**Automated Error Diagnosis Tool Clafer Compiler**

ECE750-T24

Matthew Ma, Jesus Jesus Alejandro Padilla

Department of Electrical and Computer Engineering, School of Computer Science

University of Waterloo

Waterloo, Ontario

# **Abstract**

Software bugs have caused tremendous issues in software reliability and system availabilities. Compilers like all other software, are also be prone to bugs, logical errors which in some case are more critical to performance and accuracy of the generated results.

In this paper, we present a novel compiler bug diagnosis tool that can assist the programmer to localize the error at each stage of the compiler. We apply the principle of regression testing to compare against two versions of the compiler, and determine the changes from different stage of the compiler generation represented by intermediate representations. We apply efficient algorithm to compare the IR, and filter out the legal changes from rule set which the user can specify through user interface or as a template.

We experimented our tool with Clafer compiler from the Generative Software Development Lab at the University of Waterloo, and the results show that our tool can determine most of the compiler bugs that are confined within Clafer language with acceptable performance overhead, memory usage and user experience,

Keywords: software bug; compiler; IR; regression testing; Clafer; XPath; encapsulation

1. **INTRODUCTION**

**1.1 Motivation**

Software reliability is of key concern in both industry and academia research. Although program analysis in detecting and fixing software bugs have been studied widely for the past decades, software bugs still exists, and continue to cause problems in data integrity, result accuracy and system availability to name a few.

Compilers like all other software, would also be prone to bugs, logical errors which in some case are more critical to the accuracy of the generated results.

In Microsoft Visual C++ Team Blog, hundreds of bugs are listed for well developed IDE, Visual Studio 2012[1]. The number is overwhelming considering the fact that Visual Studio has been out on market for years. Furthermore, it is clearly stated that this number are far from exhaustive; they don't contain private Connect bugs or non-Connect bugs (which come from many sources: compiler testers, STL maintainers with last names ending in J, other MS teams like Windows and Office, external customers with support contracts, etc.).  Also, some public Connect bugs aren't represented, in particular, /analyze and /ZW bugs. Thus it is not hard to imagine the scale and resources a commercial company spends in diagnosing, testing and fixing the compiler bugs.

With growing interests in solving compiler related bugs, we decided to study further to develop automated error diagnosis tool for open source compiler. For time constraints and limited resources, it is decided to focus this study on Clafer blabla compiler from the GSD Lab at the University of Waterloo.

**1.2 BACKGROUND**

Clafer is a lightweight modeling language developed at Waterloo’s GSD Lab. This language provides a simple syntax that is easy to learn and that allows the generation of models that can later be verified for consistency. It can be used in many scenarios. Currently it’s being applied to software product lines, concept modeling, and multi-objective optimization. Although Clafer is not yet widely adopted, some researchers have used it in academic teaching and various research projects. Due to its open source initiatives, Clafer is available with no cost to the public, and this modeling language is at its continual development by GSD lab researchers. It is believed that as the language matures, it is going to be used more extensively in academia and industry.

The core of Clafer is the compiler which translates the user written model into other formats, such as Alloy, Desugared Clafer, XML and others for reasoning and processing. Once the model is in Alloy, Clafer Instance Generator can be used to generate either valid instances, that prove the model is correct, or no instances if there is a problem with the model. Clafer compiler is written in functional language, Haskell blabla.

The current methodology and toolsets used to test Clafer are not the best available; there is no testing framework for compiler related bugs, and the user could only manually go through the source code while relying on experience to determine the source of the error. During the compilation process, positive examples are generated and added to a repository to be used in the future references. Through user questionnaire, we determined that the typical tests applied are to compare if the output is exactly the same as the original positive example; the comparison is done using a simple text diff which is inefficient and would introduce inaccuracies as the compiler evolves. Moreover, a simple update in the compiler might cause certain innocuous changes, for instance, different IDs, that will make all the comparison tests to start failing.

Furthermore, when a real bug within the compiler is detected, the current testing approach provides no summary, visual representation or any other form of help whatsoever to track the fault down. It is up to the developer to retrieve the useful information from all the differences identified and perform the troubleshooting to identify the root of the error.

It is obvious that to fix the bugs in Clafer compiler, it is required a more sophisticated, efficient and user friendly approach; moreover, this approach should be automated to accommodate the on-going growth of the scale of Clafer compiler. There are several problems, some of which are particular to Clafer, that need to be addressed to be able to provide a more comprehensive solution, and solving these problems is at the core of this research project.

1. **OUR CONTRIBUTIONS**

In this paper, we propose a novel error diagnosis tool for compiler which can help users to localize the error in different stages of the compiler of interests. More specifically, our tool includes the following major key ideas and contributions.

(1)By intercepting the Clafer compiler, we proposed a new approach to produce intermediate representation for each stage of the compilation process within Clafer through the API we create. By doing so, we could apply regression testing idea to find the differences at each stage of the compilation process. We realize that errors tend to propagate as differences in the format of IR so our tool always recommend the user to fix the potential bugs from the earliest stage as identified by the earliest “bad” changes.

(2)We modified existing XML difference comparison tool and output our IR in desired format to increase the efficiency in finding the IR differences from two versions of the compiler. To allow selection and filtering legal changes, we introduce the concept of rule set, where users can specify the legal changes that are allowed from the compiler version changes. This interactive approach not only gives users more flexibility but also reduces the time taken to track down to the compiler bugs as demonstrated in later sections.

(3)Last but not least, we apply XPath and XSLT techniques interfaced with Java so that our tool has extensive control on the change information within the IR. This is very novel approach as it means that we can further enhance out tool in the future not only for diagnosis purpose but also for fixing the error within the compiler.

1. **PROPOSED IDEAS AND OVERVIEW**

In higher hierarchical view, our approach is based on comparing two versions of the compilers to determine the bugs within one of them. We use the idea from regression testing, and we assume that the previous version of the compiler is often bug-free after certain time of debugging and maintenance. Although there is chances that the older version of the compiler has more bugs and our assumption might not be correct, by investigating Clafer compiler further and acquiring expert’s suggestion, it is determined that this assumption is reasonable to make for this specific case.

**3.1 IR Production**

Since one of the main goals is to reduce the user’s time spent in finding compiler bugs, we propose to give users more information about where does the changes between two compilers come from.

IR is the intermediate representation that a compiler generates during compilation process; it is encoded in XML for easier processing. We propose to generate IR at all the stages involved in Clafer’s compiling process so that it can narrow done the IR differences in each stage. By doing so, user can find directly the phase of the compiler code that implements this stage of compilation process. As discussed earlier, errors tend to propagate as differences in the format of IR so the user can pinpoint to fix the potential bugs from the earliest stage as identified by the earliest “bad” changes.

Since Clafer compiler is written in Haskell, our first task is to create an API that can ask Clafer compiler to generate the IR for our readily use.

**3.2 Difference Finder**

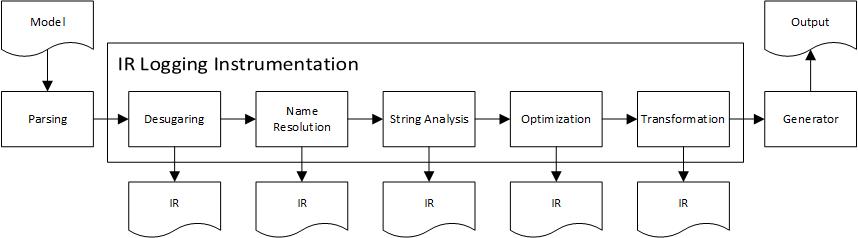
Once IR from each stage of the compilation process is present, we want to find the differences amongst them. Difference finding is of critical importance as this is one of the bottlenecks for performance issues. Since IRs can be big in size, efficient algorithm is required to compare against two IR documents and extract all the changes accurately. In our implementation which will be discussed in detail, we noticed the hierarchical structure in the IR, and we modified the IR structure accordingly to accommodate efficiently with the difference finding algorithm we are employing.

Figure 1: Stages where IR logging was implemented in Clafer.

It should be also noted that the IR found at this step should include all the differences amongst the two versions of the compilers.

**3.3 Extracting and Filtering Legal Changes**

As noted above, there are certain changes in the compiler that are perfectly acceptable. For example, when updating the compiler version, the ID number is changed accordingly. Such a change is reflected in the IR, and according to our investigation, these legal changes can be as much as 20%- 30% of the total changes.

Since overwhelming information interferes with user’s effort to determine the compiler errors, we wish to have the capability to operate on each difference found amongst the IRs so that we have the control. In this way, a user can explicitly specify the legal changes to be accepted by our tool, interactively eliminating redundant changes.

**3.4 User Interface**

At last, we want to have a visual representation for the findings by our tool so that the user can see the differences in the IR from different stage. The user should be also given ability to specify the changes and differences that are acceptable through the UI so that this information is transferred to our tool’s back-end so reflect such a decision dynamically.

We believe UI serves as effective visual aid that serves as an important presentation of our Clafer compiler error diagnosis tool. It helps the user determine the root error efficiently without becoming overwhelmed by unnecessary information.

1. **IMPLEMENTATION**

This section describes the implementation of our prototype, BugIsolator; its core is a program written in Java, which consists of roughly 1000 lines of commented code without considering external tools, like X-Diff. We also describe the changes we applied to Clafer compiler, which is written in Haskell, to be able to retrieve the required data.

**4.1 Compiler Logging API Instrumentation**

As was previously described, BugIsolator works by comparing the state of two versions of the compiler at analogous stages of operation. For Clafer, we found two potential ways of describing its state: the call graph and the compiler intermediate representation (IR). The state in a call graph is denoted by the function calls and the variables that are passed to them. This approach gives a very accurate representation, but it might return too many false positives; the tool might complain about some benign changes, like functions refactoring. The IR, on the other hand, only stores the result of transformations done to the input model during the compiler operation. Although it is not as verbose as the call graph, it only contains information about the model, so it is less likely to receive false positives because of refactoring. After careful consideration we opted for the latter approach.

To get the IR from the compiler we retrieve the representation at the end of each stage and store it in XML format. This has to be done for all the relevant stages and for all the compiler versions that want to be tested. This is obviously not the best approach, since it is very time consuming. In the future, though, we might simplify this step by instrumenting the logging code during runtime making the process completely automatic.

Figure 1 shows the points where we logged the IR in Clafer. For some of these stages, like name resolution, we considered some sub-stages. If we put all of them together, we instrumented the logging code in 12 points of the compiler. These stages cover most of the operation, but there are two important parts that are not considered: the parser and the output generator. The parsing process was not covered because it uses the abstract syntax tree (AST) structure instead of the IR, and because most of the implementation was done using automatic generation tools for Haskell, like Happy blabla and Alex blabla. The output generator was not considered either because it produces multiple types of results, like HTML and Alloy. To cover correctly this module we would have needed to translate each of the possible output formats into representations equivalent to the IR.

**4.2 XML Change Detection**

After getting the state, BugIsolator is ready to start working. The first step for each stage is to compare the IR of the two compiler versions. Since the IR is stored in XML format there are many ways in which the comparison can be done. The most natural way is using a simple algorithm like UNIX diff, compare each line and report any difference as an error. However, this is very naïve approach since there would be too many false positives caused by innocuous changes like spacing differences, and it would be very difficult to group and present these changes to the user in an understandable way. Another approach is using regular expressions (regex), but it would also be very hard to put them in practice. Regular expressions deal with plain text so a lot of contextual information could be potentially lost. Since the information is already in XML, the most sensible option seems to be an algorithm that leverages the existing structure of the information.

There are many algorithms used for change detection in XML trees. Peters [blabla] described that most of them differ by their memory consumption, time complexity, the way they report changes (update, insert, delete, and move) and if they consider the order of nodes as a change. We decided to use X-Diff [blabla] because, even though it is not the most efficient in terms of memory and time complexity (quadratic in both cases), it does not consider the order of nodes. This is particularly important since in Clafer IR the order is not important, and reporting these changes would result in false positives.

X-Diff works by comparing two xml files, in this case the IR, given as input and produces a third one called a delta. This file contains both, the part of the XML that was kept unchanged, as well as the differences, as shown in Fig. 2. We kept most of the original implementation intact except for some modifications done to the notation used in the delta file. Originally, X-Diff reports the changes as shown in Table 1 (a). The problem with this is that some standard languages such as XSLT and XPath do not recognize it. Hence, to solve the problem we changed the representation to the one shown in Table 1 (b).

Table 1 (a): Original X-Diff Notation

| Notation | |
| --- | --- |
| Operation | Example |
| Insert | <?INSERT b?> |
| Delete | <?DELETE b?> |
| Update | <?UPDATE FROM "9"?> |

Table 1 (b): New Notation

| Notation | |
| --- | --- |
| Operation | Example |
| Insert | <xdiff op='INSERT' type='attr' name='b' /> |
| Delete | <xdiff op='DELETE' type='node' name='b' /> |
| Update | <xdiff op='UPDATE' type='value' original='a' /> |

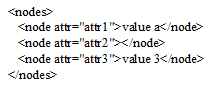


Figure 2 (a): Original XML

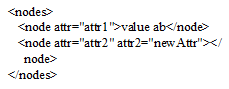


Figure 2 (b): Modified XML

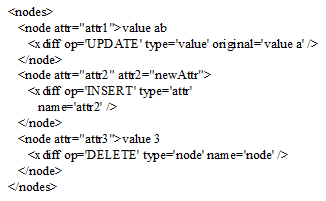


Figure 2 (c): Delta file produced by X-Diff

**4.3 Rules Engine**

As described earlier, X-Diff compares the provided IR files and reports all the differences in the XML structure that it can find in a delta file (there is one delta file for each compared stage of the compiler). Although some of these differences could represent errors, we cannot mark them as such yet; there can be some allowed changes that we need to identify and filter out to avoid false positives. To do so we created an engine that allows developers to specify a set of rules that BugIsolator will use to better analyze when a change is allowed and when it actually is an error.

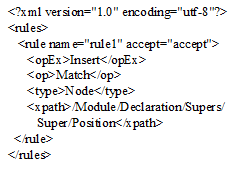


Figure 3: Rule example.

The rules in our engine are stored in an XML file and follow a format similar to the one shown in figure 3. Each rule has the following parts:

• **Name:** It identifies the rule.

• **Accept:** This attribute tells the system whether the change is allowed or not. If it is not then it represents an error that needs to be reported.

• **Type:** This is the part of an XML node that received the change. This type can have one of three values; attribute, which specifies that the attribute of a node was modified; node, used when the whole node was modified; or value, which specifies that only the value inside a node had a difference.

• **Operation Expected:** This is the type of change that the XML element suffered. It can be either update, insert, delete or any; this last one means that the rule accepts any type of change.

• **xPath:** This is the xPath expression that identifies the nodes the rule is expecting.

• **Operation:** This identifies the operation that the tool will execute when trying to determine if the node matches the rule or not. We only created one out of the box: match. This operation returns true only if the name matches the xPath expression. However, we created an interface that allows new operations to be created. The only method defined in the interface is analyze, which receives the nodes that match the xPath expression as a parameter. From these nodes the method can navigate to the rest of the XML document, so much more advanced operations can be performed, like comparing a value in the parent, or checking the number of children of the node among others.

After executing X-Diff, BugIsolator loads the rules and the delta file and starts executing the rules. As previously described, each rule has an xPath expression that identifies the nodes it expects, so the tool tries to find any node that was modified between versions that matches that expression. If any is found, it is passed as a parameter to the analyze method of the operation defined in the rule. This method determines whether or not the node matches the rule and if it should be accepted or not. All the accepted nodes are removed from the list of changes.

**4.4 Result Generation**

At this point all the differences that matched any rule are already filtered out, so only the unexpected changes are left; it is very likely that they represent errors, so it is a good moment to show the results to the user.

BugIsolator reports the results in a group of webpages. There is one that summarizes the results, and there is another one for each stage that suffered an unexpected change as shown in Fig. blabla. Each of these pages contain the differences in a table format as well as in an XML with the changes marked for easier identification. There is also a group of filters that can be used to further refine the presented information. In the future, these changes could be used to automatically generate rules for our engine.

The summary page, on the other hand shows the number of changes per stage in an ordered table. It is very likely that the first stage that had any difference is the place where the source fault was introduced. With this information the developer knows the stage where the bug was possibly introduced, how it first manifested and how it affected the following compiler stages. It is important to remark that the more fine-grained the stages are, the more accurate the results will be.

1. **EVALUATION**

In this section, we present a quantitative measurement of our prototype, BugIsolator. We analyze the effectiveness of our technique, the time to execute the tests, and the memory performance. All tests were run on a machine with a quad core Intel Core 2 Duo Extreme 2.53 GHz, 8 GB of ram

**5.2 Experiment Setup**

**5.3 Results and Analysis**

1. **RELATED WORK**

Although our approach is completely novel to our understanding, the technique is broad so there is a group of existing work that is somehow related. We discuss some of them in this section.

Traditionally, as shown by Kossatchev blabla, most of the compiler testing is done using black-box testing, like regressions. Our tool assumes that this type of testing has already been performed, and starts working once we know there is a problem in the system, but it does not perform any type of testing by itself. Hence, there is no overlapping work.

Bug isolation is an area that already has a lot of work. Program slicing, one of the most well-known bug isolation methodologies, is a research area that started in the 70’s, and has evolved a lot ever since. The idea is to extract the parts of the program that are relevant to a particular computation, so if an error is thrown it can find the exact trace that produced the error. To do so there are many methods, like dataflow equations blabla, information-flow relations blabla, and graph reachability. To our understanding, none of them uses the output of previous versions of the code, and their approach tends to be based on equations or graphs, so our technique is very different.

Delta debugging blabla, on the other hand is very similar to program slicing, but attempts to identify the lines of code that produced an error after a change was introduced. To do so it removes some of the lines of code that were introduced in the change in different combinations, until one of them stops throwing the error. This technique provides good results, but it requires recompiling the source code after each iteration; it is very time consuming and does not leverage previous working versions of the code. The area is evolving fast, though; there are other studies that explore the program state instead of the code, but it mostly works only for integers, and the approach is very different to ours too.

Finally, as we previously described, BugIsolator uses the X-Diff algorithm and implementation to identify the changes in the XML documents. However, there are a group of other tools and algorithms, like blabla, which perform similar differences detection. More than affecting the novelty of our approach, though, these are other algorithms that we could use to improve the performance of our tool, since they have different time and memory complexities.

1. **FUTURE WORK**

Although the results are promising, our tool is still a work in progress; there are lots of potential enhancements that could enhance its usability, performance and accuracy.

The first area of improvement is simplifying the way logging code is instrumented in Clafer. We mentioned before that currently we have to add the code manually to each of the versions of the compiler that we want to compare. However, this process could be simplified by instrumenting code during runtime, making this process completely automatic.

The second area that we still need to explore is how to produce rules from the filters that the user applied to refine the results presented by our tool. For this to work, we need to find an efficient way to identify the xPath expressions of the filtered nodes.

Another area of future work is improving the way XML difference works. There are many difference detection algorithms that we can test; we would like to find the one with the best balance between performance and accuracy. Also, we could merge the processes of finding XML differences and rules execution to improve the performance. This way the xml structure would only need to be traversed once.

Finally, we would like to test this technique in other software applications, like gcc. We created our technique with Clafer in mind, but we think that the approach is general enough to be applied to other compilers and tools as long as we identify a suitable way of representing their states in an xml format.

1. ACKNOWLEDGMENT
2. CONCLUSION

##### **References**

1. <http://blogs.msdn.com/b/vcblog/archive/2012/08/10/10338661.aspx>
2. Waslander S., “ME 597: Motion Modelling”, ME 597 Lecture Slides, University of Waterloo, pg. 29, October 1st 2010
3. Waslander S., “ME 597: Mapping I”, ME 597 Lecture Slides, University of Waterloo, pg. 38, October 1st 2010
4. Waslander S., “ME 597: Mapping I”, ME 597 Lecture Slides, University of Waterloo, pg. 30, October 1st 2010
5. Waslander S., “ME 597: Mapping I”, ME 597 Lecture Slides, University of Waterloo, pg. 42, October 1st 2010