Published in SAC '13 : Proceedings of the 28th Annual ACM Symposium on Applied Computing , 2013, p. 1284-1291
which should be cited to refer to this work. doi>10.1145/2480362.2480602

Dynamic Decision Tree for Legacy Use-Case Recovery

Philippe Dugerdil
Geneva School of Business Administration.
Univ. of Applied Sciences of Western Switzerland
7 route de Drize, CH1227 Geneva, Switzerland
+41 22 388 17 00
philippe.dugerdil@hesge.ch

David Sennhauser
Geneva School of Business Administration.
Univ. of Applied Sciences of Western Switzerland
7 route de Drize, CH1227 Geneva, Switzerland
+41 22 388 17 00
david.sennhauser@gmail.com

ABSTRACT
In the context of reverse-engineering project we designed a use-case specification recovery technique for legacy information systems. With our technique, we can recover the alternative flows of each use-case of the system. It is based on a dynamic (i.e. runtime) analysis of the working of the system using execution traces. But "traditional" execution trace format do not contain enough information for this approach to work. Then we designed a new execution trace format together with the associated tool to get the program's dynamic decision tree corresponding to each of the use-case scenario. These trees are then processed to find the possible variants from the main scenario of each use-case. In this paper we first present our approach to the use-case specification recovery technique and the new trace format we designed. Then the decision tree compression technique is showed with a feasibility study. The contribution of the paper is our approach to the recovery of legacy systems' use-case, the new trace format and the decision tree processing technique.

Categories and Subject Descriptors
D.3.1 [**Software Engineering**]: Requirements/Specifications - *Methodologies*

General Terms
Algorithms, Experimentation

Keywords
Use-case recovery, dynamic analysis, decision tree processing

1. INTRODUCTION
In our lab, we developed a reverse engineering technique for legacy systems based on their use-cases [6][7]. Generally, legacy systems documentation is at best obsolete and at worse non-existent. Moreover it is often the case that the developers are not available anymore to provide the maintainers with information on the structure of these systems. In such situations the only people that still have a good perspective on the system are its users: they are usually well aware of the business relevance of the programs. This information can be recorded as system use-cases. In short, our iterative and incremental reverse engineering process [8] works the following [7] :

1) Re-documentation of the system use-cases;
2) Manual redesign of the analysis diagrams associated to each of the use-cases;
3) Execution of the system according to these use-cases (i.e. their scenarios) and recording of the execution trace;
4) Analysis of this execution trace and identification of the classes involved in the trace;
5) Mapping of the classes in the trace to the classes of the analysis diagram;
6) Re-documentation of the architecture of the system by clustering the classes based on their role in the use-case implementation.

Since we fundamentally rely on users to re-document the use-cases, the completeness of the latter is an issue. Indeed the re-documented use-cases are never complete and accurate, especially regarding the alternative flows. We then need a semi-automatic way to recover the missing use-cases flows. In the literature attempts have been made to recover the use-case form the mere analysis of the source code of the legacy software (see for example [16] [18]). But in [5] we have showed this to be fundamentally impossible. We then claim that the solution to the problem, if any, could only come from the *directed* dynamic analysis of the software coupled with the static analysis of its source code. Dynamic analysis is a program analysis technique based on the execution trace of the program [1]. Basically an execution trace (or trace for short) represents the sequence of methods (or functions) called when a program is run. By *directed* we mean that our use-case recovery technique is based on an example scenario for each use-case. The problem is therefore to re-document the complete use-cases with all their relevant flows from the simple scenarios described by the users. Indeed, it is often the case that the user only knows a very limited subset of all the functions of the system. The challenge is to recover the missing part.

Our approach works the following: an execution trace is recorded when running each of the scenarios observed when the users perform their business task. Such a trace represents what we called the *backbone* trace of the corresponding use-case. Then we analyze the possible variants of the scenario by identifying the conditional statements in the execution path represented by the backbone execution trace. Next, one searches what user action could change the boolean condition of each statement. Such an

action would then change the execution flow hence create a new scenario. In this way we would have discovered an alternative path for the original scenario. By repeating this experiment we will slowly recover all the variants of the initial scenario. Specifically, to trigger the alternative behavior we must identify the control statements in the execution trace and determine what action the end user could take through the user interface to change the conditions of these control statements to execute the alternative flows. In short we must answer the following question: could the user trigger the alternative flow by manipulating some user interface control? The key feature of our approach is to allow us to recover the *sequence* in which the user actions could be performed on the systems screens, since we can observe this sequence in the trace. In particular we know when, in a valid scenario, an alternative could be triggered. By contrast any arbitrary sequence of user interface manipulation does not represent a valid scenario since, according to [15], "A use case describes sequences of actions a system performs that yield an observable result of value to a particular actor". There is very little chance that an arbitrary sequence of user interface actions could lead to such observable result of (business) value for the actor. But our approach does, since we start from a valid scenario and we rebuild variants from this scenario. In summary, our technique works the following way:

1. We redocument the main use-case of the system by observing its users. This gives us one scenario per use-case (the backbone scenario).
2. We instrument the source code of the system (i.e. we insert tracing statements), to be able to generate an execution trace. This new version of the source code is recompiled.
3. The new compiled version is run according to the backbone scenario.
4. We collect the execution trace and process it off-line to find the executed control statements. The result is stored as a decision tree (called a dynamic decision tree).
5. This dynamic decision tree is compressed to reduce the quantity of control statements to analyze.
6. The reduced tree is analyzed top-down to identify the control statements whose boolean condition could be modified by some user action on the screens. The candidate action is identified by source code analysis (program slicing [19][14]) and represents the trigger of a variant of the initial scenario.
7. If a valid variant is identified at step 6, we design a new scenario with this variant and go back to step 3. If there is no valid variant, the process is ended.

In short our technique rests on the capability to record the sequence of control statements executed for a given scenario, and to identify those for which the condition could be changed by some action of the user. Moreover we must know, for each executed control statement, if both alternatives have been executed and, if not, which one remains to be executed. It is worth noting that the recovery of such a *valid* (i.e. business-relevant) sequence of control statements by the mere analysis of the source code is fundamentally impossible. In a first version of our tool, we worked with a "standard" execution trace format containing only the method calls. Since the control statements were not explicitly represented we had to analyze the source code of the executed methods to recover these statements and determine which alternative was actually executed. But this technique has been very hard to implement, particularly because of the difficulty to analyze the types of the variables in the context of the late binding (Java). After having tried several alternative options, we gave up and decided to create an extended format for the execution trace that would explicitly record the executed conditional expressions. This worked fine on small programs. But we soon realized that the number of nodes to analyze was much too big for our scenario recovery technique to work in practice. Recall that, in our approach, each of the executed conditional statement must later be processed to identify the user-interface control that could possibly change the condition so that the alternative execution flow could be triggered. This lead us to develop a compression technique of the dynamic decision tree of the program, to keep only the relevant conditional statements. The result is what we called the Reduced Dynamic Decision Tree (RDDT).

This paper presents the techniques we developed to record and process the sequence of conditional expressions (sometimes called DD-path or Decision to Decision path in the literature [17]) gathered when running the system. The paper is structured as follows. Section 2 presents the extended trace format and model we developed to gather the relevant conditional expressions. Section 3 presents the dynamic decision tree model we use to process the conditional statements in the trace. In section 4 we give some hints on the code instrumentor. Section 5 presents a feasibility study, section 6 presents the related work and section 7 concludes the paper.

2. EXTENDED TRACE FORMAT

2.1 Limitations of the standard format

An execution trace is a sequence of trace event representing the method called when the system is run. Several trace formats have been proposed in the literature, but the fundamental information that must be recorded is:

[classID] [methodSignature]

Where *methodSignature* is the method that has been called and *classID* is the identifier of the class of the instance that executed the method. Whatever the format, it is important to note that an execution trace does not normally record the conditional statements but only the method (or procedure) calls. In the explanation below we will refer to this kind of trace as the "standard" trace. With such a format, if we must identify which control statement was executed we must *deduce* it from the recorded method calls and the source code, since no explicit information is available in the trace. Here is a simple example. Let us have the following source code statements and execution trace (in its simple standard format):

Program (in class Class1):

```
x.m1();
if (condition1)
   then y.m2();
   else y.m3();
x.m4();
```

Trace:

```
Class1 m1()
Class2 m2()
Class1 m4()
```

In the above example, it is clear that *condition1* was *true* during the execution since we observe m2() in the trace. However a simple variant of this code could be more difficult to analyze.

Program (in class Class1):

```
x.m1();
if (condition1)
    then y.m2();
    else z.m2();
x.m4();
```

Trace:

```
Class1 m1()
Class2 m2()
Class1 m4()
```

Now we must identify the *type* of the variables y and z to know which one corresponds to Class2. Only then could we deduce the boolean value of *condition1*. But the identification of the dynamic type of each variable could lead to a very difficult search in the source code since this may involve other conditional statements as well as late binding. In the case below the code may even lead to ambiguities which are hard to solve:

Program (in class Class1):

```
x.m1();
if (condition1)
    then {y.m2(); y.m3();}
    else y.m2();
if (condition2)
    then y.m3();
x.m4();
```

Trace:

```
Class1 m1()
Class2 m2()
Class2 m3()
Class1 m4()
```

In this case, were *condition1 true* and *condition2 false* or the other way around? This is important because we must know what alternative remains to be executed. In short, we found it very hard, for each conditional statement, to determine what branch was actually executed from the limited information available in the "standard" trace format. Initially the motivation to keep the standard format was to allow anyone in the dynamic analysis community [12] to reproduce our results. But we realized that this goal had to be abandoned in favor of a richer format including information on the executed control statements.

2.2 Extended trace format

The limitations of the standard trace format lead us to define a new format with explicit representation of the control information (decision nodes plus executed alternative). This new trace format allows us to identify which conditional branches were executed and if an alternative flows of execution remains to be executed. Moreover, since we need to record the hierarchy of the calls to build a decision tree, not only must we record the method entries but also the method exits. In short, our technique is to insert the tracing statement at the beginning and end of the methods as well as in the conditional statements themselves. However, in the case of the conditional statements, where should the tracing statement be inserted to know the executed alternative? Moreover, does an alternative path actually exist for the statement (is the *then* part followed by an *else* part for a given *if* statement or not)? Consequently, we decided to instrument the blocks of code associated to the *then* and *else* parts of an *if* statement and the block of code associated to loop statements (*for*, *while*,…).

2.3 Trace format grammar

To limit the impact of the insertion of the tracing statements on the performance of the system, the trace information is not directly written in a database but in a flat file. This file is later loaded offline in a database for further processing. With the new format, the trace events are method calls and conditional statements execution. Below, we present the grammar of the trace events recorded in the trace file.

Method entry:

[spn][scn][dpn][dcn]'['[tn]']' [evt]'AS'[rt]'['[tsIN]']'[pv]

Method exit:

'END'[spn][scn][dpn][dcn]'['[tn]']' [evt]'AS'[rt]'['[tsOUT]']'

Where:

[spn] := fully qualified name of the package of the class in which the executed method or conditional statement are declared.

[scn] := name of the class in which the executed method or conditional statement are declared.

[dpn] := fully qualified package name of the class of the instance that actually executed the method or conditional statement.

[dcn] := name of the class of the instance that actually executed the method or conditional statement.

[tn] := id of the thread in which the method or conditional statement is executed.

[evt] := [sign] | [cond]

[sign] := signature of a method: method name followed by the ordered list of the fully qualified types of the parameters of the method, written within brackets. This grammar element is only relevant for events corresponding to method executions.

[cond] := 'IF_TRUE()' | 'IF_TRUE_NO_ELSE()' | 'IF_FALSE()' | 'FOR_TRUE()' | 'FOR_FALSE()' | 'WHILE_TRUE()' | 'WHILE_FALSE()' | 'DOWHILE_TRUE()'. This is the identification of the conditional statement that has been executed. This grammar element is only relevant for events corresponding to conditional statement executions.

[rt] := if ([evt] = [sign]) this is the name of the type of the value returned by method. If ([evt] = [cond]) this element is **void**.

[tsIN] := timestamp when entering the method or the conditional expression.

[tsOUT] := timestamp when exiting the method or the conditional expression.

[pv] := [mpv] | [le] '@' [cl] | whitespace

[mpv] := method parameter values: list of the values of the primitive-typed parameters in the same sequence as the parameter type in the method signature. If a value is not of primitive type, it is represented by the underscore character '_'. This grammar element is only relevant for events representing method calls ([evt]= [sign]). If the method does not have any parameters, this element is empty (white space).

[le] := string representing a logical expression associated with a control flow statement in the source file. This grammar element is only relevant for events representing conditional statements ([evt] = [cond]).
[cl] := location of the control statement: [static package name] '.' [filename] ':' [line number].

The timestamps allow us to compute the time taken to process each event. In case of method calls, the event contains the values of the primitive-typed parameters (i.e. non class-typed parameters). This is used to locate, in the trace, the entry of some input parameters by the user. In case of events representing conditional statement this element represents the logical expression of the statement together with the exact location of the statement in the code. This is required since the same control statement with the same logical expression may be declared several times in the same class, to the contrary of method signatures that must be unique.

2.4 Trace model

The model of the stored format of the trace information in the database is presented in Figure 1. An ExecutionTrace instance represents a specific trace associated to a given scenario belonging to some use-case. It is composed of a collection of TraceEvents that could either be method calls or conditional statements. But the latter also contain an attribute representing their physical location (full file name and line number in the file). With this unique identifier, we are able to unambiguously identify the similar execution subtrees. As can be seen from the model, each node holds its thread number. In the database, we record a separate call tree for each of the threads.

2.5 Instrumenting conditional statements

We will now explain how *if* and loop statements are instrumented. Here is the simplest example of the location of the tracing statements in an *if* statement:

```
if(expression) {
    //THEN block
    tracingObject.traceStatement("if_true");
    …
    }
Else {
    //ELSE block
    tracingObject.traceStatement("if_false");
    …
    }
next_statement;
```

In this situation, a trace event will be generated and recorded whatever the value of the conditional expression. Therefore we will know what alternative has been executed and if there is yet another one to be executed. But the difficulty comes from the processing of the *if* statements when there is no alternative path (*then* without *else*). In this case if the condition is *false*, there is no conditional code to execute. Consequently we could not record the execution of this conditional statement since the tracing code is located in the conditional blocks. Below is an example of the instrumented code that follows the technique presented above. If *expression1* is *false*, there is no execution of the conditional block and therefore no record of any tracing event in the trace file.

```
if(expression1) {
    //THEN block
    tracingObject.traceStatement("if_true"….);
    …
    }
next_statement;
```

But we must know that a conditional flow remains to be executed for the scenario (the *then* part in the example above).

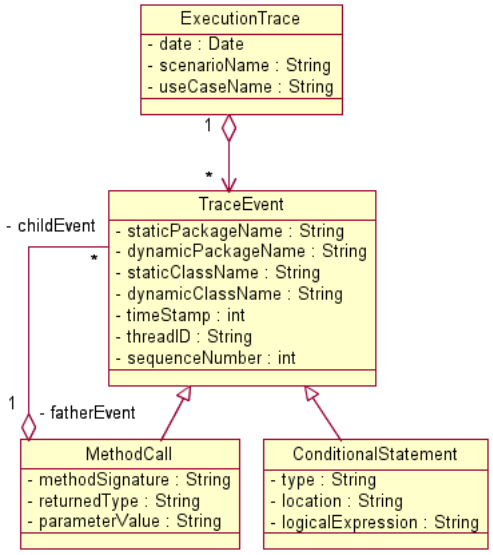

Figure 1

The solution is to *add* an *else* block during instrumentation, for the only purpose to generate an event in the trace file. This is showed below. Moreover, one should also record that in case the expression is *true* there is no alternative code to execute. Example:

Before instrumentation:

```
if(expression) {
    //THEN block
    …
    }
next_statement;
```

After instrumentation:

```
if(expression) {
    //THEN block
    tracingObject.traceStatement("if_true_no_else");
    …
    }
Else {
    //Added ELSE block
    tracingObject.traceStatement("if_false");
    }
next_statement;
```

In case the expression is *true*, the recorded event is IF_TRUE_NO_ELSE. Then we know that no alternative code remains to be executed. But is the expression is *false*, we will record it and know that an alternative path does exist. In the same

vein, we generate two events in the case of *while* and *for* loop statements. One in the case the loop is executed at least once, and another one if the loop was not executed at all. Here are the events for the *while* statements:

1. 'WHILE_TRUE' : this is recorded if the loop code was executed at least once
2. 'WHILE_FALSE' this is generated if the loop was not executed at all.

The same is done in the case of *for* loops. Finally, since *do-while* loop statements are always executed at least once, there is never any alternative to generate. But we record the corresponding event for completeness purpose only (to have a complete picture of what was executed).

3. DECISION TREE MODELS

3.1 Introduction

Our technique aims at recovering the alternative execution paths from a *backbone* use-case's scenario. These alternative paths represent variants from the backbone hence the alternative flow of the corresponding use case. But for an alternative path to be recorded as a plausible alternative, its execution must be controllable by the user. In other words, the value of the corresponding logical expression must be changeable through some user interface control. Therefore, our approach works in two steps:

1) Identify all the alternative paths to the backbone execution.
2) Determine if each path could be selected by the user through the GUI.

3.2 Definitions

In order to simplify the reading of the formal structure of the decision tree we built, we introduce the following definitions:

DN: Decision node. This represents an executed control flow statement.

PDN : **Partially** executed decision node. This is a DN for which an alternative to the current execution flow remains to be executed (in other words, there remain alternative paths to be run).

FDN: **Fully** executed decision node. This is a DN for which all the alternatives have been executed. In other words, there is no new path of execution that could be generated from this node.

DDT: Dynamic Decision Tree. This represents all the DNs in the executions of a scenario, recorded with their hierarchical dependence.

RDDT: A *Reduced* Dynamic Decision Tree. This is a DDT where all the nodes are PDN (i.e. all FDN have been removed).

3.1 Dynamic Decision Tree (DDT)

From the analysis of the trace we must build a decision tree to represents the decision statements involved in one or several execution of the program when running variants of the scenarios belonging to the same use-case. A DDT is then constructed to represent the hierarchical dependence between the DNs, since some control nodes may be dominated by other control nodes (i.e. when we change the boolean value of a DN, we may not be able to reach the DN located below it in the program). A DDT is generated for each of the separated threads in the trace (Figure 2). The DDT we generate contains different types of DNs: *if, while, for, do while*.

3.2 Reduced decision tree

The generation of a DDT produces huge trees with hundreds of DNs. These "raw" trees are much too big for our technique to work: we could not analyze thousands of decision nodes to find the one that could be influenced by some user action. In fact, what we are interested in are the nodes for which there still remains an alternative path to be executed. Therefore, our analysis algorithm will explore the tree and record all these nodes, removing the ones for which no alternative remains to be executed (FDNs). Then we rebuild a tree containing only PDNs. The resulting tree is what we call the **reduced dynamic decision tree** (RDDT).

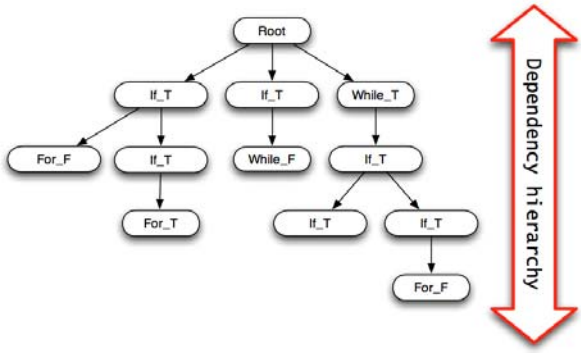

Figure 2

When the program is run according to the backbone scenario, the corresponding DDT is generated. To know what alternative paths have yet to be executed to complete the scenarios of a given use-case, we must combine the DDT of all the executed scenarios as a single RDDT to identify the remaining PDNs. The RDDT is therefore built incrementally from the DDTs of each execution of the system. After having chosen an alternative user action from the analysis of the PDNs in the RDDT (representing a variant of the scenario), the system is run again and the variant of the scenario is played. This leads to a new trace and a new DDT that is merged with the previous RDDT and so on. These scenarios represent the synthetic variants of the backbone scenario for the use-case as illustrated in Figure 3.

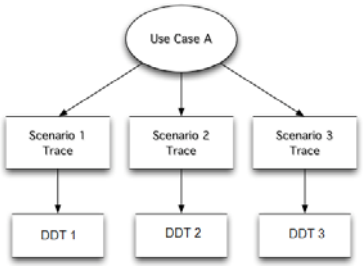

Figure 3

The principle of the construction of the RDDT is summarized in Figure 4. Here are the steps:

1) The RDDT is initialized.
2) An initial (*backbone*) scenario is executed.

3) The execution trace is captured by running the instrumented code according to the scenario and the DDT is generated from this trace.
4) This DDT is merged with the RDDT for the use-case: all the PDNs of this DDT are added to the RDDT and the similar PDNs are compared. If the two alternative execution paths of the PDN have been executed, this node becomes an FDN and is not a candidate for alternative execution anymore. It is then removed from the RDDT.
5) The resulting RDDT is analyzed to find the remaining PDNs. If there is at least one, we check if its condition could be changed by the user through some GUI action. If yes, a new scenario is built by adding this action to the previous scenario and the process go back to (3).

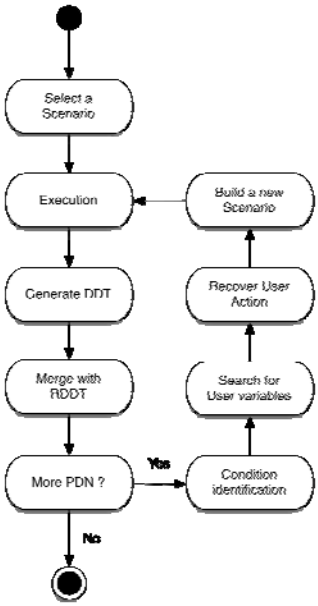

Figure 4

Technically, when a PDN is added to the RDDT we must also add all its parents. The parents of a PDN are all the decision nodes we need to travel to reach the root node from the PDN (i.e. all DNs which dominate the PDN). Indeed the execution of a given DN depends on the "state" of its parents' conditional statements. Moreover, the adding of the whole ancestor path up to the root for a PDN assures that we later compare (and compress) similar paths in the execution flow. The technique to identify the user action associated to some conditional statement works the following way. Starting from the conditional statement (the slicing criteria) we compute the associated backward slice [19][14]. If a GUI-related variable is present in the slice, this means that the value of the conditional expression could be changed by modifying the state of the variable.

Figure 5 present the initial RDDT when a new node (the central one just below the root) is added. It is worth noting that all the DNs are prefixed with their physical location in the source code so that we could easily find all the similar nodes. For example, in Figure 5, we represented three nodes with the prefix: "1" meaning that they all represent the same node in the source code. Two of them have been evaluated to *true* and the one in the middle to *false* which makes that decision node an FDN (all paths have been executed). Then this node is removed from the RDDT. This is presented in Figure 6.

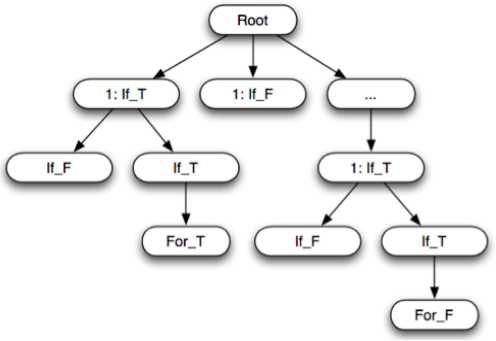

Figure 5

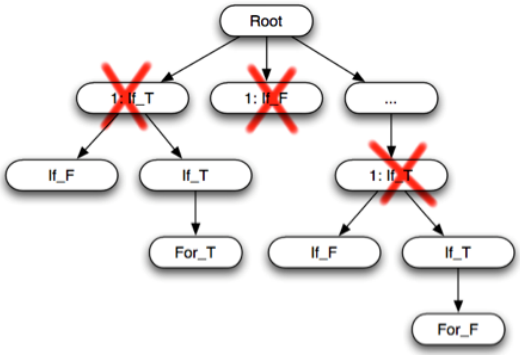

Figure 6

But if the node in the middle had been evaluated to *true*, there would still remain a path to execute (i.e. corresponding to the *false* case). Then the node would have been kept in the RDDT. Figure 7 we present the reorganization of the RDDT where the FDN is removed and the pointers from the parents redirected to the children of the removed node.

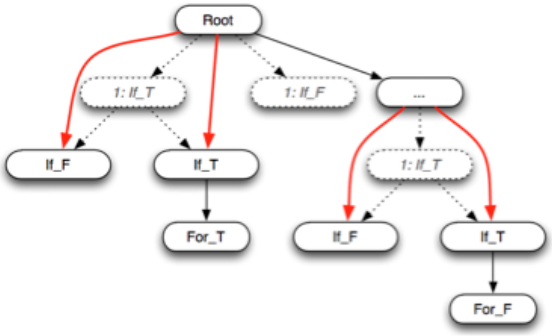

Figure 7

4. CODE INSTRUMENTOR

The tracing statements are inserted in the source code by a tool called the code instrumentor. We developed our own based on the analysis of the AST of the Java projects using a specific Eclipse library (org.eclipse.jdt.core.dom). First, the Abstract Syntax Tree (AST) of the Java source code of the system to analyze is generated one file at a time using this Eclipse library. Each file

represents one Eclipse project of the system. Then this AST is navigated and the tracing statements are added to each method entry and exit and in conditional statement as explained above. Once the AST of the project has been processed, the source code of the project is regenerated from the modified AST to get its instrumented version.

When the instrumented system is run, the tracing events are written to a flat file. Next, this file is read offline and the events are loaded in a database for further processing. This technique limits the impact of the tracing statements to the processing speed of the instrumented code. If we would have written the events directly to a database the processing speed would have been much more impacted. Currently our code instrumentor only works for Java. As an alternative to the building of our own code instrumentor, we explored Aspect Oriented Programming. However since the source code is instrumented at conditional statement level, AOP does not work. Indeed AOP does not allow to alter the behavior of the code at statement level, but only at method level. AOP can trigger the execution of additional code (advice) before or after a method call but not at a specific statement in a method. This explains why we dismissed AOP in favor of the implementation of our instrumentor.

5. FEASIBILITY STUDY

The question that remains to be answered concerns the workability of our approach. Due to the combinatorial explosion of the number of execution paths of a program, could our approach lead to a manageable set of alternatives to explore or would this set be so large that such an exploration would be prohibitive? The short answer is that our Reduced Dynamic Decision Tree (RDDT) technique leads to a workable solution.

As an illustration, we present a simple example based on the analysis of "FastUML", a small open source UML modeling tool. In this experiment we recorded 3 different scenarios, representing simple variants of each other. After having applied the tool on those three traces we obtain the following results:

- Trace 1 contained 116 007 events. Its DDT contained 45 544 Decision Nodes.
- Trace 2 contained 312 495 events. Its DDT contained 121 317 DNs.
- Trace 3 contained 104 958 events. Its DDT contained 32 849 DNs.

First, we observe that the Decision Nodes account for approximately 30% of the events in the trace. This is about what we got in all our experiments. Once we applied our decision tree reduction algorithm, the resulting RDDT from the merge of the three DDT contained only 74 PDNs. In short, with our algorithm we went from a maximum of 121 317 DNs in the trace to a final 74 PDNs. This limited number of PDNs shows that it is indeed feasible to analyze the control flow statements to identify the user manipulations leading to alternatives scenarios for a use-case.

6. RELATED WORK

Decisions to decisions graphs of programs have been used for quite some time in different contexts. In his early work, Paige reviews different approaches to partition program graph [17]. One of the techniques is decision-to-decision path that he uses to partition a program graph and propose a reduced version of the program graph only made on decision nodes. In their paper, Geoghegan and Avresky focus on adding fault detection to software. To achieve that goal they use dd-graph (decision to decision graph) to construct an execution path tree, which is used to predict normal flow of execution. Then Geoghegan and Avresky check that the current execution location follows the predicted flow of execution [10]. In a decade later Costa and Monteiro analyzed the execution of embedded software and compute observability-based statement coverage metric [3]. The particularity of this metric is that it allows knowing the statements that influence the output of the system. They compute the metric using Control Dataflow Graphs (CDFG) representing the system behavior. The result are the input data that would trigger the change in the execution flow if inputted during execution. As far as the execution trace format is concerned, Hamou-Lhadj proposed several years ago an exchange format for the traces that he called the "Compact Trace Format" [11]. Later, he proposed a metamodel for execution traces [12]. These format and metamodel corresponds to a "standard" approach, i.e. a trace in which the events are only method (or procedure) calls. The "controlNode" element present in the metamodel is not the representation of a control statement but a special construct aimed at signaling repetitions and sequence of similar sets of events in the trace. Indeed, this format is intended to represent a trace in a compressed format. In particular, each of the unique subtrees of events in the trace is represented only once. But the control statements are definitely absent from this format. Finally, to the best of our knowledge, we have not found any approach trying to recover the use-cases of a system using dynamic analysis techniques coupled with static analysis.

7. CONCLUSION

This paper presents a technique to recover the functional specification (use-cases) of a legacy program when no documentation is available (which is the common case). Our approach is based on a novel technique that uses an example scenario (that we called the *backbone*) for each use-case to recover. The latter then drives the discovery of the alternative flows for the use-case. The technique is to collect the control statement executed when running the scenarios and to identify the user manipulations that could change their state i.e. the flow of execution. The first challenge we encountered was to identify all the control flow statement involved in the scenario and what alternative was actually executed. This was very difficult to do using a standard execution trace format. We then decided to design a new trace format and we built a code instrumentor to be able to generate a trace in the new format. A second challenge was to cope with the large number of conditional statement to analyze in a single trace. Then we developed an approach where all the dynamic decision trees corresponding to the alternative scenarios to the same use-case are merged to get a reduced dynamic decision tree (RDDT). The principles for the building of such a tree have been presented in the paper. This technique has proven to be efficient in reducing the decision tree to a size compatible with our approach. The next step in the method is to take each of the remaining nodes and check if their condition can be changed by some user manipulation through the screens of the application. To go from a PDN to the screen code we use a code slicer that we presented briefly. The full explanation of the slicing technique is out of the scope of this paper. The contributions of our paper are:

1. The presentation of a novel technique to recover the functional specification of a legacy system (use cases) using dynamic and static techniques.
2. The design of an enhanced execution trace format to be able to construct the Dynamic Decision Tree of some scenario execution.
3. The design of a decision tree compression algorithm to reduce the size of the decision tree to analyze.

Future work is to tune the slicing technique to be able to cope with large GUI libraries. Indeed our approach has been tested so far on limited systems only. We must now tune our techniques to show that it is a feasible approach for large systems.

8. ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of the UAS of Western Switzerland, grant N° 24245

9. REFERENCES

[1] Ball T. The Concept of Dynamic Analysis. *Proc. 7th European Software Engineering Conference* (ESEC'99) 1999

[2] Bass L., Clements P., Kazman R. *Software Architecture in Practice*, 2nd edition. Adison-Wesley Inc.. 2003

[3] Costa J., Monteiro J. - Computation of the minimal set of paths for observability-based statement coverage. 15th IEEE Int. Conf. on Mixed Design of Integrated Circuits and Systems (MIXDES). 2008

[4] Di Lucca G. A., Fasolino A. R., De Carlini U. Recovering Use Case models from Object-Oriented Code : a Thread-based Approach. *Proc IEEE WCRE* 2000

[5] Dugerdil Ph., Jossi S. A Legacy System's Use-Cases Recovery Method. *Proc ICSOFT, Athens, Greece.* 2010

[6] Dugerdil Ph., Jossi S. Empirical Assessment of Execution Trace Segmentation in Reverse Engineering. *Proc ICSOFT* 2008

[7] Dugerdil Ph., Jossi S. Reverse-Engineering of an Industrial Software Using The Unified Process: An Experiment. *Proc IASTED SEA.* 2007

[8] Dugerdil Ph. - A Reengineering Process based on the Unified Process. Proc. IEEE International Conference on Software Maintenance, 2006.

[9] El-Ramly M., Stroulia E., Sorenson P.. Mining System-User Interaction Traces for Use Case Models. *Proc IEEE IWPC.* 2002

[10] Geoghegan S. J., Avresky D. R. - Method for Designing and Placing Check Sets Based on Control Flow Analysis of Programs. Proc. of the 7th IEEE Int. Symp. on Software Reliability Engineering, 1996.

[11] Hamou-Lhadj A.. Techniques to Simplify the Analysis of Execution Traces for Program Comprehension. PhD Thesis. Ottawa-Carleton Institute for Computer Science, University of Ottawa. 2005

[12] Hamou-Lhadj A.,Lethbridge T.. A Metamodel for the Compact but Lossless Exchange of Execution Traces. Journal of Software and Systems Modeling (SoSym), Springer, pp. 1-22, 2012.

[13] Jacobson I., Booch G., Rumbaugh J.. *The Unified Software Development Process*. Addison-Wesley Professional. 1999

[14] JavaSlicer official website : http://www.st.cs.uni-saarland.de/javaslicer/

[15] Leffingwell D, Widrig D. *Managing software requirements*, Addison Wesley. 2003

[16] Li Q., Hu S., Chen P., Wu L., Chen W.. Discovering and Mining Use Case Model in Reverse Engineering, *Proc. IEEE FSKD*. 2007

[17] Paige M.R. - On Partitioning Program Graph. IEEE Transactions On Software Engineering, Vol. 3, No. 6. 1977

[18] Qin T., Zhang L., Zhou Z., Hao D., Sun J.. Discovering Use Cases from Source Code using the Branch-Reserving Call Graph. *Proc. IEEE APSEC.* 2003

[19] Mark Weiser - Program Slicing. Proc. IEEE International Conference on Software Engineering (ICSE) 1981