ESBMC v7.7: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction

(Competition Contribution)

Tong Wu[1⋆], Xianzhiyu Li[1], Edoardo Manino[1], Rafael Sá Menezes[1,2], Mikhail R. Gadelha[3], Shale Xiong[5], Norbert Tihanyi[4], Pavlos Petoumenos[1], and Lucas C. Cordeiro[1,2]

[1] The University of Manchester, Manchester, UK
[2] Federal University of Amazonas, Manaus, Brazil
[3] Igalia, A Coruña, Spain
[4] Eötvös Loránd University, Budapest, Hungary
[5] Arm®, Cambridge, UK

Abstract. ESBMC v7.7 improves the verification of concurrent C programs by incorporating techniques such as dynamic thread scheduling, incremental SMT solving, and partial order reduction (POR). These improvements enhance the tool's performance, particularly in exploring complex multi-threaded executions. The new scheduler prioritizes higher-thread identifiers during context switches, which helps explore deeper program states. The use of incremental SMT solving and a refined POR algorithm reduces the exploration of unreachable interleavings and redundant states. These updates enable ESBMC to detect bugs faster, making it a more effective tool for ensuring the safety of multi-threaded applications.

1 Software Architecture

The Efficient SMT-based context-Bounded Model Checker (ESBMC) [**?**,**?**,**?**] is a context-bounded model checker for the verification of single- and multi-threaded software. It uses a Clang-based [**?**] front-end to transform the input C program into an intermediate representation in the GOTO language [**?**]. Then, it employs symbolic execution to generate SMT formulae and pass them to a selection of SMT solvers. For the verification of multi-threaded software, ESBMC constructs the reachability tree by depth first search under Sequential Consistency [**?**]. This way, it can symbolically and explicitly explore all possible sequential executions up to a (bounded) number of context switches. As a result, ESBMC can automatically identify many concurrency-related issues such as race conditions and deadlocks.

⋆ Jury member

2 Verification Approach

Reverse Priority Scheduling. ESBMC v7.7 introduces an advanced thread scheduling algorithm that dynamically prioritizes thread selection when a context switch occurs. In earlier versions, the scheduler would always search for eligible threads in ascending order of thread identifier t_i,[6] starting with the main thread id $t_i = 0$ [**?**]. The new approach modifies this behavior as follows:

① It first attempts to identify newer thread, which have identifier t_i higher than the current thread t_c. Among these threads, the scheduler selects the one with the smallest identifier that is eligible for scheduling.
② If no newer threads are available, the scheduler reverses the search direction and looks for older eligible threads, which have identifier $t_i \leq t_c$. Note that the scheduler can still choose t_c, which indicates it will continue executing the following steps in the current thread. In this case, it selects the eligible thread with the largest identifier.
③ If no eligible threads are found in either direction, there is no further interleaving possible in the current execution state. Thus, the scheduler reverts to exploring unexplored states by popping the current state from the reachability tree and backtracking.

The formal policy σ for our reverse priority scheduling can be expressed as:

$$\sigma(t_c, S) = \begin{cases} \min\{t_i | t_i \in S \wedge t_i > t_c\} & \text{if } \max S > t_c \quad ① \\ \max S & \text{if } \max S \leq t_c \quad ② \\ \emptyset & \text{if } S = \emptyset \quad ③ \end{cases}$$

where t_c is the current thread and S represents the set of all threads eligible for scheduling. In general, this strategy prioritizes interleavings with newly-created threads, enabling ESBMC to explore new execution paths earlier, which allows ESBMC to find bugs six times faster (see Section 3).

Incremental SMT Solving. ESBMC leverages incremental SMT solving [**?**] in an attempt to reduce the exploration of unreachable interleavings. Specifically, non-incremental mode only calls the SMT solver once when reaching the end of an interleaving. As such, it has no early mechanism to determine which boolean conditions hold (e.g., assumptions), thus producing a formula that may still contain some unreachable states. In contrast, incremental mode removes all unreachable states by checking goto guards (if and loop conditions), thread guards (interleaving conditions) and assertions immediately [**?**]. This is achieved through the push/pop interface offered by many state-of-the-art solvers [**?**,**?**,**?**,**?**]. Figure 1 illustrates the different behaviour after the thread guard `pthread_join`. In ESBMC v7.7, we add incremental checking of assumptions with the `--smt-symex-assume` option.

[6] New threads are created with higher identifiers in ESBMC.

```
1  #include <pthread.h>
2  int x = 0;
3  void *thread1() {
4    x = 1;
5    return 0;
6  }
7  int main() {
8    pthread_t t1;
9    pthread_create(&t1,
         0, thread1, 0);
10   pthread_join(t1, 0);
11   x = 2;
12 }
```

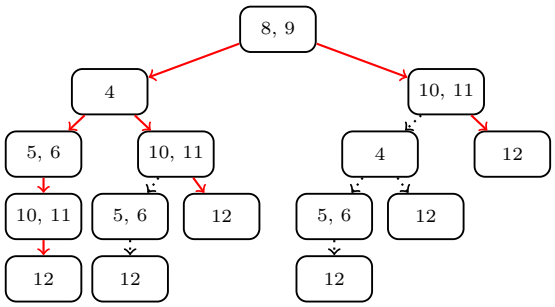

Fig. 1. Reachability tree (right) of a concurrent program (left). Boxes are execution states (by line number), red arrows are reachable paths, black dotted arrows are unreachable paths that can be cut by incremental SMT.

Partial Order Reduction. ESBMC employs an optimal partial order reduction algorithm based on the normal form in [**?**] to eliminate redundant equivalent interleavings during model checking. In ESBMC v7.7, we refine our implementation by performing a more accurate analysis of shared read/write variables that are accessed by global and local pointers. Specifically, memory leak checks are deferred until the main thread has terminated. This strategy avoids false positives, since in SV-COMP the check should happen after program termination.

Data Races. ESBMC v7.7 offers extended support for data race checking. The base method introduces a boolean flag b_x for each variable x involved in an assignment. When x is updated, b_x is set to true before the assignment and reset to false immediately after. To identify data races, an assertion ensures that b_x is false whenever x is accessed. Previous versions relied on unique flag identifiers generated from variable names. As such, the same memory address could be protected by multiple flags, especially for arrays with non-constant indices and members of compound types. The new version eliminates the dependency on variable names. Memory is represented as an infinite-size array, where variables are encoded as addresses and used as indices of the array. Additionally, we force an extra context switch when setting each flag to increase the probability of triggering a data race earlier.

Pthread Operational Models. ESBMC v7.7 features improved operational models for the pthread library, which reduces the number of unnecessary context switches. Specifically, we restrict context switches to occur exclusively on user program variables which are shared by at least two threads.

3 Strengths and Weaknesses

Table 1 reports the relative performance of each individual technique in Section 2, compared to the previous version of ESBMC. The last row reports their cumulative effect in ESBMC v7.7, which has Incremental SMT disabled and POR

partially enabled by default (more information can be found in zenodo dataset). All results are computed on the SV-COMP'25 ConcurrencySafety benchmarks.

Table 1. Experimental Results for ESBMC Improvements

Individual Technique	Correct True	Correct False	Incorrect True	Incorrect False
Reverse Priority Scheduling	+66	+18	+8	+7
Incremental SMT Solving	-3	+1	+3	+6
Partial Order Reduction	+16	-20	+15	+0
Data Races	-5	+16	-9	-14
Pthread Operational Models	+31	+3	+5	+3
ESBMC v7.7	**+97**	**+21**	**+13**	**+10**

In addition, our experiments show that Reverse Priority Scheduling accelerates bug detection by approximately 600%, and the first interleaving is enough to reach a bug in 59 additional instances. Furthermore, Incremental SMT achieves an average of 53% fewer interleavings on the 248 instances verified by both approaches and produces 18 unique correct results that would otherwise timeout. However, the repeated solver calls increase verification time, with fewer successful verifications than the non-incremental mode. Finally, Partial Order Reductions reduce the verification time required to prove correctness by 40%.

At the same time, the improvements expose 23 new incorrect results that were previously timeouts. Additionally, the Partial Order Reduction improvements turn 20 correct results into 15 incorrect results and 5 timeouts. These are mainly due to the low context-switch limit of 3 we set and the absence of a mechanism to detect shared memory access between local variables and pointers. We plan to address these issues in future work.

4 Tool Setup and Configuration

To setup and run ESBMC, follow the instructions in the README.md file. ESBMC can also be run via the Python wrapper `esbmc-wrapper.py` for simplifed usage in the competition. An example command line is: `esbmc-wrapper.py -s kinduction -a 64 -p unreach-call.prp example.c`

5 Software Project

The ESBMC development is funded by ARM, EPSRC EP/T026995/1, EPSRC EP/V000497/1, Ethereum Foundation, EU H2020 ELEGANT 957286, UKRI Soteria, Intel, and Motorola Mobility (through Agreement N° 4/2021). It is publicly available at `http://esbmc.org` under the Apache License 2.0. The participated version in SV-COMP 2025 is available at `https://doi.org/10.5281/zenodo.13867976`. A refined version and all of our experiments data are available at `https://doi.org/10.5281/zenodo.14534503`.