ESBMC-GPU
A Context-Bounded Model Checking Tool to Verify CUDA Programs

Felipe R. Monteiro[1], Erickson H. da S. Alves[1,2], Isabela S. Silva[1], Hussama I. Ismail[1], Lucas C. Cordeiro[1,3], and Eddie B. de Lima Filho[1,4]

[1]*Faculty of Technology, Federal University of Amazonas, Brazil*
[2]*Samsung Research Institute, Brazil*
[3]*Department of Computer Science, University of Oxford, United Kingdom*
[4] *TPV Technology Limited, Brazil*

Abstract

The Compute Unified Device Architecture (CUDA) is a programming model used for exploring the advantages of graphics processing unit (GPU) devices, through parallelization and specialized functions and features. Nonetheless, as in other development platforms, errors may occur, due to traditional software creation processes, which may even compromise the execution of an entire system. In order to address such a problem, ESBMC-GPU was developed, as an extension to the Efficient SMT-Based Context-Bounded Model Checker (ESBMC). In summary, ESBMC processes input code through ESBMC-GPU and an abstract representation of the standard CUDA libraries, with the goal of checking a set of desired properties. Experimental results showed that ESBMC-GPU was able to correctly verify 85% of the chosen benchmarks and it also overcame other existing GPU verifiers regarding the verification of data-race conditions, array out-of-bounds violations, assertive statements, pointer safety, and the use of specific CUDA features.

Keywords: GPU verification, formal verification, model checking, CUDA

1. Introduction

The Compute Unified Device Architecture (CUDA) is a development framework that makes use of the architecture and processing power of graphics processing units (GPUs) [1]. Indeed, CUDA is also an application programming interface (API), through which a GPU's parallelization scheme and tools can be accessed, with the goal of executing kernels [1]. Nonetheless, source code is still written by human programmers, which may result in arithmetic overflow, division by zero, and other violation types. In addition, given that CUDA allows parallelization, problems related to the latter can also occur, due to thread scheduling [2].

In order to address the mentioned issues, an extension to the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) [3] was developed, named as ESBMC-GPU [4, 5, 6], with the goal of verifying CUDA-based programs (available online at `http://esbmc.org/gpu/`). ESBMC-GPU consists of an extension for parsing CUDA source code (*i.e.*, a front-end to ESBMC) and a CUDA operational model (COM), which is an abstract representation of the standard CUDA libraries (*i.e.*, the native API) that conservatively approximates their semantics.

A distinct feature of ESBMC-GPU, when compared with other approaches [2, 7, 8, 9], is the use of Bounded Model Checking (BMC) [10] allied to Satisfiability Modulo Theories (SMT) [11], with explicit state-space exploration [12, 3]. In summary, concurrency problems are tackled, up to a given loop/recursion unwinding and context bound, while each interleaving itself is symbolically handled; however, even with BMC, state-space exploration may become a very time-consuming task, which is alleviated through state hashing and Monotonic Partial Order Reduction (MPOR) [13]. As a consequence, redundant interleavings are eliminated, without ignoring a program's behavior.

Finally, existing GPU verifiers often ignore some aspects related to memory leak, data transfer, and overflow, which are normally present in CUDA programs. The proposed approach, in turn, explicitly addresses them, through an accurate checking procedure, which even considers data exchange between main program and kernel. Obviously, it results in higher verification times, but more errors can then be identified and later corrected, in another development cycle.

Existing GPU Verifiers. In addition to ESBMC-GPU, there are other tools able to verify CUDA programs and each one of them uses its own approach and targets specific property violations. For instance, GPUVerify [2] is based on synchronous, delayed visibility semantics, which focuses on detecting data race and barrier divergence, while reducing kernel verification procedures for the analysis of sequential programs. GPU+KLEE (GKLEE) [8], in turn, is a concrete plus symbolic execution tool, which considers both *kernels* and *main* functions, while checking deadlocks, memory coalescing, data race, warp divergence, and compilation level issues. In addition, Concurrency Intermediate Verification Language (CIVL) [9], a framework for static analysis and concurrent program verification, uses abstract syntax tree and partial order reduction to detect user-specified assertions, deadlocks, memory leaks, invalid pointer dereference, array out-of-bounds, and division by zero.

In fact, ESBMC-GPU differs from the aforementioned approaches due to its combination of techniques to prune the state-space exploration (*i.e.*, two-thread analysis, state hashing, and MPOR) with COM, which demonstrated effectiveness in the verification of data-race conditions, array out-of-bounds violations, assertive statements, pointer safety, and the use of specific CUDA features (cf. Section 5).

2. Architecture and Implementation

ESBMC-GPU is built on top of ESBMC, which is an open-source context-bounded model checker based on SMT solvers for ANSI-C/C++ programs [12, 3, 14], and adds four essential models, as described below.

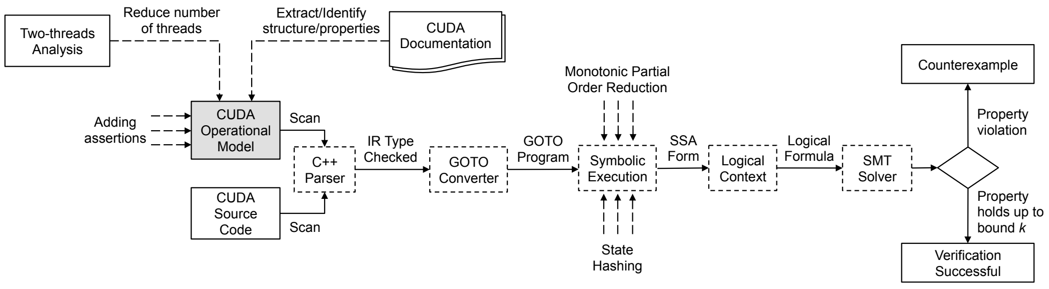

Figure 1: Overview of ESBMC-GPU's architecture.

CUDA Operational Model. An operational model for CUDA libraries that provides support to CUDA functionalities, in conjunction with ESBMC, is shown in Fig. 1. Such an approach, which was previously attempted in the verification of C++ programs [14, 15, 16, 17], consists of an abstract representation that reliably approximates the CUDA library's semantics; however, COM incorporates pre- and post-conditions into verification processes, which enables ESBMC-GPU to verify specific properties (cf. Section 3). Indeed, COM allows the necessary control for performing code analysis, where both CUDA operation and knowledge for model checking its properties are available. Importantly, COM encloses a representation for the Runtime, Math, and cuRAND APIs, which are important CUDA libraries widely used in real applications [1]. In particular, with respect to the number of methods/functions, COM covers 53%, 31%, and 17% of Runtime, Math, and cuRAND APIs, respectively.

ESBMC was designed to handle multi-threaded software, through the use of an API called POSIX – ISO/IEC 9945 [18]. In order to support the verification of CUDA kernels, COM applies code transformations to kernel calls using ESBMC's intrinsic functions [6]. In particular, thread/block configurations in a CUDA kernel call are used as parameters in such intrinsic functions, which are responsible for checking for preconditions, configuring block and threads dimension, and translating GPU threads to POSIX ones. Thus, COM is able to support thread interleaving (to create execution paths) and dynamic creation of threads, in order to check data race and specific C/C++ programming language failures (*e.g.*, array out-of-bounds and pointer safety). ESBMC models the Pthreads API [12] to support thread synchronization, *i.e.*, mutex locking operations and conditional waiting, and dynamic creation of threads, which makes that representation very similar to the official CUDA scheduler [6], in such a way that our multi-threading model approximates to that of GPU kernels. Therefore, COM is able to use such aspects present in ESBMC, in order to handle variables in different types of memory and also in inter-warp communication.

The ESBMC's memory model uses static pointer analysis, padding in structures, with the goal of making all fields align to word boundaries, memory access alignment rules enforcement, and byte array allocation, when the type of memory allocation is unclear [19]. Since ESBMC-GPU is built on top of ESBMC, its memory model for the different types of memory is complete.

Two-threads Analysis. Similarly to GPUVerify [2] and PUG [7], ESBMC-GPU also reduces the number of threads (to only two elements), during the verification of CUDA programs, by considering a NVIDIA Fermi GPU architecture [1], in order to improve verification time and avoid the state-space explosion problem. In CUDA programs, whilst threads execute the same parametrized kernel, only two of them are necessary for conflict check. Thus, such an analysis ensures that errors (*e.g.*, data races) detected between two threads, in a given subgroup and due to unsynchronized accesses to shared variables, are enough to justify a property violation [6].

State Hashing. ESBMC-GPU applies state hashing to further eliminate redundant interleavings and also reduce the state space, based on SHA256 hashes [20]. In particular, its symbolic state hashing approach computes a summary for a particular state that has already been explored and then indexes the resulting set, in order to reduce the generation of redundant states. Given any state computed during the symbolic execution of a specific CUDA kernel, ESBMC-GPU simply summarizes it and efficiently determines whether it has been explored before or not, along a different computation path. When this behavior is confirmed, which happens during the ESBMC-GPU's symbolic-execution procedure, then the current computation path does not need to be further explored in the associated reachability tree (RT). This way, if ESBMC-GPU reaches such a state, *i.e.*, where a context switch can be taken (*e.g.*, before a global variable or synchronization primitive) and all shared/local variables and program counters are similar to another explored node, then ESBMC-GPU just considers that an identical node to be further explored, since reachability subtrees associated to them are also similar [6, 21].

Monotonic Partial Order Reduction. MPOR is used to reduce the number of thread interleavings, by classifying transitions inside a program as dependent or independent. As a consequence, it is possible to determine whether interleaving pairs always lead to the same state and then remove duplicates in a RT, without ignoring any program's behavior [21].

3. Functionalities

COM models CUDA libraries and provides a multi-threading model much similar to the CUDA scheduler (cf. Section 2). Moreover, it is able to simulate CUDA's program structure and memory, being susceptible of handling CUDA programs. Thus, through the integration of COM into ESBMC (*i.e.*, ESBMC-GPU), one is able to analyze CUDA programs and verify the following properties:

- **Data race.** ESBMC-GPU checks data race conditions, in order to detect if multiple threads perform unsynchronized access to the same memory locations;
- **Pointer safety.** ESBMC-GPU also ensures that *(i)* a pointer offset does not exceed object bounds and *(ii)* a pointer is neither `NULL` nor invalid;
- **Array bounds.** ESBMC-GPU performs array-bound checking, in order to ensure that any variable, used as an array index, is within known bounds;
- **Arithmetic under- and overflow.** ESBMC-GPU checks whether a sum or product exceeds the memory limits that a variable can handle, which can cause an error capable of spreading through the entire execution path;
- **Division by zero.** ESBMC-GPU analyzes whether denominators, in arithmetic expressions, lead to divisions by zero;
- **User-specified assertions.** ESBMC-GPU considers all assertions specified by users, which is essential to a thorough verification process, as some specific possible violations must be explicitly pointed out.

In order to check the aforementioned properties, ESBMC-GPU explicitly explores the possible interleavings (up to the given context bound) and calls the single-threaded BMC procedure on each one, whenever it reaches a RT leaf node. Then, the mentioned procedure will stop if it finds a bug or when all possible RT interleavings have been systematically explored [6]. Furthermore, ESBMC-GPU has the following additional command-line options:

- `--no-assertions`: to ignore assertions;
- `--no-bounds-check`: to skip array bounds check;
- `--no-div-by-zero-check`: to skip division by zero check;
- `--no-pointer-check`: to skip pointer check;
- `--memory-leak-check`: to enable memory leak check;
- `--overflow-check`: to enable arithmetic under- and overflow check;
- `--deadlock-check`: to enable global/local deadlock check with mutex;
- `--data-races-check`: to enable data races check;
- `--lock-order-check`: to enable for lock acquisition ordering check;
- `--atomicity-check`: to enable atomicity check at visible assignments;
- `--force-malloc-success`: to consider that there is always enough memory available in the device;

Thus, ESBMC-GPU is able to check CUDA programs for: deadlock, assertion, lock acquisition error, division by zero, pointer safety, arithmetic overflow, and/or out-of-bounds array violation. The precision and performance of ESBMC-GPU will be further discussed in Section 5.

4. Illustrative Example

In this part, ESBMC-GPU usage is demonstrated, by using the CUDA program shown in Fig. 2. First of all, users must replace the default kernel call (line 16) by an intrinsic function of ESBMC-GPU (line 17). Then, the resulting CUDA program can be passed to the command-line version of ESBMC-GPU, as follows:

```
esbmc-gpu <file>.cu --unwind <k> --context-switch <c>
        --state-hashing -I <path-to-CUDA-OM>,
 1 #include <...>
 2 #define BLOCKS 1
 3 #define THREADS 2
 4
 5 __global__ void kernel(int *A) {
 6   A[threadIdx.x + 1] = threadIdx.x;
 7 }
 8
 9 int main(){
10   int *a;
11   int *dev_a;
12   int size = THREADS*sizeof(int);
13   a = (int*)malloc(size);
14   cudaMalloc((void**)&dev_a, size);
15   for (int i = 0; i < THREADS; i++)
16     a[i] = 0;
17   cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
18   // kernel<<<BLOCKS, THREADS>>>(dev_a);
19   ESBMC_verify_kernel(kernel, BLOCKS, THREADS, dev_a);
20   cudaMemcpy(a, dev_a , size , cudaMemcpyDeviceToHost);
21   for (int i = 0; i < THREADS; i++)
22     assert(a[i]==i);
23   cudaFree(dev_a);
24   free(a);
25   return 0;
26 }
```

Figure 2: Illustrative CUDA code example.

where `<file>.cu` is the CUDA program, `<`k`>` is the maximum loop unrolling, `<`c`>` is a context-switch bound, `--state-hashing` reduces redundant interleavings, and <`path-to- CUDA-OM`> is the location of the COM library.

In the mentioned example, ESBMC-GPU detects an array out-of-bounds violation. Indeed, this CUDA-based program retrieves a memory region that has not been previously allocated, *i.e.*, when `threadIdx.x = 1`, the program tries to access $a[2]$. Importantly, the `cudaMalloc()` function's operational model has a precondition that checks if the memory size to be allocated is greater than zero. In addition, an assertion checks if the result matches to the expected postcondition (line 22). The verification of this program through ESBMC-GPU produces 54 successful and 3 failed interleavings. For instance, one possible failed interleaving is represented by the threads executions t_0 : $a[1] = 0$; t_1 : $a[2] = 1$, where $a[2] = 1$ represents an incorrect access to the array index a. It is worth noticing that CIVL, ESBMC-GPU, and GKLEE are also able to detect this array out-of-bounds violation, but GPUVerify fails, as it reports a true incorrect result (*i.e.*, a missed bug).

5. Experimental Evaluation

In order to evaluate ESBMC-GPU's precision and performance, a benchmark suite was created, which comprises 20 CUDA kernels from NVIDIA GPU Computing SDK $v2.0$ [22], 20 CUDA kernels from Microsoft C++ AMP Sample Projects [23], and 114 CUDA-based programs that explore a wide range of CUDA functionalities. In summary, the chosen suite contains 47.4% bug-free and 52.6% buggy benchmarks, which were organized into 5 sets, in order to simplify our discussion, according to the properties it tackles: array bounds (5), assertive statements (7), data-race conditions (17), pointer safety (7), and other specific CUDA functionalities (118). The latter includes `__device__` function calls, general CUDA functions (*e.g.*, `cudaMemcpy`), general libraries in CUDA (*e.g.*, *curand.h*), type modifiers (*e.g.*, `unsigned`), type definitions, and intrinsic CUDA variables (*e.g.*, `uint4`).

The present experiments answer two research questions: *(i)* How accurate is ESBMC-GPU when verifying the chosen benchmarks? *(ii)* How does ESBMC-GPU's performance compare to other existing verifiers? In order to answer both research questions, all benchmarks were verified with 4 GPU verifiers (ESBMC-GPU *v*2.0, GKLEE *v*2012, GPUVerify *v*1811, and CIVL *v*1.7.1), on an otherwise idle Intel Core i7-4790 CPU 3.60 GHz, with 16 GB of RAM, running Ubuntu 14.04 OS. Importantly, all presented execution times are actually CPU times, *i.e.*, only the elapsed time periods spent in the allocated CPUs, which was measured with the `times` system call (POSIX system). An overview of the experimental results is shown in Fig. 3, where *True* represents bug-free benchmarks, *False* represents buggy benchmarks, *Not supported* represents benchmarks that could not be verified, *Correct* represents the percentage of benchmarks correctly verified, and *Incorrect* represents the percentage of benchmarks incorrectly verified (*i.e.*, a verification tool reports an unexpected result).

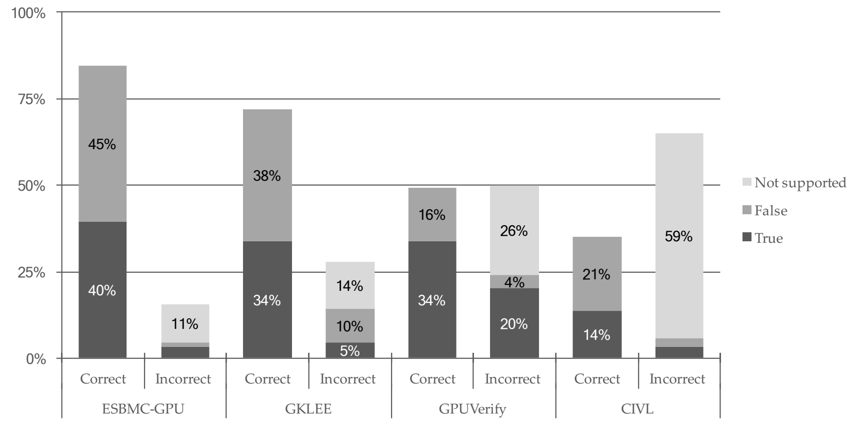

Figure 3: Experimental evaluation of ESBMC-GPU against other verifiers.

As one may notice, the present experimental results show that ESBMC-GPU reached a successful verification rate of approximately 85%, while GKLEE, GPUVerify, and CIVL reported 72%, 50%, and 35%, respectively. More precisely, ESBMC-GPU correctly detected all data-race conditions present in the benchmarks, which is due to the COM's multi-threading model that under-approximates the GPU kernels. It also outperformed GKLEE, GPU-

Verify, and CIVL, in the verification of array out-of-bounds violations (100%), assertive statements (86%), and pointer safety (72%), which is related to ESBMC's capacity to handle arrays and pointers [3]. Furthermore, ESBMC-GPU presented the highest coverage rate for specific CUDA functionalities (82%) that is once again due to COM, which incorporates specific pre- and post-conditions into its verification processes.

Limitations. ESBMC-GPU was unable to correctly verify 24 benchmarks, which are related to constant memory access (2%), CUDA's specific libraries (*e.g.*, `curand.h`) (4.5%), and the use of pointers to functions, structures, and `char` type variables, when passed as kernel call arguments (4.5%). In addition, it only reported 3% of incorrect true, which are due to `NULL` pointer accesses, and 1% of incorrect false results, due to partial coverage of the *cudaMalloc* function for copies over `float` variables. The remaining verifiers (*i.e.*, GKLEE, GPUVerify, and CIVL) were unable to detect mostly data-race conditions, assertive statements, and array out-of-bounds violations. In addition, they lack support of CUDA specific features, *e.g.*, GPUVerify does not support the use of the `memset` function nor function pointers and CIVL does not support several CUDA features, such as `atomic` functions, `cudaThreadSynchronize`, `threadIdx`, `curand` functions, `dim3`, `math_functions`, `uint4`, `__constant__` variables, among others.

Performance. MPOR resulted in a performance improvement of approximately 80%, by decreasing the verification time from 16 to 3 hours, while the two-threads analysis further reduced that to 789.6 sec. Although such techniques have considerably improved the ESBMC-GPU's performance, it still takes longer than the other evaluated tools: GPUVerify (98.36 sec), GKLEE (105.18 sec), and CIVL (708.52 sec). On the one hand, this is due to thread interleavings, which combine symbolic model checking with explicit state-space exploration [6]. On the other hand, ESBMC-GPU still presents the highest accuracy, with less than 6 seconds per benchmark.

Availability of Data and Tools. The performed experiments are based on a set of publicly available benchmarks. All benchmarks, tools, and results, associated with the current evaluation, are available at `www.esbmc.org/gpu/`.

6. Conclusions and Future Work

ESBMC-GPU marks the first application of an SMT-based context-BMC tool that recognizes CUDA directives [6]. Besides, it also applies MPOR, two-thread analysis, and state hashing, in order to further simplify verification models and provides fewer incorrect results, compared with GKLEE, GPUVerify, and CIVL. Indeed, it presents improved ability to detect array out-of-bounds and data race violations.

Future work aims to extend ESBMC-GPU, in order to fully support the verification of CUDA (parallel) streams and events [1]. In addition, more models of libraries will be integrated into COM, with the goal of increasing the coverage of CUDA's API such as CUDA Driver API, NPP, and cuSOLVER. Finally, we also aim to implement further techniques (*e.g.*, invariant inference via abstract interpretation [24]), in order to prune the state-space exploration, by taking into account GPU symmetry.

Acknowledgements

This paper is based on research sponsored by the Institute of Development and Technology (INdT) and by the National Council for Scientific and Technological Development (CNPq) under agreement number $475647/2013-0$.

References

[1] J. Cheng, M. Grossman, T. McKercher, *Professional CUDA C Programming,* John Wiley and Sons, Inc., Indianapolis, Indiana, USA, 2014.

[2] A. Betts, N. Chong, A. Donaldson, S. Qadeer, P. Thomson, *GPUVerify: A Verifier for GPU Kernels,* in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, 2012, pp. 113–132, URL `http://doi.acm.org/10.1145/2384616.2384625`.

[3] L. Cordeiro, B. Fischer, J. Marques-Silva, *SMT-Based Bounded Model Checking for Embedded ANSI-C Software,* IEEE Trans. Software Eng. 38 (2012) 957–974, URL `https://doi.org/10.1109/TSE.2011.59`.

[4] P. Pereira, H. Albuquerque, H. Marques, I. Silva, C. Carvalho, V. Santos, R. Ferreira, L. Cordeiro, *Verificação de Kernels em Programas CUDA usando Bounded Model Checking,* in: XV Brazilian Symposium on High-Performance Computing (WSCAD-SSC), Florianópolis, Brazil, 2015, pp. 24–35.

[5] Pereira, P., Albuquerque, H., Marques, H., Silva, I., Carvalho, C., Santos, V., Ferreira, R., and Cordeiro, L. *Verifying CUDA Programs using SMT-Based Context-Bounded Model Checking.* in: Proceedings of the 31^{st} Annual ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2016, pp. 1648–1653, URL `http://doi.acm.org/10.1145/2851613.2851830`.

[6] P. Pereira, H. Albuquerque, I. Silva, H. Marques, F. R. Monteiro, R. Ferreira, L. Cordeiro, *SMT-Based Context-Bounded Model Checking for CUDA Programs,* Concurrency Computat.: Pract. Exper. (2016), URL `http://dx.doi.org/10.1002/cpe.3934`.

[7] G. Li, G. Gopalakrishnan, *Scalable SMT-based Verification of GPU Kernel Functions,* in: Proceedings of the 18^{th} ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, 2010, pp. 187–196, URL `http://doi.acm.org/10.1145/1882291.1882320`.

[8] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, S. Rajan, *GKLEE: Concolic Verification and Test Generation for GPUs,* in: Proceedings of the 17^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, New York, NY, USA, 2012, pp. 215–224, URL `http://doi.acm.org/10.1145/2145816.2145844`.

[9] M. Zheng, M. Rogers, Z. Luo, M. Dwyer, S. Siegel, *CIVL: Formal Verification of Parallel Programs,* in: Proceedings of the 30^{th} IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA, 2015, pp. 830–835, URL `http://dx.doi.org/10.1109/ASE.2015.99`.

[10] A. Biere, *Bounded Model Checking,* in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, Amsterdam, The Netherlands, 2009, pp. 457–481.

[11] C. Barrett, R. Sebastiani, S. Seshia, C. Tinelli, *Satisfiability Modulo Theories,* in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, Amsterdam, The Netherlands, 2009, pp. 825–885.

[12] L. Cordeiro, B. Fischer, *Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking,* in: Proceedings of the 32^{nd} ACM/IEEE International Conference on Software Engineering, ACM, New York, NY, USA, 2011, pp. 331–340, URL `http://doi.acm.org/10.1145/1810295.1810396`.

[13] V. Kahlon, C. Wang, A. Gupta, *Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique,* in: Proceedings of the 21^{st} International Conference on Computer Aided Verification, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 398–413, URL `http://dx.doi.org/10.1007/978-3-642-02658-4_31`.

[14] M. Ramalho, M. Freitas, F. R. Monteiro, H. Marques, L. Cordeiro, B. Fischer, *SMT-Based Bounded Model Checking of C++ Programs,* in: Proceedings of the 20th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems, IEEE Computer Society, Washington, DC, USA, 2013, pp. 147–156, URL `http://dx.doi.org/10.1109/ECBS.2013.15`.

[15] F. R. Monteiro, L. Cordeiro, E. B. de Lima Filho, *Bounded Model Checking of C++ Programs Based on the Qt Framework,* in: IEEE 4th Global Conference on Consumer Electronics, IEEE Consumer Electronics Society, Washington, DC, USA, 2015, pp. 179–447, URL `https://doi.org/10.1109/GCCE.2015.7398699`.

[16] M. Garcia, F. R. Monteiro, L. Cordeiro, E. B. de Lima Filho, *ESBMC^{QtOM}: A Bounded Model Checking Tool to Verify Qt Applications,* in: 23rd International Symposium on Model Checking Software, Springer International Publishing, Cham, 2016, pp. 97–103, URL `http://dx.doi.org/10.1007/978-3-319-32582-8_6`.

[17] F. R. Monteiro, M. Garcia, L. Cordeiro, E. B. de Lima Filho, *Bounded Model Checking of C++ Programs based on the Qt Cross-Platform Framework,* Softw Test Verif Reliab. 27 (2017) e1632, URL `http://dx.doi.org/10.1002/stvr.1632`.

[18] Institute of Electrical and Electronics Engineers Inc., *IEEE Standard for Information Technology-Portable Operating System Interface (POSIX) Base Specifications,* IEEE Std. 1003.1-2008 (Revision of IEEE Std. 1003.1-2004). (2008) c1-3826, URL `https://doi.org/10.1109/IEEESTD.2008.4694976`.

[19] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, B. Fischer, *ESBMC* 1.22 *(Competition Contribution),* in: 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, Berlin, Heidelberg, 2014, pp. 405–407, URL `http://dx.doi.org/10.1007/978-3-642-54862-8_31`.

[20] National Institute of Standards and Technology, *Secure Hash Standard (SHS),* Federal Information Processing Standards Publication 180-4. (2015), URL `http://dx.doi.org/10.6028/NIST.FIPS.180-4`.

[21] J. Morse, *Expressive and Efficient Bounded Model Checking of Concurrent Software,* University of Southampton, PhD Thesis, Southampton, UK, 2015.

[22] NVIDIA Corporation, *CUDA Toolkit Archive.* `https://developer.nvidia.com/cuda-toolkit-archive`, 2015 (accessed 01.06.2017).

[23] D. Moth, *C++ AMP Sample Projects for Download.* Microsoft Corporation, `blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx`, 2012 (accessed 01.06.2017).

[24] W. Rocha, H. Rocha, H. Ismail, L. Cordeiro, B. Fischer, *DepthK: A k-Induction Verifier Based on Invariant Inference for C Programs,* in: 23^{rd} International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, Berlin, Heidelberg, 2017, pp. 360–364, URL `http://dx.doi.org/10.1007/978-3-662-54580-5_23`.

Required Metadata

Current executable software version

Ancillary data table required for sub version of the executable software.

Nr.	**(executable) Software metadata description**	**Please fill in this column**
S1	Current software version	2.0
S2	Permanent link to executables of this version	`http://esbmc.org/gpu/`
S3	Legal Software License	Apache v2.0
S4	Computing Operating System	Ubuntu Linux OS
S5	Installation requirements & dependencies	GNU Libtool; Automake; Flex & Bison; Boost C++ Libraries; Multiprecision arithmetic library developers tools (libgmp3-dev package); SSL development libraries (libssl-dev package); CLang 3.8; LLDB 3.8; GNU C++ compiler (multilib files); libc6 and libc6-dev packages
S6	Link to user manual	`http://esbmc.org/gpu/`
S7	Support email for questions	`lucas.cordeiro@cs.ox.ac.uk`

Table 1: Software metadata (optional)

Current code version

Ancillary data table required for subversion of the codebase.

Nr.	**Code metadata description**	**Please fill in this column**
C1	Current code version	$v2.0$
C2	Permanent link to code/repository used for this code version	`https://github.com/ssvlab/esbmc-gpu`
C3	Legal Code License	GNU Public License
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	C++
C6	Compilation requirements, operating environments & dependencies	GNU Libtool; Automake; Flex & Bison; Boost C++ Libraries; Multi-precision arithmetic library developers tools (libgmp3-dev package); SSL development libraries (libssl-dev package); CLang 3.8; LLDB 3.8; GNU C++ compiler (multilib files); libc6 and libc6-dev packages
C7	Link to developer documentation	`http://esbmc.org/gpu`
C8	Support email for questions	`lucas.cordeiro@cs.ox.ac.uk`

Table 2: Code metadata (mandatory)