A Generic Framework for Automated Quality Assurance of Software Models –Implementation of an Abstract Syntax Tree

Darryl Owens and Dr Mark Anderson
Department of Computing
Edge Hill University
Ormskirk, Lancashire

***Abstract*—Abstract Syntax Tree's (AST) are used in language tools, such as compilers, language translators and transformers as well as analysers; to remove syntax and are therefore an ideal construct for a language independent tool. AST's are also commonly used in static analysis. This increases the value of ASTs for use within a universal Quality Assurance (QA) tool. The Object Management Group (OMG) have outlined a Generic AST Meta-model (GASTM) which may be used to implement the internal representation (IR) for this tool. This paper discusses the implementation and modifications made to the previously published proposal, to use the Object Management Group developed Generic Abstract Syntax Tree Meta-model core-components as an internal representation for an automated quality assurance framework.**

Keywords—software quality assurance; software testing; automated software engineering; programming language paradigms; language independence; abstract syntax tree; static analysis; dynamic analysis

I. INTRODUCTION

To ensure the reliability of output, it is imperative that Software Quality Assurance (QA) is adopted in the development and maintenance of scientific software systems [1]. The integration of such techniques can either be performed manually, which is labour intensive, or utilise automated toolkits [2] [3] which alleviate these problems. The automated toolkits are limited in the respect that they are language-, paradigm- or problem-specific. This paper proposes a framework that would address these limitations by introducing a taxonomy of generic techniques combined with a generic internal representation (IR) of languages. The framework also covers a range of different language paradigms. This paper also proposes a form of IR representation that could be used as an intermediary between QA techniques and source code.

When considering the broad range of programming paradigms the differences between the languages, such as the constructs and data types, need to be addressed. This paper focuses on addressing issues in procedural and object-oriented languages as these are the most widely adopted paradigms in the development of scientific software [4].

II. ABSTRACT SYNTAX TREES

In order to address syntactical differences, Abstract Syntax Trees (AST) are adopted as 'a formal representation of the software syntactical structure' [5]. At a surface level, the underpinning constructs of many procedural languages appear similar, and removing syntax from these would make all ASTs analogous. However, the resulting ASTs produced following analysis of source code are based on a broad range of factors, such as the context-free grammar used to define the language syntax [6]. It is therefore highly likely that the generated ASTs for simple algorithms implemented in different programming languages can prove to be fundamentally different. These differences can become even more significant when addressing additional language features, such as data types.

III. CURRENT APPLICATIONS OF ASTS

The primary usage of ASTs is to facilitate the implementation of compiler tools. For this purpose, ASTs are built from token streams after lexical analysis of source code [7]. However, the usage of ASTs now encompasses the implementation of many language related tools, such as interpreters, document generators and syntax-directed editors, etc. [8].

One such use that an AST can support is in duplicate code detection, whereby an AST designed to support data matching efficiency [8] requires only a pattern to be found. The significance of this is that only an initial node needs to be identified which is subsequently followed by a predetermined pattern of nodes. This technique is similar to that found in the plagiarism detection techniques, which makes use of code comparison, described by Cui et al [9]. Equally, code analysis techniques can be implemented using node counting. Removing code comments and disregarding layout metrics produces better comparison metrics from this technique compared to those collected from source code [10].

A major development of ASTs lies in language translation, which occurs by producing an AST for a specific language. The AST can then be parsed whilst introducing the syntax of the output language.

After the tree has been parsed, the resultant code should be complete and functional in the output language syntax [11] [12]. A working example of this technique is adopted by Mono as a working and functional example of the approach in action [13]. Mono is a framework that allows the use of the .NET platform upon other devices than just windows via the use of language translation.

IV. Generating ASTs

In order to generate ASTs, ANother Tool for Language Recognition (ANTLR) is a tool which uses a grammar input and can produce recognisers, compilers and translators [14]. Of significance to the project presented within this paper is that the ANTLR compilers can build ASTs from source code [14]. Utilising ANTLR, an example of the fundamental differences that can be generated in ASTs from simple algorithms is presented. In this example, a simple "Hello World" program written in Java and C#.

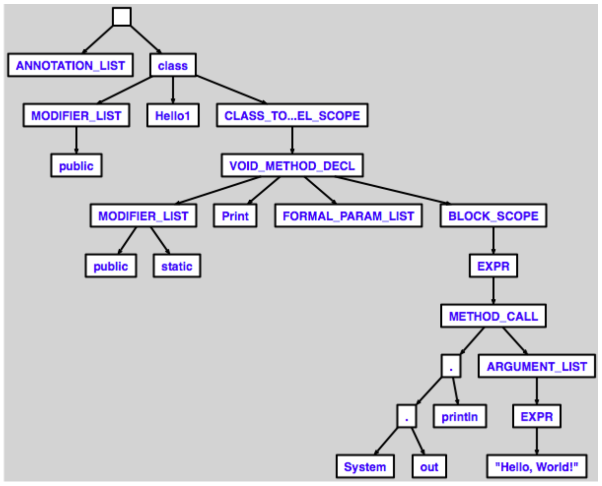

Fig. 1. Java Hello World AST

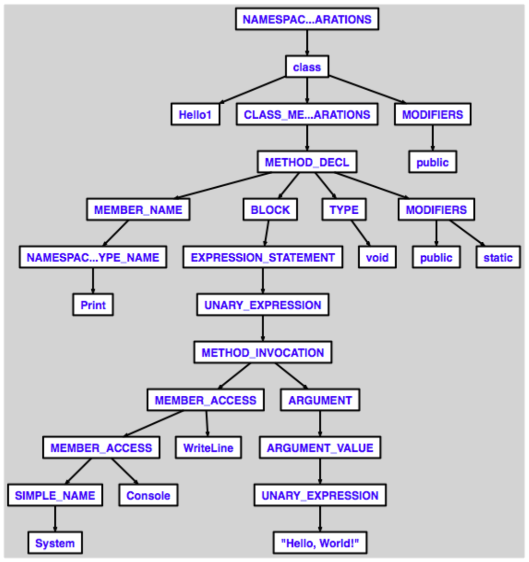

Fig. 2. C# Hello World AST

It can be seen from the ASTs depicted in Figure 1 and Figure 2 that there are fundamental differences between the representations of a simple program in two similar object-oriented programming languages that are included in the ANTLR repository [15].

Clearly there are some similarities between the ASTs; for example the class node has the name, modifiers and body. However, the significant differences between the AST's is directly related to the grammar files.

V. Language Independence

A key requirement for the successful implementation of the proposed analysis framework requires language independence to be implemented in order to separate any reliance on the QA procedures from the syntax and semantics of the source code programming language. The Object Management Group (OMG) has initiated a number of projects to investigate the development of generic ASTs. Broadly there have been two tiers adopted for the approach, the Abstract Syntax Tree Metamodel (ASTM) and the Knowledge Discovery Metamodel (KDM). The KDM is a standard to facilitate interoperability for exchange of data between tools that may be provided by different vendors [16]. The KDM complements the ASTM and both are designed to work together. The extent to which they do is questionable as the link between the ASTM and KDM can best be described as fuzzy [17]. However the KDM is less relevant in the development of the proposed framework, as the KDM focus on migration of software artifacts and not representation of language, so the focus is on ASTM and the Generic ASTM (GASTM) which is defined in the ASTM specification [18].

VI. Testing / Quality Assurance and Automated Approaches

There is a wide range of toolkits developed for testing software [19] [20] [21] [22] [23] [24] and, broadly, these are targeted at the automation of testing to reduce workload required to test software applications. An initial survey of the available toolkits has revealed that most tools that apply QA techniques to multiple languages only do so on a small-scale. Generally this is in the range of 2 – 5 languages [2], and also focused on languages which share a programming paradigm. It is also noted that the more generic toolkits with a broader coverage of programming paradigm instead have a restriction in terms of the areas of testing which are covered [2].

There are two types of analysis within QA; dynamic and static [25] [26]. Whilst both offer advantages, combining these techniques results in a broader impact as a result of QA [18] [19]. Static analysis facilitates an abstract view of a program and examination of source code without code execution [27], and also supports the identification of such potential issues as memory corruption errors, buffer overruns, out-of-bound array accesses, or null pointer de-references [28].

Dynamic analysis is the analysis of code as it is executing, and therefore extracting accurate values of variables under set circumstances is a key target [25]. This technique can be used to run functional, logical, interface and bottom-up tests amongst a range of supported testing [25]. The combination of static and dynamic analysis allows for a larger coverage of QA techniques and a tool which implements both of these analysis types would be more comprehensive [29][30].

VII. Proposal for Abstract Syntax Tree

A proposal for the use of the GASTM as a form of Internal Representation (IR) for automated quality assurance was theorized under the ideas that both static and dynamic analysis could be implemented upon this IR via the processes described by the flow diagrams in figure 3, 4 and 5 [31].

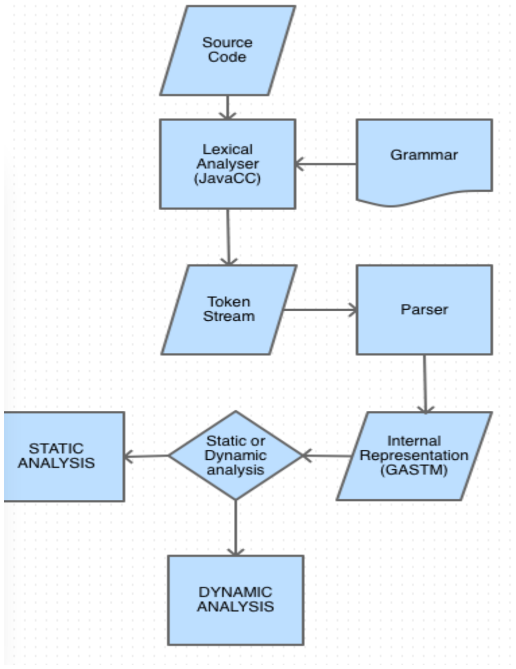
Source Code
Lexical Analyser (JavaCC)
Grammar
Token Stream
Parser
STATIC ANALYSIS
Static or Dynamic analysis
Internal Representation (GASTM)
DYNAMIC ANALYSIS

Fig. 3. System Data Flow

It was identified that by using the GASTM, static analysis could be possible via implementing tree walkers. In essence this would entail replacing source code analysers, and could implement automated quality assurance techniques such as metric and pattern matchers as well as allowing the GASTM to be converted to a control flow graph for data flow analysis.

Dynamic analysis however is a more complicated matter. By using a generic monitor class, nodes could be inserted into a program before conversion into a runnable language. These inserted nodes would call method in the generic monitor class allowing for information about data, properties or runtime information to be pulled out and recorded or analyses whilst the program is running.

VIII. Implementation

LIQA (Language Independent Quality Assurance), is the implementation of the research discussed in this paper. Utilizing tools that have been previous developed by third parties, LIQA implements a middle layer to facilitate interaction using bespoke code and breaks down a subset of the Java language forming the GASTM representation of the source code.

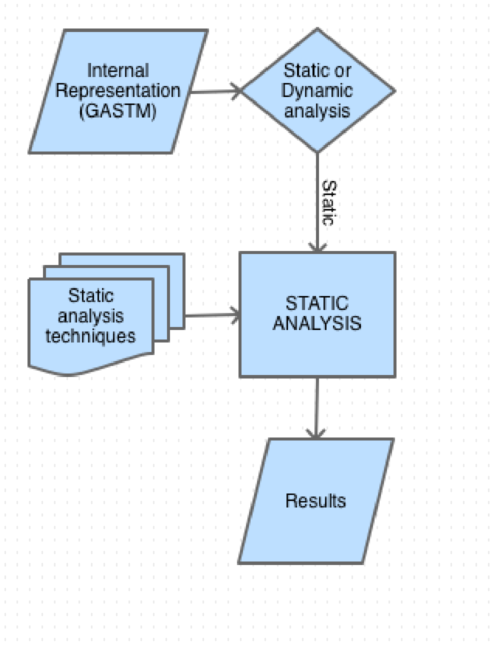
Internal Representation (GASTM)
Static or Dynamic analysis
Static
Static analysis techniques
STATIC ANALYSIS
Results

Fig. 4. Static Analysis Data Flow

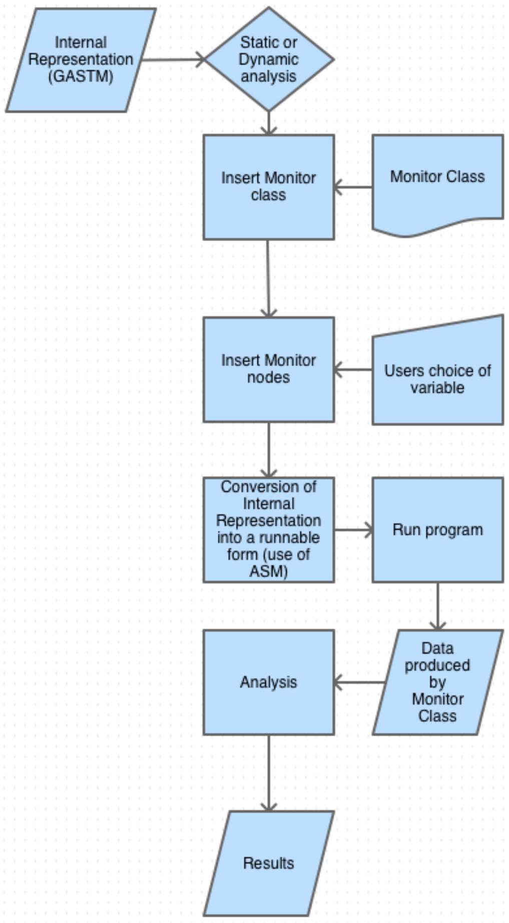
Internal Representation (GASTM)
Static or Dynamic analysis
Insert Monitor class
Monitor Class
Insert Monitor nodes
Users choice of variable
Conversion of Internal Representation into a runnable form (use of ASM)
Run program
Analysis
Data produced by Monitor Class
Results

Fig. 5. Dynamic Analysis Data Flow

A. LIQA Functionality

LIQA was designed for practical use within a lab setting. A GUI (Graphic User Interface) was developed to control the overall work flow as well as visual representation of the GASTM IR (Internal Representation) structure to allow quick assessment of correctness. This has been accompanied by a feature to create and load projects into the software.

B. GASTM Representation

The diagram shown below (figure 7) is the graphical output from LIQA and is a sample of the function definition 'HelloWorld' as shown in figure 6.

```
public class HelloWorld {
    public static void print() {
        printout("Hello, World");
    }
}
```

Fig. 6. HelloWorld Java

As was identified earlier, the IR can become very complicated from even a simple program. After the Java code has been parsed into the classes that represent the GASTM nodes, the object is then walked using a separate class to 'pretty print' the IR into XML which is then taken and placed in a SVG (Scalable Vector Graphics) format culminating in figure 7.

Several modifications have had to be made to the GASTM representation developed by Modisco [32], these modifications have been made for one of two reasons. The small change of implementing java.io.Serializable on the classes GASTMFactoryImpl, GASTMObjectImpl,

GASTMPackageImpl, GASTMSemanticObjectImpl, GASTMSourceObjectImpl and GASTMSyntaxObjectImpl, was to enable the IR to be saved as a binary file thus making it simple to implement a project based file system and allow users to save their work.

The other modifications listed below were implemented to better mirror the properties of some of the selected procedural languages. The ClassType and ClassTypeImpl were modified to include a link to the AccessKind class via the methods getAccessKind and setAccessKind. This was because in Java, C#, C++ and recent high-level languages allow programmers to assign classes with the access modifier i.e. public, private or protected. The FunctionMemberAttribute and FunctionMemberAttributeImpl have had the property IsStatic added, which is a boolean variable. The methods to modify the property setIsStatic and getISStatic have also been added. This was as Java, C#, and C++ allow programmers to assign functions with the static modifier.

C. Tools used in development

LIQA utilizes several tools to achieve various tasks, these tasks (and therefore tools) are not all necessary however they make LIQA easier to use and simple to test for issues. The tools used as listed below:

- Modisco - GASTM Core Model [32]
- JavaCC - Produced tokinizer for Java (Grammar from library) [33]
- XsdVi - Used to generate a .svg file from .xsd [34]
- Batik - Toolkit to visualize .svg file in JFrame [35]

The Modisco library has a Java representation of the GASTM core objects and therefore can be used instead of having to write the object in Java or another language. This links with the JavaCC tool which generated a Java tokenizer using a grammar located in the JavaCC library which is used by LIQA to convert the source code and generate the GASTM IR via the Modisco library. These two tools were required to make the production of LIQA a quicker and simple process. However the other tools are used to simplify the use of LIQA and simplify fault finding within the parsing and IR generating process. After the IR is generated it is then walked by LIQA and written to and .xsd file.

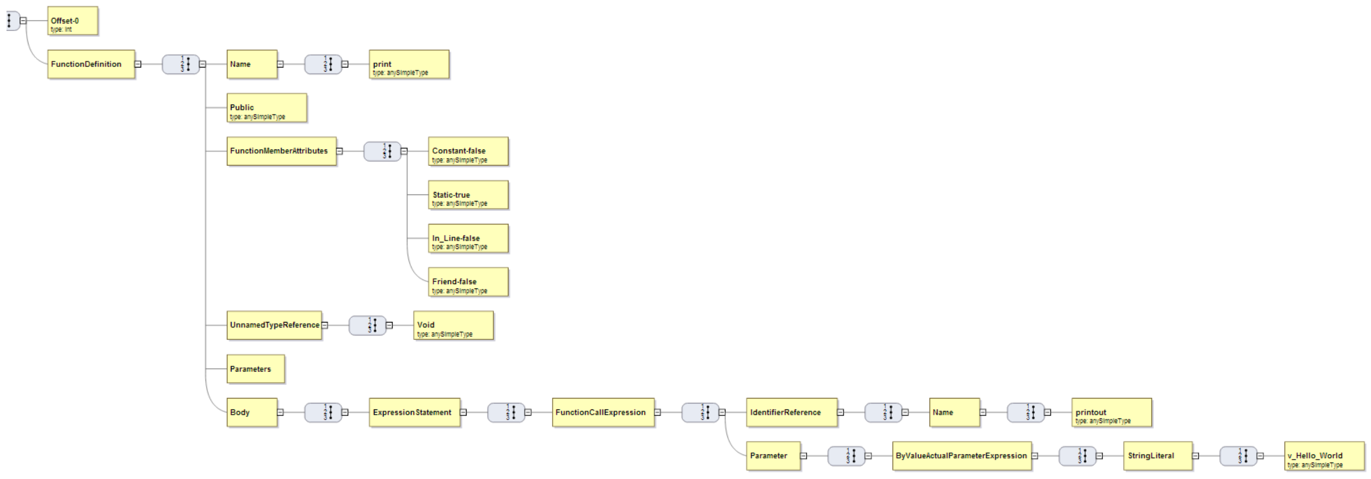

Fig. 7. HelloWorld GASMT

XsdVi is then used to generate a .svg file from the .xsd, this is so a graphical representation of the IR is viewable. Following this, the Batik toolkit has been implemented into LIQA to allow the .svg file to be viewed in a java JForm utilizing the JSVGScrollpane and JSVGCanvas.

D. Limitations

The limitations of LIQA are underpinned by a variety of contributing factors and are segmented to specific sections of the program. LIQA itself has the limitation of only being able to handle single file programs rather than full programs/projects built up over multiple files. This is due to time constraint. Although extending LIQA to handle full programs/projects would be a simple process, at this stage of the research it is not required as LIQA is only a 'proof of concept' for the larger framework.

The following limitations are to the Java language parser and are either due to time constraints for this research however will be part of future development and not being required to test the basic function of the IR, or are due to the GASTM not supporting the specific syntax. The following limitations are due to the GASTM core limitations, the program that is being analyzed cannot contain:

<= , >= , += , -= , /= , *=

The GASTM cannot handle these operators however considering the operators can be broken down i.e. 'x += 1' = 'x = x + 1' and 'x <= 2' = 'x < 3' it would be possible to integrate these into the IR. Due to time constraints, the lack of importance with these operators, as they can be replaced and effort it would take to code the conversion, they have not been included within the parser that generates the IR from the tokens.

The following limitations are due to be implemented in further developments of LIQA. However they are not necessary for testing the framework at this stage.

- Operators that are not implemented are '?' and '!'
- List types are not implemented i.e. 'List<String>'
- Re-type casting has not been implemented i.e. 'String str = (String) x;'
- The assignment of arrays via block statement has not been implemented i.e. 'int[] x = {3,2,1};'
- Inline if statements have not been implemented, if statements must have a block containment i.e. 'if (condition) statement;' is not supported and 'if (condition) {statement}' is supported.

E. Analysis Test

After the initial implementation of the GASTM IR was finished, a proof of concept addition was made to LIQA, this was to implement a single form of static and dynamic analysis to test the proposal before a deeper analysis of quality assurance techniques takes place.

For dynamic analysis a simple profiler was developed, this required a tree walker which analyses the IR to find all the function definitions and the variable definitions in them allowing the user control over which methods and variables were monitored, after the user made their choice LIQA then inserts nodes to monitor the variable values wherever modified and counts method calls through the monitor class interface. The monitor class must be written in the original language as it may require language specific method calls itself in later development.

For static analysis a simple metric was written using a similar tree walker as the one for the profiler however this returned Logical Lines of Code value and is currently setup for further metrics to be included.

F. Proposal Modifications

During the implementation small changes had to be made to the proposal as technological limitations arose, the only change that was significant is the formulation of a plausible and feasible way of running a GASTM IR that has been modified to perform dynamic analysis. This was achieved by running a conversion back into source code from the GASTM IR via a tree walker, this would have to be implemented with every language due to library and specific method calls that are unique to that language. A further implementation of LIQA could include a method and parameter mapping system which would also allow for language conversion.

The tree walker however provides a further form of quality assurance as some coding standards require code to be formatted is a specific way, as the tree walker generates the code a formatting system can be applied for easier maintenance.

A smaller modification is to the flow of data with regards to how static analysis is run, the DFD (figure 8) shows the initial stage of the IR being converted to a CFG, though not all static analysis techniques utilize a CFG the conversion must take place for those that do before the application of a static quality assurance techniques can be applied. The tree walker is utilized by all techniques not just the conversion to a CFG and would also be required before a technique can be applied.

IX. Conclusion

This paper presented the implementation of an internal representation fit to allow both quality assurance via static and dynamic analysis and also to allow the representation of multiple languages. The major issues of this implementation are differences in languages and how to apply the analysis upon the final structure.

The ASTM is a standard prepared by OMG for language based tools and implements a set of core components that can represent a subset with many procedural and object oriented languages. It is therefore an obvious choice for this internal representation.

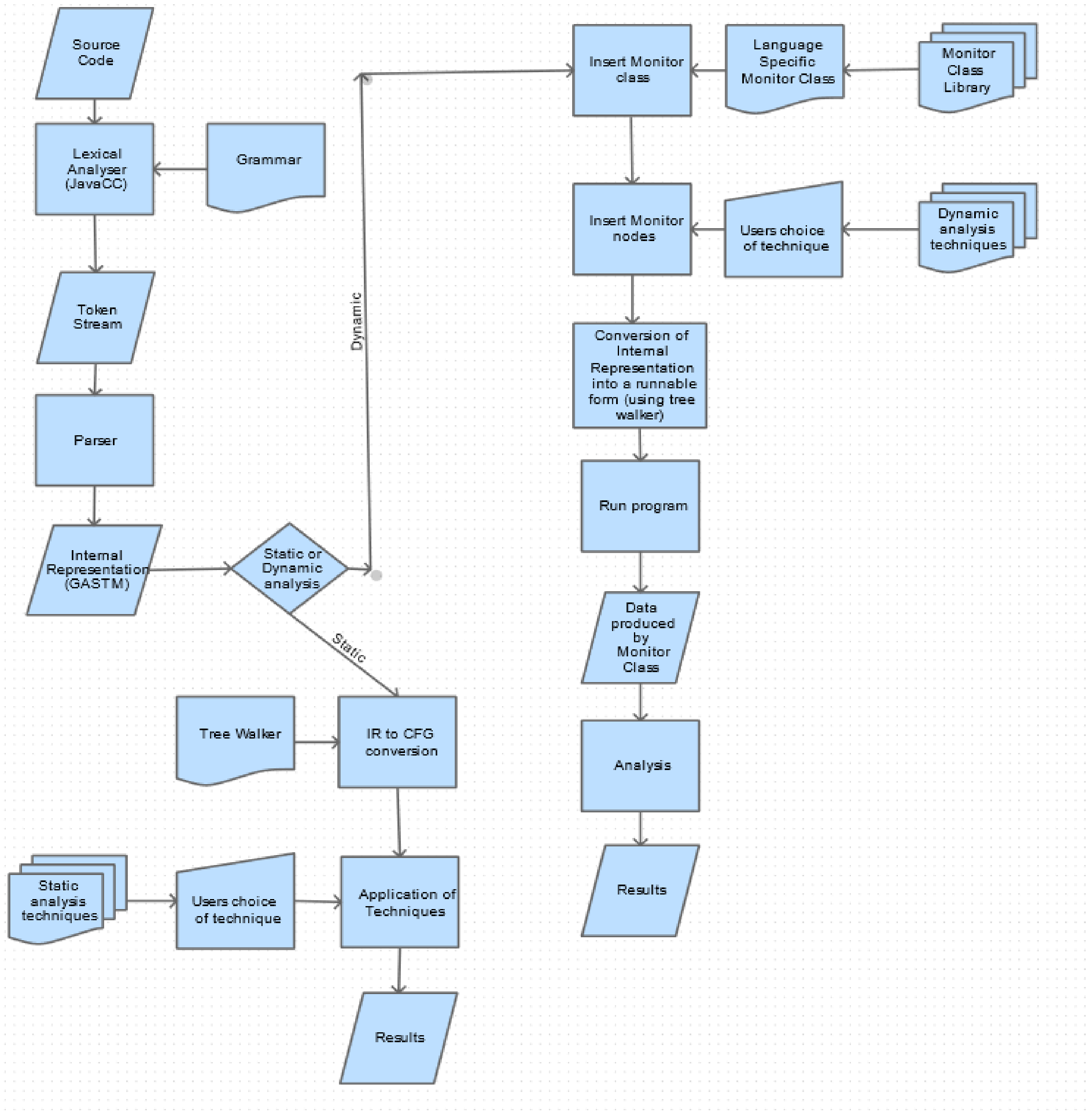
Source Code
Lexical Analyser (JavaCC)
Grammar
Token Stream
Parser
Internal Representation (GASTM)
Static or Dynamic analysis
Dynamic
Static
Tree Walker
IR to CFG conversion
Static analysis techniques
Users choice of technique
Application of Techniques
Results
Insert Monitor class
Language Specific Monitor Class
Monitor Class Library
Insert Monitor nodes
Users choice of technique
Dynamic analysis techniques
Conversion of Internal Representation into a runnable form (using tree walker)
Run program
Data produced by Monitor Class
Analysis
Results

Fig. 8. LIQA Data Flow

The implementation of the proposal within the tool LIQA represents a proof of concept showing that the GASTM is a suitable IR and that quality assurance techniques can be applied to this. However exactly what techniques can be applied is uncertain, it is certain that at least a simple level of static and dynamic analysis can be performed. Further work will demonstrate what techniques can be applied and these techniques will be derived from many tools currently used as industry standards.

REFERENCES

[1] Rosenberg, L. (2002) Software quality assurance engineering at NASA. *Aerospace Conference Proceedings, 2002. IEEE*. 5 pp. 5-2569 - 5-2575.

[2] Owens, D. and Anderson, M. "A Generic Framework for Automated Quality Assurance of Software Models: Supporting Languages of Multiple Paradigms". In 5th International Conference on Computer Engineering and Technology (ICCET), Vancouver, Canada, 13-14 April

[3] Collins, J., Farrimond, B., Anderson, M., Owens, D., Bayliss, D. and Gill, D. "Automated Quality Assurance Analysis: WRF – a case study". Accepted for 5th International Conference on Computer Engineering and Technology (ICCET), Vancouver, Canada, 13-14 April

[4] Pickering, R. (2010). *Beginning F#.* Apress.

[5] Newcomb, P. (2005, October). *Abstract Syntax Tree Metamodel Standard ASTM Tutorial 1.0.* Retrieved 2 5, 2013, from Object Managment Group: http://www.omg.org/news/meetings/workshops/ADM_2005_Proceedings_FINAL/T-3_Newcomb.pdf

[6] Tripp, A. (2006, February 22). *Manual Tree Walking Is Better Than Tree Grammars.* Retrieved 2 5, 2013, from ANTLR v2: http://www.antlr2.org/article/1170602723163/treewalkers.html

[7] Fischer, G., Lusiardi, J., & Gudenberg, J. (2007, August 25-31). Abstract Syntax Trees – and their Role in Model Driven Software Development. *Software Engineering Advances, 2007. ICSEA 2007. International Conference on* , 38.

[8] Van Den Brand, M., Moreau , P., & Vinju, J. (2005). A generator of efficient strongly typed abstract syntax trees in Java. *EE Proceedings - Software Engineering 152, 2 (2005) 70--87* , 70-87.

[9] Cui, B., Li, J., Guo, T., Wang, J.-X., & Ma, D. (2010). Code Comparison System based on Abstract Syntax Tree. *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*, 668- 673 .

[10] Fischer, G., Lusiardi, J., & Gudenberg, J. (2007, August 25-31). Abstract Syntax Trees – and their Role in Model Driven Software Development. *Software Engineering Advances, 2007. ICSEA 2007. International Conference on* , 38.

[11] Ichisugi, Y. (2003). *Patent No. 6516461*. United States.

[12] ASM. (2012). *Model Driven Modernization* . Retrieved 2 5, 2013, from Automated Software Modernization: http://www.automatedsoftwaremodernization.com/component/content/article/3.html

[13] OMG. (2012, July 19). *Catalog Of Omg Modernization Specifications*. Retrieved 2 5, 2013, from Object Management Group: http://www.omg.org/technology/documents/modernization_spec_catalog.htm

[14] Parr, T. (n.d.). *ANTLR*. Retrieved 2 4, 2013, from ANother Tool for Language Recognition: http://www.antlr.org

[15] *Grammar List.* (n.d.). Retrieved 2 5, 2013, from ANTLR v3: http://www.antlr3.org/grammar/list.html

[16] Deltombe, G., & Goaer, O. L. (2012). Bridging KDM and ASTM for Model-Driven Software Modernization . *SEKE* , 517-524.

[17] OMG. (2011, January). *OMG Architecture-driven modernization: Abstract Syntax Tree etamodel (ASTM).* Retrieved 2 5, 2013, from Object Managment Group: http://www.omg.org/spec/ASTM/1.0/PDF/

[18] Salah, M., Mancoridis, S., Antoniol, G. & Di Penta, M. (2006) Scenario-driven dynamic analysis for comprehending large software systems. *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on* pp. 80 - 90.

[19] Fairley, R. (1978) Tutorial: Static Analysis and Dynamic Testing of Computer Software. *Computer*. 11(4) pp. 14-23.

[20] Anywhere, A. (n.d.) *TestingAnywhere*. HYPERLINK "http://www.automationanywhere.com/Testing/" http://www.automationanywhere.com/Testing/ [accessed 09 November 2012].

[21] Systems, Q. (n.d.) *Cantata - The Unit Testing Tool for C/C++*. HYPERLINK "http://www.qa-systems.com/cantata.html%20" http://www.qa-systems.com/cantata.html [accessed 09 November 2012].

[22] Artho, C. et al. (2004) JNuke: Efficient Dynamic Analysis for Java. *Proc. CAV '04*.

[23] MathWorks (1994) *Static Analysis with Polyspace Products*. HYPERLINK "http://www.mathworks.co.uk/products/polyspace/" http://www.mathworks.co.uk/products/polyspace/ [accessed 09 November 2012].

[24] SimCon (1995) *SimCon - Fortran Analysis, Engineering & Migration*. HYPERLINK "http://www.simconglobal.com/" http://www.simconglobal.com/ [accessed 09 November 2012].

[25] Fairley, R. (1978) Tutorial: Static Analysis and Dynamic Testing of Computer Software. *Computer*. 11(4) pp. 14-23.

[26] Austin, A. & Williams, L. (2011) One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on* pp. 97-106.

[27] Austin, A. & Williams, L. (2011) One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on* pp. 97-106.

[28] Bell, D. & Brat, P.G. (2008) Automated Software Verification & Validation: An Emerging Approach for Ground Operations. *Aerospace Conference, 2008 IEEE* pp. 1 - 8.

[29] Harrison, K. (1999) Static Code Analysis on the C-130J Hercules Safety-Critical Software. *UK Iternational Systems Safety Conferance*.

[30] Wong, W.E. (2000) An Integrated Solution for Creating Dependable Software. *Computer Software and Applications Conference* pp. 269 - 270.

[31] Owens, D. and Anderson, M. "A Generic Framework for Automated Quality Assurance of Software Models - Application of an Abstract Syntax Tree". Science and Information Conference 2013, Heathrow, London, 7-9 October 2013

[32] Modisco. (n.d.). Modisco. Retrieved from Eclipse: http://www.eclipse.org/MoDisco/

[33] JavaCC. (n.d.). Java Compiler Compiler tm (JavaCC tm) - The Java Parser Generator. Retrieved from JavaCC: http://javacc.java.net/

[34] Slavětínský, V., & Kosek, J. (2013, March 22). XsdVi. Retrieved from Source Forge: http://sourceforge.net/projects/xsdvi/

[35] Apache. (n.d.). The Apache™ Batik Project. Retrieved from The Apache™ XML Graphics Project: http://xmlgraphics.apache.org/batik/