Advances in the ANaConDA Framework for Dynamic Analysis and Testing of Concurrent C/C++ Programs*

Jan Fiedor, Monika Mužikovská, Aleš Smrčka, Ondřej Vašíček, and Tomáš Vojnar
Brno University of Technology, Faculty of Information technology, IT4Innovations Centre of Excellence, Czech Republic

ABSTRACT

The paper presents advances in the ANaConDA framework for dynamic analysis and testing of concurrent C/C++ programs. ANaConDA comes with several built-in analysers, covering detection of data races, deadlocks, or contract violations, and allows for an easy creation of new analysers. To increase the variety of tested interleavings, ANaConDA offers various noise injection techniques. The framework performs the analysis on a binary level, thus not requiring the source code of the program to be available. Apart from many academic experiments, ANaConDA has also been successfully used to discover various errors in industrial code.

CCS CONCEPTS

• **Software and its engineering → Dynamic analysis**; **Software testing and debugging**; *Concurrency control*;

KEYWORDS

Dynamic analysis, testing, concurrency, noise injection, PIN

ACM Reference Format:
Jan Fiedor, Monika Mužikovská, Aleš Smrčka, Ondřej Vašíček, and Tomáš Vojnar. 2018. Advances in the ANaConDA Framework for Dynamic Analysis and Testing of Concurrent C/C++ Programs. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3213846.3229505

1 INTRODUCTION

Nowadays, multi-threaded software can be found in nearly all application areas, including embedded systems. As a result, the programs now contain not only program-logic-related errors but also various kinds of synchronisation-related errors caused by the non-deterministic nature of multi-threaded computation. These errors are not only easy to cause but also very hard to discover.

While static analysis has made huge progress, it still suffers from false alarms and scalability issues, especially when dealing with concurrent programs. This is mainly due to the huge number of thread interleavings and complex program data to be analysed. Static analyses thus usually abstract the interleavings and data, which causes imprecision and false alarms. Therefore, testing and dynamic analysis are still widely used. They scale better as they analyse concrete executions, and seeing the data the program uses allows them to be more precise. However, common testing is not sufficient to discover rarely occurring concurrency-related errors since it will typically not cover sufficiently many thread interleavings. To cope with this problem, special approaches are necessary. Notably, *dynamic analyses* such as [8, 16] extrapolate the witnessed behaviour and warn about possible errors even though they did not happen in the given execution. *Noise injection* [3] inserts in a randomised way various context switches, delays, or additional synchronisation into the run of a concurrent program to stimulate rare (but legal) interleavings that may yield so far undiscovered errors. Other alternatives then include *systematic testing* [14] or *coverage-based testing* [18] briefly mentioned below.

In this paper, we discuss the ANaConDA framework for dynamic analysis and testing of concurrent C/C++ programs, which was first introduced in [6]. ANaConDA comes with several predefined analysers for common concurrency problems (e.g., deadlocks, data races, atomicity violations) and is designed to simplify the development of further extrapolating dynamic analysers. ANaConDA also offers a wide range of noise injection techniques. It is built on top of Intel PIN [13], a framework for dynamic binary instrumentation, and thus the analysis is done on a binary level. This has several advantages. ANaConDA does not require the source code of the program or its libraries and it can handle self-generating, self-modifying, and assembly code. ANaConDA[1] is open source, runs both on Linux and Windows, and supports various concurrency libraries.

Since its introduction in [6], ANaConDA was extended with several noise heuristics [4], a support for monitoring transactional memories [5], and a capability of detecting violations of contracts for concurrency [2]. Moreover, we newly added hierarchical filters allowing one to exclude various functions from the analysis (motivated by an industrial application of ANaConDA), several new analysers implementing detectors originally developed for Java programs, construction of precise backtraces for diagnostic purposes, support of C++11 concurrency extensions, as well as automation scripts simplifying the installation and usage of the framework.

Related work. The closest tools to ANaConDA are IBM ConTest [3] and RoadRunner [9]. Both are able to monitor the execution of multi-threaded Java programs and notify analysers built on top of them about important events. The analysers available over these frameworks differ: an analyser so far unique to the ANaConDA framework is the detector of violations of contracts for concurrency. IBM Contest supports noise injection too—though not the *read/write* noise of ANaConDA. For monitoring multi-threaded C/C++ programs, the options are much more limited. The closest tool to ANaConDA in this area is Fjalar [10]. However, its purpose is to simplify access to various compile-time and memory information. It does not provide any concurrency-related information.

*This work was supported by the Czech Science Foundation (project no. 17-12465S).

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3229505

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda

An alternative to noise-based testing is systematic testing [14], exploring all schedules up to some bound (e.g., in the number of context switches). It offers higher guarantees of discovering concurrency errors, but it is less scalable and has problems with some program constructions (user input, networking, etc.). Another approach is coverage-driven testing [18] that influences thread scheduling to maximize coverage of several important synchronization idioms. However, it does not support some kinds of errors, and, somewhat like noise injection, it is partially based on randomisation.

2 FEATURES AND USAGE OF ANACONDA

ANaConDA allows one to readily use several predefined detectors of common concurrency errors—namely, AtomRace, Eraser, FastTrack, and GoodLock. *AtomRace* [11] is a simple detector of data races that does not report any false alarms since it looks for two unsynchronized memory accesses (with at least one being a write access) to be ready to execute in two threads at the same time. To make such scenarios show up more likely, noise injection is used. *Eraser* [16] and *FastTrack* [8] are well-known extrapolating data race detectors, originally developed for Java. *GoodLock* [1] is an extrapolating deadlock detector.

ANaConDA also offers an original detector of violations of *contracts for concurrency* [2]. These contracts can specify that the execution of some sequences of function calls, called *targets*, must not be mutually interleaved with other sequences of calls, called *spoilers*. For instance, a test of the presence of some element in a collection followed by its modification must not be interleaved with a removal of the element. This detector covers as special cases various classes of errors such as *atomicity violations* or *order violations*.

Further, ANaConDA allows its users to easily define *new dynamic analysers* by simply listening to selected concurrency-related program events (memory accesses, locking, unlocking, etc.).

ANaConDA can also be easily configured to work with different *concurrency libraries* by simply specifying which functions are used to implement various synchronisation operations and what the roles of their parameters are. Currently, ANaConDA supports the `pthread` library, Win32 API, and C++11 concurrency extensions.

ANaConDA allows one to increase chances of finding rare thread interleavings, which are more likely to hide concurrency-related errors, by inserting various kinds of *noise* into the execution. In particular, the following kinds of noise are available: *sleep noise* that suspends a thread for some time, *yield noise* that gives up the CPU several times, the *busy-wait noise* that actively loops for some time, and the *inverse noise* that suspends all threads but one for some time. The noise is parametrized by *frequency*, the probability to insert some noise, and *strength*, the intensity of the noise. The strength is given in milliseconds for the sleep, busy-wait, and inverse noise; and in the number of applications for the yield noise. Finally, the user may also specify where to put the noise—be it before *every monitored event*, before *shared variables* only, before *read/write* accesses, or into some specific sequences (*patterns*) of accesses.

An important feature of ANaConDA (compared, e.g., with ConTest [3]) is a support of *read/write noise* [7], i.e., insertion of different kinds of noise at different memory accesses, which turns out to be very efficient in practice, especially when detecting data races. In particular, one can inject a strong noise before the rarer type of memory accesses and a much weaker noise before the other accesses. The analyser then checks the rare accesses against a large number of potentially conflicting accesses from other threads which are not delayed by the strong noise. Moreover, the fine-grained noise has recently been extended to functions, allowing specific noise to be inserted before specific functions. This functionality is useful when detecting contract violations as it can prolong the execution of specific sequences of function calls to increase the window during which they can be violated by functions from other threads.

As monitoring many memory accesses can significantly slow down the analysis, ANaConDA was recently extended by *hierarchical filters* allowing the user to specify that all memory accesses from some library or all accesses from some functions should be ignored. The introduction of such filters was motivated by an industrial application of ANaConDA where it was applied on code heavily using memory-access-intensive encryption and communication libraries. To facilitate the use of such filters, ANaConDA newly provides means to *collect statistics* about the number of memory accesses performed by each function to pinpoint the problematic functions.

To aid diagnosis of discovered errors, ANaConDA can provide *on-demand backtraces*. A problem is that this requires the program to properly create stack frames, which nowadays programs rarely do for performance reasons. Alternatively, the user may use the newly added *precise backtraces* that are created by tracking all function calls in the program. To keep such backtraces valid, ANaConDA needs to track returns from functions and pair them with the corresponding calls in order for the function calls to be removed from the backtraces. Unfortunately, not all functions return. A typical example is that of trampolines, stub functions whose only goal is to redirect the call to an external library where the called function is located. To deal with such functions, ANaConDA allows one to specify functions that will not be tracked for backtrace creation.

Using the recently added automation scripts, ANaConDA can be easily installed both on Linux and Windows using the script `tools/build.sh` from its distribution. As ANaConDA is primarily a command-line tool, all of its functionality is either available from the command line or via configuration files (using a specialised simple syntax for describing new detectors, concurrency libraries, or filters). The simplest way to run ANaConDA is to use the script `tools/run.sh <analyser> <program> [<program-params>]` where `<analyser>` is the chosen analyser and `<program>` the program to be analysed. ANaConDA also supports repeated testing via the `tools/test.sh` script. It allows one to specify the number of test runs or the amount of time to repeat the tests, saving the obtained output (and configuration used) for each test run performed.

3 ARCHITECTURE

We now explain the architecture of ANaConDA and discuss its functioning during an analysis. An overview of the architecture is shown in Fig. 1. The top left part shows a fragment of assembly code (compiled from a C code) of two threads being analyzed: one of the threads increments a variable `i` under a lock `L`, while the other increments it without the lock held.

An important part of the architecture is a set of available analysers (top right of Fig. 1). When ANaConDA is invoked, it first initialises the chosen analyser (e.g., AtomRace) and lets the analyser register which events in the run of an analysed program it is interested in, out of those supported by the monitoring layer.

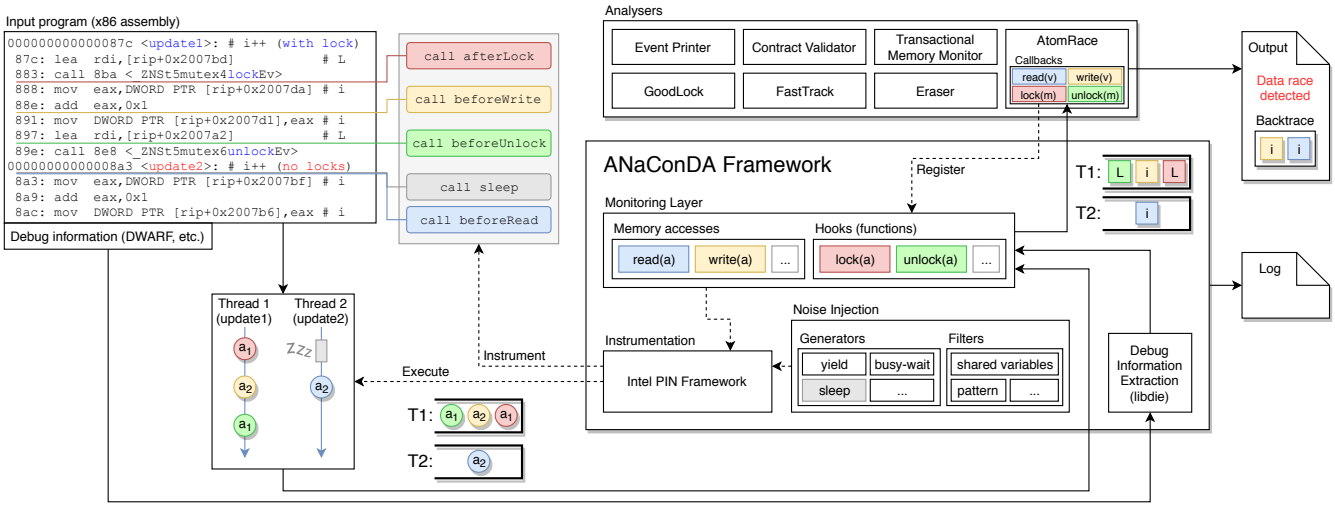

Figure 1: Overview of the architecture of ANaConDA

It can be informed, e.g., that a thread in the monitored program is before/after reading/writing some variable v, that the thread is before/after locking/unlocking a lock m, and likewise for other synchronisation operations. (In fact, the monitoring works with binary addresses a of memory locations, synchronisation primitives, etc.; but the framework then strives to convert the binary information back to the names of variables v, locks m, and so on.)

The program to be analysed is then loaded into memory and instrumented using the Intel PIN framework [13]. The instrumentation inserts only the code that is needed to extract the information requested by the analyser and to inject noise generation at some instructions. The latter is done only if the user decides that some of the available noise generators should be applied. This fine-grained instrumentation reduces the slowdown of the analysed program caused by ANaConDA. The in-memory instrumentation also allows ANaConDA to transparently analyse libraries used by the given program without affecting other programs using the same libraries.

Once the program is instrumented, the framework executes it. If noise injection is enabled, noise is inserted at various points in the execution based on the generators, their parameters, and filters used. For example, Fig. 1 illustrates insertion of random `sleep` noise at accesses to the address a_2 corresponding to the variable $v = \texttt{i}$. The instrumented code generates events that are send back to the framework, associated with binary data about the particular event—e.g., memory access events carry the size of the accessed data and their address (e.g., a_2 in our figure), the lock acquire events carry the address of the synchronization primitive (e.g., a_1), etc. The framework processes this data and refines it.

The refinement depends on the type of event. For synchronisation events, the name of the function encountered determines the synchronisation operation performed (e.g., `pthread_mutex_lock` triggers the *lock acquired* operation). Based on the type of operation, the address of the synchronisation primitive is transformed to a platform-independent identifier (e.g., a lock identifier for lock operations). This is done using the hooks and mapper objects which are declared when instantiating ANaConDA for a particular concurrency library. This way, the detectors become independent on the concrete concurrency library used. If debugging information is available, a conversion to the corresponding program identifiers is possible too (e.g., transforming the address a_1 to the lock $m = \texttt{L}$).

For memory access events, if there is debugging information, the address of a memory access is converted to the name of the variable residing on this address (e.g., instead of the address a_2, the analyser is informed about the variable $v = \texttt{i}$). Moreover, the data type of the variable and the program location of the instruction performing the access are also available. As obtaining such information is expensive, it is done only if the analyser requests this during its initialisation.

The refined data are sent to the analyser, which performs the analysis and outputs the results (e.g., providing a backtrace for the data race between the unsynchronised accesses to `i` in our example).

4 EXPERIMENTS AND APPLICATIONS

The first works on ANaConDA [6, 7] included a number of experiments with the AtomRace detector that discovered a number of errors in student projects as well as (previously unknown) errors in real-life software: namely, the `unicap` libraries for video processing (40 KLOC). Later, in [2], we used ANaConDA to detect (previously known) atomicity violations even in a program as large as the Chromium browser (7.5 MLOC). Moreover, together with a major international producer of home automation (whose name we are, unfortunately, not allowed to reveal), we used ANaConDA to discover a very rarely occurring order violation in a component of a cloud-connected thermostat (1.5 KLOC) used for managing parallel task processing that common testing was unable to identify.

After the extension of ANaConDA by further detectors, we performed new experiments targeted at their evaluation. In particular, we first evaluated our implementation of AtomRace, Eraser, and FastTrack on the benchmark suite DataRaceBench [12], comparing the detectors with the well-known Helgrind data race detector of Valgrind [15] and the LLVM/Clang ThreadSanitizer data race detector [17]. The benchmark suite is designed to evaluate the effectiveness of data race detection tools. It includes microbenchmarks either manually written, extracted from real scientific applications, or automatically generated optimization variants. It also defines

Table 1: Results of the DataRaceBench benchmark.

Detector	TP	TN	FP	FN	P	R	A
AtomRace*	102	108	0	123	1.00	0.45	0.63
FastTrack*	184	72	36	41	0.84	0.82	0.77
FastTrack	213	3	105	12	0.67	0.95	0.65
Helgrind	213	3	105	12	0.67	0.95	0.65
ThreadSanitizer	213	3	105	12	0.67	0.95	0.65

metrics for effectiveness and efficiency of data race detection tools based on the ratio of false-positives (FP), false-negatives (FN), true-positives (TP), and true-negatives (TN). Concretely, the metrics for *Precision* $P = TP/(TP + FP)$, *Recall* $R = TP/(TP + FN)$, and *Accuracy* $A = (TP + TN)/(TP + TN + FP + FN)$ are defined. The precision reflects the confidence that a reported positive is a real one. The recall shows an ability of a tool to find an existing data race. The accuracy summarises correctness of all the reports. The bigger the value of P, R, and A, the better.

The results are shown in Table 1, aggregating results from 80 different microbenchmarks (median 18 LOC) tested with different variations in the number of threads (3, 45, 256) and, for array microbenchmarks, the size of the array it operates on (32, 128, 1024). Results for Eraser are not shown as they are almost identical to FastTrack. The lines marked with "*" use noise, which is necessary for AtomRace. FastTrack is examined with and without noise. Table 1 shows that AtomRace can find many races while not reporting any false alarms. Helgrind and ThreadSanitizer can find more races but are plagued by many false alarms, which is a problem in practice. FastTrack without noise gives the same results as Helgrind and ThreadSanitizer. With noise, some races are missed but the precision is improved, yielding a compromise between the other results. (We also see that noise can hide some errors too, which happens in particular for the relatively simple microbenchmarks where the extrapolating power of FastTrack is by itself strong enough.)

Next, we performed new analyses on the complete implementation of the thermostat software (145 KLOC) mentioned above. First, we looked for data races using the *AtomRace* and *Eraser* analysers. Compared with detectors from Valgrind, *Eraser* reported many more data races. While some of them were real errors, a majority were just false alarms. *AtomRace* reported a few data races only, a subset of those found by *Eraser*, but all of them were real errors. Interestingly, when helped by the *read/write* noise, both analysers reported a data race causing a segmentation fault that Valgrind failed to find due to its rare occurrence. However, nobody paid attention to the warning raised by *Eraser* as it was hidden among hundreds of false alarms. Only when *AtomRace* forced the problem to manifest and precisely localized it, the error was corrected.

We also used ANaConDA to find a blocked thread error. While there is no dynamic analysis targeting this type of errors, just executing the program with the right noise forced an execution leading to the error, causing the whole program to be irrecoverably stuck. This demonstrates that one does not always need to perform a dynamic analysis: simple noise-based testing may suffice.

Finally, we used ANaConDA to detect missing synchronisation between a thread disposing locks and threads using these locks. This problem was very specific to the target program, but, in about 2 hours, we were able to write a simple analyser checking the operations on these locks and reporting a use of disposed locks. The analyser provided us with backtraces, making it easy to localize the issue and fix it. This nicely illustrates how the ANaConDA framework can be used to detect application-specific issues.

The slowdown of programs running under ANaConDA is hard to quantify as it depends on many aspects. The base slowdown imposed by Intel PIN is about 5 times. Next, if monitoring function calls, e.g., for detecting contract violations [2], the slowdown is about 10 times. However, if memory accesses need to be monitored too, the slowdown can go up to 100 times, for data-intensive functions even up to 1000 times. In such a case, hierarchical filters are usually needed to exclude such functions from monitoring and keep the slowdown on a reasonable level. The process of injecting noise has a negligible impact on the slowdown, but one must keep in mind that the injected noise itself may delay the execution considerably.

5 FUTURE WORK

As a part of further improvements of ANaConDA, we are now working on indexing data about functions, instructions, and variables to speed up repetitive work with them. ANaConDA is also being ported to Mac OS. Another planned extension is to support trace processing, allowing ANaConDA to analyse executions of programs written in any language and/or running on systems unable to run ANaConDA directly, e.g., ARM-based systems or systems with specialized operating systems. Another open question is a support for a different backend beside Intel PIN that would ideally be less intrusive in regards of the slowdown of the execution.

REFERENCES

[1] S. Bensalem and K. Havelund. 2006. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proc. of HVC'05 (LNCS 3875)*. Springer.

[2] R. Dias, C. Ferreira, J. Fiedor, J. Lourenço, A. Smrcka, D. Sousa, and T. Vojnar. 2017. Verifying Concurrent Programs Using Contracts. In *Proc. of ICST'17*. ACM.

[3] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. 2003. Framework for Testing Multi-threaded Java Programs. *Concur. and Comp.* 15, 3-5 (2003).

[4] J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. 2014. Advances in Noise-based Testing. *STVR* 24, 7 (2014).

[5] J. Fiedor, Z. Letko, J. Lourenço, and T. Vojnar. 2014. On Monitoring C/C++ Transactional Memory Programs. In *Proc. of MEMICS'14 (LNCS 8934)*. Springer.

[6] J. Fiedor and T. Vojnar. 2012. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In *RV'12 (LNCS 7687)*. Springer.

[7] J. Fiedor and T. Vojnar. 2012. Noise-based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of PADTAD'12*. ACM.

[8] C. Flanagan and S. N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. of PLDI'09*. ACM.

[9] C. Flanagan and S. N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proc. of PASTE'10*. ACM.

[10] P. J. Guo. 2006. *A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs*. Master's thesis. MIT.

[11] Z. Letko, T. Vojnar, and B. Křena. 2008. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proc. of PADTAD'08*. ACM.

[12] C. Liao, P. Lin, J. Asplund, M. Schordan, and I. Karlin. 2017. DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools. In *Proc. of SC'17*. ACM.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of PLDI'05*. ACM.

[14] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI'08*. USENIX.

[15] N. Nethercote and J. Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of PLDI'07*. ACM.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *SOSP'97*. ACM.

[17] K. Serebryany and T. Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proc. of WBIA'09*. ACM.

[18] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. 2012. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *Proc. of OOPSLA'12*. ACM.