VERISMART 2.0: Swarm-Based Bug-Finding for Multi-Threaded Programs with Lazy-CSeq

Bernd Fischer
Stellenbosch University, South Africa
bfischer@cs.sun.ac.za

Salvatore La Torre
University of Salerno, Italy
slatorre@unisa.it

Gennaro Parlato
University of Molise, Italy
gennaro.parlato@unimol.it

***Abstract*—Swarm-based verification methods split a verification problem into a large number of independent simpler tasks and so exploit the availability of large numbers of cores to speed up verification. Lazy-CSeq is a BMC-based bug-finding tool for C programs using POSIX threads that is based on sequentialization. Here we present the tool VERISMART 2.0, which extends Lazy-CSeq with a swarm-based bug-finding method. The key idea of this approach is to constrain the interleaving such that context switches can only happen within selected tiles (more specifically, contiguous code segments within the individual threads). This under-approximates the program's behaviours, with the number and size of tiles as additional parameters, which allows us to vary the complexity of the tasks. Overall, this significantly improves peak memory consumption and (wall-clock) analysis time.**

***Video*—https://youtu.be/m1GwUWCdxdI**

***Sources*—https://www.southampton.ac.uk/~gp1y10/cseq/verismart.tgz**

***Index Terms*—program analysis, verification, concurrency, sequentialization, swarm verification**

I. INTRODUCTION

Swarm-based verification methods take advantage of the large number of cores that are now available by splitting a verification problem into a large number of independent simpler tasks. They are effective when the "task swarm" can be built cheaply and the resulting tasks are indeed simpler than the original problem. For bug finding, only some of the tasks that exhibit the bug need to be simpler, because the analysis can be stopped as soon as a bug is found in one task.

This idea is explored by Holzmann et al. [6], who obtain the tasks by randomizing and diversifying the search process in SPIN (e.g., using different hash polynomials and search algorithms), thus running *different strategies* over the *same program*. We introduced a complementary swarm-based method for finding bugs in multi-threaded programs [12] where the *same strategy* is run over *different variants* of the original program. Each variant is obtained from the original program by enabling preemptions only within selected code segments (*tiles*), thus capturing a subset of the program's interleavings. The variants can be built such that each interleaving of the original program is captured by at least one verification task.

Here we describe the VERISMART 2.0 tool that implements the swarm-based approach from [12] as an extension of Lazy-CSeq [8], [10], a BMC-based tool for finding bugs in C programs using POSIX threads. Lazy-CSeq uses lazy sequentialization [9]: it first translates a multi-threaded C program P into a nondeterministic sequential C program P' that preserves reachability for all round-robin schedules with a given bound on the number of rounds, and then exploits CBMC [3] as sequential verification engine to analyze P'. Experiments show that the swarm-based approach can significantly improve Lazy-CSeq's peak memory consumption and (wall-clock) verification times, in particular for complex programs: on some implementations of lock-free data structures, verification times went from 8-12 hours down to 15-30 minutes, using only a modest number (5-50) of processors.

VERISMART 2.0 is aimed at software developers that face the challenging task of developing correct multi-threaded software. It is implemented as stand-alone, command line tool, and can thus be integrated into the normal development workflow. In comparison to the early prototype used in [12] (in the following referred to as VERISMART 1.0), VERISMART 2.0 reflects a major system refactoring and substantial extensions, providing three main novelties. First, we added a verification manager module that orchestrates all steps of the verification process, and so frees the user from collecting intermediate files, starting verification tasks, and collating results. Second, we re-designed the old batch-style architecture, where all tasks were generated before the verification could start, into an incremental pipeline. This allows the verification manager to control the task generation in a demand-driven fashion, and so resolves a bottle-neck that chocked the prototype for problems spawning a very large number of tasks. Third, we improved the integration with the Lazy-CSeq sequentialization and can now produce counterexamples for the original (multi-threaded) program; however, we do not detail this feature here.

II. SWARM VERIFICATION VIA PROGRAM TILINGS

VERISMART 2.0 inherits the swarm-based bug-finding methodology from VERISMART 1.0. We briefly recall the main notions here. For a bounded multi-threaded program P, this methodology relies on the generation of variants of P, and is based on the notions of program tiling and tile selection.

a) Program tiling: A *tiling of a thread* $t \in T$ is defined [12], as a partition of t's statements; each element of a tiling is called *tile*. A *tiling of a multi-threaded program* P is a set of thread tilings, one for each thread of P. Let $\Theta_P = \{\Theta_t\}_{t \in T}$ be a tiling of P. A *z-selection* of Θ_P is a set $\{\theta_t\}_{t \in T}$ where $\theta_t \subseteq \Theta_t$ contains exactly z tiles or it coincides with Θ_t if $|\Theta_t| \leq z$, i.e., all tiles in the partition of t are selected if there are at most z. For a z-selections of a given tiling Θ_P of

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00124

IEEE computer society

Authorized licensed use limited to: UNIVERSITY OF SOUTHAMPTON. Downloaded on August 11,2021 at 21:21:07 UTC from IEEE Xplore. Restrictions apply.

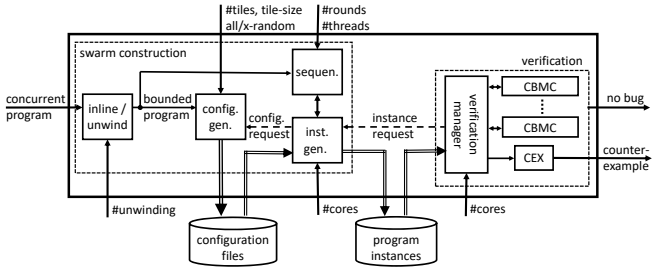

Fig. 1. VERISMART tool architecture.

P, a program variant P_ϑ of P is obtained by instrumenting P in a way such that each thread t can only be preempted at statements belonging to the tiles of θ_t and at any other blocking statement of t (in order to allow an execution to continue when a statement is blocked and there are other threads that are ready to execute). In VERISMART 2.0, we use *window tilings*, where each tile contains only consecutive statements of P. The *size* of a tile is the number of visible statements that it contains; a statement is *visible* if it involves an access to the shared memory or an invocation of a function from the POSIX threads library.

b) Bounded round analysis and tile selection: Let $k \in \mathbb{N}$ and ϑ be any z-selection of Θ_P with $z = \lceil \frac{k}{2} \rceil$. We recall from [12] that to cover all of P's behaviors up to k context switches, we need to consider all possible variants P_ϑ of P: in any interleaving with at most k context switches, each thread is preempted at most $\lceil \frac{k}{2} \rceil$ times. A *round* is a portion of computation where each thread is preempted at most once. Since the context-bounded analysis of Lazy-CSeq is parameterized over the number of rounds, in order to guarantee that all of P's behaviors up to k rounds are covered, it suffices to restrict the analysis to the set of k-selections of Θ_P.

III. TOOL ARCHITECTURE

Figure 1 illustrates the architecture of VERISMART 2.0. It comprises two main parts. The *swarm construction* takes as input a multi-threaded program P along with additional parameters, and generates a set of program variants. These variants are fed into the *verification manager* that controls the swarm construction and orchestrates the analysis possibly using multiple cores. These two parts are set up as an *incremental pipeline* where the program variants are generated on demand.

a) Swarm construction: The *inline/unwind* module transforms the input multi-threaded program P into a bounded multi-threaded program P' by function inlining and loop unwinding (up to a bound `#unwinding`, which is given as input parameter). The resulting program has thus a different *thread function* associated with each thread. We use a slightly modified version of the existing inline/unwind module provided by the CSeq framework [10].

The *configuration generator* module takes the bounded program P' and the tiling parameters (`#tiles` and `tile-size`), and generates one or more configuration files, each containing a list of tile selections encoded in JSON format. It first computes the number of visible statements for each thread of P', then determines a window tiling of P' where each tile covers `tile-size` consecutive visible statements (except for the last tile in each thread, which can be shorter), and finally produces a list of `#tiles`-selections. The tile selections can be generated deterministically and exhaustively (`all`) or randomly and selectively (`x-random`). Configuration files can be also provided externally as an input. This enables the user to express any kind of tilings and tile selections (e.g., selections of a window tiling with tiles of different sizes) as the configuration file format allows to disable preemptions at any chosen visible statement.

The *sequentialization* module sequentializes the bounded program P' for each tile selection described in the configuration file. This follows the approach in Lazy-CSeq [9] except that it runs under control of the *instance generator*. This ensures that numerical labels and related control code are injected only at the visible statements indicated in the tile selection. Since the numerical labels are used in this sequentialization to control the simulation of preemptions, restricting the injected control code only to the visible statements within the selected tiles results in much smaller verification conditions (i.e., SAT formulas) produced by the backend verifier.

b) Verification: The *verification* subsystem analyzes each generated program variant independently with the bounded model-checker CBMC. This is orchestrated by the *verification manager*, which launches the generated tasks in parallel, again exploiting the number of specified cores. It also collects the output of CBMC on all the variants asynchronously. As soon as a counterexample is found, all the remaining tasks can be aborted if specified by the user. Any counterexample found in a program variant is mapped into a counterexample of the original program P by the CEX module. This is implemented by using the line map mechanism provided by the CSeq framework [10]. Note that the generated tasks can in principle be processed independently by any sequential verification tool for C, but we tested VERISMART 2.0 only with CBMC v5.11.

c) Incremental pipeline: The sequentialized programs can be generated in parallel, using the number of specified cores. This is relatively fast compared to verification but since the number of instances can grow very quickly, even a parallel upfront generation of all instances can still lead to long delays and out-of-memory errors before the verification phase itself can actually start. VERISMART 2.0 therefore generates the instances lazily, driven by the demand from the verification manager (cf. the backward dashed arrows in Figure 1).

IV. TOOL IMPLEMENTATION

Like its predecessor, VERISMART 2.0 is implemented in Python and uses the `pycparser` to parse a C program into an abstract syntax trees (ASTs), and then executes the code-to-code transformations described in Section III at the AST level. VERISMART 2.0 also uses the Python `multiprocessing`

Authorized licensed use limited to: UNIVERSITY OF SOUTHAMPTON. Downloaded on August 11,2021 at 21:21:07 UTC from IEEE Xplore. Restrictions apply.

library for managing a thread pool in the *verification manager*. The thread pool is initially filled with the specified number of verification jobs; this triggers the lazy generation of the corresponding program variants. Each completed job is removed from the pool and replaced by a job for the next—lazily computed—variant.

V. Tool Usage and Analysis Scenarios

VeriSmart 2.0 is called on the command line as `./verismart.py` *`[options] file`*, where *`file`* contains the C program to be analyzed. It produces a configuration file *`FILE`*`_auto_config.json` and a directory *`FILE`*`.swarm` with the sequentialized instance files. For example,

```
./verismart.py ex/ex1.c
```

generates all possible five instances and runs the default verification backend. This finds a bug in the second configuration attempted.

Compared to its predecessor, VeriSmart 2.0 now provides a refactored and more comprehensive set of options for the user to easily manipulate the swarm verification process.

The number of threads that are started can be modified with the `--cores` option (default: 4):

```
./verismart.py --cores 2 ex/ex1.c
```

only starts up two threads but this still finds the error quickly.

The tool implements several tiling techniques for the thread code. The default tiling splits the code into windows of the same size (possibly except for the last tile of each thread). The window length can be specified with the `--window-length` (or `-l`) option; the argument gives the number of visible statements in the tile:

```
./verismart.py -l 2 ex/ex1.c
```

produces only three configurations but still finds the error. Alternatively, the window length can be specified via the `--window-percent` option as percentage of the threads' length; this needs to be used with care because small percentages can lead to empty windows. This technique is very useful when the sequentialized output has threads that differ substantially in size.

The (maximum) number of tiles selected in each thread can be controlled by the `--picked-window` (or `-p`) option:

```
./verismart.py -p 2 ex/ex1.c
```

picks two tiles in each thread, if the thread is long enough to allow this, and so generates ten configurations, of which four demonstrate the error. We can force the analysis to stop after the first error using the `--exit-on-error` option; note that other pending tasks may still finish (and so actually produce multiple errors), due to delays in killing the processes.

The `--shift-window` option allows us to randomly shift the windows up and down by half a window size. This has empirically shown to lead to better results.

The number of configurations can grow large very quickly, in particular for small window lengths and larger numbers of picked windows; it can be limited with the `--instances-limit` option:

```
./verismart.py --instances-limit 2 ex/ex1.c
```

Note that VeriSmart 2.0 chooses random configurations if the instance limit (which defaults to 100) is smaller than the number of possible instances. Hence, different runs can yield different results, and may fail to find the error, if any.

The tool also provides options to control the bounding in Lazy-CSeq. `--unwind<X>` (or `-u<X>`) sets the unwind bound for all loops, while `--while-unwind<X>` resp. `--for-unwind<X>` (or `-w<X>` resp. `-f<X>`) set the unwind bounds for potentially unbounded resp. bounded loops. `--rounds<X>` (or `-r<X>`) sets the number of round-robin schedules. All bounds default to one. Similarly, the tool also provides options to control the backend; the `--help` (or `-h`) option lists them in more detail.

By default, VeriSmart 2.0 manages the full verification process: it generates the configuration and instance files, and runs the verification backend over all instances on a single machine. However, this can be changed: the tool can stop after generating the configuration and instance files, respectively, so that the verification can in a second stage be re-started on a different machine (or at a later time); it can also generate a set of configuration files that can then be used to run several instances of VeriSmart 2.0 in parallel, e.g., on a cluster.

Specifically, with the `--config-only` option, the tool stops after writing the configuration file:

```
./verismart.py --config-only ex/ex1.c
```

Note that the contents of this file depend on any other control parameters such as `-l` or `-p`. These configuration files can then independently be used with the `--config-file` (or `-c`) option to re-start the verification process; configuration files can also be created manually or by other tools.

Similarly, with the `--instances-only` option, VeriSmart 2.0 stops after writing the instance files (i.e., the sequentialized program variants):

```
./verismart.py --instances-only ex/ex1.c
```

Finally, VeriSmart 2.0 can also be used to generate a set of configuration files, each describing the specified number of configurations:

```
./verismart.py --cluster-config 2 ex/ex1.c
```

This can then be used in a second stage to run several instances of the tool in parallel, each using the `--config-file` option to read one of the generated configuration files and then to generate and verify the described instances.

VI. Related Work

Parallel Verification: Attempts to parallelize verification by partitioning the problem and distributing the workload have been implemented in explicit-state model checking [2], [16] and SAT solving [13]. However, these suffer from the overhead introduced by exchanging information between the instances. *Portfolio approaches* that run several tools with different strategies and heuristics in parallel on the unpartitioned problem have been more successful in automated

Authorized licensed use limited to: UNIVERSITY OF SOUTHAMPTON. Downloaded on August 11,2021 at 21:21:07 UTC from IEEE Xplore. Restrictions apply.

theorem proving [14], [15] and SAT/SMT solving [18], [19]. Our approach leverages the *swarm verification* introduced by Holzmann explicit-state model checking [6], [7], where computing instances do not collaborate directly in finding a solution, but solve independent subproblems that cover the original problem. We lift this idea to symbolic model checking through sequentialization.

Concurrency testing: Automated testing tools such as CHESS [11] have been very successful for finding concurrency bugs in large code bases because of their ability to handle code independently of its sequential complexity. Nonetheless, their success depends on the proportion of schedules that lead to a bug w.r.t. the total number of schedules, as shown by a recent empirical study [17] on testing of concurrent programs. Preemption sealing [1] consists of inhibiting preemptions in some program modules which corresponds in our approach to choose a tiling where tiles exactly correspond to program modules. This strategy was aimed to tolerating errors for finding more ones and compositional testing of layered concurrent systems. The uniform tiling we use is irrespective of the structure of the program and looks more appropriate for an exhaustive bug-finding search up to a given number of context-switches. There are also differences in the implementation of the two techniques, we do not seal portions of code with scope functions but rather we implement tiles statically, that in general makes the underlying BMC analysis simpler.

In analogy to concolic testing [4], our approach could be called *randolic testing*, as it combines randomness and symbolic representation of non-determinism; in fact, it enables us to fine-tune randomness and nondeterminism used for the analysis by varying the tile sizes and the way instances are selected. On one end, tiles cover only a single visible statement and our approach can be seen as symbolic random testing, where only randomness is used for the analysis. On the other end, the entire thread is treated as a tile and our approach can be seen as BMC, where only nondeterminism is used.

VII. Discussion and Conclusions

As already demonstrated in [12], VeriSmart's main strength is its ability to handle benchmarks with rare bugs that are "out of reach" for other tools based on testing and BMC. For programs with rare concurrency bugs, most instances do not contain a bug and the analysis is time-consuming. However, in tasks that do, the bugs are generally more frequent than in the original program and thus can be found faster and with fewer resources. VeriSmart 2.0 improves over VeriSmart 1.0 in the way the program variants are generated, but it ultimately produces the same program variants from the same inputs. Therefore, it retains the bug-finding ability of VeriSmart 1.0. In fact, the results of the backend analysis are the same as those reported in [12] for VeriSmart 1.0.

The main improvement in VeriSmart 2.0 is that verification now starts as soon as the first variant is generated while VeriSmart 1.0 first generated all the variants up to the given bound before starting verification. This has two main advantages. First, since the analysis can be stopped as soon as a bug is found, VeriSmart 2.0 can save the potentially significant time spent on generating the remaining variants. Second, guessing an appropriate number of variants to be generated was crucial in VeriSmart 1.0 to ensure that if the program has a bug, one of the generated variants will exhibit it. This has become almost irrelevant in VeriSmart 2.0 since we can now go on generating variants until a bug is found.

VeriSmart 2.0 can now also be used to orchestrate (sequential and) parallel verification over all cores of a single computer; it can already prepare the jobs for use on a cluster, but the verification manager does not yet distribute them over an entire cluster; this work is currently in progress. We are also working on a variant that uses testing instead of a bounded symbolic analysis; this may scale easier to large programs but may need (many) more instances to find a bug.

Acknowledgements. La Torre's work was partially supported by GNCS 2019 and MIUR-FARB 2017-2018 grants.

References

[1] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer Preemption sealing for efficient concurrency testing. *TACAS*, LNCS 6015, pp. 420–434, 2010.
[2] J. Barnat, L. Brim, and I. Cerná. Cluster-based LTL model checking of large systems. *FMCO*, LNCS 4111, pp. 259–279, 2005.
[3] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *TACAS*, LNCS 2988, pp. 168–176, 2004.
[4] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *PLDI*, ACM, pp. 213–223, 2005.
[5] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. *SPAA*, ACM, pp. 206–215, 2004.
[6] G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification techniques. *IEEE Trans. Software Eng.*, 37(6):845–857, 2011.
[7] G. J. Holzmann. Cloud-based verification of concurrent software. *VMCAI*, LNCS 9583, pp. 311–327, 2016.
[8] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, G. Parlato. Lazy-CSeq: A Lazy Sequentialization tool for C (Competition Contribution). *TACAS*, LNCS 8413, pp. 398–401, 2014.
[9] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, G. Parlato. Bounded Model Checking of Multi-Threaded C Programs via Sequentialization. *CAV*, LNCS 8559, pp. 585–602, 2014.
[10] O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded C-programs. *ASE*, IEEE, pp. 807–812, 2015.
[11] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. *OSDI*, USENIX, pp. 267–280, 2008.
[12] T. L. Nguyen, P. Schrammel, B. Fischer, S. La Torre, and G. Parlato. Parallel Bug-finding in Concurrent Programs via Reduced Interleaving Instances. *ASE*, IEEE, pp. 753–764, 2017.
[13] K. Ohmura and K. Ueda. c-sat: A parallel SAT solver for clusters. *SAT*, LNCS 5584, pp. 524–537, 2009.
[14] J. Schumann. Sicotheo: Simple competitive parallel theorem provers. *CADE*, LNCS 1104, pp. 240–244, 1996.
[15] A. Wolf and R. Letz. Strategy parallelism in automated theorem proving. *IJPRAI*, vol. 13, no. 2, pp. 219–245, 1999.
[16] U. Stern and D. L. Dill Parallelizing the mur*phi* verifier. *CAV*, LNCS 1254, pp. 256–278, 1997.
[17] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using controlled schedulers: An empirical study. *TOPC*, vol. 2, no. 4, pp. 23:1–23:37, 2016.
[18] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura. A concurrent portfolio approach to SMT solving. *CAV*, LNCS 5643, pp. 715–720, 2009.
[19] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, vol. 32, pp. 565–606, 2008.

Authorized licensed use limited to: UNIVERSITY OF SOUTHAMPTON. Downloaded on August 11,2021 at 21:21:07 UTC from IEEE Xplore. Restrictions apply.