Towards a New Approach for Reverse Engineering of UML 2 Sequence Diagrams

Wahab Rouagat
University of Batna
Batna,Algeria
wahab.rouagat@gmail.com

Tewfik ZIADI
LIP6 & University of Paris 6
104 avenue du President Kennedy,
75016 Paris.
Tewfik.Ziadi@lip6.fr

Absract—**Many legacy systems are developed without providing the necessary documentation for future maintenance and evolution. In such cases maintainers and developers will spend more time and will make a great effort (more cost) in system structure and behavior understanding phase. Despite the great amount of system comprehension step, results are error-prone even if maintainers are themselves the inner developers. Reverse engineering aim to alleviate developers and maintainers in their work by restraining cost and increasing system comprehension precision; and to present results in a human understandable manner. The topic of our new approach is the UML dynamic models reverse engineering, in particular sequence diagrams. We present an overview for this domain, we set the global context for our solution and we highlight our approach implementation details.**

Keywords ***reverse engineering, UML sequence diagram, meta-model, static analysis, dynamic analysis.***

I. INTRODUCTION

Since systems environments and users needs are changing continuously, maintainers and developers must periodically evolve their software in response to those changes. The maintenance and the evolution of the system can not be successfully conducted without a real comprehension of its structure and behavior. When the documentation is incomplete and-or obsolete (e.g. in the case of the legacy systems), reverse-engineering techniques represent the only way to realize to evolve the system.

Reverse engineering is the process of analyzing a system with the goal to create representations of the system in another higher level abstraction [1]. To be useful, reverse-engineering techniques should be associated with a well-known representation way. The Unified Model Language (UML) [3] seems to be the language for such aim. It represents the facto standard for object-oriented system representations. UML is composed of a set of diagrams which allow specifying the several aspects of a system. The two main aspects are: static and behavior aspects. UML class diagrams are an example of diagrams for static aspect representation while UML sequence diagrams are used to specify the behavior aspect.

Reverse-engineering for the static aspect (e.g. the class diagram) of an object-oriented system are already available in many UML CASE tools (e.g. IBM RSA[1] and Objecteering[2]). However there is a little work on reverse engineering of sequence diagrams. As underlined by Briand et al. [5], one motivation of reverse-engineering of UML sequence diagrams is to help maintainers and developers to understand the behavior of systems with incomplete documentation. Such reverse-engineering can also be used to verify the conformance of the implementation to the existing design. This can be realized by comparing the reverse-engineered sequence diagrams with the sequence diagrams including in the design.

This paper presents the results of our first investigation on reverse-engineering for UML2 sequence diagrams. It first presents an overview for this domain. Then it outlines our approach for reverse-engineering of UML2 sequence diagrams.

The rest of the paper is organized as follows: Section 2 briefly presents reverse-engineering and UML2 sequence diagrams. Section 3 outlines our approach. We present the implementation details and we show the results of a testing case in Section 4 while the Section 5 concludes our work and addresses some perspectives.

II. REVERSE ENGINEERING AND UML2 SEQUENCE DIAGRAMS

A. *Reverse-Engineering*

Reverse-Engineering (RE) is initially defined by Chikofsky and al. [1] as:

"..RE is the process to analyzing a subject system with two goals in mind:

[1] http://www.ibm.com/developerworks/rational/products/rsa/

[2] http://www.objecteering.com/

- *To identify system component's and their relationships,*
- *To create representations of the system in another higher level abstraction. ."*

Reverse-engineering aims to provide design models from existing software. Hence, this can facilitate program comprehension and by consequence maintenance and evolution of systems. In this paper, we consider UML diagrams as target design models for reverse-engineering. This means that we focus only on RE of UML diagrams (i.e. the process that analyzes the execution of the system and creates UML diagrams that depicting its structure and its behavior).

UML [3] is an object-oriented language for software system modeling. It was standardized by the OMG (Object Management Group) basing on three precedent methods which are: OMT (Object Modeling Technique) [7], Booch [8] and OOSE [9] (Object Oriented Software Engineering). UML is composed of a set of diagrams which allow specifying the several aspects of a system. The two main aspects are: static and behavior (also called dynamic) aspects. The static aspect of system can be specified using class or component diagrams, while the behavior aspect can be specified using sequence diagrams, state machines and activity diagrams. In this paper we focus only on UML sequence diagram.

B. UML2 Sequence Diagrams

1) Description and notations

Sequence Diagrams (SD) have been significantly changed in UML2 [11]. Notable improvements include the ability to define what is called combined SD. A Combined SD is a sequence diagram that refers to a set of SD and composes them using a set of interaction operator. The main operators are: seq, alt, opt, loop, and par. The seq operator specifies a weak sequence between the behaviours of two SD. The alt operator defines a choice between a set of SD. The opt operator specifies the option, while the loop operator define an iteration of a SD. The par operator allows specifying concurrency between SD.

Figure 1 shows three basic Sequence Diagrams (SD)[3]: SD1, SD2, and SD3. The basic SD SD1 describes the interactions between two instances a1 (instance of the A class) and b1 (instance of the B class). The vertical lines represent life-lines for the given instances. Interactions between instances are shown as horizontal arrows called messages (like m1). Each message is defined by two events: message emission and message reception, which induces an ordering between emission and reception. Events situated on the same lifeline are ordered from top to down. Figure 1 also shows a combined SD called CombinedSD that refers to three basic SD and composes them using interaction operators. The combined SD illustrates the use of three operators: seq, loop and par. The behaviour specified in the CombinedSD of Figure 1 is equivalent to the expression loop (SD1 seq (SD2 par SD3)).

[3] A basic SD is a SD without interaction operators. It shows only simple interactions between instances.

C. Reverse-engineering of UML2 sequence diagrams

We distinguish two main categories of existing UML SD reverse-engineering approaches [2]: the first category gathers approaches which are based on static analysis while the second concerns dynamic analysis based approaches.

Static analysis is done on static information which describes the structure of the software as it is written in the source code. This can be realized using language key words and following the sequence of method calls to determine system objects and their interactions. However; dynamic analysis is based on the system runtime behavior information which can be captured by separated tools as in [4], by instrumentation techniques as in [5], or by debugging techniques.

Both of two axes have their advantages and drawbacks, but they complete each other. Static approaches have limits with dynamic links and polymorphism states, which not the matter in dynamic approaches. Dynamic approaches have problems to overcoming all possible system instances, when the solution is in static methods. In our approach we follow the dynamic one.

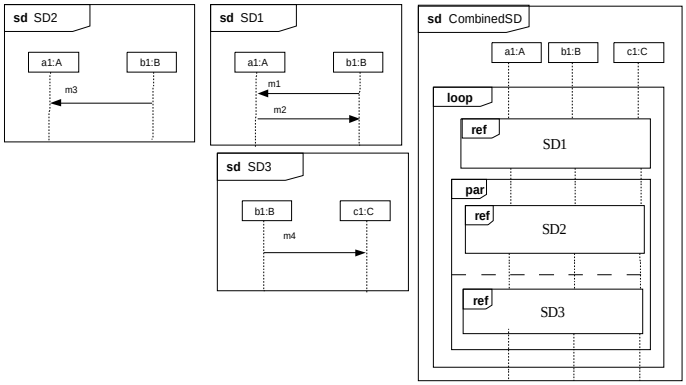

Fig 1. Example of UML2 Sequence diagrams.

III. OUR APPROACH

Our approach for UML2 SD reverse-engineering is illustrated in the Figure 3. It takes as input, the traces of several system executions and it aims to construct a higher level abstraction of the system by producing a complete UML2 sequence diagram that shows the entire interactions.

Definition1. *A STATEMENT is a tuple T = <SENDER, METHOD, RCEIVER> where:*

(i) SENDER is the class that invoked the method; (ii) RECEIVER is the class that performed the procedure; (iii) METHOD is the method.

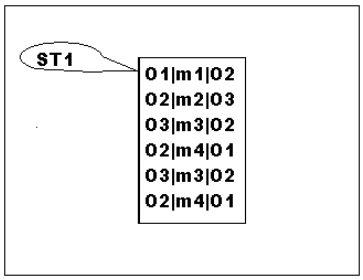

Fig 2. Example of a trace

Definition2. *A TRACE is a tuple TR = <T, L, <> such that:*

(i) ST is a set of STATEMENT;

(ii) L Є N is the size of the trace where N is the set of positive integers;
(iii) <: is a total ordering relation in ST.

Example Fig .*2 shows an example of trace where :*

St1=‹O1, m1, O2› Є STATEMENT *where: SENDER =O1, RECEIVER =O2, and METHOD =m1.*

Definition3. *A BasicSequenceDiagram is a tuple BasicSD = <M, L, <> where:*

(i) M is a set of messages. A message is defined by: m = (sender, name, receiver) where: sender, receiver Є L;
(ii) L is a set of lifelines involved in sequence diagram;
(iii) <is a total ordering relation in M.

Definition4. *A CombinedFragment is a tuple FC= <Operand, operator> where:*

(i) Operand is a set of BasicSequenceDiagram;

(ii) Operator is an enumeration: seq, opt, alt, loop.

Definition5. *A SequenceDiagram is a tuple SD = <CFs, <> where:*

(i) CFs is a set of CombinedFragment;

(ii) <is a total ordering relation in CFs.

In what follows we define the set of functions that will be used later in our algorithms:

1) RANG is the function that returns the order of a STATEMENT in a trace;
2) SubTrace is the function that returns the sub-trace of a trace;
3) FirstStatement is the function that returns the first STATEMENT of a trace;
4) LastStatement is the function that returns the last STATEMENT of a trace;
5) Successor is the function that returns the Statement which follows a particular Statement;
6) Predecessor is the function that returns the Statement which precede a particular Statement;
7) SuccessorSet is the function that returns the set of Statements which follows a particular Statement ;
8) PredecessorSet is the function that returns the set of Statements which precede a particular Statement;
9) isEquivalent is the function that verify if two sub-traces are equivalent or not ;
10) SetOperand(Messages) is the function that relate a set of messages to a BasicSequenceDiagram .
11) newOperand (name) is the function that create a basicSequenceDiagram in a particular CombinedFragment.
12) newCombinedFragment(name, operator) is the function that create a combinedFragment in a sequenceDiagram.

Our approach is divided on two Phases: traces collection and sequence diagram extraction.

A. Phase I: Traces collection

Our aim at this stage is to collect the major events occurring during the system executions. At instance, we will consider just four types of events: 1) object creation 2) object destruction 3) message sending event 4) message receiving event. At this phase, we have to use tracer tools to gather traces. Instead of creating new tracer tool, we have reused existing one. There are many tracer tools such as MoDec [4], Caffeine [12] and JTracor [6].

The system behavior is related to the environment entry data, in particular, values introduced by the user to initialize specific system variables. Thus, one execution session is not enough to identify all system behaviors. Indeed, the alternative and iterative interactions can't be detected, only when the appropriate value for "if, elseif" and "while, for, repeat …" conditions has been specified. Hence, we must iterate this phase (for each system execution session we get a trace) as much as the number of different possible system entries. This needs a system expert user who knows the different system entries values and environment states. However the number of execution sessions; yet some interactions is might still out of sight, but not the major.

B. Phase II: Sequence diagram extraction

At this phase, we aim to use traces obtained previously to construct our UML2 sequence diagrams. The obtained sequence diagrams are instances of the UML2 meta-model. This construction can be realized using the correspondence between the trace concepts and the SD elements. Objects in traces correspond to lifelines in SD and events correspond to messages. However, the critical question is: how to detect

combined fragment? In other words: how we can detect interaction operators, such as *alt*, *loop* and *par*?

Our solution is an incremental construction of the sequence diagram as described in Figure 3. Indeed, the sequence diagram obtained from the first trace is enriched by using the second trace. Then the new SD is enriched by the third trace, etc.

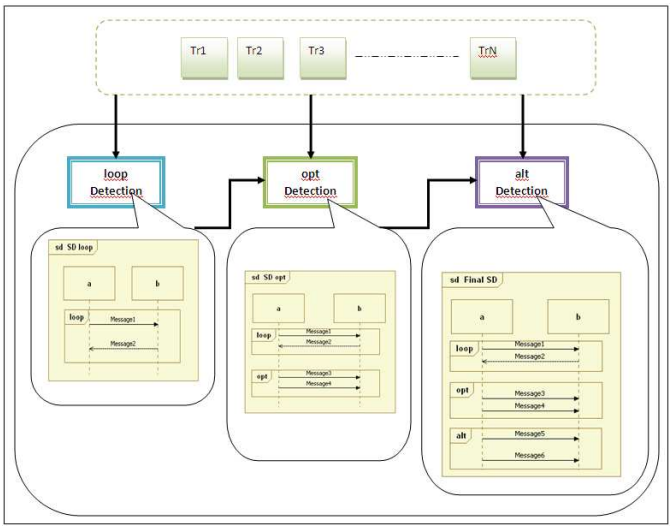

Fig 3. Incremental approach of SD reverse engineering

1) Anatomy of traces

Depending on the system behavior, several formats of traces can be obtained at the end of execution; these formats might be simple, complex or more complex:

Simple traces are composed of a series of permanent messages without any complex structure (conditional or iterative structures). In such cases, there is no benefit to use our solution for the reason that will be no combined fragment to be detected (all messages belong to the same basic interaction fragment).

Complex traces are constituted of series of blocs (bloc: *sequence of messages belonging to the same basic combined fragment*) of eventual different structures kinds: optional, alternative, iterative or permanent.

More Complex traces are the same as complex traces format, though blocs may contain other sub-blocs (imbricate combined fragment). This format hasn't been yet supported by our solution; we plan to deal with it at the nearest future work.

Bloc notion:

Our current work deal with complex traces format. The final vision to be obtained at the end is a sequence of combined fragments (sequence of blocs) of different kinds, which represent the complete behavior of the system.

Blocs are considered as messages containers; these containers exist in a specific order in all traces, however blocs can be empty from trace to other, due to conditional structures ('opt' and 'loop' cases) or permanently filled ('permanent' or 'alt' cases), and all traces have to respect the same template of blocs (look fig.4).

Tip: Each bloc can be addressed by its previous and next bloc.

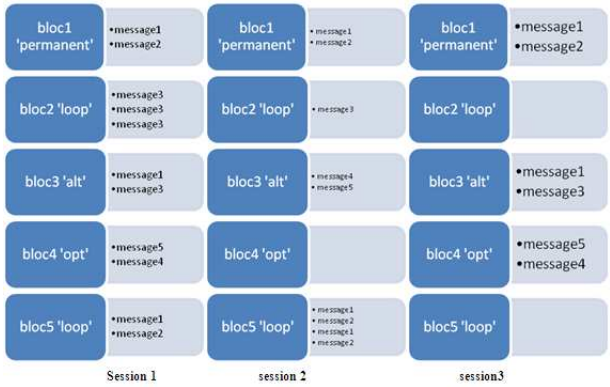

Fig 4. example of trace blocs

2) Detecting combined fragment

a) Combined fragment with the "loop" operator

Idea: contiguously seen the same sequence of messages (more than one time); prove existence of 'loop' combined fragment (CF).

The algorithm below shows steps to detect loop CF, others algorithms will be not presented in detail due to the restricted number of pages:

Algo 1 :
Inputs : Tr=(M,T,<) ∈ *Trace,* SD=(Fc,<) ∈ *SequenceDiagram*
Outputs : SD avec fc=(F,Op) ∈ *fragment-combiné* where Op = loop
Variables i :*integer*; m, succ , next: *Statement; detected: Boolean; Tr1: Trace;*

```
1:    Begin
2:    SD:=null;
3:    m:= FirstStatement (Tr) ;
4:    For i :=1 to T do begin
5:        detected:=false;
6:        succ= successor (m, Tr);
7:        If (m=succ) then
8:            Detected:= true
9:        Else begin
10:           succ:= successor(succ,Tr);
11:           If (succ = null) go to 14:;
12:           Else go to 7:;
13:       end
14:       If (detected) then begin
15:           If (isEquivalent(successorSet(succ,Tr),predecessorSet(succ,Tr)))
      then begin
16:               SD.newCombinedfragment(cf,loop);
```

```
17:            cf. newOperand(opd) ;
18:            opd. SetOperand(succ+ successorSet (succ,Tr)) ;
19:            opd. SetOperand(m+ predecessorSet(succ,Tr)) ;
20:            next := successor(LastStatement (successorSet(succ,Tr)),Tr) ;
21:            Tr1:= successorSet (next,Tr);
22:            While (Tr1<> Φ) then
23:               If (isEquivalent(successorSet (succ,Tr),Tr1) then;begin
24:                  opd. SetOperand(next + successorSet (next,Tr));
25:                  next := successor(LastStatement (successorSet (next)),Tr) ;
26:                  Tr1:= successorSet (next,Tr) ;
27:               end
28:               Else begin m:=next; go to 31:; end
29:         End
30:         Else begin succ:= successor(succ,Tr); go to 7:;end
31:      End
32:      Else m :=successor(m,Tr);
33:   End
34:   end algo1
```

We mast execute this algorithm for each trace session and more parsed traces led to more 'loop' CFs to be detected.

b) Combined fragment with the "opt" operator

Idea: every missed message from any trace in the appropriate bloc is marked as an 'opt' message, so include optional and alternatives ones.

At this point all optional messages are detected, as well including those ones which are alternatives, because every alternative message is optional and not the reverse. Also; the common template for traces is just to be identified; it contains 'loop', 'permanent' and 'opt' blocs' kinds.

The optional messages that appear and hide together are expected to be at the same alternative or optional combined fragment operand. As the same as the "loop" interactions case, if a condition haven't been realized in all traces, then, the concerned "opt" interaction fragment is never to be detected.

c) Combined fragment with the "alt" operator

The case of the "alt" operator is the most complex one, it extremely depend on discrepancies between different traces. At this level our sequence Diagram contains ordered messages, some belong to "loop" operands, others are permanent and the others are optional. We have defined basic rules to emphasize our judgments on a group of messages if are alternatives whither not, these rules are:

a) All messages seen at the same trace bloc (i.e. bloc's included messages in a particular trace) will be not alternatives.
b) Each message in a trace bloc is candidate to be alternative to missed messages.
c) Every trace bloc mast contains at least one of the alternatives groups of messages.
d) The most grouped number of alternatives messages is selected.
e) The rest of messages are optional, then, all messages seen at the same trace bloc are belonging to the same operand and every message missed from one or more trace bloc will be disqualified.

The longest alternative chain:

According to the rule **d)** we compose the longest chain which the case when every message is alternative to other messages:

For the set: {mess***1***, mess***2***, .., mess***k***} we get the chain: "mess***1***+ mess***2***+ ..+ mess***k***"

'+' : alternative.

Chain decomposition:

A chain has to be decomposed to specific number of sub-chains when the case of the rule a) is faced; the old chain must be streamed to obtain the root chain; then new sub-chains are constructed basing on the root chain:

<root-chain> = <old-chain> – <trace bloc's messages>

According to the rules b) and d) we have to construct a number of sub-chains as long as the number of possible combinations of the trace bloc's messages combined without repetition as follow:

$$\text{NbrChains} = \sum_{t=1}^{j} \frac{j!}{(j-t)!\,t!}$$

The algorithm begins by surrounding all possible alternatives groups, and then it follows a method of exclusion according to the bloc's messages combinations in each trace. At the end we'll get zero, one or more chains that support the messages combinations in all traces, we choose automatically the longest one.

We remind that the order of execution of these three algorithms is seriously important, and results will be not reachable if we change the order.

B. Sequenced combined fragment:

As we have indicated above, there is two kind of sequences 1) with the "seq" operator and 2)with the "strict" operator. We can't confine all the combined fragments with sequencing operators since the number of interaction fragments is unknown. In fact, we mast firstly select some interaction fragments to know sequences. However, all combined sequence diagrams that have been detected (with 'loop', 'opt' or 'alt' operator) have"strict" sequences according to the order of their messages.

C. Case study:

To prove the effectiveness of our approach, we chose the sales process as a use case. It is a simple code that can provide different types of behavioral interactions (permanent, iterative, optional and alternative) that are the subject of our study. The seller sends an order to create a new sale (permanent behavior), then the routine addition of items and calculation of the sum is

repeated as the number of items ordered (repetitive behavior). After that, a delivery order or an invoice must be established and then be signed by the seller (alternative behavior). Finally, the creation of a form of payment is subject to customer choice (optional behavior). It should be noted that the position of our current approach does not support applications that contain nested blocs; this makes it difficult to find real applications for validation. With such simple example our approach has proved good results; figure 5 shows the extracted sequence diagram where SD3.4 is a combined fragment of type loop, SDA is a combined fragment of type alt, and SDO a combined fragment of type opt.

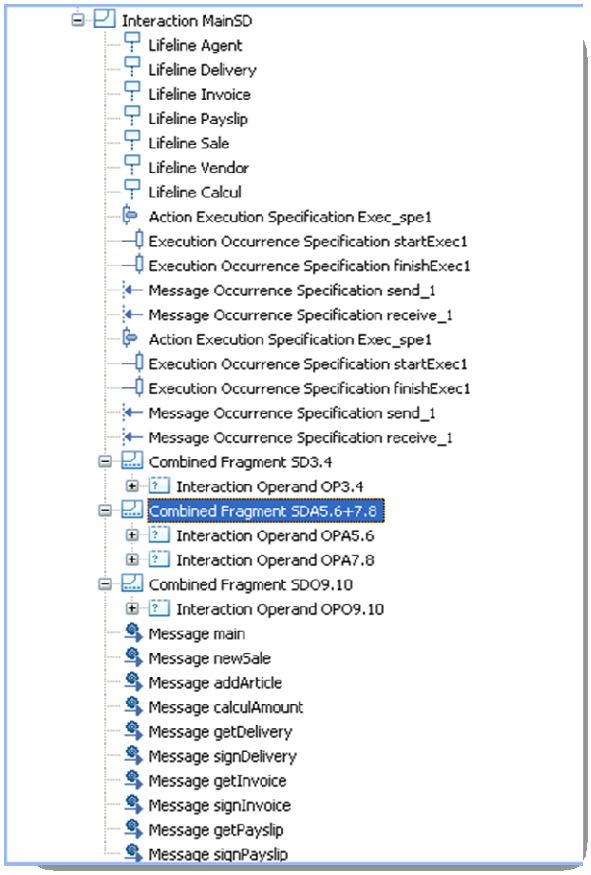

Fig 5. A complete SD displayed with an Eclipse editor

II. Conclusion and perspectives

In this paper we have presented an overview on the reverse-engineering domain. The proposed approach is similar to the Briand et al. ‘s approach [5]. However, Briand et al. ‘s approach is only based on the use of a single trace to construct the SD. The Briand et al. ‘s approach is also related to a specific-language: java. However, our approach can be reused with multiple languages (we just need to change the tracer tool to support the new language).

We have only focused on the main concepts of UML2 sequence diagrams and we have evolved different constraints, nevertheless, yet exists some exceptional cases that we will consider in of future works. Also, future work will concern the integration of the advanced concepts in sequence diagrams such as “gates”, “continuants”, ”interactionUse”, ..etc.

In addition to sequence diagrams, UML proposes State Machine as formalism for system behavior specification. One of the authors of this paper has presented a method allowing automatically translating UML2 sequence diagrams to state machine [10]. So, we plan in the future to reuse this method to generate state machines from the SD obtained after reverse-engineering.

III. References

[1] Chikofsky, E.J.; J.H. Cross II (January 1990). "Reverse Engineering and Design Recovery: A Taxonomy in IEEE Software". IEEE Computer Society: 13–17.

[2] TARJA SYSTA . "Static and Dynamic Reverse Engineering Techniques for Java Software Systems". PhD these. Thesis University of Tampere. May 8th 2000.

[3] OMG Unified Modeling Language (OMG UML), Superstructure. V2.1.2; November 2007.

[4] http://www.ptidej.net/material/inanutshell. August 2009.

[5] L. C. Briand, Y. Labiche, J. Leduc, "Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software", IEEE Transactions on Software Engineering, vol. 32 (9), pp. 642-663, September 2006

[6] F. Fleurey et al. JTracor. http://franck.fleurey.free.fr/JTracor/index.htm

[7] G. Booch. Object Oriented Design with Applications. Benjamin Cummings, 1991.

[8] J. Rumbaugh, M. Blaha, F. Eddy, P. W., and W. Lorensen. Object Oriented Modeling and Design. Prentice Hall Inc. Englewood Cli_s, New Jersy, USA, 1991.

[9] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. Object-Oriented Software Engineering : A Use Case Driven Approach. Addison-Wesley, 1992.

[10] T. Ziadi, L. Hélouët, J-M. Jézéquel, Revisiting Statechart Synthesis with an Algebraic approach, in IEEE International Conference on Software Engineering (ICSE 04), May 2004, Edinburgh, UK.

[11] Robert B. France, Sudipto Ghosh, Trung Dinh- Trong, and Arnor Solberg. Model-driven development using uml 2.0: Promises and pitfalls. Computer, 9(2):59_66, 2006.

[12] Yann-Gael Gueheneuc, Rémi Douence, Narendra Jussien. "No Java without Caffeine A Tool for Dynamic Analysis of Java Programs". Ecole des Mines de Nantes; May 16, 2002.