A SURVEY ON TESTING CONCURRENT AND MULTI-THREADED APPLICATIONS TOOLS AND METHODOLOGIES

Desoky Abdelqawy, Amr Kamel and Fatma Omara,
Department of Computer Science, Faculty of computer and information
Cairo University,
{d.abdelqawy, a.kamel, f.omara}@fci-cui.eg.edu

ABSTRACT

Recent advances in microprocessor technologies have increased the need for concurrent multi-threaded programming. These advances imposed an immediate mandate in the researcher to develop methodologies and tools to support the development process especially in testing and debugging of concurrent multi-threaded programs. In this paper, we are surveying the available tools and techniques for testing and debugging concurrent and multi-threaded applications as well as highlighting the different ways of implementing these techniques. At the time of writing and as fare as our search had reached we didn't find published surveys that cover these tools techniques since 2006 in [1], [2].

KEYWORDS

Race conditions, concurrency, multi-threaded application analysis, testing and debugging tools, dynamic analysis, static analysis.

1 INTRODUCTION

Recent advances in microprocessor technologies have increased the need for concurrent multi-threaded programming. These advances imposed an immediate mandate in the researchers to develop methodologies and tools to support the development process especially in testing and debugging of concurrent and multi-threaded programs.

Testing concurrent multi-threaded programs is not straightforward due to the nature of concurrent bugs. Concurrency bugs are difficult to detect due to the nondeterministic behaviour exhibited by parallel applications. Even if these bugs are detected they remain hard to reproduce. Further after fixing a bug it's difficult to ensure that the bug was truly corrected and not simply masked.

Data-race, *deadlocks*, *starvations* and *poor application performance* are identified examples of concurrency bugs.

Data-race condition according to [3] it's a situation that occurs when two or more threads can access (either reading or writing) a shared or global resource and at least one of the two accesses is write and according to [2] it's a situation that occurs when program read a variable and then has to take an action based on the contents of this variable and its possible to another thread or process to snake in and change in the variable value between the read and the action in such a way that, the action about to be taken is no longer appropriate. Data-race condition is un-deterministic bug since whether or not a race condition lead to actual error totally depends on the interleaving of the threads during a given run of the programs as well as

its hard to reproduce since its manifest themselves in a corrupted or incorrect variable data.

Deadlocks according to [4] it's an unintended condition in which a set of threads blocks forever because each thread in the set is waiting to acquire a lock already held by another thread in the set. Deadlock sometimes introduced by developer when they are trying to avoid the race-condition using synchronization primitives incorrectly.

Starvation according to [5] it's an infinite delay or permanent blocking of one or more run-able threads; threads that are not being scheduled to run even though they are not blocking or waiting on anything else are said to be starving. Starvation is typically the result of scheduling rules and policies.

Live-Locks according to [5] it's a situation which occurs when threads are scheduled but are not making forward progress because they are continuously re-acting to each other's state changes; High CPU utilization with no sign of real work being done is considered a classic warning sign of a live-lock.

Concurrency type-state bugs according to [6] occurs when type-state-altering operations from one thread are not well synchronized with other operations from different threads, causing the object being applied with un-permitted operations in some type-state, i.e., type-state violation.

In this paper we will survey the different types of concurrent bugs, analyze their features and representing different available strategies and techniques used to discover these bugs as well as highlighting the available tools and its implementation techniques focusing on data-race condition bug detectors.

2 CONCURRENT APPLICATION TESTING APPROACHES AND TECHNIQUES

Concurrent applications 's testing entails testing for correctness, reliability, performance and scalability. According to Testing for correctness is determined through utilizing one of two principles approaches *verifying access order* through allowing no unordered access to a shared resource such that at least one access is writing or *verifying locking disciple* approach through assuring that all accesses to shared variables obey a locking discipline.

Based on the previous two approaches there exist a number of techniques that used to support in

recognizing and handling data-races; these techniques include *race-condition prevention* in which system or programming language designed such that data-race are impossible by definition, *race-condition detection* in which we accept that language allow races but ensure that races are notice or detected [7].

In the following sections we will present the common used strategies for detecting race-condition i.e. race free-typed systems, *static analysis*, *dynamic analysis technique and model-checking* and *hybrid technique*.

3 RACE FREE-TYPED TECHNIQUE

In this technique a type-based system introduced to help eliminating race-condition from the code.

According to [2] a type-based system for preventing race-condition can offer a strong assurance of race-free code. This technique also doesn't suffer from any performance penalties.

The main suffering issue in this technique is the expressiveness of the implemented type system i.e. how fare it will restrict the programmer's ability to use the idioms and coding styles he familiar with it, It's also enforce one discipline which is protecting the shared resources by using the locks and finally its required a burden of annotations.

EPAJ (Extended Parameterized Atomic Java) introduced In [8] as a type system for specifying and verifying atomicity in java programs, which combines atomicity type introduced in [9] with a significantly expressive type system.

4 STATIC ANALYSIS TECHNIQUE

Tools based on static analysis techniques analyze code without actually running the program.

Static analysis immediately finds bugs in obscure code paths that are difficult to reach with testing according to [1], [2].

Locking at metadata from a compiled application or annotated source code performs static analysis. The analysis typically includes some formal inspection to ensure that the intent and assumption of the developer prevent incorrect behaviours some time called annotation.

Static analysis could perform escape analysis to determine shared instance fields in the program and then type analysis can determine the consistency of lock protection using derived or annotated locksets.

Static analysis tends to report a high number of false positives due to the coarse-grained nature of the analysis. The main cause of this high false positive rate is that, the static analysis has an overall knowledge of the program under analysis but has imprecise information with-respect to a particular control follow or interleaving. It may determine two accesses may allies when considering the overall program however its generally lacks the ability to determine which control follow these two accesses will actually occurs; there for the static analysis will treat those accesses as always may-alias even if no feasible control flow at runtime will fulfil this alias relationship.

A static analysis tool that uses flow sensitive, inter-procedural analysis to detect both data race and deadlocks was introduced in [10]. It's explicitly designed to find bugs in large and complex concurrent multi-threaded systems. Its aggressively infers checking information such as which locks protect which operations, which code contexts are multi-threaded, and which shared access are dangerous as well as it tracks a set of code features to sort errors both from most to least severe.

RacerX starts by building a control-flow graph of the entire application. Using a flow-sensitive procedure, the analysis adds locks to the lockset when its encounter locking statements and removes them when its encounter unlocking statements. As the tool encounters accesses of thread-shared variables, it verifies that their locks are in the current lockset, and if not a warning is produced.

RacerX determine which lock should protect which variable through counting the number of times that variable is protected by the giver lock verses the number of times that it is not and if the number of times is statistically significant then an association is created.

Look-Smith a static analysis tool for finding data race in C programs that utilize the context sensitivity greatly reduces the number of false warnings but also limits the tool scalabilities as well as utilizing field sensitivity that's improved both precision and performance and this is because more precise modelling of aliasing speeds up subsequent phases of the tool. They also introduce a lazy field modelling and specific approach to deal with void* pointers was introduced in [11].

5 DYNAMIC ANALYSIS TECHNIQUE

Tools based on dynamic analysis techniques actually execute the program and the bugs are detected by looking at footprint of execution. It operates on runtime and visits only the visible paths and has accurate views of shared resources values and other resources state.

There are two types of dynamic analysis techniques *online* and *offline*. *Online Dynamic analysis* is a technique in which the analysis done while the application is executed. *Offline Dynamic analysis* is a technique in which the analysis is done through recording traces during the execution and analyzes them later to detect bugs.

Dynamic analysis tries to trigger different interleaving in test runs. This could be achieved through randomly seeding sleep statements or systematically exploring all interleaving using an explicit state model checker.

According to [1], [2] Dynamic analysis tool can use very intuitive knowledge about the runtime behaviour of the application in order to increase the precision i.e. it doesn't suffer from false positive but its actually imposes a heavy computational overhead making it's time consuming to run test cases and impossible on programs that have a strict timing requirements. Dynamic race detectors can only find bugs in the execution paths that are actually taken at runtime so it suffers from coverage issue since a random triggering can never guarantees that all partial ordering are tested and a systematic exploration using an explicit static mode checker can lead to program state explosion problem.

The main cause of the previous problem is that dynamic analysis has a precise local knowledge of the program "a particular control flow or interleaving" but little to no overall knowledge of the program.

[14] Dynamic analysis technique based on two major algorithms *lockset* and *happen-before* introduced in [12].

Lockset algorithm based tool potential race is deemed to have occurred if two threads access a shared memory resource without holding a common lock it doesn't depend on thread interleaving. Race is potential because two threads may not have interfered with each other. This approach can be implemented with very low overhead at least when the whole program static analysis are available it doesn't produce false negative but It still has a poor

precision since its produce superfluous false-positives [2].

Happen-before algorithm based tool potential race is deemed to have occurred if two threads access a shared memory location and the accesses are causally unordered in a precise sense as defined in [13]. This approach produces fewer false positives than lockset-based detection; Unfortunately happens- before race detection has proven difficult to implement efficiently.

Eraser a dynamic detector in lockset based multi-threaded programs introduced in [14]. Eraser use binary rewriting technique to monitor every shared-memory reference and verify that consistence locking behaviour is observed. Eraser can only process mutex synchronization operations and its fails when other synchronization primitives are built on top of it. It suffers from poor performance due to the instrumented code and on the fly analysis of race condition. Eraser ownership model extended in [7], [15] Extends Eraser and allow detects races at the object level of each memory locations.

Modern dynamic race detectors combine both approaches lockset and happen-before in order to improve their efficiency. This kind of hybrid dynamic tools introduced in [16], [17] MultiRace and RaceTrace. Both of the tools make specific engineering choices in order to make their analyses more practical to use. For instance, one of the problems with previous dynamic race detectors was an artificially impose granularity on the size of shared memory to be checked. Therefore, RaceTrack and MultiRace both dynamically reduce the detection granularity once a race condition has been detected on an object to reduce the false positives problem. ThreadSentizer introduced in [18] is a dynamic detector tool of data race that combine happen before and locksets approaches as well as introducing what they called a dynamic annotations that allow the user to inform the detector about any tricky synchronization in the program. Its support hybrid dynamic race detection and pure happen-before mode.

Even the best dynamic race detectors have their own share of problems. The general problem of dynamic detectors of any kind is that you have to actually run your program for them to do any good. Dynamic race detectors still can only find bugs in the execution paths that are actually taken at runtime. False positives are still a problem, but the combination of lockset and happens-before analysis has helped to alleviate this issue.

PCT Probabilistic Concurrency Testing introduced in [19] was introduced as a randomized algorithm for concurrency testing. Given a concurrent program and an input test harness, PCT randomly schedules the threads of the program during each test run. In contrast to prior randomized testing techniques, PCT uses randomization sparingly and in a disciplined manner. As a result, PCT provides efficient mathematical probability of finding a concurrency bug in each run. Repeated independent runs can increase the probability of finding bugs to any user-desired level of certainty.

RACEZ introduced in [20] which is a race detection mechanism that uses a sampled memory trace collected by the hardware performance monitoring unit rather than invasive instrumentation. The approach introduces only a modest overhead making it us- able in production environments.

Falcon introduced in [21] proposed a pattern-based dynamic analysis technique for fault localization in concurrent programs that combines pattern identification with statistical rankings of suspiciousness of those patterns and apply this technique to both atomicity and order violations. For each pattern, the technique uses the pass/fail statistics to compute a measure of suspiciousness that is used to rank all occurring patterns, in the spirit of Tarantula in the sequential case.

A data race witnesses algorithm in multithreaded java programs introduced in [22] which based on analyzing a single execution trace. Mainly their idea is to perform a post-mortem analysis on a log of the access events generated from executing one of the implemented data-race algorithms [10], [11], [23] that compute a set of potential data races, that act as input to their witness generation algorithm. Using the generated trace and a set of potential data races. The algorithm models the access events of that trace using suitable classes of constraints and formulating the witnesses generation problem as constraint solving. These constraints represent a maximal set of inter-leavings of events of that trace, and all these alternative traces are guaranteed to be actual program executions. The constraints generated by the their algorithm are in a quantifier-free first-order logic that can be decided by off-the-shelf Satisfiability Modulo Theory (SMT) solvers.

6 MODEL CHECKING TECHNIQUE

According to [5] Model checking is a method for verifying the correctness of a finite state concurrent system. Model checker tries to simulate race and deadlock conditions through exploring every possible execution path for all possible variables in order to determine if certain undesired behaviour might occur in our case data race or deadlock; this simulation is done with using software abstraction that model the control follow and the data values of the program to be modelled.

Using model checking, we can formally prove the absence of races and deadlocks.

According to [3] Race detection is a safety verification problem for concurrent programs a race occurs when two threads can access a shared data variable simultaneously and at least one of the accesses is a write. The program is race-free if no such state is reachable so races can be detected using model checking the major practical obstacle to model checking is the interleaving of concurrent threads cause an exponential explosion of the control state and if the threads can be dynamically created the number of control state is unbound.

A system-C compiler with integrated precise formal race analysis through model checking this compiler introduced in [24] produce a simulator that use the outcome of the analysis to perform partial order reduction and make the model scaled through applying it to a tiny fraction of system-C model. (System-C is a system level modelling language that offers a wide range of features to describe concurrent systems at different level of abstraction).

CHESS introduced in [23] use model-checking for performing concurrency scenario testing of systems programs through using model checking techniques to systematically generate all interleaving of a given scenario it's also scales to large concurrent programs. Chess is able to reproduce an erroneous execution that manifesting the bug.

Model checking provides superior confidence in design and architecture, provide high level of coverage and require minimal external drivers. But at the other hand it’s difficult to automatically extract the model from the code, it’s suffer from state space explosion, may prove that the design is error free but the implementation may still be incorrect.

7 HYBRID TECHNIQUE

Due to the advantage and limitations for each of the individual testing techniques, recent researches have been tried to combine more than one technique to leverage the advantage of each other.

This collaboration may be either *unidirectional* or *bidirectional*.

Unidirectional, where information follows from one technique to another i.e. to have static analysis identifying a set of relevant artefacts that will help guide dynamic analysis. In general, static analysis provides some guidance to dynamic analysis to ensure a low false positive rate by running the relevant interleaving.

Bidirectional, where information transferred between both techniques i.e. one technique provide initial information to the other one which use it during its operation and feedback the other one with results. During this approach, static analysis can tell dynamic analysis more than which instruction should be considered for permutation; static analysis can tell the dynamic analysis also how to efficiently test these permutations at runtime taking advantage of what already been tested. Moreover we can leverage the precise interleaving specific information collected by the dynamic analysis to improve the quality of the static analysis, by taking the flow of control into account.

An iterative technique introduced in [25] in which model checking and static analysis are combined to check large software systems. Static analysis identifies safe statements, which are used by the model checker to perform partial order reduction. The model checker performs state exploration (using partial order reduction) and computes precise alias sets used by the static analyzer to identify safe statements.

A hybrid testing technique introduced in [26], which enable tighter collaboration between static and dynamic technique where static analysis and dynamic analysis interact iteratively throughout the testing process. Static analysis uses coarse-grained analysis to guide the dynamic analysis to concentrate on the relevant search space, while dynamic analysis collects concrete runtime information during the guided exploration. The runtime information provided by the dynamic analysis helps the static analysis to refine its coarse-grained analysis and provides better guidance on dynamic analysis.

Based on fact that, concurrency bugs are caused by non-deterministic interleaving between shared memory

accesses and Their effects propagate through data and control dependences until they cause software to crash, hang, produce incorrect output, etc; So the lifecycle of a concurrency bug thus consists of three phases: (1) triggering, (2) propagation, and (3) failure [27] introduce a ConSeq which is a consequence-oriented approach to improving the accuracy and coverage of state-space search and bug detection. The proposed approach first statically identifies potential failure sites in a program binary (i.e., it first considers a phase (3) issue). It then uses static slicing to identify critical read instructions that are highly likely to affect potential failure sites through control and data dependences (phase (2)). Finally, it monitors a single (correct) execution of a concurrent program and identifies suspicious interleaving that could cause an incorrect state to arise at a critical read and then lead to a software failure (phase (1)).

8 IMPLEMENTATION TOOLS

During this section we will have a deep look in the available ways to implement a multi-threaded program-testing tool.

Dynamic analysis based tools due to its nature that require actually running the code to be test so it's need a way code instrumentation. *PIN dynamic instrumentation tool*, *INSTR java instrumentation tool [28]*, *DynamoRIO dynamic instrumentation tool [29]*, *Valgrin [30] and Microsoft phoenix* [31] are good tools for either c/c++ or java instrumentals.

Static analysis based tools due to its nature that requires extract a model from the code and process that model most of the cases that model is CFG (control follow graph) and perform CFA (control follow automata).

Soot framework [32], *BLAST [33]*, *CIL [34] and Microsoft phoenix framework [31]* are good tools for building static analyzers.

9 MICROSOFT PHOENIX FRAMEWORK

Microsoft Phoenix framework is a framework that could be used in implementing either static or dynamic analyzers tools.

Phoenix is Microsoft next generation state of the art infrastructure for program analysis and transformation. It's a compilation and tools framework with an infrastructure that is robust, retarget-able, extensible, configurable and scalable. It's built on C++/CLI and compiles either as managed or native code, the process of building program with C++/CLI described in figure 1.

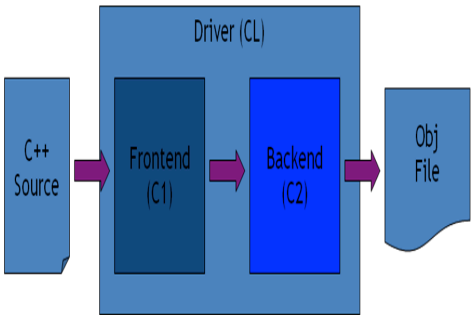

Figure 1. Program‘s building process with C++/CLI.

Front end compiler (C1) accept C++ source code and make the pre-processing, tokenization, parsing, semantic processing, CIL emission, types and symbol debug info and metadata for managed code and produce a CLI output format.

Backend compiler (C2) read the CLI and makes program analysis, optimization, lowering to target, COFF emission and source level debug info.

Microsoft phoenix can be used in set of areas i.e. AST Tools, MSR Advanced language. Academic RDK and Native code generation.

10 CONCLUSION

In this paper we have studied different strategies and techniques for testing concurrent and multi-threaded programs. Static analysis technique based tools are an easy to use tool when no annotation required and the tool is scalable and could be run in large real code bases but it produce lots of false alarms. Dynamic analysis technique based tools although the result is precise i.e. doesn’t suffer from false alarm but it still suffer from poor confidence since it only analyze paths that have been actually run and poor performance due to code instrumentation and runtime analysis. Due to the advantage and limitations for each of the individual testing techniques, recent researches have been tried to combine more than one technique to leverage the advantage of each other. There are different ways for implementing a multi-threaded program analysis tool and we figured Microsoft phoenix framework is a promising framework that contains implementation of different methods that help focusing in the algorithm or research idea itself instead of the implementation.

11 REFERENCES

[1] A. Raza, “A Review of Race Detection Mechanisms,” in *Computer Science – Theory and Applications*, vol. 3967, D. Grigoriev, J. Harrison, and E. A. Hirsch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 534–543.

[2] N. E. Beckman, “A Survey of Methods for Preventing Race Conditions,” 2006.

[3] T. A. Henzinger, R. Jhala, and R. Majumdar, “Race checking by context

inference," *SIGPLAN Not.*, vol. 39, no. 6, pp. 1–13, Jun. 2004.

[4] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," 2009, pp. 386–396.

[5] V. P. Rahul and G. Boby, "Tools And Techniques to Identify Concurrency Issues." [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc546569.aspx. [Accessed: 25-Feb-2012].

[6] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2ndStrike: toward manifesting hidden concurrency typestate bugs," in *Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 239–250.

[7] C. von Praun and T. R. Gross, "Object race detection," *SIGPLAN Not.*, vol. 36, no. 11, pp. 70–82, Oct. 2001.

[8] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller, "Automated type-based analysis of data races and atomicity," 2005, p. 83.

[9] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, New York, NY, USA, 2003, pp. 338–349.

[10] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," 2003, p. 237.

[11] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Practical static race detection for C," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 1, pp. 3:1–3:55, Jan. 2011.

[12] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," 2003, p. 167.

[13] L. Lamport, "Ti clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[15] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, New York, NY, USA, 2002, pp. 258–269.

[16] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs," *SIGPLAN Not.*, vol. 38, no. 10, pp. 179–190, Jun. 2003.

[17] Y. Yu, "RaceTrack: Efficient detection of data race conditions via adaptive tracking," *IN SOSP*, pp. 221–234, 2005.

[18] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer," 2009, p. 62.

[19] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 167–178.

[20] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng, "RACEZ: a lightweight and non-

invasive race detection tool for production applications," in *International Conference on Software Engineering*, 2011, pp. 401–410.

[21] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *International Conference on Software Engineering*, 2010, pp. 245–254.

[22] M. Said, C. Wang, Z. Yang, and K. Sakallah, *Generating Data Race Witnesses by an SMT-Based Analysis*. 2011.

[23] M. Musuvathi and S. Qadeer, *CHESS: Systematic Stress Testing of Concurrent Software*. 2006.

[24] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, Piscataway, NJ, USA, 2008, pp. 356–363.

[25] G. Brat and W. Visser, "Combining Static Analysis and Model Checking for Software Analysis," *PROC. ASE 2001*, pp. 262–271, 2001.

[26] J. Chen and S. MacDonald, "Towards a better collaboration of static and dynamic analyses for testing concurrent programs," in *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*, New York, NY, USA, 2008, pp. 8:1–8:9.

[27] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps, "ConSeq: detecting concurrency bugs through sequential errors," in *Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 251–264.

[28] "Pin - A Dynamic Binary Instrumentation Tool." [Online]. Available: http://www.pintool.org/tutorials/asplos04/. [Accessed: 26-Feb-2012].

[29] "DynamoRIO Dynamic Instrumentation Tool Platform." [Online]. Available: http://dynamorio.org/. [Accessed: 26-Feb-2012].

[30] "Valgrind Home." [Online]. Available: http://valgrind.org/. [Accessed: 26-Feb-2012].

[31] "Phoenix Compiler and Shared Source Common Language Infrastructure - Microsoft Research." [Online]. Available: http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx. [Accessed: 06-Mar-2012].

[32] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, 1999, p. 13–.

[33] E. A. Strunk, M. A. Aiello, and J. C. Knight, "A survey of tools for model checking and model-based development," *University of Virginia*, 2006.

[34] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction*, vol. 2304, R. N. Horspool, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 213–228.