Parallel SMT-Constrained Symbolic Execution for Eclipse CDT/Codan

Andreas Ibing

Chair for IT Security TU München
Boltzmannstrasse 3, 85748 Garching, Germany

Abstract. This paper presents a parallel symbolic execution engine as a plug-in extension to Eclipse CDT/Codan. It uses the CDT parser and the control flow graph builder from CDT's code analysis framework (Codan). Path satisfiability and bug conditions are checked with an SMT solver in the logic of arrays, uninterpreted functions and nonlinear integer and real arithmetic (AUFNIRA). Each worker of the parallel engine keeps the symbolic program states along its current program path in memory, to allow for quick backtracking. Dynamic redistribution of work between workers is enabled by splitting a worker's partition of the execution tree at the partition's top decision node, where a partition is defined by the start path leading to its root control flow decision node. The runtime behaviour of the parallel symbolic execution engine is evaluated by running it on buffer overflow test programs from the NSA's Juliet test suite for static analyzers. Both the speedup of backtracking the symbolic program state over a previous single-threaded implementation with path replay and the speedup with an increasing number of workers are investigated.

1 Introduction

Symbolic execution (SE, [1]) is an attractive approach for automated discovery of common software weaknesses. SE treats program input as variables and translates operations on them into logic equations. For a path through a program, SE builds a path constraint from the control flow decisions. Path satisfiability and the presence of bugs is decided with an automatic theorem prover (constraint solver [2,3]). Current SE tools normally rely on Satisfiability Modulo Theories (SMT, [4]) solvers. A more detailed overview of the current state and available tools is given in [5,6].

Many SE tools first transform the source code into an intermediate representation (IR) and run the symbolic execution on the IR. In [7], C/C++ code is compiled into LLVM [8] bytecode before symbolic execution, while [9] uses CIL [10] as intermediate code. [11] analyzes Java bytecode with symbolic execution.

In order to achieve high code coverage in a limited time, parallelization of SE has been investigated. [12] presents a parallelized version of [11], which initially performs a breadth-first exploration of the symbolic execution tree up to a certain depth, and then runs multiple workers on disjunct static partitions of the execution tree. [13] presents a parallelized version of [7] with dynamic redistribution of work between workers.

While SE of intermediate code does have its advantages, there is also a motivation for symbolic execution of source code: an IR loses source information by discarding high-level types and the compiler lowers language constructs and makes assumptions about the evaluation order. However, rich source and type information is needed to explain discovered bugs to the user [14].

In order to detect errors as early as possible, bug detection tools should be integrated into IDEs. The Eclipse IDE is widely used, open source and designed for extensibility (OSGi architecture [15]). For C/C++ development, Eclipse CDT features a code analysis framework (Codan, [16]), which includes a control flow graph (CFG) builder and several code checkers. Codan does not, however, feature path-sensitivity or symbolic execution, which may lead to detection inaccuracies for many analyses (false negative and false positive detections).

This paper presents a parallelized SMT-constrained symbolic execution engine with dynamic work redistribution and backtracking of symbolic program states as plug-in extension for Eclipse CDT. It builds on previous work [17], which developed a sequential SE engine with replay of start paths after backtracking path decisions. The remainder of this paper is organized as follows. Architecture and design are described in section 2. Section 3 evaluates the implementation with buffer overflow test programs from the Juliet test suite [18] and benchmarks both the speedup of backtracking symbolic program states over [17] and the speedup with a varying number of workers. Section 4 discusses the results.

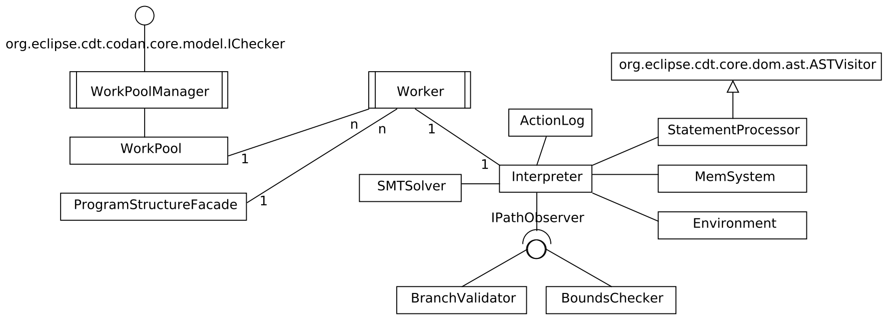

Fig. 1. Overview of main classes. WorkPoolManager and Worker are active classes.

2 Architecture and Design

2.1 Trade-offs in Memory, Computation, Communication and Parallelism

The exploration of (at least a finite) execution tree can in principle be performed in a straight-forward manner with the Worklist algorithm [19], in which unexplored tree nodes (frontier nodes) are put together with the corresponding symbolic program state in a work list (queue-based tree traversal). In practice however, this may lead to memory exhaustion, even in distributed computation setups.

An alternative to reduce memory consumption is to store program paths without symbolic program states. A program path can further be compressed as a sequence of branches. Minimum memory consumption is achieved when only one path and one program state are kept in memory. This approach was used in [17] for a sequential implementation with depth-first exploration of the execution tree (with a configurable loop depth bound). Restoration of a program state for an unexplored (frontier) node, on the other hand, requires redundant computation along the new start path.

The memory versus computation trade-off extends to storing the history of program states along a path (which requires memory) versus the possibility of restoring a program state on this path by backtracking (which avoids redundant computation). The symbolic program state can be backtracked as far as the required information (variable definitions, equations, etc.) is available. Thus, there are effectively three possibilities:

Path replay: only the current symbolic program state is kept in memory, with the possibility for garbage collection of dead symbolic variables. The current path's control flow decisions are used to generate the next path [17].

State cloning: the open symbolic program states (frontier states) are kept in memory. The program state at a decision node is cloned for each child branch node. This is used in a distributed implementation in [13].

State backtracking: program states along the current path are kept in memory. This can be efficiently implemented using single assignment form and not garbage-collecting dead symbolic variables. This approach is used here.

In a parallelized implementation, it is desirable to balance computation complexity and communication complexity. The communication complexity can be rated differently for communication between multiple threads on a shared memory architecture (multi-core and/or hyper-threading CPUs) versus network communication in a distributed setup. There are basically two possibilities:

- symbolic program states are transmitted to new or idle workers (requires state cloning), or
- start paths are transmitted to new or idle workers (less transmitted data, but path replay is needed).

The most adequate parallelization depends on the available hardware resources as well as the size of the software to be analyzed.

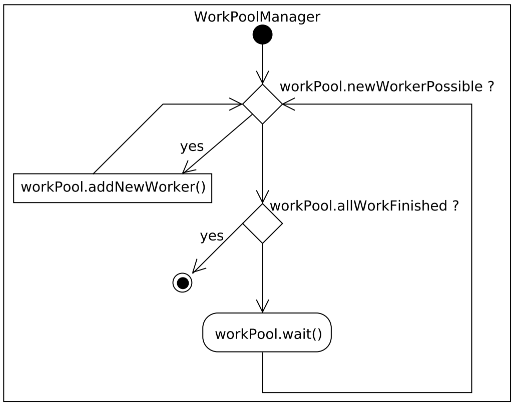

Fig. 2. Activity diagram of WorkPoolManager. WorkPool is used as synchronization object.

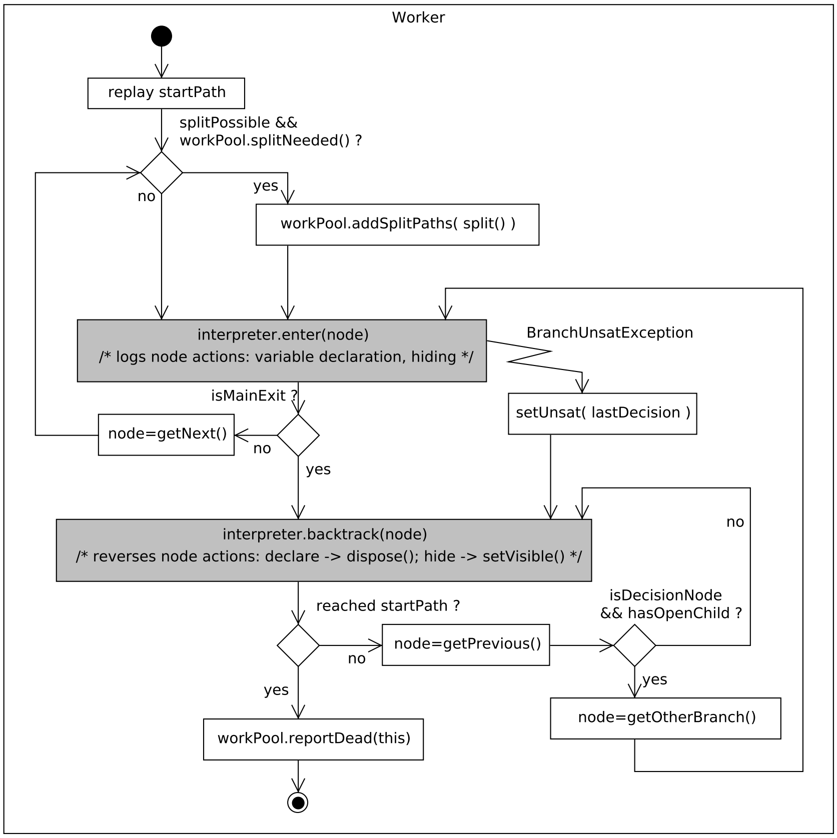
Worker
replay startPath
splitPossible && workPool.splitNeeded() ?
no
yes
workPool.addSplitPaths(split())
interpreter.enter(node)
/* logs node actions: variable declaration, hiding */
BranchUnsatException
isMainExit ?
node=getNext()
no
yes
setUnsat(lastDecision)
interpreter.backtrack(node)
/* reverses node actions: declare -> dispose(); hide -> setVisible() */
no
reached startPath ?
isDecisionNode && hasOpenChild ?
node=getPrevious()
no
yes
yes
node=getOtherBranch()
workPool.reportDead(this)

Fig. 3. Activity diagram of Worker. WorkPool is used as synchronization object.

2.2 Design Decisions

The symbolic execution engine performs whole-program analysis with a call string approach [20] for interprocedural analysis.

Symbolic Program State and Backtracking. The translation into SMT logic is the same as in [17]. For program variables, the interpretation generates symbolic variables, whose values are logic formulas in SMTLIB AUFNIRA syntax. The formulas may contain other symbolic variables as terms. Symbolic variables must not be overwritten (no destructive assignments) if they might still be needed in other formulas. A variable is called 'live' if it may still be needed in the future, and 'dead' otherwise. Therefore single assignments are used: for each assignment to a program variable, a new symbolic variable version with unique name is generated. Pointers and structs are not directly translated into SMT logic, they are represented internally during interpretation (e.g. a pointer has a target and an offset formula). Logic equations are generated at pointer dereference and at field access to a struct. A path through a program is a sequence of control flow graph nodes along edges in the CFG, and function calls are treated as edges between different functions' CFGs. A symbolic program state comprises all declared symbolic variables and the internal representation of pointers and structs along a program path. Due to single assignment form, a symbolic program state contains all previous states along the path. To allow for backtracking, an ActionLog keeps track of the actions performed during interpretation of each CFG node on the path. An action may be the declaration of a symbolic variable or hiding stack variables at the exit from a function. Through backtracking, 'dead' variables may become 'live' again. To backtrack a CFG node, the actions are reversed: a declared symbolic variable is disposed, and hidden variables are set visible again (in case of backtracking a function exit).

Execution Tree Exploration and Splitting into Subtrees. A configurable number of workers analyzes disjunct partitions of the execution tree. Each worker performs a depth-first exploration of its partition with backtracking of the symbolic program state. For dynamic work redistribution, a worker can split its partition at the partition's top decision node. The child branches not taken by the current worker are returned as start paths for other workers. After a split, the partition start path is adjusted (prolonged by one branch node). Analysis starts with one worker, who splits until the configured number of workers is busy. A worker is initialized by replaying its partition start path. The maximum loop depth to be explored can be bounded. If a worker reaches an unsatisfiable branch or a satisfiable leaf of the execution tree, it backtracks and changes a path decision according to depth-first tree traversal. If backtracking reaches the end of the partition start path, the partition is exhausted. The algorithm is illustrated in the activity diagrams Fig. 2 and Fig. 3.

2.3 Main Classes

A diagram of the main classes is shown in Fig. 1. The implementation is multi-threaded, where control flow graphs and syntax trees are shared between worker threads.

WorkPoolManager extends Codan at the extension point org.eclipse.cdt.codan.core.model.IChecker. The WorkPoolManager starts workers and reports found errors through the Codan interface to the Eclipse marker framework.

ProgramStructureFacade provides access to control flow graphs.

WorkPool is used as synchronization object (synchronized methods). It is used to track the number of active workers and to exchange split paths.

Worker has a forward and a backward (backtracking) mode. It passes references to control flow graph nodes for entry (forward mode) or backtracking to the Interpreter.

Interpreter follows the tree-based interpreter pattern [21]. SMT syntax is generated by the StatementProcessor (which implements CDT's ASTVisitor) by bottom-up traversal of AST subtrees (visitor pattern), which are referenced by CFG nodes. Symbolic variables are stored in and retrieved from MemSystem. Backtracking additionally relies on ActionLog, which links certain actions to nodes on the current path, like hiding stack variables at function exit. The Environment class provides symbolic models of Standard library functions. The interpreter further offers an interface to BranchValidator and to checker classes.

SMTSolver wraps the interface to the currently used external solver, which is [22].

BranchValidator is triggered when entering a branch node. It generates a satisfiability query for the path constraint. For an unsatisfiable branch it throws an exception, which is caught by the worker.

BoundsChecker is triggered for memory access. It generates satisfiability queries for violation of lower and upper buffer bounds and reports an error in case of satisfiability.

2.4 Communication and Synchronization

Activity diagrams for the active classes are shown in Fig. 2 and Fig. 3. Synchronization of multiple local worker threads for sharing control flow graphs and abstract syntax trees (ASTs) relies on the following methods:

WorkPool all methods are synchronized. The WorkPoolManager waits if the configured number of workers is busy or no further split path is available and is notified for changes (compare Fig. 2).

ProgramStructureFacade offers synchronized methods to retrieve CFG references.

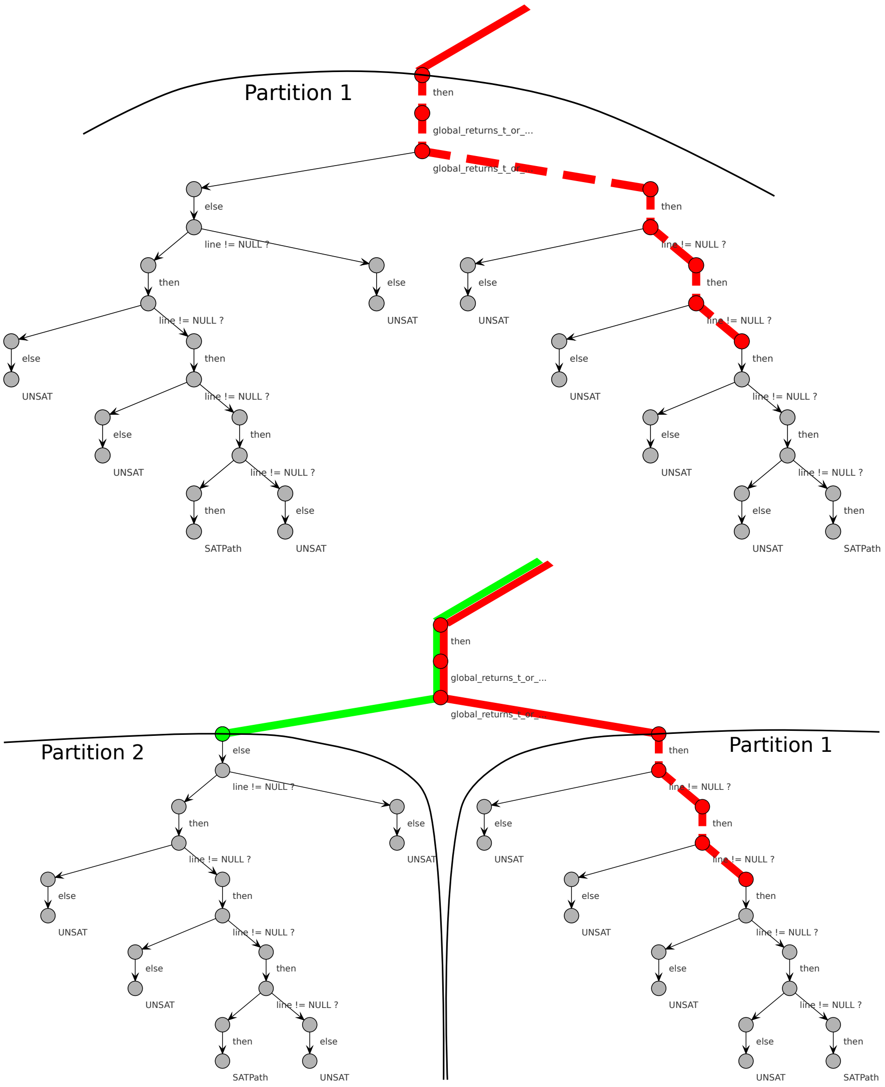

Fig. 4. Illustration of the partition split operation. Red indicates worker 1, green worker 2. Partition start paths are indicated with solid lines, the current worker state with a dashed line. Partition borders are indicated by curves and the text "Partition n". Unexplored parts of the execution tree are shaded.

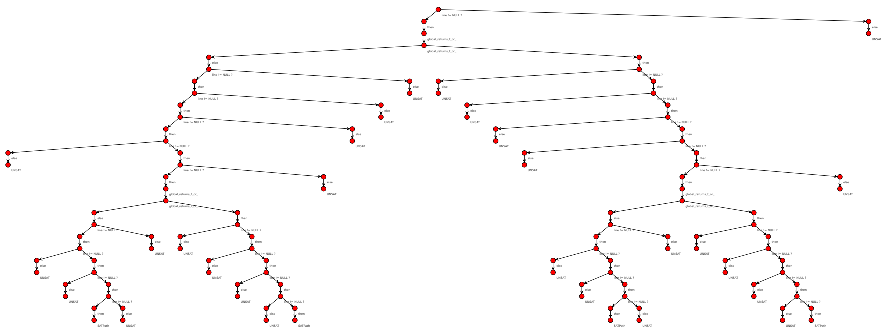

Fig. 5. Example execution tree (showing only decision and branch nodes) for test program CWE121_char_type_overrun_memcpy_12 from the Juliet test suite [18]. The subtree on the lower left was used to illustrate the partition split operation in Fig. 4.

AST nodes are not thread-safe, so Workers lock AST subtrees at the CFG node level (the AST subtree which is referenced by the currently interpreted CFG node).

Index each CDT project has an index, which is the persisted document object model (DOM). Access results in (possibly blocking) I/O operations on a database stored in small files, so that workers acquire a read lock for accesses.

The split operation is illustrated in Fig. 4, which shows part of the execution tree of the Juliet test program CWE121_char_type_overrun_memcpy_12. The tree shows only decision nodes and branch nodes; other CFG node types are not shown. A path through the execution tree normally contains alternating decision nodes and branch nodes. This example tree contains an exception because of a function call expression in a decision node (corresponding to a function call in the condition of an if-statement). This leads to a repetition of the decision node as a call node, so that the interpreter can conveniently continue interpretation with the return value.

The unexplored part of the (sub)tree is shaded. Red lines indicate the paths which have been explored by worker 1, green lines correspond to worker 2. Partition start paths are shown as solid lines, and a dashed line indicates the current position of a worker. Worker 1 splits its partition and generates a split path, which becomes start path for worker 2. After the partition split, worker 1's start path is prolonged by one branch node.

The execution tree is normally not generated during analysis, it is only traversed on-the-fly. The complete execution tree for this example is shown in Fig. 5.

2.5 Visualization

CFGs and explored execution trees can be visualized with the Java Universal Network/Graph library (JUNG, [23]) and exported as vector graphics with Apache Batik [24]. These two libraries are therefore loaded as Eclipse plug-ins. Execution tree visualization has been used for Fig. 4, 5.

Table 1. Duration of analysis for two sets of test programs from Juliet (on a quad-core processor). Shows increase of analysis speed for backtracking the symbolic program state over path replay and for different number of worker threads.

	CWE121_memcpy (18 test programs)	**CWE121_CWE129_fgets** (30 test programs)
Single-threaded with path-replay [17]	131 s	1026 s
Backtracking, 1 Thread	25 s	102 s
Backtracking, 2 Threads	27 s	58 s
Backtracking, 3 Threads	31 s	54 s
Backtracking, 4 Threads	33 s	54 s

3 Evaluation

The parallelized symbolic execution engine is evaluated with stack based buffer overflow test programs from the Juliet suite [18]. In order to achieve a certain coverage of bugs, language constructs and context depths, Juliet combines 'baseline' bugs with different control and data flow variants into test programs. The test programs contain 'good' functions in addition to 'bad' functions to provide enough possibilities for false positive detections. Juliet contains 39 flow variants for C programs, and the maximum context depth spanned by a flow variant (flow 54) is five functions in five different source files (necessary context depth for accurate bug detection for that flow). The flows are not numbered consecutively.

Two sets of test programs are used, which contain buffer overflows with the *memcpy* (set 1) and *fgets* (set 2) functions. Analyses are run with time measurement as JUnit plug-in tests in Eclipse. Run times are only evaluated for those programs for which bug detection is accurate, i.e. no false positives and no false negatives. Therefore flow 18 is excluded, because it contains a goto statement which leads to an exception in the current version of the CFGBuilder, resulting in a false negative detection. Flow 54 uses unions, which is not yet implemented in the translation to SMT syntax, also resulting in a false negative. A false positive occurs for flow 66, because the current solver version (version 5.1.9 is used) gives an incorrect satisfiability answer for the corresponding mixture of array logic and arithmetic. On the other hand, accurate detection is achieved for flow 12: the solver reports that the contained modulo function is not yet implemented, but luckily guesses the correct satisfiability answer. As in [17], bugs have been accurately detected with 36 of the 39 C flow variants (90%), while the percentage of detectable 'baseline' bugs is unsatisfying, because only a small part of the standard library functions is interpreted.

Table 1 shows benchmarks for single-thread execution with path replay, single-thread execution with backtracking, and multi-threaded execution with partition splitting for a varying number of threads. The plug-in is run in Eclipse 4.2 on a Core 2 Quad CPU Q9550 on 64-bit Linux kernel 3.2.0. Even for the tiny test programs, backtracking already shows a 5-10x speedup over path replay.

The overhead of partition splits and thread creation hampers multi-threading speedup, and actually leads to a setback for the tiny *memcpy* programs. The *fgets* test programs contain several loops, which leads to bigger execution trees and a 2x speedup with 3 threads.

4 Discussion

This paper presented a parallelized symbolic execution engine with Eclipse CDT integration and showed significant speedup over a previous sequential implementation. Workers are run as multiple local threads on shared control flow graphs and syntax trees. While the symbolic execution currently aims at path coverage (with a loop depth bound), less comprehensive coverage criteria also need to be supported in order to scale analyses to bigger programs. Future work includes a straightforward extension to a distributed setup with a dynamic two-level hierarchical partitioning of the execution tree (first over Eclipse processes on different machines, then over local threads).

Acknowledgement. This work has been partially funded by the German Ministry for Education and Research (BMBF) under grant 01IS13020.

References

1. King, J.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
2. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
3. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard Version 2.0 (December 2010), `http://goedel.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.12.21.pdf`
5. Cadar, C., Sen, K., Godefroid, P., Tillmann, N., Khurshid, S., Visser, W., Pasareanu, C.: Symbolic execution for software testing in practice – preliminary assessment. In: Int. Conf. Software Eng. (2011)
6. Pasareanu, C., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. Int. J. Software Tools Technology Transfer 11, 339–353 (2009)
7. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX Symp. Operating Systems Design and Implementation (2008)
8. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: Int. Symp. Code Generation and Optimization (2004)
9. Correnson, L., et al.: FRAMA-C User Manual, release oxygen-20120901. CEA LIST (2012), `http://frama-c.com/download/frama-c-user-manual.pdf`
10. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002), `http://dl.acm.org/citation.cfm?id=647478.727796`

11. Visser, W., Pasareanu, C., Khurshid, S.: Test input generation with Java Pathfinder. In: Int. Symp. Software Testing and Analysis (2004)
12. Staats, M., Pasareanu, C.: Parallel symbolic execution for structural test generation. In: Int. Symp. Software Testing and Analysis, pp. 183–193 (2010)
13. Bucur, S., Ureche, V., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: EuroSys (2011)
14. Kremenek, T.: Finding software bugs with the Clang static analyzer. LLVM Developers' Meeting (August 2008), `http://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf`
15. Archer, S., VanderLei, P., McAffer, J.: OSGi and Equinox: Creating Highly Modular Java Systems. Addison Wesley (2010)
16. Laskavaia, A.: Codan- C/C++ static analysis framework for CDT. In: EclipseCon (2011)
17. Ibing, A.: SMT-constrained symbolic execution for Eclipse CDT/Codan. In: Workshop on Formal Methods in the Development of Software (2013)
18. United States National Security Agency, Center for Assured Software: Juliet Test Suite v1.1 for C/C++ (December 2011), `http://samate.nist.gov/SRD/testCases/suites/Juliet_Test_Suite_v1.1_for_C_Cpp.zip`
19. Nielson, F., Nielson, H., Hankin, C.: Principles of Program Analysis. Springer (2010)
20. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnik, S., Jones, N. (eds.) Program Flow Analysis: Theory and Applications, pp. 189–233. Prentice-Hall (1981)
21. Parr, T.: Language Implementation Patterns. Pragmatic Bookshelf (2010)
22. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
23. Madadhain, J., Fisher, D., Smyth, P., White, S., Boey, Y.: Analysis and visualization of network data using JUNG. J. Statistical Software (2005)
24. Apache: Batik Java svg toolkit, `http://xmlgraphics.apache.org/batik/`