Introduction to BUG Finder for C Language

Ms.Warsha M.Choudhari
Professor, Information Technology
Datta Meghe Institute of Engineering,
Technology & Research, Wardha, India

Mrs. Rinku S. Astankar
Professor, Computer Science & Engineering
ITM College of Engineering, Nagpur, India

Ms. Mausami Sawarkar
Professor, computer Science & Engineering
J.L.Chaturvedi college of Engineering, Nagpur

Ms. Shalini Kharkate
Professor,Computer Science & Engineering
Govt.Polytechnic, Bramhapuri

Abstract— **Today need better tools for C, such as source browsers, bug finders, and automated refactorings. The problem is that large C systems such as Linux are software product lines, containing thousands of configuration variables controlling every aspect of the software from architecture features to file systems and drivers. The challenge of such configurability is how do software tools accurately analyze all configurations of the source without the exponential explosion of trying them all separately. C tools need to process two languages: C itself and the preprocessor. The latter improves expressivity through file includes, macros, and static conditionals. But it operates only on tokens, making it hard to even parse both languages. SuperC is a complete, performant solution to parsing all of C. First, a configuration-preserving preprocessor resolves includes and macros yet leaves static conditionals intact, thus preserving a program's variability. Detecting linker errors across all compilation units in the Linux kernel demonstrates each part of the Cilantro framework and is evaluated on the Linux source code.**

***Keywords*: Cilantro framework, SuperC**

1. INTRODUCTION

First, Kmax determines the complete set of compilation units and their presence conditions, then SuperC's capacity for implementing semantic analysis is used to detect errors within those compilation units. Abaleal. showed that bugs are caused by Linux's variability and lack automated tool support; they found already-patched bugs by looking through the Linux kernel mailing list [4].

2. CROSS-CONFIGURATION BUG FINDERS

There are five challenges to implementing a cross-configuration bug finders for all variations of a C project like Linux.

(1) The tool needs to find the feature model that defines constraints on feature selection, because not all configurations are valid builds of the software product line. Kmax extracts Linux's feature model from the Kconfig files using the Kconfig parser bundled with the Linux build system.

(2) The tool needs to find all compilation units comprising the source code, in order to perform
project-wide analyses.

(3) The tool needs to find the presence conditions of the compilation units, because the build system chooses compilations units according to feature selections. Kmax also handles these two challenges by extracting all compilation units and their variability information.

(4) The tool needs to first parse the compilation units across all configurations, which SuperC does.

(5) Finally, the tools needs to perform cross-configuration static analysis.

Build a linker error bug finder. This requires not SuperC's semantic actions, but also Kmax's ability to extract variability from the build system. Linker errors happen when one compilation unit calls a function that has no definition. C's separate compilation is used for modularity, and a compilation unit roughly defines a set of related functions. Compilation units that use these functions include a header that declares the imported functions, but the definition of the function is not available until link time. Even if the function definition exists in some compilation unit, a linker error is still possible if there is a configuration in which that compilation unit is excluded by the build system.

Such errors are typically only discovered when building the errant configuration in which the error appears, making it difficult to check for such errors. With Kmax's ability to clean the presence conditions
of compilation units.

1. Data structures for implementing semantic analysis across all configurations with Super C's crossconfiguration parsing framework,
2. Implementations of a project-wide linker error bug finder, and
3. An evaluation of the linker error finder on the complete Linux kernel.

3. SEMANTIC ANALYSIS

Semantic actions are functions that run when the parser recognizes a specific grammar construct. As with the bison parser generator, SuperC supports actions written directly in the grammar specification file. Semantic actions are often used to build an AST during parsing. SuperC supports declarative AST generation, so writing such semantic actions is unnecessary. However, semantic actions can also be used for bug finding.

```
1 #ifdef CONFIG_CRYPTO_BLKCIPHER
2 void *crypto_alloc_ablkcipher()
3 {
4 return (void*)0;
5 }
```

```
6 #endif
7 #ifdef CONFIG_CRYPTO_TEST
8 static void test_cipher()
9 {
10 crypto_alloc_ablkcipher();
11 }
 12 #endif
```

Figure 3.1: An example of a variability bug from the variability bug database by Abalet al.

On line 2, the function crypto_alloc_ablkcipher is defined only if the CONFIG_CRYPTO_BLKCIPHER feature variable is defined. Line 10 makes a call to crypto_alloc_ablkcipher inside the function test_cipher. But text_cipher, defined on line 8, is only defined when CONFIG_CRYPTO_TEST is enabled. With two boolean features, there are four possible configurations of this code block, and all of these configurations compiles correctly except one. When CRYPTO_CRYPTO_TEST is enabled but CONFIG_CRYPTO_BLKCIPHER is not, there is an unidentified symbol error; line 10 calls crypto_alloc_ablkcipher, which is not defined in this configuration.

Then, checking for a bug in some configuration is checking whether the boolean expression is there some combinations of features that leads to the bug, i.e., satisfiability.

```
1 #ifdef A
2 duped int x;
3 #else
4 int x;
5 #endif

6 int main() {
7 int y;
8 x *y;
9 }
```

Figure 3.2: An example of the same C identifier declared as a typedef name in one configuration, but a variable in another.

Since SuperC's parser already tracks configurations, performing bug checking during parsing is ideal. Each subparser maintains the current presence condition it's parsing, and semantic actions in the grammar are executed by the subparsers as usual; actions are written in-line in grammar productions and are executed after the grammar construct is recognized. Each subparser maintains its own parsing context, allowing it not only to parse constructs from a different configurations but to record configuration-specific semantic information, such as symbol definitions. Subparsers are temporary, being created and destroyed by forking and merging as new configuration are encountered. Being managed by the subparser, the parsing context must follow suit, and SuperC provides an interface for implementing cross-configuration a parsing context. It has hooks to fork and and merge corresponding to subparser forking and merging. To store semantic information for bug finders, the parsing context is used to manage a conditional symbol table. As in SuperC and Kmax, this a conditional symbol table maps identifiers to each possible value across all configurations.

To illustrate how cross-configuration semantic analysis works in practice, illustrate the implementation of typedefs, because SuperC's C parse already performs some semantic analysis to support them. This context-sensitive aspect of the C language requires maintaining a table of typedef names and ref-erencing it, to reclassify identifier tokens as typedef names during parsing. Implementing this behavior using semantic actions. Typedef declarations take a C identifier and convert it to a typename, making C a context-sensitive language, which cannot be recognized by context-free parser generators without extra support. Typedefs are context-sensitive, because the same string can be recognized with a different grammar construct depending on whether an identifier has been declared a typedef name earlier in the program.

Figure 3.2 illustrates this context-sensitivity. Line 2 defines x as a typedef name if A is true, otherwise it is a variable. The statement on line 8 is either a multiplication expression, when both x and y are C identifiers, or it is a pointer declaration when x is a typedef name. Implementing typedefs for a single configuration is simple: the parsing context maintains a symbol table of typedef declarations, mapping C identifiers to a boolean flag. A semantic action embedded with the declaration grammar construct looks for the typedef keyword and maps the declared identifier to true in the symbol table. When reading tokens from the lexer, the parser consults this symbol table and reclassifies identifiers to typedef names tokens as necessary. Like variables, typedef declarations may appear in any lexical scope, so the parsing context maintains scope. Our implementation of the context uses a stack of symbol tables to represent scope, which makes sharing context between forked subparsers as it does with the LR state stack. As a further optimization, even forked subparsers point to the same symbol table, possible because the subparsers' presence conditions are always mutually exclusive; any updates to the symbol table are independent. The only time subparsers point to different symbol tables is when they enter a new scope or leave the scope after forking.

SuperC's parsing context interface support arbitrary implementations of cross-configuration semantic information, with hooks called by the parser upon forking and merging. The interface contains the following methods:

1. forkContext creates a new context and is called when SuperC forks a subparser.
2. mayMerge determines whether two contexts allow merging, if not, SuperC will delay merging subparsers until their contexts allow, for instance by delaying a merge until the subparsers return to the same lexical scope.
3. mergeContexts combines two contexts, merging their state, and is called when SuperC merges two subparsers.
4. reclassify takes a token and changes or adds the token and is used to implement typedef names.

Illustrate how semantic state is processed and stored while parsing the typedef example Figure 3.2. On line 1, SuperC forks two subparsers, one to enter the #ifdef branch under the A presence condition and one to enter the #else under the mutually-exclusive :A presence condition. Initially, the parsing context contains an empty table. After parsing their respective branches, each subparser encounters the same semantic action for declarations. By default, the x keyword is mapped to false. The first subparser, seeing the

typedef keyword, updates the entry for x in the symbol table. The subparser computes the new
entry by disjoining its own presence condition, Csubparser with the original presence condition in the table, Coriginal, i.e.,
Cnew ←Coriginal V Csubparser

The new condition for x in the table becomes ⊥ V A. This means that x is a typedef whenever the expression A is true. The second subparser sees that that x is declared as a variable and removes this configuration from the entry by conjoining the negation of its presence condition, i.e.,
Cnew ←Coriginal ^ ¬subparser

Since the #else branch's presence condition is :A, the new condition becomes (⊥ V A) ^ ¬(¬A), which when simplified, is still A. After the static conditional, the subparsers merge, leaving a single parser on line 6. Parsing continues until line 8, which uses the x identifier. The parser consults the symbol table to find that the identifier is a typedef in only some configurations, and forks two subparsers, one for the typedef presence condition and one for the non-typedef presence condition. The same principles used to support typedef names apply to cross-configuration bug finders, albeit with more semantic information and extra semantic actions. For example, to support detection of undefined symbol uses, the bug finder deduces whether there exists some combination of features where the undefined symbol gets used. It models the bug by taking the presence condition Cdef under which the symbol is defined and the presence condition Cuse for a use of the symbol and constructs the following expression:
Cuse ^ ¬Cdef

If the above expression is satisfiable, then there is some configuration where the bug exists. Further constraints to the set of configurations may be conjoined to the expression, for example, the Kconfig feature model.

```
1 #ifdef CONFIG_TRACING
2 void trace_dump_stack(int skip) {
3 // do something
4 return;
5 }
6 #else
7 static inline void trace_dump_stack(void) { }
8 #endif
9
10 int main(int argc, char** argv) {
11 trace_dump_stack(0); // ERROR
12 return 0;
13 }
```

Figure 3.3: An example of an error caused by the wrong number of arguments to a function that only appears on one configurations found by Abal et al [3].

The undefined symbol finder updates the parsing context in the same way that the typedef implementation does, except that the symbol table stores the conditions in which the symbol is defined. To use this information, a new semantic action for C expressions, where functions and variables get used, constructs the model for the bug's presence condition and uses a SAT solver to determine whether the bug appears in any configuration.

To store more semantic information for more sophisticated bug finders, a conditional symbol table is useful. The conditional symbol table is useful for all variability-aware tools, including Kmax and SuperC themselves. A conditional symbol table maps keys to a list of values, where each value is tagged with a presence condition. Figure 3.3, also from Abal etal, is a function that has a different number of arguments depending on the configuration. To create a finder for this bug, a conditional symbol table stores an entry for each possible number of arguments and its presence condition. After parsing the mutually exclusive function definitions on lines 1–8, the symbol table maps the trace_dump_stack function to two entries, one entry records one argument under the CONFIG_TRACING presence condition and other entry records zero arguments for ¬ CONFIG_TRACING. A semantic action function calls checks for this bug. It takes the presence condition at the call site on line 11, which passes one argument to trace_dump_stack. The finder collects the presence conditions for all symbol entries other than the entry recording one argument, and conjoins it with the presence condition at the call site to deduce whether any configurations have a bug. Figure 3.3 does have a bug when CONFIG_TRACING is not enabled.

4. CONCLUSION:

SuperC and Kmax are components of all variability-aware software tools. It introduces the fork-merge parsing context, which enables a cross-configuration parser to maintain state while sub parsers fork and merge. Symbol tables for semantic analysis, as with other configuration-preserving tools, are conditional symbol tables that maintain state for all configurations simultaneously. With only SuperC and these data structures, cross-configuration bug finders are possible by modeling the conditions of bugs with a boolean expression and using a SAT solver to discover the erroneous configurations.

REFERENCES

[1] N. Andersen, K. Czarnecki, S. She, and A. W˛asowski. E_cient synthesis of feature models. In Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC '12, pp. 106–115, New York, NY, USA, 2012. ACM.
[2] T. Berger, S. She, K. Czarnecki, and A. Wasowski. Feature-to-code mapping in two large product lines. Tech. report, University of Leipzig (Germany), University ofWaterloo (Canada), IT University of Copenhagen (Denmark), 2010.
[3] T. Berger, S. She, R. Lotufo, A. W˛asowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, pp. 73–82, New York, NY, USA, 2010. ACM.
[4] A. Bessey et al. A few billion lines of code later: Using static analysis to find bugs in the real world. CACM, 53(2):66–75, Feb. 2010.
[5] R. Bowdidge. Performance trade-o_s implementing refactoring support for Objective-C. In Proceedings of the 3rd WRT, Oct. 2009.