Flumine: A Change Impact Analysis Tool For Typescript

Nicolas Larranaga Cifuentes[1] and Andres Mauricio Rondon Patino[2]

***Abstract*— Typescript was created as a superset of JavaScript that allows the creation of large projects using class based Object Oriented Programming, this however creates a scenario where refactoring and restructuring code in later development stages becomes an arduous task, as the source code becomes more convoluted an implicit dependency graph is created between the functions, classes and even Lines of code. Flumine is proposed as a Tool to ease the process of modifying code in a project, it operates as an implementation of the EAT algorithm by making use of hooks through the Compiler of Typescript itself, as an alternative of more classical algorithms like PathImpact and CoverageImpact, EAT offers a high accuracy rate at a low memory and time cost, this proved to be specially beneficial to large projects.**

***Index Terms*— Impact, Analysis, Software, Typescript, Trace, Dependency, Control Flow Graph, Execute after sequences.**

I. Introduction

A software system is a very volatile construct because it requires to be adapted to new environments or necessities very often. Indeed, due to the high complexity nature of software, cases when a very simple change cause the system to behave in unpredictable ways are often seen [1]. That is the reason why software developers often make changes in a system and then spend considerable amount of time of getting the system back to a consistent state. Thus, a change in a software system may have hidden side effects that are difficult to see even for experienced programmers [2].

The type of analysis that a software developer does when trying to make a change is similar to executing the program in his head and the trying to see test cases on which his change would fail. The problem with this approach is that it requires a very experienced person in the system or a tremendous amount of time to test at least an acceptable amount of functionality. Although, by using a discipline like Test driven development[2], we can check to see if a given change affected unexpected parts of the system, the information that this give us is that a test is failing, the developer then would need to investigate why is the test failing [3].

Given the previous considerations, an automatic way of detecting change impact has been subject of study by several authors on computer science and software engineering[4]. What follows are the definitions of this mechanisms and advances that some researches have made in the area. Followed by our approach, design decisions and trade offs of implementing an automatic impact analysis tool for a the Typescript Language.

A. Change Impact Analysis

Change impact analysis (CIA) is the process of identifying the set of impacted procedures of a computer program given a change must be made in another method. Specifically talking about Object-oriented programming languages, its evident that the extensive use of sub-typing and dynamic dispatch make understanding the flow of values and procedures a nontrivial task[5]. Even though recently functional programming has been increasing in popularity, OOP is still the main paradigm being used in software development, moreover theres a recent wave of programming languages which uses transpilers to pass from OOP code to functional programming, and other paradigms, examples of these are Typescript and Coffeescript [6].

Maintenance is the most time-money consuming stage of any development cycle [7], starting from this premise the need of a tool that improves the work-flow inside the migration and maintenance stage becomes evident. Many solutions have been implemented to specific languages such as Java [Chianti Reference], these tools are based on Path impact and execution tracing algorithms [8].

B. Theory and Definitions

We define a software change as a change of its source code except otherwise noted. It is true that configuration files do change the behavior of a system but almost all the methods discussed in this work are focused on the source code.

There are different properties that a Change Impact Analysis algorithm may have. The following is a list that the referenced authors have identified to be pertinent.

- *Static and Dynamic*: A CIA algorithm is said to be static if it uses the source code and perform static analysis on it and it does not need runtime information. On the other hand, a CIA algorithm is said to dynamic if it uses information that is only available at run time, such as the execution trace of a program given a set of executions.
- *Safe and Unsafe*: A CIA algorithm is said to be safe it finds all the places that are affected by a software change. That is to say, all of the lines of code that would

[1] Systems and Computing undergrad, Universidad Nacional De Colombia, Bogota, Colombia nlarranagac@unal.edu.co

[2] Systems and Computing undergrad, Universidad Nacional De Colombia, Bogota, Colombia amrondonp@unal.edu.co

possibly need to be changed given the proposed initial change. In contrast, an unsafe CIA algorithm is one that gives a subset of the places affected. At first glance, one can imagine that unsafe algorithms are not useful. However, we will see why unsafe algorithms are being researched and their some of their applications.

- *Dependency based*: A CIA algorithm is said to be dependency based if its procedure makes inferences with the dependency structure of the source code. For example, if the software change consists on changing a class name, then all the components that depend on that class will be affected. This property is the most common method on the CIA algorithms among the referenced articles.
- *Traceability based*: A CIA algorithm is said to be based on traceability if it does not compute syntactic or semantic dependencies and rather makes data-mining and associations on files that are often changed together, rising the level of abstraction and letting other type of files apart from than source code files to be considered.

Some of the authors use software instrumentation which is defined as the ability to monitor software runtime information such as the call traces in a given execution or the values of the variables on each step of the program. Debuggers implement this functionality to allow step by step execution and monitoring.

C. Dynamic Analysis

There are two main known algorithms for dynamic analysis, **PathImpact** and **CoverageImpact**, both of them will be discussed in the following sections, however we opted to implement a third algorithm called **EAT** [9] as it provided more flexibility in time and memory constraints while maintaining a high accuracy rate, this will be explained later in section II.

The dynamic analysis algorithms operate over two important data structures; a call graph (fig 1) and an execution trace associated to a program run (fig 2). In this case the hypothetical run's execution trace indicates that method M was called which then invoked method A twice, then it invoked method B who called itself method C and then returned, finally method M calls again B who finalizes the run.

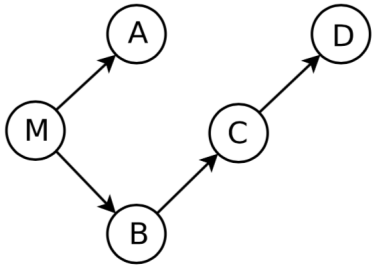

Fig. 1. Call Graph example

$$\boxed{\mathtt{M}_e\ \mathtt{A}_e\ \mathtt{A}_r\ \mathtt{A}_e\ \mathtt{A}_r\ \mathtt{B}_e\ \mathtt{C}_e\ \mathtt{C}_r\ \mathtt{B}_r\ \mathtt{B}_e}$$

Fig. 2. Execution trace for a hypothetical run

In the execution trace let's denote X_e as the event where method X is entered into, and X_r as the event where method X returns.

1) PathImpact: PathImpact works at the method level and uses compressed execution traces to compute impact sets. Given a set of changes, PathImpact performs a series of forward and backwards walks over the execution trace of a given program, the objective of the forward walk is to determine all the methods that were called after the changed method, whereas the backwards traverse identifies methods into which the execution can return [9]. From these walks the set is created using the following rules.

- In a forward walk, the algorithm starts from the immediate successor of X_e, and includes every method called after X in the impact set while counting the number of unmatched returns (i.e. a return that do not have a corresponding method entry in the trace segment examined).
- In a backward walk, PathImpact starts from the immediate predecessor of X_e and for every unmatched return counted in the forward walk it includes a unmatched method (i.e. a method that does not have an associated return in the segment of the trace examined).
- Finally X is added to the impact set.

This algorithm provides a good precision on the impact set created, avoiding adding methods that should not belong to it, however the time and memory that costs the computation of an execution trace makes this approach unfeasible as it can consume over 2 GB of memory to analyze a relatively small project [9].

2) CoverageImpact: This algorithm also works at the method level, however it does not use the execution trace to compute the impact sets. The information gathered by the algorithm is stored in a bitset that contains one bit per method in the program. If a method is executed in the specified run, its corresponding bit is turned on. As an example the coverage bitset associated to the execution trace in figure 2 can be found in figure 3.

M	A	B	C	D
1	1	1	1	0

Fig. 3. Coverage bitset associated to figure 2

CoverageImpact computes the impact sets in two steps.

1) Using the coverage information, it identifies the executions that traverse at least one method in the change set C and marks the methods covered by such executions.

2) Compute a static forward slice from each change in C considering only marked methods.
3) The impact set is the set of methods resulting in the computed slices.

Opposite to PathImpact, CoverageImpact implies no overhead in either memory nor time as it runtime and space is linear ($O(n)$) in the number of methods of the program, however this comes at a cost and it's precision; the way

II. Flumine

We have described two dynamic algorithms in section I-C however both of these result impractical in large scale projects, **PathImpact** provides the problem of resource starvation and time complexity overhead, although it provides good accuracy in the resulting impact set it is simply not possible to use it in a project with 500+ lines of code. On the other hand **CoverageImpact** provides a low complexity in both time and memory, however it does not offer the same accuracy in the resulting impact set. As an alternative to these problems we decided to use a third way, following the details specified in [9] to implement an algorithm called Execution After Sequences, we could create an Execution after sequences Tool (EAT).

A. Execution After Sequences Analysis

EAT is born from the realization that the main dynamic CIA algorithms compute redundant information to obtain a basic correspondence called the Execute-after Relation which is defined as following.

1) Execute-after Relation: Given a program P, a set of executions E, and two methods X and Y in P, $(X, Y) \in EA$ for E if and only if, at least one execution in E

1) Y calls X directly or transitively.
2) Y returns into X directly or transitively.
3) Y returns into a method Z and a method Z later calls X.

This relation can be computed directly without the use of a bitset as in **CoverageImpact** or a full execution trace as in **PathImpact**.

Lets redefine the execution trace in the following way; instead of mark a method return event, register the method returned into events, this means that if a method X calls a function Y and then this function returns into X the execution trace will be written as X_e, Y_e, X_i, as an example the figure 4 shows the modified version of the trace showed in the figure 2.

$$\boxed{\mathtt{M}_e\ \mathtt{A}_e\ \mathtt{M}_i\ \mathtt{A}_e\ \mathtt{M}_i\ \mathtt{B}_e\ \mathtt{C}_e\ \mathtt{B}_i\ \mathtt{M}_i\ \mathtt{B}_e}$$

Fig. 4. Modified execution trace with EA

This traces provides the information about the EA relationships in an obvious way, however it can be further improved to eliminate redundant information starting from the fact that if the first event for a method Y_f occurs before the last event of another method X_l then we can affirm that $(X, Y) \in EA$. From this we can reduce our event array to only the first and last events for every method reducing our space to $2n = O(n)$ where n is the number of methods in the program, see the reduction from figure 4 to 5 where each event can be expressed in terms of first and last as in figure 6.

$$\boxed{\mathtt{M}_e\ \mathtt{A}_e\ \mathtt{A}_e\ \mathtt{B}_e\ \mathtt{C}_e\ \mathtt{M}_i\ \mathtt{B}_e}$$

Fig. 5. reduced EA trace

$$\boxed{\mathtt{M}_f\ \mathtt{A}_f\ \mathtt{A}_l\ \mathtt{B}_f\ \mathtt{C}_f\ \mathtt{C}_l\ \mathtt{M}_l\ \mathtt{B}_l}$$

Fig. 6. execution trace using first and last events

2) Execute-after generation Algorithm: In terms of implementation two arrays (or Hash maps) are used to store the initial and final events for every method, more specifically the time where they occur, these arrays update in the following way.

Let C be the counter that specifies the time when each method is called and increments by 1 on every event, F and L be the data structures used to store the first and last events for each method, the proposed algorithm executes these steps.

1) Declare L and F as arrays of size n where n is the number of methods in the program
2) Fill the arrays with λ as no event has occurred yet.
3) For every entry on a method M if $F[M] = \lambda$ set $F[M] = C$ as we would have found a new method.
4) set $L[M] = C$ as we have found a new last event for the method M.
5) On a return into a method M set $L[M] = C$ as we have found a new last event for M.
6) On every step increase C by 1.

This algorithm generates the information needed to create an impact set without causing an overhead in memory or time ($O(n)$ for both) and with a high rate of accuracy. Now the only task left is given an changed method iterate through the arrays of events and add to the set those whose first event time is lesser than the last event time of the specified method.

B. Typescript Compiler

Flumine implements the algorithm described in II-A.2 by making use of the Typescript compiler through the APIs it provides to embed hooks that allow external functions to

update the F and L arrays as well as the counter C.

As this is a dynamic analysis tool it requires a piece of code to be compiled an run to work over it, so we chose to use the set of tests defined inside the projects to perform the analysis, this based on the assumption that a large project counts with a suite of tests that armours the code base, specially as the use of practices as TDD have increased in popularity in recent years.

Talking about the specifics of the Typescript superset, there are some considerations that must be taken, as there is not only one way of returning into a method from another one, this can happen in three different scenarios.

- Return from a call of a function.
- Return from a callback.
- A Catched exception from an inner function.

We have successfully implemented the first two cases, however the last one is still unsupported. This is because of the way we introduce the aforementioned hooks into the code, each *methodEntered* hook is inserted before the first statement in each method, and the *returnedIntoMethod* hook is introduced after the call of a function and the declaration of a callback.

Flumine was conceived as an open source project its source code can be found in `https://github.com/larranaga/flumine`.

III. Outlook and Conclusions

Although it may be assumed that a tool which operates over a set of tests on a project may limit it's usefulness, most Typescript projects count with a well defined set of tests, as these schemes tend to increase rapidly in code volume, most developers shield their source code with unit tests.

OOP paradigm tends to enlarge projects rapidly, so the need of a tool such as Flumine is evident on a relatively new Programming language as Typescript whose objective is focused on the support of OOP for JavaScript which is a language characterized for it's Functional programming alignment.

Flumine is still a work in progress, it's core functionality can still be extended, other approaches are still an open area to apply in the analysis methodologies, a static analysis alternative as suggested in [10] and [11] may offer more flexibility for those projects which do not have a suit of tests ready to use dynamic approaches. On the other hand there are still cases where the EAT implementation must be polished, for starters the handle of exceptions returning into another method, the implementation of event hooks for arrow functions and higher order functions and finally the case of IIFEs.

References

[1] M. Usman, E. Mendes, F. Weidt, and R. Britto, "Effort estimation in agile software development," in *Proceedings of the 10th International Conference on Predictive Models in Software Engineering - PROMISE '14*, (New York, New York, USA), pp. 82–91, ACM Press, 2014.

[2] Robert C. Martin, "Clean code," 2009.

[3] B. Tanveer, L. Guzmán, and U. M. Engel, "Effort estimation in agile software development: Case study and improvement framework," *Journal of Software: Evolution and Process*, vol. 29, p. e1862, nov 2017.

[4] B. Tanveer, A. M. Vollmer, and U. M. Engel, "Utilizing Change Impact Analysis for Effort Estimation in Agile Development," in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 430–434, IEEE, aug 2017.

[5] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "JRipples: A Tool for Program Comprehension during Incremental Change," in *13th International Workshop on Program Comprehension (IWPC'05)*, pp. 149–152, IEEE.

[6] B. Meyer, "Object-Oriented Software Construction SECOND EDITION,"

[7] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, pp. 613–646, dec 2013.

[8] J. Law, G. R. S. R. Engineering, undefined 2003, and undefined 2003, "Incremental dynamic impact analysis for evolving software systems," *ieeexplore.ieee.org*.

[9] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proceedings of the 27th international conference on Software engineering*, pp. 432–441, ACM, 2005.

[10] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, p. 9 pp., IEEE, 2005.

[11] S. A. Bohner and R. S. Arnold, *Software change impact analysis*. IEEE Computer Society Press, 1996.