Detecting Source Code Changes to Maintain the Consistence of Behavioral Model

Yuankui LI,
Linzhang WANG[*]
Xuandong LI
State Key Laboratory for Novel Software Technology,
Department of Computer Science & Technology
Nanjing University
lyk@seg.nju.edu.cn,
{lzwang, lxd}@nju.edu.cn

Yuanfang CAI
Department of Computer Science,
Drexel University
Philadelphia, PA
yfcai@cs.drexel.edu

ABSTRACT

It is well-known that as software system evolves, the source code tends to deviate from its design model so that maintaining their consistence is challenging. Our objective is to detect code changes that influence designed program behaviour which are referred as *design level changes* and update the behavioural model timely and automatically to maintain consistence. We propose an approach that filters out low-level source code changes that do not influence program behaviour, abstracts code changes into updating operations for behavioral model, and automates the integration and update of activity diagrams to maintain consistence. We've recognised that it is not uncommon for developers to introduce quick and dirty implementation that unnecessarily increases program complexity or introduces suboptimal behaviour changes. So while merging code changes into behaviour model, our approach also calculates cyclometric complexity variation before and after the process so that developers can be alerted of significant and/or detrimental changes. Our tool allows the user to approve the change in code before merging and updating the model.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

EXPERIMENTATION

[*]Corresponding author

Keywords

behavioral model, software maintenance, source code differencing

1. INTRODUCTION

Maintaining the consistence between code and model is challenging. As source code evolves, developers either do not have enough time to update the model due to the pressure of pending deadlines or do not have sufficient support to evolve the model correctly and continuously, which usually is at much higher level of abstraction. Our objective is to abstract detailed source code changes into higher level changes in behaviour models so that significant behavioural changes can be understood, tracked and documented.

Our work is related to both one way code-model transformation, recovering model from code or transforming model into code, and two-way transformation and tracing. As a representative work of recovering model from code, Briand et al.[2, 3] proposed an approach that generates UML sequence diagrams from Java source code, with automated tools. The idea is to combine static semantic analysis and run time tracing to restore the behaviour of a system. Work of Hearnden et al.[6] describes how to propagate the modification from the source model to target model lively, dealing with more formalised models such as Ecore/MOF-/QVT. Those models have well-defined rules, based on which transformation can be performed. By contrast, our work aims to generate activity diagram which is at a higher level to ease program understanding, and leverage a light-weighted method where only source code is needed.

Yu et al. [12] just published their recent work about maintaining the traceability between model and code. The basis of this work is the well-defined bidirectional transformation framework *GRoundTram*[7] which could guarantee the round-trip property in bidirectional model transformation. However, it doesn't distinguish the behavioral changes from normal changes. László et al. described a technique to generate code from model[1], their framework could translate domain specific model(DSM) to platform specific model (Abstract Syntax Tree, AST) and finally the source code. While the existing work aims at consistence between code and static model, we are interested in abstracting away de-

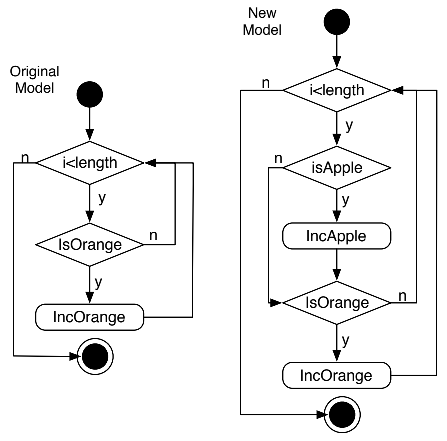
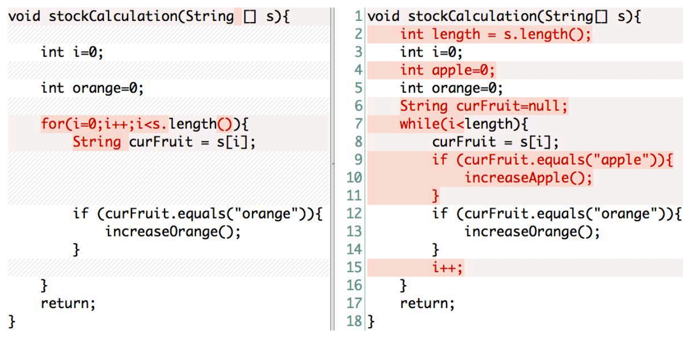


Figure 1: The behavioral models



```
void stockCalculation(String [] s){

    int i=0;

    int orange=0;

    for(i=0;i++;i<s.length()){
        String curFruit = s[i];



            if (curFruit.equals("orange")){
                increaseOrange();
            }

    }
    return;
}
```

```
void stockCalculation(String[] s){
    int length = s.length();
    int i=0;
    int apple=0;
    int orange=0;
    String curFruit=null;
    while(i<length){
        curFruit = s[i];
        if (curFruit.equals("apple")){
            increaseApple();
        }
        if (curFruit.equals("orange")){
            increaseOrange();
        }
        i++;
    }
    return;
}
```

Figure 2: The original code versus the new code

tailed code changes, and incrementally update higher level behavioural model to renew the inconsistent model and ease program understanding.

We are proposing a framework, which would detect the changes between distinct versions of source code, and find out model related changes such as behavioural changes for UML Activity Diagram. By analysing changes between original and new code, the framework decides how to modify the original model to yield a new model which conforms to the new code. After the model delta updating operations are generated, we calculate cyclometric complexity variation, if it exceeds a threshold, which indicates that the model changes might be detrimental, then the changes captured from the source code should be reviewed by developers instead of propagated backward to the model.

The work we are describing is novel in that it's suggesting an integrated framework, to detect the changes of source code, filtering behavioural changes, and to distill model updating details from the behavioural change information. Instead of updating the model directly, we also would analyse the updating operations to see if the changes are actually corruptive to the software system.

2. MODEL UPDATING FRAMEWORK

Our framework is illustrated in Figure 3. It takes the original behavioural model M_0, original code C_0 which is implemented according to M_0 and the evolved new code C_n as input. First the framework detects the behavioural changes from C_0 to C_n, then it generates operations which could update the original model to resolve the inconsistence. Just before the model updating operations are performed, the framework calculates the difference measurement between the original model and the to-be-verified model, to check if the changes being propagated back from code to model are corruptive to the software system, on which situation a warning would be raised to request the developers' review and confirmation to update the model.

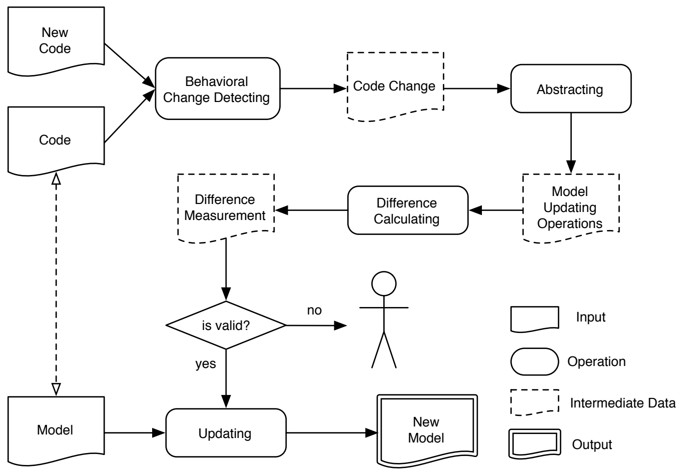


Figure 3: Overview of the framework

In the following sections, we are going to demonstration the details of our framework with models and code snippets in Figure 1 and Figure 2. There are two source code snippets in Figure 2, on the left side is the original source code, which is implemented according to the UML activity diagram on the left side of Figure 1. After the source code has evolved for a while and undergone many modifications, finally evolved to the shape on the right side of Figure 2. There are many differences between them, as shown with shaded areas, the original model could no longer represent the new source code. By detecting code changes between those versions of source code and abstract code changes into model updating operations, the framework could decide how to rectify the obsolete model, yielding a new model as on the right side of Figure 1, which conforms to the new code concerning the program behaviour.

2.1 Behavioral change detection on evoloving source code

To detect the changes on original source code, we precompile the source code to Abstract Syntax Tree, i.e., AST, take java code as an example. **Java-AST** Having inherited many features from C++ and C, Java views everything as objects with their own states and operation. Programs written in Java would complete tasks by sending messages to each other. Grammar of Java is rather complex, with hundreds kind of AST nodes.

The basic units in a Java program can be categorised into two worlds, i.e., statement and expression. Expressions are used to command the computer to compute, and statements are used to coordinate the sequence. For example, an assignment 'a=b+c', the righthand side 'b+c' is an expression, telling the computer to evaluate 'b plus c', while the whole assignment is a statement, in which no computation is intentionally involved because an assignment only conducts the data to flow from right to left. In structural programming, statements are used to conduct the control flow and data flow, such as if statement, while statement and assignment, those mathematically speaking, 'if' statement can be constructed by the elementary recursive functions, and 'while' statement is just an generic form of 'if' statement. Object-oriented programming is on a higher layer than structural programming, which means it's more abstract than the tedious structural programming. Software are divided into modules and sub-modules, while a single functionality might spread over many methods or classes.

The AST of code is stored in an xml document, while the implementation of an raw tree diff algorithm. The AST model we are adopting is simpler than a real AST model. Without the loss of generality, we choose the following types of nodes: MethodDeclaration, MethodInvocation, IfStatement, WhileStatement. Almost all the control flow semantics in Activity Diagram are depicted with these nodes.

To update the obsolete model, we must detect behavioural changes between the code. We perform semantics analysis to filter out non-semantics changes and non-behavioural changes. In line 1 of Figure 2 the elimination of a white space has no actually impact to the semantics of the program. The code differences in line 2, 4, 6, 15 might affect some subtle behaviours of the program, however, the high level behaviour would not change. And in line 7, the original version uses *for* to state a loop, but in the latest version the coder preferred a semantically similar *while* statement. In line 9-10, some new conditional statements are inserted, and the control flow is affected, causing high level behavioural change. So code changes could be classified into three categories:

1. Those changes such as reformatting of the code structure, reordering of some sibling fields, and in most programming languages(except languages such as Python and Haskell), white space elimination and addition, though the source code would look different, they have no apparent effect on the semantics of a program.

2. Those non-behavioural changes that have no direct effect on the behaviour of a program, i.e., the structural changes such as the definition of a local variable or a field member, the rename of some variables.

3. Those behavioural changes that would affect the control flow in a program, for example adding or removing an invocation, conditional statement etc.

The change of source code must be captured with regard to its semantic properties. The source code is transformed by a compiler to AST, which is a compiling intermediate representation and could retain the essential meaning from source code. By default, any semantically significant changes in code will result in changes in AST, it retains all the useful information conveyed in its original source code, while neglecting the coding format styles, such as the position of a brace for an *if* statement, or the number of white spaces, which contribute nothing in the final execution of programs as in the first category. Since detecting the changes in ASTs would extract the change details of source code, the change detection task is reduced to the comparison of ASTs. Each node in AST has its own semantical

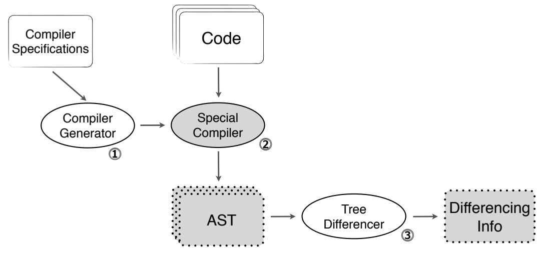


Figure 4: Specifying and detecting certain sort of changes

functionality, so we can pick up certain behavioural parts in a complete AST and construct a minified AST which only consists the information we care.

To filter out the behavioural changes, we specifies the compiler in a well-known compiler domain specific language ANTLR[8] thus controlling the format of the generated AST as in Figure 4. Also, it would produce the same AST structure for the semantically similar language components, thus normalising the AST output, to ignore the different implementation for the same control flow. For example, when the compiler recognises *while* statement and *for* statement, it could yield almost identical AST structure.

The generated AST is a tree structure, the change detection of which is different from the lines-of-text situation and a little complicated[4]. Fluri et al. [5] improved the algorithms in [4] to detect the changes on AST generated by Eclipse JDT component. They are aiming to identify particular changes that occur across version history of a program, while calculating the significance level of changes with respect to the granularity and type. Many tree differencing algorithms have been devised[9]. To detect the difference in two ASTs, we must notice that the AST's nodes, though stored in a xml file, are ordered.

Since the generated ASTs in our framework are stored in xml files, we deployed an xml file differencing algorithm to detect and output the edit scripts(A.K.A. editing sequencing). After comparison, we devised an algorithm based on Wang et al. in[11]. This algorithm supports three kinds of edit operations: *changing* the attributes of a node, *deleting* a sub-node and *inserting* a new node. More edit operations such as *moving* some nodes or *reordering* the sibling nodes could be considered. Performing the tree-differencing algorithm on versions of AST, the output, i.e., the changes will be represented as a list of tree edit scripts.

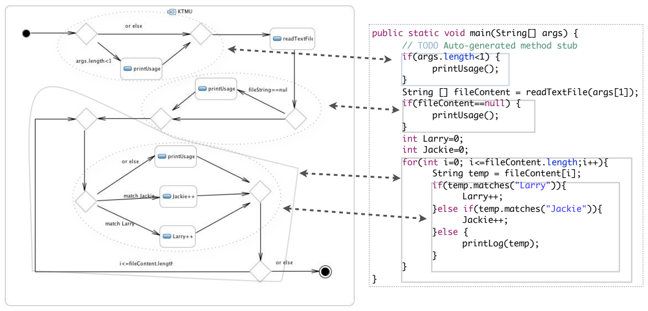
The advantage of our change detection method includes:

- To neglect the format of source code, such as the position of brackets, the number of white spaces.

- To neglect certain subtle changes, for example, through the specification, one can neglect the change from type *int* to *long*.

- To neglect the non-behavioural, for example, we can specify the definition of a local variable as insignificant, thus ignoring such changes.

- To neglect some semantically invariable changes, by normalising the statements with similar semantics. For example, the loop structure statement with keywords



```
public static void main(String[] args) {
      // TODO Auto-generated method stub
      if(args.length<1) {
            printUsage();
      }
      String [] fileContent = readTextFile(args[1]);
      if(fileContent==null) {
            printUsage();
      }
      int Larry=0;
      int Jackie=0;
      for(int i=0; i<=fileContent.length;i++){
            String temp = fileContent[i];
            if(temp.matches("Larry")){
                  Larry++;
            }else if(temp.matches("Jackie")){
                  Jackie++;
            }else {
                  printLog(temp);
            }
      }
}
```

Figure 5: Illustrating the relationship between a model and the corresponding source code

for, *foreach*, *while*, *do...while* could be grouped as similar, while *if...else* and *switch...case* are grouped as conditional components.

2.2 Abstracting the source code edit script to model updating operations

Software engineering is the art of handling the complexity, through abstracting, a smaller model could reflect some aspect of a huge system. While in our case, for a certain granularity, the model represents its corresponding source code. Many research work has been done to connect or reconnect the source code to its corresponding models or documentation. The connection between them are usually referenced as *Traceability Links*.

The matching between code and model is a critical stage. As far as we know, no literature has ever discussed about such a process. One non-trivial issue about it is that there are no explicit rules to map the statements of a language to the elements of a model. Meanwhile it is not possible to devise a set of rules mapping between them without information loss. To implement a same module, different programmers might write various programs, which have similar functionality, but differ a lot in details. Even with the same design model as guide, different programmers might still yield different programs. However, the programs would share some common characteristics which could be mined by empirical study.

In this way we find that the mapping info between source code and its corresponding model is not easy to find. Possible technology includes dynamic analysis, name matching, semantics induction, and assigning manually. For simplified Models that are used in Regression Testing and such scenario, we could outline simple rules to direct the matching. To ensure the soundness of the process, we would suggest programmer to confirm the mapping.

The source code edit scripts, represented as AST edit scripts in our framework, specify the changes between the original code and the new code. To propagate the modifications back to the model, the edit scripts must be distilled to decide how to change the model. The AST edit script includes these operations:

The Types of AST Edit Script

Operation	*Semantics*
Deletion	Delete([x,y]) removes the y-th child node of node x, as well as its children.
Insertion	Insert(n,[x,y]) adds the new node n as the y-th child of node x.
Update	Update(p(n),n) sets the property of node n as p(n).

The artifacts in the development of a software are all generated by tools or by the related staff. The models are usually used as a blueprint or a detailed guide for programmers to implement, so that the model and the code are somehow correlated. Take the Activity Diagram as example, an action might be implemented as an invocation of a method or an initialization of an object or any statements or expressions, as long as they are semantically consistent. Notwithstanding those mapping relationships are usually obvious and clear for proficient software engineers, we must admit that generally the model and the code are neither surjective nor injective. Sometimes, an element in model might be neglected during implementing because of some certain reasons, especially when the element appears in the model only for the consideration of syntax. For example, the start node and the end node in activity diagram do have their semantics, but no programmers would explicitly write anything for them. On the other hand, models are usually abstract so that some details are for the programmers to fill up in their code.

What effect does a change in AST have on the related design model? When the AST edit scripts generated in the prior stage come to this stage, we need to classify those edit operations by its significance to the model as well as by the type and position of their referring AST nodes.

No Significance: Those changes on code reflect no feedback for the model, and real modeler usually don't care about such changes. So we don't need to modify the model accordingly. For example, a variable declarations might not be shown in an Activity Diagram, thus its renaming or deletion will not be a concern.

Design-level Significance: Source code related to those changes are correlated to the model, which is important for the rebuilding of consistence between them. For example, if a method is written according to the model, and somehow it's renamed in later versions of code, the corresponding component in the original model should also be renamed.

To judge if a change is design-level significant for a model, we have to concentrate on certain sorts of changes, for example, the changes happened to the code which describes the behaviour of a programs might be more important, while the structural changes could be neglected as long as they are consistent. The AST reflects the structure of its original source code, and the components of the language are organised as layered nodes in AST. Each node has its semantical functionality, for instance, the **forInit** node represent the initial statement of a *for* statement. With this observation, we can pick up certain behavioural parts in a complete AST and construct a minified AST which consists the information we care only. To pick up the nodes, we'd recognise the named tree nodes and describe the behavioural patterns. To sum up, the specification mainly covers up such details: *Class Declaration*, *Method Declaration*, *While Statement*, *If Statement*, *Method Invocation* and other properties which are related to those mentioned nodes.

Since the code is written according to the model, there certainly exist some traceability links between them, on basis of which, we could bridge the code and model, yielding the mapping information, which is useful to decide where to perform the modification. The basic assumption is that the original code and model are soundly consistent, which is fair, considering those situations such as model driven development or regression testing to ensure the property. Though it is notable that the mapping might be neither bijective nor surjective. As long as they are consistent, we can map some parts of the source code to some part of the model. In our framework, elements in the original model and the components of AST are correlated by name matching and simple rules. For example, we relate the AST to the behavioural model Activity Diagram with some key observations:

- A Method Invocation in the source code could be mapped as an opaque action or a composite action.
- The control flows such as the order of statements in a Method Declaration are usually the same as the control flow in Activity Diagram.
- IfStatement or WhileStatement could be mapped to a structured or composite action. The internal control flow could be revealed if necessary, because we could zoom in the details of a composite action, in which case the conditional statements are mapped to the guard on the flows, while the control flows are intuitive.

In Java AST, the changes of a node might affect some other nodes, for example, the removal of a single 'if' phrase will change the behavior of the nodes in its sub-phrases (be them conditional phrases or method invocation).

The artefacts in the development of a software are generated by tools or manually. The models are usually used as a blueprint or a detailed guide for programmers to implement the real software system, so the model and the code are somehow correlated. Take activity diagram as example, an action might be implemented as an invocation of a method or an initialisation of an object or any statements or expressions, as long as they are semantically consistent.

Activity Diagram As a part of UML, Activity Diagram could be used in modelling of workflows, business processes, and web-services, emphasising the sequence and conditions for coordinating lower-level behaviours. Also, for the ease of analysing, the Activity Diagram is degenerated. We take up only a few types of elements to form models for Java code.

A simple UML Activity Diagram AD is formally defined as follows.

- $AD = \{S, E, N, F\}$
- $S = StartNode$
- $E = EndNode$
- $N = A \cup D \cup M$
- $A = \{ActionNode\}$
- $D = \{DivideNode\}$
- $M = \{MergeNode\}$
- $F = \{Flow\}$

Table 1: Translation of AST Edit Scripts

Operations	Invocation	If	While	Else
Update	Change the related attribute.			
Insert	Create a new node and connect it to the diagram.			
Removal	Delete the mapped action and connect its pre- and post- node.			Redirect the flow.

The nodes have properties, all the nodes in N have property **id**, nodes in A have property **name**. Flows have property **id**, **in**, **out** and optional **guard**. To guarantee the semantically and structurally correctness of an Activity Diagram, some constraints must be satisfied.

- All the nodes have distinct ids.
- $\exists f \in F, f.in = S.id$
- $\exists f \in F\ satisfying\ f.out = E.id$
- $\forall a \in A, \exists\ unique\ f \in F\ satisfying\ f.in = a.id$
- $\forall a \in A, \exists\ unique\ f \in F\ satisfying\ f.out = a.id$
- $\forall d \in D, \exists\ unique\ f \in F\ satisfying\ f.out = d.id$
- $\forall d \in D, \exists\ two\ f \in F\ satisfying\ f.in = d.id$
- $\forall m \in M, \exists\ unique\ f \in F\ satisfying\ f.in = m.id$
- $\forall m \in M, \exists\ two\ f \in F\ satisfying\ f.out = d.id$

Notwithstanding those mapping relationships are usually obvious and clear for proficient software engineers, we must admit that generally the model and the code might be neither surjective nor injective. Sometimes, an element in model might be neglected during implementing, especially when the element appears in the model only for the consideration of syntax. For example, the start node and the end node in activity diagram do have their semantics, but programmers wouldn't explicitly write anything for them. On the other hand, models are usually simple so that some details are for the programmers to fill up in their code. So the AST edit scripts should be further investigated with the help of its context to dig out their meanings to decide how to modify the correlated elements in the model. Once the traceability between AST and model is established, behavioural changes could be abstracted to model updating operations.

2.3 Verifying the updating

The final stage is to update the original model according to the model updating operations as Table 1, however, it is notable that not all the modifications performed on source code by programmers are correct. So before committing the updating operations, the result must be verified to preview the impact as a safe-belt and warnings shall be raised if necessary and the engineers should confirm the warning and supervise the updating process. We calculate model metrics such as cyclometric complexity density to evaluate the new model.

Due to the restrict of time and costing, programmers have a tendency to mix up the design structures to corrupt the software, especially when its size and the complexity are gradually growing. Also when the deadline is approaching or they are eager to fix bugs, which is quite common during software maintenance, the design models and increase the complexity of the whole systems. So the assumption that all the modifications on source code are always reasonable and improving does not hold, it is necessary to analyze the source code changes even the evolving system does not appear to have distinctive defects.

The cyclometric complexity is a well-known metric to quantify the complexity of the software system and to indicate the cost of testing. To monitor and limit the cyclometric complexity of the software under development is also an efficacious method to control the number of defects. We calculate the cyclometric complexity of the new model which would be an assisting index for the developers to reconsider about the quality of the new evolving software system.

3. CONCLUSION AND FUTURE WORK

The work contained in this paper is motivated by the need of model recovery, which is in turn motivated by model-based testing, why we are pursuing such an automatic framework is inspired by the observation of the importance role that the model plays in testing such as the previous work by one of the authors [10]. We have presented a behavioral model consistence maintenance framework based on code change detection and model measurement calculation.

The framework in this paper should be enhanced step further, and there are some interesting research issues to be addressed to extend and improve its ability. For example the change detection part could be enhanced by source code retrieving, or edit script retrieving, to find the targeted changes with more preciseness. Rules about the translation should be defined in a formal declarative language explicitly. The framework are using very simple and basic transformation rules to abstract the change information to generate the model updating operations, which limits its ability. To specify the transformation rules we are currently working on devising a formal declarative language based on the formalization of AST and the behavioral model. The model measurement like cyclometric complexity should be calculated incrementally *w.r.t* the updating operations, on a basis of formal definition. Future research will also include migrating the framework to assist model based testing for legacy software systems, especially regression testing.

4. ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No.61021062,61170066), the National Grand Fundamental Research 973 Program of China (No.2009CB320702), and by the National 863 High-Tech Program of China (No.2012AA011205).

5. REFERENCES

[1] L. Angyal, L. Lengyel, and H. Charaf. A synchronizing technique for syntactic model-code round-trip engineering. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 463–472. IEEE, 2008.

[2] L. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *Software Engineering, IEEE Transactions on*, 32(9):642–663, 2006.

[3] L. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of uml sequence diagrams. In *proceedings of the 10th Working Conference on Reverse Engineering*, page 57. IEEE Computer Society, 2003.

[4] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *ACM SIGMOD Record*, volume 25, pages 493–504. ACM, 1996.

[5] B. Fluri, M. Wursch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[6] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. *Model Driven Engineering Languages and Systems*, pages 321–335, 2006.

[7] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. Groundtram: An integrated framework for developing well-behaved bidirectional model transformations. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 480–483. IEEE, 2011.

[8] T. Parr, J. Lilly, P. Wells, R. Klaren, M. Illouz, J. Mitchell, S. Stanchfield, J. Coker, M. Zukowski, and C. Flack. Antlr reference manual. *MageLang Institute, document version*, 2(0), 2003.

[9] L. Peters. Change detection in xml trees: a survey. In *3rd Twente Student Conference on IT*, 2005.

[10] L. WANG, J. YUAN, X. YU, J. HU, X. LI, and G. Zheng. Generating test cases from uml activity diagram based on gray-box method. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 284–291. IEEE, 2004.

[11] Y. Wang, D. DeWitt, and J. Cai. X-diff: An effective change detection algorithm for xml documents. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 519–530. Ieee, 2003.

[12] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux. Maintaining invariant traceability through bidirectional transformations. In *34th International Conference on Software Engineering (ICSE)*, pages 540–550, 2012.