Concept Assignment as a Debugging Technique for Code Generators

Jeremy Singer
University of Manchester, UK
jsinger@cs.man.ac.uk

Abstract

Code generators are notoriously difficult to debug, yet their correctness is crucial. This paper demonstrates that concept assignment can be performed in an entirely syntax-directed manner for code generators and other highly structured program modules. Anomalies in this concept assignment indicate the possible existence of bugs. These insights enable the formulation of a new debugging technique for code generators. This paper describes the procedure, a practical implementation, and results from the application of this debugging technique to an experimental code generator.

1 Introduction

Code generators feature in compiler backends. Similar modules are necessary in interpreters, binary translators and other program transformation systems. A code generator transforms an input instruction stream into an output instruction stream, according to some well-defined rules.

In general, programmers find code generator debugging to be a frustrating, arduous task. There are several reasons for this difficulty.

1. *Second order debugging* is required. The principal way to find errors is to exercise the code generator with some test input instructions, then execute the output instruction stream. Bugs in the output instruction stream indicate bugs in the code generator. It is necessary to deduce the code generator bugs by working backward from the generated code bugs. This approach operates at a higher level of abstraction than usual, since there is increased separation from the original source code. It would be more intuitive to debug the code generator source code directly rather than to analyse the generated output code.

2. When code generators target *exotic or experimental platforms*, it can be difficult or impossible to run the generated code and test for bugs. The platform may be unstable, or not yet developed, or not available in hardware or simulator.

3. Backends are often the *most ugly* areas of compilers, since they are not exposed directly to users. Sometimes code generator source code is itself auto-generated (entirely or in part) and then modified by hand. Code generators contain lots of repetitive code, and there are always many special cases to handle.

4. When a compiler backend is retargetted to a new platform, sometimes the backend for an existing platform is adapted. Developers may fail to modify sufficient code, comments and conventions. This style of *cut-n-paste retargetting* is particularly susceptible to subtle bugs.

Code generator maintainers employ one of the following elementary techniques when hunting down their elusive bugs.

- They insert plenty of trace `printf` statements into the code generation methods, and trawl through huge files of trace output.
- They 'eyeball' the code generator source code for irregularities.
- They single-step through generated output code instructions and check for errors.

Experienced developers make good use of tools like grep, diff, awk and perl to analyse generated code or trace files, but even this is tiresome. As both compilers and target platforms become more complex, there is an urgent need for better debugging techniques to handle 21st century code generators. Debugging tools should operate at a higher level and be more user-friendly.

This paper applies *concept assignment* (CA) to the problem of code generator debugging. Section 2 reviews CA, which is a well-established technique for program understanding. There are many approaches to performing CA, mostly involving artificial intelligence theory. However, Section 3 shows how CA may be simply applied to code

generators, by means of basic syntax-directed analysis. Section 4 shows how this syntax-directed CA can help to discover bugs, by comparing the results of different CAs on a code generator code base. Section 5 describes an automated debugging tool that implements this procedure. Section 6 reports on results when this tool is applied to an experimental code generator. Section 7 discusses related work. Finally Section 8 concludes.

This paper makes four significant contributions.

1. It shows that the CA problem can be solved in an entirely syntax-directed manner for highly structured source code modules like code generators.
2. It evaluates several software metrics as a basis for grouping conceptually related methods.
3. It presents a technique for debugging code generators that compares different CAs for inconsistencies.
4. It describes a functional implementation of this debugging technique, complete with a case study involving an experimental code generator.

2 Concept Assignment

This section introduces the notion of concept assignment (CA). It also discusses the computation and application of CA information.

2.1 What is Concept Assignment?

CA is a process for high-level program comprehension [2]. It relates human-oriented concepts to implementation-oriented artefacts. Often, human-oriented concepts are expressed using UML diagrams or other high-level specification schemes, which are far removed from the typical programming language sphere of discourse. In contrast, implementation-oriented artefacts are expressed directly in terms of source code features, such as variables and method calls.

CA is a form of reverse engineering. In effect, it attempts to work backward from source code to recover the 'concepts' that the original programmers were thinking about as they wrote each part of the program. This conceptual pattern matching assists maintainers to search existing source code for program fragments that implement a concept from the application. This is useful for program comprehension, refactoring, and post-deployment extension.

Generally, each individual source code entity implements a single concept. The granularity of CA may be as small as per-token or per-line; or as large as per-block or per-method. Often, CA is visualised by colouring each source code entity with the colour associated with that particular concept. CA can be expressed mathematically. Given a set U of source code units $u_0, u_1, \ldots, u_n$, and a set C of concepts $c_0, c_1, \ldots, c_m$, then CA is the construction of a mapping from U to C. Often the mapping itself is known as the concept assignment.

Note that there is some overlap between CA and aspect mining [5]. Both attempt to recover high-level information from low-level program descriptions. The principal difference is that concepts are universal. Every source code entity implements some concept. In contrast, only some of the source code implements aspects. *Aspects* encapsulate implementation-oriented cross-cutting concerns, whereas *concepts* encapsulate human-oriented concerns which may or may not be cross-cutting.

2.2 Approaches to Concept Assignment

There are two phases in the CA process.

concept selection: determine the set C of human-oriented concepts that are implemented in the source code.

concept mapping: map each source code entity (at some specified granularity) to a particular concept from C.

Either or both of these phases can take place manually or automatically. Automated CA may require human guidance or intervention. In manual CA, the maintainer traverses the source code and manually assigns a plausible concept for each source code entity. In automated CA, a software tool analyses the source code and automatically selects the most appropriate concept for each source code entity (for some definition of appropriate). Most automated systems incorporate some level of artifical intelligence to determine 'appropriate' concepts. Biggerstaff et al describe a semi-automated design recovery system called DESIRE [2]. This uses a precomputed domain model and a connectionist inference engine to perform the CA. Gold and Bennett describe a hypothesis-based CA system [12]. This applies information from a human-generated knowledge base to source code, using self-organizing maps to perform the CA.

The simplest approach to automated CA is syntax-directed, also known as program plan assignment by parsing. Rich and Wills [21] describe a system that operates in this manner. Biggerstaff et al argue that syntax-directed CA is not powerful enough to recover genuine human-oriented concepts, since it is restricted to a 'parsing-oriented' view of the world [2]. However, this paper shows that such a restricted view is sufficient for highly structured code bases like code generators. The first novel idea in this paper is that syntax-directed CA is as accurate as manual CA for a restricted class of highly structured programs. Section 3 provides further detail.

```
emitADD          emitSUB
  pop              pop
  pop              pop
  add              sub
  push             push
```

Figure 1. Addition and subtraction code generation

2.3 Applications of Concept Assignment

Existing CA applications are entirely concerned with program understanding. This does not seem to be taken further. Biggerstaff et al [2] simply present a tool for software comprehension and visualization. Gold and Bennett [12] state that the primary motivation for CA is 'providing the maintainer with an additional knowledge source from which to work.' Kontogiannis et al [15] use CA information to assist in the detection of cloned source code, but there is no automated client that uses this information.

This paper argues that a machine-level program understanding should facilitate further automated analysis or transformation of the subject program. The second novel idea in this paper is that CA supplies information to support automated debugging. Section 4 elaborates.

3 Code Generators

This section claims that, for the restricted domain of code generators, syntax-directed CA can produce as accurate results as manual CA.

Code generators usually have standard output patterns for related input instructions. For instance, consider Figure 1 which shows the code generation methods for addition and subtraction operations in a stack-based machine. Indeed, all diadic arithmetic and logical operations will have the same style.

Groups of conceptually related instructions have similar code generation patterns in the program source code. These patterns are easily identifiable by explicit syntactic clues, such as number of lines of code and control flow structure. Each related code generation method can be assigned the same concept. This is the first insight:

> Concept assignment for code generators can be entirely syntax-directed.

Note that this observation does not apply exclusively to code generators. Other code base genres may exhibit this phenomenon, so long as they have the same properties that the code is highly structured, and stereotypical structure indicates conceptually related code.

Basically, syntax-directed CA attempts to discover units of code (in our case, methods) that are quite similar, but not exactly the same. It uses simple software metrics such as:

1. number of lines of code (numlines)
2. number of method calls (nummethodcalls)
3. cyclomatic complexity [18] (cyccomp)
4. ABC complexity [7] (abc)

If two methods have the same score for one or more of these metrics, then they are assigned the same concept. In other words, they are *conceptually related*. This is self-clustering CA. Rather than doing an explicit initial concept selection step, the process simply clusters the methods into related groups. Each group is assumed to implement a concept. Note this is similar to Cimitile et al [4], only we assume the clusters are valid without performing any adequacy validation process.

The above metrics are *absolute*, since each method can be given a score independent of all other methods. There are also some *relative* metrics. String similarity is relative, since it compares all methods at once, and clusters them according to how similar their method names (methodnames) or method bodies (methodbodies) are to each other. String similarity is expensive to compute, since it uses dynamic programming.

All of these software metrics are entirely syntax-directed. They require no domain-specific knowledge about input programs. However, the results presented in Section 6 show that such metrics enable as accurate CA analysis as the best domain-specific knowledge possible.

The above six metrics are *primitive*. It is possible to combine two or more primitive metrics into a *compound* metric. The compound union a+b of metrics a and b is defined as follows: a+b relates method m_1 to method m_2 if either a or b relates m_1 to m_2. The union operation generally decreases the number of concepts and increases the size of each concept set. The compound intersection a*b of metrics a and b is defined as follows: a*b relates method m_1 to method m_2 if both a and b relate m_1 to m_2. The intersection operation generally increases the number of concepts and decreases the size of each concept set.

4 Debugging Technique

This section presents the debugging technique for code generators, that leverages the CA information.

Recall that manual CA should group related code generation methods correctly, since it relies on human guidance and domain-specific knowledge. The previous section showed that syntax-directed CA clusters related code generation methods. However, syntax-directed CA can only give

correct answers if conceptually related methods are syntactically similar, which they should be. If related methods are not syntactically similar, there is a possible bug! Incorrect syntax implies an incorrect code generator. Section 1 has already described why such code generation bugs are difficult to detect and correct via manual inspection. So an automated tool that discovers such errors is highly desirable. This results from the second insight:

> Anomalies in syntax-directed CA indicate potential bugs in code generators.

A comparison between syntax-directed CA and manual CA highlights unexpected relationships in syntax-directed CA. These anomalies indicate that further investigation is required. So the automated debugging tool does not fix the problem, but it provides a zoom tool to focus the (human) debugger's attention.

A procedure for code generator debugging follows.

1. perform manual CA.
2. perform syntax-directed automated CA.
3. detect inconsistencies between manual and automated CAs.
4. check each inconsistency—is it caused by bugs in the code generator source code?

Steps 1 and 4 are manual. Steps 2 and 3 are automated. In effect, the procedure enables expert humans to concentrate on source code locations containing potential bugs. The human maintainers then perform manual checks to determine whether each inconsistency indicates the presence of a genuine bug.

The automated inconsistency detection is important. A good heuristic seems to be 'odd-one-out' detection. It can be defined as follows. Let $f : M \rightarrow A$ be the mapping for concept assignment CA_A. Let $g : M \rightarrow B$ be the mapping for a different concept assignment CA_B over the same set of methods M. Method $m \in M$ is an 'odd-one-out' exactly when $f(m) = a_x$ and $g(m) = b_y$ and there is no other method $m' \in M$ with $f(m') = a_x$ and $g(m') = b_y$. This heuristic highlights anomalies while taking account of the fact that two different CAs are never exactly aligned. Concept names are generally different. Often there is a different number of concepts in each CA. These details are examined empirically in Section 6.

5 Implementation

This section describes the software tool that implements the debugging procedure described in the previous section.

I implemented a syntax-driven CA tool. It is written in Java and it operates on Java programs. The Java parser is generated by the JavaCC parser generator. This parser is extended to compute the appropriate metric scores for each method's abstract syntax tree. The tool outputs the scores for each analysed method into plain text files.

I developed Perl scripts to transform the plain text files into HTML-based web pages. This enables hyperlinked cross references, colour highlights and navigation panels. The auto-generated HTML can be navigated using a familiar web browser interface. This is an intuitive tool to visualise the CA results. Note that source code navigation via web browsing was originally proposed in the elucidative programming paradigm. [20].

The simplest CA presentation displays the source code with colour highlights, where different colours correspond to different concepts. The output shows a concept map that gives a high-level overview of the source code, with different coloured blocks corresponding to different concepts. A close-up frame shows the source code of the currently selected method. A different method is selected by clicking on a new area of the concept map, or by choosing a method from the list of methods in each concept. Figure 2 shows an example screenshot of this visualization tool in action. HTML code should be refreshed each time CA reoccurs, so the visualization information is dynamically regenerated.

More Perl scripts analyse the CA results. Various tools compute the union, intersection and difference of two CAs. The difference tool applies the 'odd-one-out' heuristic (outlined above) to highlight inconsistencies between CAs.

Once the automated CA information has been generated, it can be compared with manual CA. For the manual CA data, I grouped Java bytecode operations into concepts, largely based on the groupings suggested by standard JVM textbooks [17, 22]. These groups include integer arithmetic operations, conditional branch operations, and floating point arithmetic operations.

Differences between manual and automated CA indicate potential bugs in the code generator source code, which need to be investigated further. Again, the web browser interface presents an intuitive interface to the source code. Methods that have inconsistently assigned concepts are highlighted, and the source code can be examined through the browser.

The three most desirable properties of an automated debugging tool are:

1. scalability, and
2. intuitivity, and
3. precision.

The implementation presented is scalable, since the concept map frame (left panel in Figure 2) enables a high-level overview as well as a zoom in for detail. The tool is intrinsically intuitive since it relies on the familiar underlying

Figure 2. Screenshot of the concept visualization tool

web browser infrastructure. The next section discusses the precision of the debugging tool.

6 Results

This section describes a case study of the application of this debugging tool to an experimental code generator. I developed the debugging tool for the specific purpose of debugging an ARM/Linux code generator for the IBM Jikes RVM compiler. ARM is a standard RISC architecture [9]. RVM is an open-source adaptive Java virtual machine (JVM) [1]. RVM code generator modules are written in Java. The experimental ARM backend is a non-optimizing code generator that simply macro-expands Java bytecode operations into fixed sequences of ARM instructions. This conforms to the model of code generator assumed by this paper, for which similar input operations have similar code generation methods.

Residual bugs are highly likely in the ARM backend, since it has been developed by copying large portions of source code from PowerPC and IA-32 backends [3]. These two architectures are quite different from each other and from ARM. The relevant source code has been copied based on its match with the features of the ARM architecture. Bugs could be caused by mismatches between the original target and the new target (for instance, PowerPC has 32 general purpose registers, ARM has less than 16) or by inconsistencies between several original targets (for instance, PowerPC method calling conventions differ from IA-32). Such incompatibilities are likely sources of error.

The empirical results presented in this section show that structural similarities in code generator source code correlate with conceptual similarities. Also, the results show that anomalies in automated CA can detect a significant number of bugs.

6.1 Consistency of Manual and Automated CAs

Figure 3 shows the differences between manual CA and various automated CAs. A score of zero indicates perfect agreement. The abstract measure of difference is computed from the proportion of methods that are not 'odd-ones-out' in automated CA, and also the total number of concepts in automated CA compared with the total number of concepts in manual CA. These results show that intersection-based compound CAs are most accurate, and that methodbodies*abc gives the closest level of agreement. (Note that all bar one merges all primitive metrics except for methodbodies, which is the most expensive to compute.) A low difference score means that this automated CA should give few false positives in the debugging tool. Recall that a difference between manual and automated CAs is interpreted as a potential bug that needs further investigation.

6.2 Bug Detection

This section discusses the relative capabilities of the various automated CAs for finding bugs. A potential bug is indicated by a difference between the manual and automated CAs, using the 'odd-one-out' heuristic. I measured this capability by artificially injecting known bugs into the code generator source code, and seeing whether these caused anomalies in automated CA.

The first set of tests injects a single bug at a time into the ARM RVM code generator source code. There are 10 different cases. These range from generating too few instructions, generating too many instructions, generating the wrong instructions, using incorrectly typed registers or opcodes in generated instructions. These are typical bugs from a cut-n-paste error that has not been fixed up properly afterwards. Figure 4 shows how many of the 10 bug injections can be detected by each automated CA, when each bug is injected in a separate run of the debugging tool. Only a few automated CAs are shown, these include all primitive CAs and the most accurate compound CAs. Note that the numlines metric is among the most successful for bug detection. This shows how obvious the bugs are in terms of syntactic clues. However numlines may not be used for bug detection in practice, since the previous study shows that it has a higher number of false positives than, say, numlines*abc. Note that methodnames string similarity metric cannot detect any bugs at all. This is because it does not take any account of syntactic differences in method bodies, only looks at their names. While this may be a useful metric for clustering related methods into concepts, it is almost useless for actual bug detection.

The second set of tests injects all 10 bugs into a single instance of the source code file at once. This may inhibit the debugging technique, since there will obviously be less structural similarity. More bugs imply less structural similarity between conceptually related code. If the same bug occurs in two different methods in same concept, that bug is more difficult to detect since the concept does not have such stereotypical structure as it should. Figure 5 shows how many of the 10 bug injections can be detected by each CA when all 10 bugs are injected at once.

The results show that the debugging technique is less effective. The maximum number of bugs that can be detected at once is 6, whereas previously, 8 out of 10 bugs could be detected on an individual basis. Note that the most accurate automated CAs are the same as before.

The real measure of the success of this debugging technique is its ability to detect genuine bugs in the code base. This was the original motivation for the tool's development. Now the tool is reaching maturity, I am about to deploy it for real bug detection on the ARM code generator, as well as on other backends for the RVM, notably the newly developed

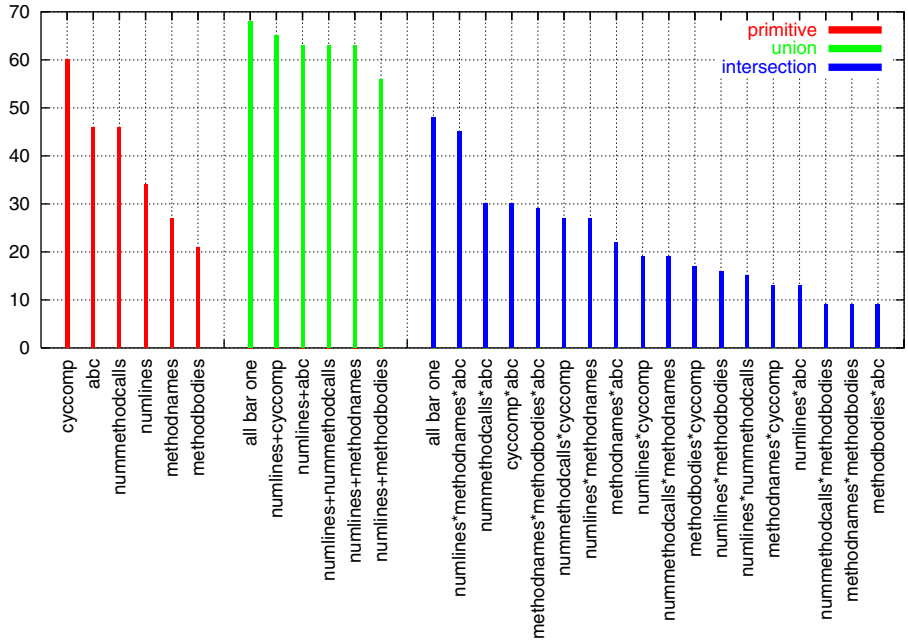

Figure 3. Difference between manual CA and various automated CAs

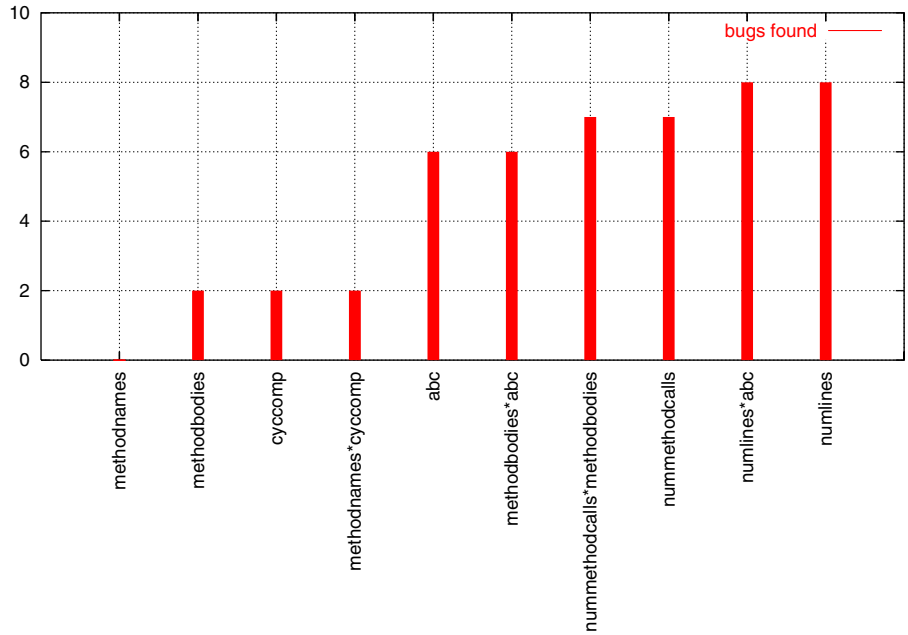

Figure 4. Number of individual bugs detected using various automated CAs

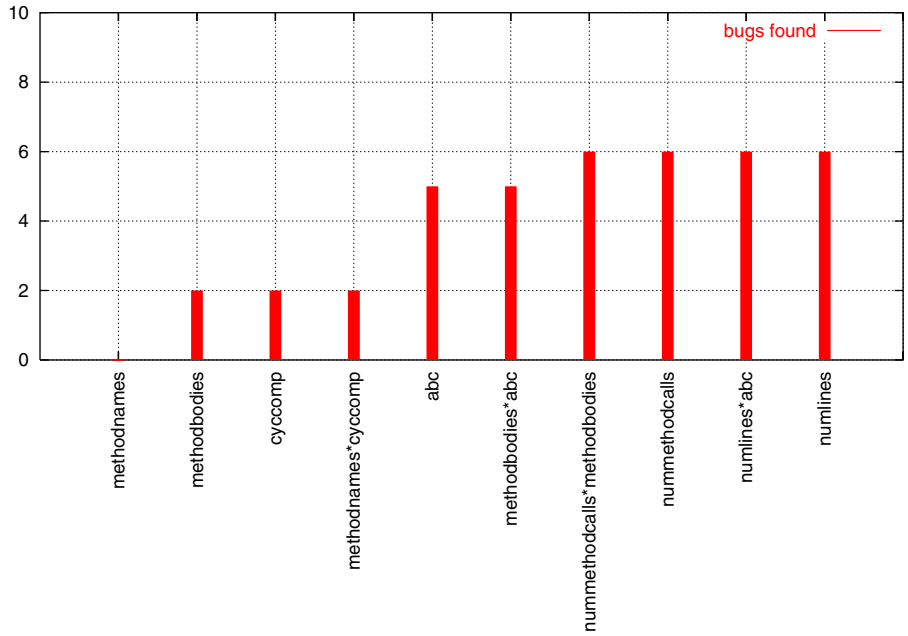

Figure 5. Number of bugs detected in composite test using various automated CAs

prototype Jamaica backend [23]. Another validation step is to mine the code generator's revision history files, and see if I could have used the CA analysis to detect previous bugs that were found by hand.

6.3 Performance Details

New debugging techniques will not be adopted by the community unless they are efficient, and they work for real-world programs. The Jikes RVM ARM/Linux backend used in this case study is 5900 lines of code (loc). This is comparable to other non-optimizing backends for the RVM system, which vary between 3800 and 6200 loc. For most of the metrics, the system completes the analysis in under 5 seconds. This includes the time to parse the Java source code, to calculate the CA information, to generate the HTML output and to render the initial HTML output in a web browser window. The actual calculation of CA information is a small proportion of this total time. The only slow metrics are methodbodies and compound metrics involving methodbodies, since these must perform many expensive string similarity computations. In these cases, CA takes over 10 minutes on the ARM backend. However, the tests show that it is not normally necessary to use methodbodies, since cheaper metrics provide as good or better results.

7 Related Work

7.1 Syntax-Directed Concept Assignment

Section 2 discussed various other approaches to CA. This section concentrates on syntax-directed CA and its recent applications. Cimitile et al [4] analyse the structural characteristics of source code, via abstract syntax trees. They divide a program into clusters, which are suggested units for manual CA. Kontogiannis et al [15] use source code metrics to identify similar sections of code, which are then identified as possible clones or concepts. Again, the tool suite requires programmer intervention as soon as the concepts are identified. Kontogiannis et al use some different metrics to this paper, but similar clustering behaviour is induced.

The contrast between these two approaches and my tool is that I leverage the syntax-directed CA information to detect inconsistencies, which may indicate bugs.

Other syntax-directed techniques detect similar sections of code in order to facilitate procedural abstraction for code size reduction. For instance, Johnson uses pattern matching on a data dependence graph [14]. However, these techniques are not applied to debugging.

7.2 Code Generator Debugging

There are few specialized debugging tools for code generators. For instance, Engler [6] laments the lack of symbolic debugging facilities for his VCODE code generation system. He states that, 'Debugging dynamically generated code requires stepping through it at the level of host-specific machine code ... Making sense of the debugging output is challenging.'

Obviously, general purpose debugging tools can apply, but for the reasons listed in Section 1 they are not always as effective as for general purpose programs. One specialized code generator debugger is bugpoint, for the LLVM compiler infrastructure [16]. Unlike the approach in this paper, bugpoint is a dynamic analysis tool. It compiles and runs code using 'good' and 'bad' versions of the code generator to try and discover inconsistencies. This theme that 'inconsistencies imply bugs' is common to my work, but all other principles are very different.

Compiler backends are often derived automatically using formalisms such as bottom-up rewrite systems (BURS) [8] or attribute grammars [10]. There appears to be a thriving field of research in verification of these systems. For instance, Glesner shows how to use program checking to ensure the correctness of compiler backends generated by BURS [11]. Ikezoe et al use algorithmic debugging and program slicing to detect bugs in attribute grammar specifications of compiler backends [13]. However, the CA-based debugging technique is more widely applicable than these formal approaches, since it can be used for human-generated compiler backends, which are common in retargetted compilers.

8 Concluding Remarks

There are two key points in this paper.

1. Syntax-directed CA works for code generators.

2. Anomalies in syntax-directed CA indicate possible bugs.

This research leverages these two insights to produce an automated debugging technique for code generators. A prototype implementation finds 80% of injected bugs in an experimental code generator module.

In future, I hope to apply this tool to other RVM backends. The same manual CA is valid, so its acquisition cost is amortized over many applications of the tool.

This debugging technique is lightweight and easily retargetable. No deep domain-specific knowledge is required. The manual CA is domain-specific, but this is often obtainable with relative ease from design documentation or similar specification material. One approach that I have not yet investigated is to search for inconsistencies between different automated CAs. If this also highlights potential bugs, then it completely obviates the need for domain-specific knowledge encoded in manual CA.

Note that the technique is not exclusive to code generator debugging. It should apply to other source code modules that have a similar style, with small, stereotypical units of code.

However, the fact that CA-based debugging is effective indicates significant flaws in current code generator development practice. Code generators should instead be constructed by means of abstraction layers. Ideally, a code generator generator would auto-generate source code from a high-level specification file. This automated approach is in use, for instance [8], but the practice is far from universal. Note that CA-based debugging could be relevant even for high-level specification files. It should still be possible to detect patterns that indicate conceptually related sections of the specification.

The underlying message of this paper is that concepts are a useful debugging aid. I hope that concepts will become more widespread. Two future developments seem promising.

1. Programmers and tools may drive concepts explicitly through the compilation cycle. Concepts could be represented in a conceptual assembly language in a similar manner to types in typed assembly language [19].

2. Runtime CA should be possible. In effect, the debugging technique in this paper identified sequences of machine instructions statically, in code generator source code. It should not be much more difficult to identify sequences of machine instructions dynamically, in program execution traces.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes research virtual machine project: Building an open source research community. *IBM Systems Journal*, 44(2):1–19, Feb 2005.

[2] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, pages 482–498, 1993.

[3] M. Chen. A Java virtual machine for the ARM processor. Master's thesis, University of Manchester, 2004.

[4] A. Cimitile, A. R. Fasolino, and P. Maresca. Reuse reengineering and validation via concept assignment. In *Proceedings of the Conference on Software Maintenance*, pages 216–225, 1993.

[5] A. v. Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of the First International Workshop on Refactoring: Achievements, Challenges, Effects*, 2003.

[6] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 160–170, 1996.

[7] J. Fitzpatrick. Applying the ABC metric to C, C++, and Java. *C++ Report*, Jun 1997.

[8] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG: fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, Apr 1992.

[9] S. Furber. *ARM System Architecture*. Addison-Wesley, 1996.

[10] M. Ganapathi and C. N. Fischer. Description-driven code generation using attribute grammars. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 108–119, 1982.

[11] S. Glesner. Using program checking to ensure the correctness of compiler implementations. *Journal of Universal Computer Science*, 9(3):191–222, Mar 2003.

[12] N. Gold and K. Bennett. Hypothesis-based concept assignment in software maintenance. *IEE Software*, 149(4):103–110, 2002.

[13] Y. Ikezoe, A. Sasaki, Y. Ohshima, K. Wakita, and M. Sassa. Systematic debugging of attribute grammars. In *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000.

[14] N. E. Johnson. Code size optimization for embedded processors. Technical Report 607, University of Cambridge Computer Laboratory, Nov 2004.

[15] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1–2):77–108, 1996.

[16] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 75–88, 2004.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[18] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[19] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[20] K. Nørmark. Elucidative programming. *Nordic Journal of Computing*, 7(2):87–105, 2000.

[21] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, Jan 1990.

[22] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 2nd edition, 2000.

[23] G. Wright, A. El-Mahdy, and I. Watson. *Java Microarchitectures*, chapter 10: Java Machine and Integrated Circuit Architecture (JAMAICA), pages 187–206. Kluwer, 2002.