to u).

The CFG for the MSC is not explicitly built, but represents implicitly the behavior of the algorithm we have developed in Xtend to generate all its possible paths. The algorithm uses recursion to traverse the graph, since we permit

the unlimited use of combined fragments inside the body of other combined fragments (this is something that not many related proposals allow).

A string with the text of a SOIL file is created at the beginning of this process, and initialized with the set of SOIL commands that create the corresponding instances and links. Then, the recursive algorithm starts.

Every time an action node is found, the text corresponding to the SOIL call is added to the file we are building. Recursion continues until a final node of the MSC is reached. Then, a SOIL file is produced with the current contents of the text string, and the recursion backtracks to let the algorithm explore further nodes, continuing the CFG traversal.

3.3 Some Optimizations

Although in theory the algorithm described above should work well with all types of fragments and their combinations, its performance severely degrades under the presence of nested loops. This is why we had to change its behavior when a loop fragment is detected in the MSC AST. Instead of using just one string to store the current SOIL file, we change the behavior of the algorithm when a loop is detected. In this case, the body of the loop is treated as a separate and individual MSC, for which one SOIL file (or many, in case of combined fragments are contained inside the loop body) is created. Such a file (or set of files) is then copied as many times as required by the possible iterations of the loop in each alternative execution. For example, a `loop<2..4>` instruction will generate 3 alternative executions, one where the body is repeated twice, one where the body is repeated three times, and a final one where the body of the loop is repeated four times. Although the number of files that are produced in this way can be large (especially when a loop happens inside a loop, not to mention when nesting happens at two levels), the performance of the algorithm is much better (in terms of a few seconds instead of hours).

3.4 Evaluation

To validate our proposal, we have tested it with different kinds of system models. First, we used a test suite composed of models of artificial systems that contained the different combinations of the supported fragments (`alt`, `loop`, `opt`, `par`), with varying numbers of interactions and nesting levels. These tests covered most basic cases and allowed us to check the behavior of our plugin. For example, with just three nested loops one can easily get thousands of SOIL scripts for the same MSC. This made us realize about the intrinsic complexity of the MSC specifications, whose execution traces can grow beyond the user expectations. The performance of our plugin when dealing with these cases was acceptable, with responses that in the worst cases went up to a few minutes for producing tenths of thousands of SOIL scripts. As part of our future work we plan to conduct a proper evaluation of the performance of our plugin, and check its current limits regarding the number of nested loops it is able to support and how its response time varies.

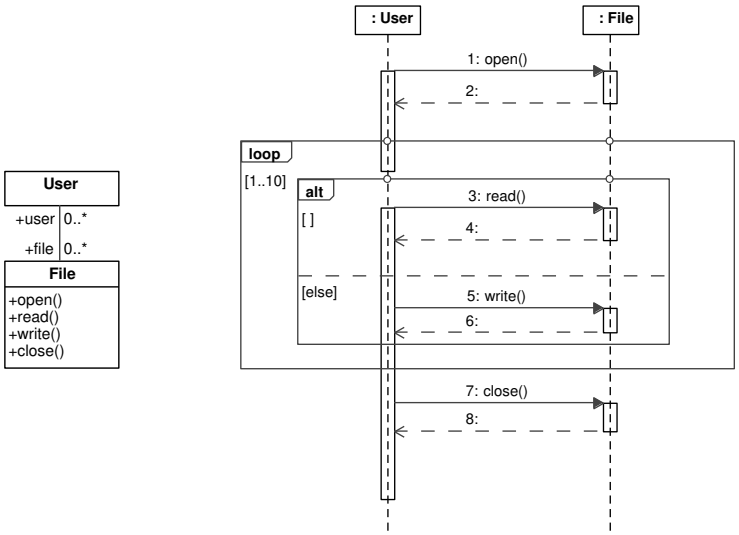

Fig. 11: The File Manager system.

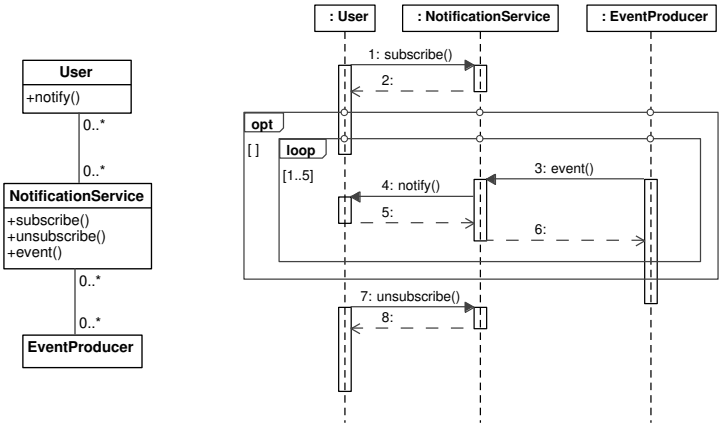

Fig. 12: The Publish & Subscribe system.

In addition to the synthetic tests, we also validated our plugin with exemplar applications that cover various kinds of SD specifications. Although smaller than the previous models, they were useful for understanding the responses we obtained, and for checking that the behavior of the system was as expected. In particular, some of these exemplar models used were the following:

– A File Manager system (Fig. 11), which represents the typical use of a file: it has to be opened, then the user can read or write several times, and then it needs to be closed. Assuming a loop with between 1 and 10 iterations, within an optional block, we obtain 21 alternative behaviors.
– A Publish and Subscribe system (Fig. 12) that represents the typical application where listeners register in a notification service, and are notified every time an event producer tells the server that an event has happened. Six SOIL files are produced in this case.
– A Library system (not shown here) that represents an application where a user can borrow a book and then either return it or not, and could be optionally fined.
– A system that accesses the services of a large High-Performance Computer using a security manager for registration and identification for clients (not shown here for space reasons).

All the artifacts associated to these examples (USE models, MSCs and soil files) are available from out git server.[5]

4 Related Work

UML Sequence diagrams have been used by several authors to perform different software engineering activities with them. For example, some works focus on generating code from UML class and sequence diagrams [8,12,15]. They normally build the CFG explicitly, and generate the code from it. They also support a subset of all SD operators, although most of them do not support their arbitrary combination and nesting, in particular for loops.

Other group of works (e.g., [13, 18]) focuses on generating test cases from UML sequence diagrams. They also use the CFG for generating the test cases. However, unlike us, they do not seem to use any executable notation for the specification of the produced test cases. In our case, it is important that all traces (test cases in their terminology) are formally specified and can be directly executed. Another key feature of our solution is that it remains at the same level of abstraction as the modeled system, and within the environment of the same tool. In this sense, our produced artifacts can be seamlessly manipulated and executed by USE, with the goal of improving the usability of our solution.

5 Conclusions and Future Work

This work has introduced an approach to specify UML sequence diagrams using a textual notation (a subset of ITU-T's MSCs) and how all valid behaviors of the system according to that MSC can be automatically generated. We have developed an Eclipse plugin to support these tasks, and the USE environment as the UML modeling tool and executable engine. Thus, all valid behaviors

[5] `https://github.com/atenearesearchgroup/msc-use/`

are generated in terms of SOIL scripts (sequences of commands) which can be simulated by USE.

We have presented here the first prototype of our tool, in which we plan to continue working in several directions. First, we would like to support more combined operators such as `seq`, `critical` or `break`, in addition to the four that we currently support. Second, we would like to consider the guards of the combined fragments, since now we abstract this information away. A more tight embedding into the USE environment could be envisaged, too. Of course, validating our plugin with more and larger case studies would be required, as well as studying its performance. The application of our proposal to other UML and OCL tools could also be an interesting extension of this work. Finally, we would like to analyze the usability of our proposal, conducting some empirical experiments with real modelers that could help assess if specifying SDs with a textual notation is acceptable, and whether our proposal indeed helps understanding the semantics of SDs.

Acknowledgements. We would like to thank the reviewers for their constructive comments on previous versions of this paper. This work has been partially supported by Spanish Research Projects PGC2018-094905-B-I00 and TIN2016-75944-R.

References

1. Burgueño, L., Vallecillo, A., Gogolla, M.: Teaching UML and OCL models and their validation to software engineering students: an experience report. Computer Science Education **28**(1), 23–41 (2018)
2. Büttner, F., Gogolla, M.: On OCL-based imperative languages. Sci. Comput. Program. **92**, 162–178 (2014)
3. Chai, M., Schlingloff, B.: Monitoring with parametrized extended life sequence charts. Fundam. Inform. **153**(3), 173–198 (2017)
4. Garousi, V., Briand, L.C., Labiche, Y.: Control flow analysis of UML 2.0 sequence diagrams. In: Proc. of ECMDA'05. LNCS, vol. 3748, pp. 160–174. Springer (2005)
5. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. **69**(1-3), 27–34 (2007)
6. Gogolla, M., Hilken, F., Doan, K.: Achieving model quality through model validation, verification and exploration. Computer Languages, Systems & Structures **54**, 474–511 (2018). https://doi.org/10.1016/j.cl.2017.10.001
7. ITU-T Z.120: Message Sequence Chart (MSC) (Feb 2011)
8. Kundu, D., Samanta, D., Mall, R.: Automatic code generation from unified modelling language sequence diagrams. IET Software **7**(1), 12–28 (2013)
9. Micskei, Z., Waeselynck, H.: UML 2.0 Sequence Diagrams' Semantics. Tech. Rep. 08389, Laboratoire d'Analyse et d'Architecture des Systemes (LAAS) (Aug 2008), `http://mit.bme.hu/~micskeiz/sdreport/uml-sd-semantics.pdf`
10. Object Management Group: Object Constraint Language (OCL) Specification. Version 2.4 (Feb 2014), OMG Document formal/2014-02-03
11. Object Management Group: Unified Modeling Language (UML) Specification. Version 2.5 (Mar 2015), OMG document formal/2015-03-01

12. Parada, A.G., Siegert, E., de Brisolara, L.B.: Generating java code from UML class and sequence diagrams. In: Proc. of SBESC'11. pp. 99–101. IEEE Computer Society (2011). https://doi.org/10.1109/SBESC.2011.22
13. Seo, Y., Cheon, E.Y., Kim, J., Kim, H.S.: Techniques to generate UTP-based test cases from sequence diagrams using M2M (Model-to-Model) transformation. In: Proc. of ICIS'16. pp. 1–6. IEEE Computer Society (2016). https://doi.org/10.1109/ICIS.2016.7550832
14. Siau, K., Loo, P.: Identifying difficulties in learning UML. IS Management **23**(3), 43–51 (2006)
15. Usman, M., Nadeem, A.: Automatic Generation of Java Code from UML Diagrams using UJECTOR. International Journal of Software Engineering and Its Applications **3**(2), 21–37 (2009)
16. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
17. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**(5), 297–312 (Aug 2012)
18. Vu, T., Hung, P.N., Nguyen, V.: A method for automated test data generation from sequence diagrams and object constraint language. In: Proc. of SOICT'15. pp. 335–341. ACM (2015)