A Method to Localize Faults in Concurrent C Programs

Erickson H. da S. Alves[12], Lucas C. Cordeiro[13*], Eddie B. de L. Filho[14*]

[1] *Electronic and Information Research Center, Federal University of Amazonas, Brazil*

[2] *Samsung Research Institute Brazil, Amazonas, Brazil*

[3] *Department of Computer Science, University of Oxford, United Kingdom*

[4] *FPF Tech, Amazonas, Brazil*

Abstract

We describe a new approach to localize faults in concurrent programs, which is based on bounded model checking and sequentialization techniques. The main novelty is the idea of reproducing a faulty behavior, in a sequential version of a concurrent program. In order to pinpoint faulty lines, we analyze counterexamples generated by a model checker, to the new instrumented sequential program, and search for a diagnostic value, which corresponds to actual lines in a program. This approach is useful to improve debugging processes for concurrent programs, since it tells which line should be corrected and what values lead to a successful execution. We implemented this approach as a code-to-code transformation from concurrent into non-deterministic sequential programs, which are used as inputs to existing verification tools. Experimental results show that our approach is effective and capable of identifying faults in our benchmark set, which was extracted from the SV-COMP 2016 suite.

Keywords: Concurrent Software, Bounded Model Checking, Fault Localization, Non-determinism, Sequentialization

1. Introduction

Discovering errors in software is an activity that needs to be performed at the early stages of development processes, so that costs are reduced and,

*Corresponding author

Email address: lucas.cordeiro@cs.ox.ac.uk (Lucas C. Cordeiro[13])

in some cases, lives are saved [1]. Several methods have been proposed, in order to find errors in software, such as testing [2] and model-based methods [3, 4, 5, 6, 7, 8, 9]. More deeply, when an error is found, its cause needs to be tracked inside source code files, which is one of the most time-consuming activities in program debugging [10]. Nonetheless, when it comes to concurrent programs, asserting their correctness turns out to be complex, since the number of possible interleavings can exponentially grow, with the number of threads and statements, and finding errors then becomes an exhausting task to developers. It is worth noticing that, in the worst case, faulty lines may not even be identified.

In this study, we describe a method to localize faults in concurrent C programs, based on the method initially proposed by Griesmayer *et al.* [11], which consists in using non-determinism, in a sequential program, to find values for a successful program run, thus correcting original errors. It is based on *sequentialization*, which is an approach proposed by Qadeer *et al.* [12] that employs verification tools originally developed for sequential programs, in order to check safety properties in concurrent programs, and *bounded model checking* (BMC), which is a technique introduced by Biere *et al.* [13], as an approach to bound the search for counterexamples through a depth k, thus producing formulae that are easier to be solved.

In our prior work [14], we extended the method described by Griesmayer *et al.* [11], with the goal of finding faults in concurrent programs, translated concurrent statements to sequential ones, and applied ESBMC [15] as our BMC tool. Here, we continue this work and improve our methodology to precisely model concurrent programs. In summary, we extend our sequentialized Pthread library [16] model, mainly regarding mutex type modeling, address benchmarks with deadlocks, and expand Grismayer's method, in order to diagnose bad formulated assertions. We also use Lazy-CSeq [17], as another BMC tool, which aims to improve our effectiveness in handling concurrent programs. Finally, we exploit more benchmarks, in order to evaluate our method and point out its advantages and disadvantages.

1.1. Availability of Data and Tools

Our experiments are based on a set of publicly available benchmarks. All tools, benchmarks, and results of our evaluation are available on a supplementary web page `http://esbmc.org/benchmarks/ejss2016.zip`.

1.2. Organization of this Work

The remainder of this paper is organized as follows. Section 2 describes background theories and concepts used in this work. Section 3 is the main section of this paper, where we detail our approach to localize faults in concurrent C programs. Section 4 provides experimental results, analysis, and discussions, while related work is briefly described in section 5. Finally, section 6 summarizes the main obtained results and highlights future work.

2. Background

In this section, we discuss the verification process of the two model checking tools used in our work, show how counterexamples can provide useful information to characterize faults, and then demonstrate a method to localize faults in sequential programs.

2.1. Pthread Library

We have to model concurrent statements of a program P to corresponding sequential statements, in order to simulate its original behavior in a sequential approach. Such a transformation is focused on the C programming language, since our focus is on concurrent C programs, and, in particular, the UNIX environment. In that programming language and its environment, threads are defined through the Portable Operating System Interface (POSIX) standard and are called POSIX Threads (Pthread), as implemented in the *pthread* library [16]. Regarding *pthread* library statements, we model thread creation, termination, and synchronization. In particular, our focus is on mutexes and conditionals, since they are the structures where the majority of concurrent bugs occur [18]. Besides, since our method reproduces a faulty behavior of a concurrent program and not all possible executions, thread creation and termination can be abstracted as the beginning and the end of the thread function execution, respectively [19]. Therefore, we currently do not support thread cancelling and joining.

2.2. ESBMC

In ESBMC, the program to be verified is modeled as a state transition system $M = (S, R, s_0)$, which is extracted from its control-flow graph (CFG) [20, 21]. S represents the set of states, $R \subseteq S \times S$ represents the set of transitions (*i.e.*, pairs of states specifying how the system can jump from state s_i to state s_{i+1}), and $s_0 \subseteq S$ represents the set of initial states. A state

$s \in S$ consists in values of the associated program counter pc and all program variables, and an initial state s_0 assigns the initial program location of the related CFG to pc. We identify each transition $\gamma = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1}, with a logical formula $\gamma(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter and other program variables.

Given the transition system M, a safety property ϕ, a context bound C, and a bound k, ESBMC builds a reachability tree (RT) that represents the program unfolding for C, k, and ϕ [19, 22]. We then derive a verification condition (VC) ψ_k^π for each given interleaving (or computation path) $\pi = \{v_1, ..., v_k\}$, which is given by the following logical formula

$$\psi_k^\pi = \overbrace{I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})}^{constraints} \wedge \overbrace{\neg\phi(s_i)}^{property}, \tag{1}$$

where I characterizes the set of initial states M and $\gamma(s_j, s_{j+1})$ is the relation of M between time steps j and $j+1$. Hence, $I(s_0) \wedge \bigvee_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents executions of M of length i and ψ_k^π can be satisfied if and only if, for some $i \leq k$, there exists a reachable state along π, at time step i, in which ϕ is violated. ψ_k^π is a quantifier-free formula in a decidable subset of first-order logic, which is checked for satisfiability by an SMT solver. If ψ_k^π is satisfiable, then ϕ is violated along π and the SMT solver provides a satisfying assignment, from which we can extract values of program variables to construct a counterexample. The latter, regarding a property ϕ, is a sequence of states $s_0, s_1, ..., s_k$, with $s_0 \in S_0$, $s_k \in S$, and $\gamma(s_i, s_{i+1})$ for $0 \leq i < k$. If ψ_k^π is unsatisfiable, we can conclude that no error state is reachable in k steps or less, along π. Finally, we can define $\psi_k = \bigwedge_\pi \psi_k^\pi$ and use it to check all paths. Nonetheless, ESBMC combines symbolic model checking with explicit state space exploration; in particular, it explicitly explores the possible interleavings (up to the given context bound), while symbolically treating each one. ESBMC implements different variations of this approach, which differ in the way they exploit the RT. The most effective variation simply transverses the RT depth-first and calls the single-threaded BMC procedure on the interleaving, whenever it reaches an RT leaf node, and stops when a bug is found or all possible RT interleavings were systematically explored.

2.3. Lazy-CSeq

Lazy-CSeq is a code-to-code transformation tool based on Lazy Sequentialization techniques, initially introduced by La Torre *et al.* [23] for non-deterministically sequentializing concurrent C programs, which re-uses existing BMC tools as backends, in order to find code violations [17]. Its main idea is to pre-process a concurrent program and convert it into a non-deterministic sequential one, with the goal of simplifying verification tasks.

In Lazy-CSeq, a program P is sequentialized w.r.t. a number of rounds K and a depth bound k. In particular, each thread is simulated as a function and the main one of the sequentialized program P_K^{seq} is responsible for scheduling each simulated thread, by using a lazy approach and a round-robin scheduling schema. This process reduces the non-determinism present in a program, which reduces the amount of possible interleavings, but leads to efficient verification tasks. Finally, it model checks P_K^{seq} in an existing BMC tool (similar to the description in 2.2), such as ESBMC [15] and CBMC [24], which states whether a given program is safe or not. Lazy-CSeq is also able to provide a counterexample to unsafe programs; however, developers have only made that available when using CBMC [24] as backend.

2.4. Using Counterexamples to Localize Faults

In model checking, the most essential activity, with respect to fault localization, is to generate a counterexample, which is produced when a program does not satisfy a given specification. A counterexample does not merely provide information about the cause-effect relation of a given violation, but it can also assist in fault localization, as mentioned by Clarke *et al.* [25, 26]. Nonetheless, actual faulty lines may not be easily identified, given the massive amount of information that is obtained from a counterexample.

Several methods have been proposed, in order to localize possible fault causes, using counterexamples. Ball *et al.* [27] introduced an approach that tries to isolate possible causes of counterexamples, generated by the SLAM model checker [28]. Its main idea is that potential faulty lines can be isolated by comparing transitions among obtained counterexamples and successful traces, since the ones not present in correct traces are potential causes of errors. Groce *et al.* [29], in turn, state that if a counterexample exists, a similar but successful trace also exists and can be obtained using BMC techniques. This way, program elements related to a given violation are suggested by the minimal differences between that counterexample and a successful trace,

which is known as the Java Pathfinder approach [30] and can also provide execution paths that lead to error states, regarding concurrent programs (*e.g.*, data race). The key concept of the approach described by Groce *et al.* [31] is similar to the latter and uses alignment constraints to associate states, in a counterexample, with corresponding states in a successful trace, which are generated by a constraint solver. The mentioned states are abstract states over predicates, which represent concrete states in a trace. By using distance metric properties, constraints can be applied to represent program executions, and non-matching constraints that represent concrete states possibly lead to faults. Additionally, if a distance metric property is not satisfied, a counterexample is generated by the respective BMC tool [31].

2.5. Fault Localization in Sequential Programs

Griesmayer *et al.* [11] proposed a method based on BMC techniques, which can directly identify potential faults in programs. In particular, that method uses additional numerical variables, *e.g.* `diag` elements, in order to pinpoint faulty lines, in a given program.

Each line of a program, representing a statement `S`, is transformed into a logic version of such statement, while the value held by `S` is either non-deterministically chosen by a model checker (if the value of `diag` is the same as the one representing the line related to statement `S`) or the one originally specified. If a model checker identifies a `diag` value, by correcting this line in the original program, such fault can be avoided. In the case of multiple `diag` values, correcting the associated lines lead to a successful program execution. In order to find the full set of lines that cause faulty behavior in a program, a new specification[1] can be added to its source code, which is then rerun by a model checker. This process is repeatedly executed until no more `diag` values are obtained, *i.e.*, the run succeeds [32].

In order to illustrate how this method works, one may consider the following digital controller based on the function for motion with constant acceleration [33], as shown in Eq. 2, whose behavior is defined in Eq. 3 (the values were arbitrarily assigned).

$$s(t) = at^2/2 + v_0 t + s_0 \text{ [33]} \tag{2}$$

[1] `assume(diag != a)`

$$c(t) = t^2 - 3t + 2 \tag{3}$$

A model in the C language of the controller is addressed in Fig. 1.

```
#include <stdio.h>
#include <assert.h>

#define C_0 2
#define C_1 0
#define C_2 0
#define C_3 2

const int A = 1;
const int B = -2; // Incorrect modeling
const int C = 2;

int controller(int t) {
  int output = A * t * t + B * t + C;
  return output;
}

int main() {
  assert(controller(0) == C_0 &&
         controller(1) == C_1 &&
         controller(2) == C_2 &&
         controller(3) == C_3);
  return 0;
}
```

Figure 1: Sequential code of a trivial controller.

One may notice that the model is not in conformity with the given equation, in particular, the B term is assigned with the value of -2 instead of -3. Thus, the assertion is expected to fail while executing the program in a model checker, as shown in Fig. 2.

```
...
Violated property:
  file model.c line 19 function main
  assertion
  FALSE

VERIFICATION FAILED
```

Figure 2: Counterexample trace to the model.

By using ESMBC as the model checker, a non-deterministic instrumented program is obtained, as in Fig. 3. After executing the program, the *diag* values from Fig. 4 are obtained.

```
#include <stdio.h>
#include <assert.h>
#define C_0 2
#define C_1 0
#define C_2 0
#define C_3 2
const int A = 1;
const int B = -2;
const int C = 2;
int nondet(int i) {
  int ret;
  __ESBMC_assume(ret != i);
  return ret;
}
int controller(int t) {
  int diag = nondet(0);
  int ta = (diag == 1 ? nondet(A) : A) *
           t * t;
  int tb = (diag == 2 ? nondet(B) : B) * t;
  int tc = (diag == 3 ? nondet(C) : C);
  int output = ta + tb + tc;
  return output;
}
int main() {
  __ESBMC_assume(controller(0) == C_0 &&
                 controller(1) == C_1 &&
                 controller(2) == C_2 &&
                 controller(3) == C_3);
  assert(0);
  return 0;
}
```

Figure 3: Instrumented sequential code with the described method applied.

According to the counterexample obtained using ESBMC, it is possible to notice that the value of *diag* is 2 in three cases and a signed integer in another case. Therefore, the problem is indeed in the calculation of the second term. The complete counterexample shows that the assignment needed for repairing the faulty behavior is -3, which can be done and followed by a rerun of the model checker.

After correcting the mentioned fault, the verified code in Fig. 5 is executed in ESBMC and the lines in Fig. 6 are obtained, meaning that there are no left faults. In digital controllers, it is of great importance that models are precisely specified, in order to avoid faults while operating in real en-

```
griesmayer::controller::1::diag=-2012462479 (-2012462479)
griesmayer::controller::1::diag=2 (2)
griesmayer::controller::1::diag=2 (2)
griesmayer::controller::1::diag=2 (2)
```

Figure 4: Faulty lines obtained by the program execution 3.

```
#include <stdio.h>
#include <assert.h>
#define C_0 2
#define C_1 0
#define C_2 0
#define C_3 2
const int A = 1;
const int B = -3;
const int C = 2;
int controller(int t) {
  int output = A * t * t + B * t + C;
  return output;
}
int main() {
  assert(controller(0) == C_0 &&
         controller(1) == C_1 &&
         controller(2) == C_2 &&
         controller(3) == C_3);
  return 0;
}
```

Figure 5: Repaired sequential code.

vironment, since they can lead to unexpected behaviors and even damages, which increases the associated costs. Finally, it was possible to observe how the method proposed by Griesmayer *et al.* [11] works, when applied to a sequential program.

```
griesmayer::controller::1::diag=-934770697 (-934770697)
griesmayer::controller::1::diag=-1 (-1)
griesmayer::controller::1::diag=-1 (-1)
griesmayer::controller::1::diag=-1 (-1)
```

Figure 6: Faulty lines obtained by the program execution 5.

3. A Method for Concurrent Programs

In this section, we fully describe our method to localize faults in concurrent C programs. First, we show a motivating example, in order to describe

our approach, and discuss pre-fault localization steps using BMC. Then, the employed transformation rules are tackled, with the goal of sequentializing a concurrent program. Finally, we reason about the fault localization process, in the transformed concurrent program.

3.1. Illustrative Example

We use a simple concurrent program, which is shown Fig. 7, in order to illustrate our approach [19]. It has two shared variables, *i.e.*, mutexes `mutex` and `lock`, which are used to synchronize threads `A` and `B`. The `main` function initializes each thread's integer counter and spawns two threads, executing `threadA` and `threadB` functions, respectively. Each thread function acquires `mutex`, increments its counter (`A_count` or `B_count`), checks if it is possible to acquire `lock` (if its counter equals one), releases `mutex`, tries to acquire `mutex` right after, decrements its counter, checks if it is possible to release `lock` (if its counter equals zero), and finally releases `mutex`. Since we do not have any assertions, we should not have any other types of violation, but concurrent errors. Nonetheless, one may notice that we are using local variables to control access between two different threads to the mutex `lock`. Assuming that context switches might occur at any line of the program, it is likely that one specific execution leads to a deadlock error. For instance, the following execution leads to a deadlock: thread `A` executes until line 5, acquiring mutex `lock`, thread `B` then starts executing, but when it tries to acquire mutex `lock`, a classic deadlock condition is presented, since mutexes `lock` and `mutex` cannot be released by their respective threads, so that the other one can proceed execution.

The translated code is shown in Fig. 8. Here, we added non-deterministic assignments to variables, in order to find not only lines that are related to faults in the program, but also what assignments can correct them. We also added a sequential framework that simulates the faulty behavior of the original program, according to the counterexample obtained by the model checker. When this new instrumented program is run in a model checker, it produces a counterexample that contains which lines cause the faulty behavior and what assignments can be done to produce a successful run. Therefore, our methodology ultimately enables developers to pinpoint what faults are presented in a given program and how they can be fixed.

```
void *threadA(void *arg) {
    pthread_mutex_lock(&mutex);
    ++A_count;
    if (A_count == 1) pthread_mutex_lock(&lock);
    pthread_mutex_unlock(&mutex);
    pthread_mutex_lock(&mutex);
    --A_count;
    if (A_count == 0) pthread_mutex_unlock(&lock);
    pthread_mutex_unlock(&mutex);
}
void *threadB(void *arg) {
    pthread_mutex_lock(&mutex);
    ++B_count;
    if (B_count == 1) pthread_mutex_lock(&lock);
    pthread_mutex_unlock(&mutex);
    pthread_mutex_lock(&mutex);
    --B_count;
    if (B_count == 0) pthread_mutex_unlock(&lock);
    pthread_mutex_unlock(&mutex);
}
int main() {
    pthread_t A, B;
    A_count = 0; B_count = 0;
    pthread_create(&A, NULL, threadA, NULL);
    pthread_create(&B, NULL, threadB, NULL);
    pthread_join(A, NULL);
    pthread_join(B, NULL);
    return EXIT_SUCCESS;
}
```

Figure 7: Illustrative example.

3.2. Method Overview

Here, we briefly describe our present method, as shown in Fig. 9, and a more detailed explanation is addressed in the following sections. Given a concurrent program P, we first check whether it presents a failing execution w.r.t. a specific interleaving. In order to accomplish that, we run P in a model checker twice: the first run checks deadlocks and the second one is in charge of verifying other types of violations, such as lock acquisition errors, division by zero, pointer safety, arithmetic overflow, and out-of-bounds arrays. We cannot verify it only once because model checkers usually separate this verification, *i.e.*, one needs to add a command line option to enable deadlock check and ignore violations due to assertions. If we obtain a counterexample from this step, we can proceed with our method. Then, the next step defines our transformation rules, which are sequential statements that will replace the original concurrent ones, and a sequential framework, which aims to simulate the concurrent execution of the failed interleaving. The third step

```
... int non_det(), diag;
int A_count, B_count;
h_mutex mutex, lock;
void A_1(void *arg) {
    int t; cs cs;
    lock(&mutex, &cs, 1, 9);
    t = A_count;
    A_count = (diag == 10 ? non_det() : t + 1);
    if ((diag == 11 ? non_det() : A_count) == 1) lock(&lock, &cs, 1, 11);
    unlock(&mutex, &cs, 1, 12);
}
void A_2(void *arg) {
    int t; cs cs;
    lock(&mutex, &cs, 1, 13);
    t = A_count;
    A_count = (diag == 14 ? non_det() : t - 1);
    if ((diag == 15 ? non_det() : A_count) == 0) unlock(&lock, &cs, 1, 15);
    unlock(&mutex, &cs, 1, 16);
}
void B_1(void *arg) {
    int t; cs cs;
    lock(&mutex, &cs, 2, 20);
    t = B_count;
    B_count = (diag == 21 ? non_det() : t + 1);
    if ((diag == 22 ? non_det() : B_count) == 1) lock(&lock, &cs, 2, 22);
}
void B_2(void *arg) { ... } ...
```

(a) Translated code part 1

```
... #define NCS 4
int cs[] = {11, 21, 31, 22};
int main() {
    int i;
    diag = non_det();
    for (i = 0; i < NCS; i++) {
        switch (cs[i]) {
            case 1: {
                case 11: {
                    A_count = 0; B_count = 0; if (cs[i] == 11) break;
                }
            } break;
            case 2: {
                case 21: {
                    A_1(NULL); if (cs[i] == 21) break;
                }
                case 22: {
                    A_2(NULL); if (cs[i] == 22) break;
                }
            } break;
            case 3: {
                case 31: {
                    B_1(NULL); if (cs[i] == 31) break;
                }
            } break;
        }
    }
    assert(0);
}
```


(b) Translated code part 2

Figure 8: The porposed method applied to the example given in Fig. 7.

consists in using Griesmayer's method [11] to instrument assignments and expressions, in order to pinpoint faulty program locations and statements. Such an instrumented program can then be run, using a model checker, and it is now possible to collect faulty lines, until the associated verification does not produce different elements. In Fig. 9, this iteration is described in the relation between steps 3, 4, and the check for new counterexamples, in order to decide whether a new faulty line was found. Finally, we have faulty lines and the assignments needed to produce a successful execution of P.

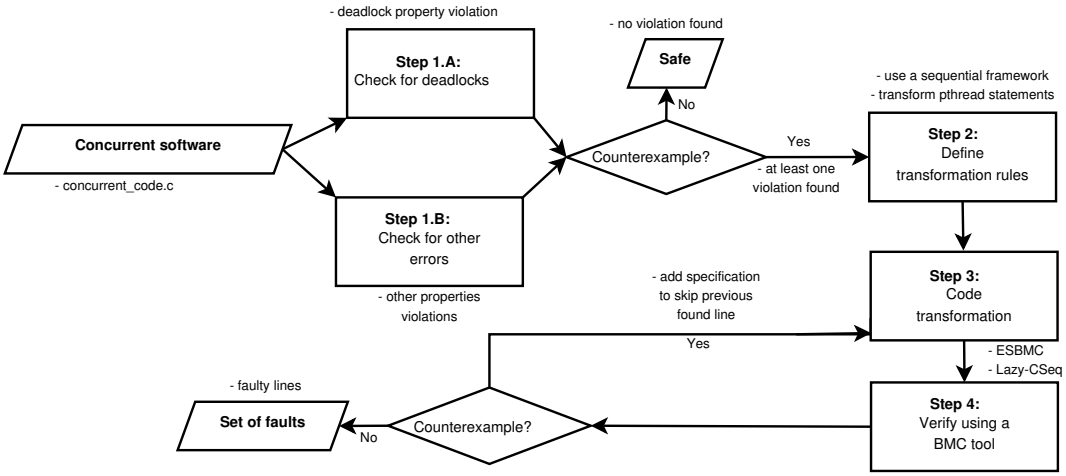

Figure 9: The proposed method for fault localization, in concurrent software.

3.3. Using BMC to Assist in Fault Localization

As one can notice, we use BMC techniques to assist our proposed method in localizing faults. First, we must show that the concurrent program under verification presents a faulty execution under some specific interleaving, *i.e.*, when it is run in a model checker, it produces a counterexample. Second, it is necessary to extract context switch information from the obtained counterexample, in order to model a sequential program with the same aspects of the original one.

3.3.1. Asserting Unsafety of Concurrent Programs

In order to apply our methodology, we must assert the unsafety of the concurrent program P under verification. That can be accomplished with, at least, one violated property (deadlock, assertion, lock acquisition error,

division by zero, pointer safety, arithmetic overflow, and/or out-of-bounds array) and its counterexample (see Section 2.2 for more details), which can be obtained by verifying P in a BMC tool twice. If a counterexample for P cannot be found, we either increase the bound k (limited by computer resources) or claim that P is safe up to the bound k (P does not have any faults), *i.e.*, we check whether loops have been fully unrolled and whether the safety property holds in all states reachable within k iterations.

3.3.2. Extracting Context Switch Information from Counterexamples

With a counterexample C_{ex} for P, we must extract context switch information, in order to further use it in finding a sequential program P_{seq}, which reproduces the same faulty behavior of P.

Such information can be obtained by assuming that C_{ex} is composed by a set of states $s_0, s_1, ..., s_k$, where each one (s_i) contains a line l_{s_i} and a thread T_{s_i}, to which such state belongs. Annotating the tuple (T_{s_i}, l_{s_i}), where $T_{s_i} \neq T_{s_{i+1}}$, gives the context switches occurred in P, w.r.t. a given faulty behavior. For instance, regarding Fig. 7, the context switch information obtained from its counterexample is $CS = [(0, 27), (1, 5), (2, 14), (1, 6)]$, where thread 0 represents the `main` function, thread 1 represents thread A, and thread 2 represents thread B.

3.4. Sequentializing Concurrent Programs

In order to localize faults in a concurrent program, we first sequentialize it, which aims to obtain lines related to a fault, by simulating the faulty execution and the assignments needed to repair it. Indeed, that can be achieved by modeling Pthread synchronization primitives, through a framework to simulate a concurrent execution (w.r.t. to the failed interleaving), and finally applying a sequential fault localization method.

3.4.1. Modeling Pthread Synchronization Primitives

For the sake of sequentialization, we must model originally concurrent statements, in order to obtain a sequential version of them and still keep their functionality. First, a `struct` is used to represent a context switch and `thread_ID` and `program_line` are stored, which are identifiers to the thread and the program line, respectively, where the context switch occurred. This `struct` is named `cs` and is defined as in Fig. 10.

Then, another `struct` is implemented, in order to represent a `pthread_mutex_t`. We store `status`, which is an identifier to state whether such mutex is acquired or not, and `last_cs`, which identifies the thread and the program

```
typedef struct context_switch {
    int thread_ID;
    int program_line;
} cs;
```

Figure 10: Struct to represent a context switch.

where an acquisition or release of such mutex occurred. This `struct` is named `mutex` and is defined as in Fig. 11.

```
typedef struct mutex {
    int status;
    cs last_cs;
} mutex;
```

Figure 11: Implementation for `pthread_mutex_t`.

It is also necessary to provide implementations for mutex manipulation, which are the acquisition of a mutex, *i.e.*, `pthread_mutex_lock`, and the release of a mutex, *i.e.*, `pthread_mutex_unlock`. The acquisition function has 4 arguments, which are `m` (the mutex a thread is trying to acquire), `cs` (a context switch variable), `id` (an identifier to the thread which called the acquisition function), and `line` (an identifier to the program line where this function was called from). The release function also has 4 arguments, which are `m` (the mutex in which a thread is trying to release) `cs` (a context switch variable), `id` (an identifier to the thread which called the release function), and `line` (an identifier to the program line where this function was called from). The acquisition function is named `lock` and is defined as in Fig. 14, while the release function is named `unlock` and is defined as in Fig. 15.

Conditionals are implemented in *pthread*, as shown in Fig. 12; however, in order to simulate the same behavior, we do not need to stop an execution. Since we are simulating only a faulty behavior, the necessary condition is already satisfied. So, we basically represent a `pthread_cond_t` variable as an integer one and `pthread_cond_signal` and `pthread_cond_signal` assign 0 and 1 to it, respectively. In Fig. 12, the usual implementation of conditionals is described and Fig. 13 presents how they are modeled in the presented methodology. Regarding other `pthread` statements, such as `pthread_create`, `pthread_exit`, and `pthread_join`, they are further discussed in section 3.4.2.

```
pthread_cond_t c;
...
    pthread_cond_signal(&c, &m);
...
while (!condition) {
    pthread_cond_wait(&c, &m);
}
```

Figure 12: Standard usage of conditionals.

```
int c;
...
    c = 0;
...
while (!condition) {
    c = 1;
    break;
}
```

Figure 13: Implementation for conditionals.

Table 1 summarizes all rules employed to transform concurrent programs into sequential ones. Column *"#"* represents the two groups of program statements, *i.e.*, (1) regular and (2) concurrent, column *"Code fragment"* shows the code to be transformed, and columns *"No deadlock"* and *"Deadlock"* tell what transformation rule must be performed, in case of a program that presents or not a deadlock, respectively.

3.4.2. Adding a Framework

A framework provides the same execution sequence as in the original program. It consists basically in writing each thread code inside a `case` statement, whose execution sequence is specified in `cs`. Such a framework is used as the basic structure for new sequential versions, in order to simulate the faulty behavior of concurrent programs, and Fig. 16 shows how it is encoded.

As one can notice, the mentioned framework provides new fixed positions, for each part of the original code, and Table 2 shows the relation between new positions and code-fragment types, *i.e.*, it summarizes how the new sequential code is structured. In particular, global elements, global variables, header file declarations, and other types of global declarations are placed before the sequential code's `main` function. The body of `main`, in the original code, is placed between the statement `case 1` and its respective `break` command, the

```
void lock(mutex *m,
          cs *cs,
          int id,
          int line) {
    int status = m->status;
    if (status == 1) {
        __VERIFIER_error();
    } else {
        cs->thread_ID = id;
        cs->program_line = line;
        m->status = 1;
        m->last_cs.thread_ID = cs->thread_ID;
        m->last_cs.program_line = cs->program_line;
    }
}
```

Figure 14: Implementation for `pthread_mutex_lock`.

```
void unlock(mutex *m,
            cs *cs,
            int id,
            int line) {
    if (m->status == 1 &&
        m->last_cs.thread_ID == id) {
        cs->thread_ID = id;
        cs->program_line = line;
        m->status = 0;
        m->last_cs.thread_ID = cs->thread_ID;
        m->last_cs.program_line = cs->program_line;
    } else {
        __VERIFIER_error();
    }
}
```

Figure 15: Implementation for `pthread_mutex_unlock`.

body of the first thread is placed between `case 2` and its respective `break` command, and so on. This process is repeated until there are no more threads to be inserted into the sequential code version. Additionally, arguments passed to the original program's `main` function are all passed to the sequential version's `main`. In cases where threads are partially executed and a context switch occurs, *i.e.*, another thread is executed or a previous thread continues to execute from where it stopped, the respective code pieces are inserted into each `case` inside the outer N^{th} *case* (which represents the Nth thread), in such a way that the execution order remains the same. The counterexample obtained from the model checker is fundamental to the framework, since it shows not only the context-switches inside a program, but also the number

Table 1: Rules to transform concurrent programs

#	Code fragment	No deadlock	Deadlock
1	Variable declaration	No changes	No changes
	Expression	Unwind	Unwind
	Statement	No changes	No changes
2	`pthread_t`	ϵ	ϵ
	`pthread_attr_t`	ϵ	ϵ
	`pthread_cond_attr_t`	ϵ	ϵ
	`pthread_create`	ϵ	ϵ
	`pthread_join`	ϵ	ϵ
	`pthread_exit`	ϵ	ϵ
	`pthread_mutex_t`	ϵ	Mutex variable is declared
	`pthread_mutex_lock`	ϵ	Lock is called using variable
	`pthread_mutex_unlock`	ϵ	Unlock is called using variable
	`pthread_cond_t`	ϵ	Conditional variable is declared
	`pthread_cond_wait`	ϵ	Wait is called using variable
	`pthread_cond_signal`	ϵ	Signal is called using variable

of threads spawned during a faulty execution.

Table 2: Relation between positions and codes

Code Fragment Type in the Original Code	Position in the New Sequential Code
global elements	before line 1
main function body	between "case 1" and "break"
thread body n	between "case $n+1$" and "break"

In order to maintain the same execution order found in the original program, order-control switching is required. A fixed context-order switching, from a counterexample generated for a concurrent program, can be copied to a new sequential one by controlling `case` and conditional statements[2], in the framework `switch` statement, and, in general, adding context-switching order control to a new sequential program can be divided into two steps. In order to show a simple example, it is assumed that there are less than 10 context switches, in each thread ($\forall N_{ti}, N_{ti} < 10$), a counterexample, given by a BMC

[2] `if (cs[i]) == Y) break;`, where `Y` represents the number of the context switch

```
#define NCS X
int cs[] = {...};
int main(int argc, char *argv[]) {
    int i;
    for(i = 0; i < NCS; i++) {
        switch(cs[i]) {
            case 1:
            case 11: { ... }
            ...
            case 20: { ... }
            break;
            case 2:
            case 21: { ... }
            ...
            case 30: { ... }
            break;
            case 3:
            case 31: { ... }
            ...
            case 40: { ... }
            break;
            ...
            default:
            break;
        }
    }
    return 1;
}
```

Figure 16: The standard framework to sequentialize a concurrent program.

tool, N context switches, and, from those, N_{t0} occur in the main function, N_{t1} occur in thread 1, N_{t2} in thread 2, and so on ($N_{t0} + ... + N_{tn} = N$).

The first step consists in acquiring information from counterexamples generated by a BMC tool, *i.e.*, the total number of context switches in the original program and in each thread, the order of all context switches in the entire program and also in a single thread, and the corresponding position where a context switch occurred. With such data, it is possible to add the previously referred conditional statements for maintaining the same execution order of the original program, so that when a line is executed, the sequential code executes the next `case` statement, which represents the next thread in a original source code excerpt.

One can notice that if there are iteration statements in the original concurrent program, a global variable named `loopcounter` is added for each one. Besides, a statement to increment the value of `loopcounter` is also added to the end of each loop body. This newly added global variable is used

as a condition to directly control the validity of `break` statements, so that when a context switch occurs, inside a loop, the value held by `loopcounter` must also be used in the respective statement `break`, in order to maintain the original program execution sequence.

The second step consists in modifying values related to `cs`, in such a way that the execution order is kept, in a new sequential program. By changing lines 1 and 2, in Fig. 16, according to a specific number of context switches and their execution order, it is possible to guarantee the original execution order, given that a `switch` statement (line 6) selects which piece of code (representing threads from the original program) is executed, based on `cs[i]`. It is worth noticing that the process of obtaining context-switches from a counterexample is fully automatic, but the sequential code generation is semi-automatic.

For instance, in Fig. 7, the execution order is thread 0, thread 1, thread 2, and finally thread 1 (this information was obtained by analyzing the counterexample generated by a BMC tool, as described in section 3.3.2). The `cs` array will hold 11, 21, 31, and 22, meaning that the first `case` will be executed, then the first inner `case` inside the second one, the third, and, finally, the second inner `case` inside the second one.

3.4.3. Applying a Sequential Fault Localization Method

Finally, the method described by Griesmayer [11] is applied. We simply convert every assignment in P to a non-deterministic version of it, whose value is chosen by the BMC tool and is then related to a diagnostic variable, *i.e.*, `diag`. This way, if a counterexample is provided for P_{seq}, there are `diag` variables in this trace, which can compose the set of faulty lines in P. Correcting those lines then lead to a successful execution of P.

For instance, in Fig. 7, our methodology identifies lines that increment counters `A` and `B` and also provides assignments to them, in order to repair the program fault. In particular, changing the assignments for one of the counters enables the other thread to acquire and release one of the shared mutex freely, which then fixes the deadlock. It is important to stress out that this is only a possible fix, since this change leads to a successful program execution. Other fixes are also possible, such as a rearrangement of mutex acquisition and release orders or the addition of join statements between thread calls; however, here we do not explore them, since they are not possible to be achieved by only changing assignments to variables, in the original program.

4. Experimental Evaluation

Here, the conducted experiments are discussed. We start by briefly describing the experimental objectives and later discuss the experimental setup and the employed benchmarks. Then, we move to the experimental results and lead a structured discussion, by describing advantages and disadvantages of the proposed methodology.

4.1. Experimental Objectives

Using the proposed benchmarks, our experimental evaluation aims to:

1. Show that our methodology can be applied to reason about concurrent C programs, regarding fault localization;
2. Evaluate the elapsed time, when executing the model checker, in order to verify the instrumented code generated by our methodology.

4.2. Experimental Setup

In order to verify and validate the proposed method, we used ESBMC v3.0.0, allied to the SMT solver Boolector [34], and Lazy-CSeq v1.1, with CBMC [24] v5.3 (backend). All experiments were conducted on an otherwise idle Intel Core i7 – 4500 1.8Ghz processor, with 8 GB of RAM and running Fedora 24 (64-bits operating system).

The benchmarks in Table 3 include the same programs used for evaluating ESBMC, regarding concurrent C programs [19], and other that were extracted from the concurrency suite of SV-COMP 2016, which are available on the ESBMC website[3]. *account_bad.c* is a program that represents basic operations in bank accounts: deposit, withdraw, and current balance, with a mutex to control them. *arithmetic_prog_bad.c* is a basic producer and consumer program, using mutex and conditional variables for synchronizing operations. *carter_bad.c* is a program extracted from a database application, which uses mutex to synchronize threads. *circular_buffer_bad.c* simulates a buffer, using shared variables to synchronize receive and send operations. *queue_bad.c* is a program that simulates a data-queue structure. *sync*01*_bad.c* and *sync*02*_bad.c* are producer and consumer programs: the former never consumes data and the latter initializes a shared variable with some (arbitrary) data. *token_ring_bad.c* propagates values through

[3]http://esbmc.org/benchmarks/ejss2016.zip

shared variables and checks whether they are equivalent, through different threads. *twostage_bad.c* simulates a great number of threads running simultaneously, and *wronglock_bad.c* simulates a large number of producer threads and the propagation of their respective values to other threads. $race-1\_1-join\_true-unreach-call.c$ is a program that tests race conditions in a shared variable, using a mutex to synchronize access to such. $bigshot\_p\_false-unreach-call.c$ is a program that allocates and copies data to a string pointer, in two different threads, and ultimately asserts if the pointer was operated as expected. $fib\_bench\_false-unreach-call.c$ and $fib\_bench\_longer\_false-unreach-call.c$ are programs that loop a predefined number of times and increment two shared variables, without a mutex for synchronizing access, and then assert whether one of them reached some specific value or not. $lazy01\_false-unreach-call.c$ uses a mutex to control summation operations over a shared variable and then checks its value. $stateful01\_false-unreach-call.c$ uses two different shared variables guarded by two different mutexes, performs arithmetic operations in them, and checks their final value. $qw2004\_false-unreach-call.c$ tests non-guarded arithmetic operations over shared variables. Finally, *half_sync.c* and *no_sync.c* share a counter to increase its value and check a property without fully synchronizing threads.

The experimental evaluation procedure can be split into three different steps. First, it is necessary to obtain a counterexample for a given program. If the result given by ESBMC/Lazy-CSeq is *verification failed*, then the benchmark is unsafe, *i.e.*, it presents a deadlock, assertion, lock acquisition error, division by zero, pointer safety, arithmetic overflow, and/or out-of-bounds array violation, and we can translate this faulty behavior into a sequential program that simulates such an execution. In the second step, it is necessary to extract context-switch information, *i.e.*, number of context-switches (statements where context-switches occurred) and the number of running threads, through the method presented in section 3, which is achieved by removing the `--deadlock-check` option in the issued ESBMC command line[4] (in case of Lazy-CSeq, removing the `--deadlock` option from the command line[5] will produce the expected result), and parsing the generated counterex-

[4] `esbmc --no-bounds-check --no-pointer-check --no-div-by-zero-check --no-slice --deadlock-check --boolector <file>`

[5] `cseq --deadlock --cex --rounds 5 -i <file>`

ample. In the third step, the original program is transformed into a sequential one, with the information obtained from steps 1 and 2, by applying the rules in section 3.4.2 and the method proposed by Griesmayer *et al.* [11]. Finally, the sequential version of a program can then be verified in ESBMC, by using a command line[4] without the `--deadlock-check` option, changing the specified file, and applying the same strategy demonstrated in section 2.5.

4.3. Experimental Results

Table 3 summarizes the experimental results. **F** describes the name of the benchmark, **L** represents the number of code lines, **T** is the number of threads, **D** identifies whether a deadlock occurred (if its value is `1`), **FE** is the amount of errors found during the fault-localization process, that is, the number of different `diag` values retrieved by ESBMC/Lazy-CSeq, **AE** is the number of actual errors, **R** stands for the actual result (`1` if the information retrieved by ESBMC/Lazy-CSeq is helpful), and, finally, **VT** is the time that ESBMC/Lazy-CSeq took to verify the benchmark.

The verification of *account_bad.c* presented 3 different `diag` values, which are in different code parts; however, they ultimately identified the actual fault in the original program, which was a bad assertion.

The 7 diagnosed values regarding *circular_buffer_bad.c* led to a bad assertion in the program, which is related to a loop. This way, the `diag` values indicate this loop.

When checking *arithmetic_prog_bad.c*, the proposed methodology informed 2 different `diag` values, which address a loop in thread 2 of this program, meaning that the fault is in that specific loop.

The analysis of *queue_bad.c* presented 4 errors. The identified faults are related to flags providing access control to a shared variable and a loop where they are changed, that is, the problem lies again on bad handling.

sync02_bad.c presented 2 different values, related to a consumer thread in the original program, whose lines are related to a deadlock present in this benchmark.

carter_bad.c, *token_ring_bad.c*, *twostage_bad.c*, and *wronglock_bad.c* presented 1 faulty line when diagnosing each sequentialized program. In the first, the proposed methodology was able to find an assignment to a shared variable that fixes the deadlock present in this program. In the second, the diagnosed value leads to a bad formulated boolean expression in the assertion. The third was diagnosed with a faulty line that points to a bad formulated

Table 3: Experiment results

F	L	T	D	FE/AE	VT	R
account_bad.c	49	2	0	3/3	0.102	1
arithmetic_prog_bad.c	82	2	1	2/2	0.130	1
carter_bad.c	43	4	1	1/1	0.289	1
circular_buffer_bad.c	109	2	0	7/7	0.227	1
queue_bad.c	153	2	0	4/4	0.934	1
sync01_bad.c	64	2	1	1/0	0.451	0
sync02_bad.c	39	2	1	2/2	0.116	1
token_ring_bad.c	56	4	0	1/1	0.307	1
twostage_bad.c	128	9	0	1/1	0.284	1
wronglock_bad.c	111	7	0	1/1	0.310	1
race-1_1-join_true-unreach-call.c	58	1	0	1/1	0.399	1
bigshot_p_false-unreach-call.c	35	2	0	2/2	10.724	1
fib_bench_false-unreach-call.c	44	2	0	2/2	8.966	1
fib_bench_longer_false-unreach-call.c	44	2	0	2/2	11.147	1
lazy01_false-unreach-call.c	51	3	0	1/1	0.259	1
stateful01_false-unreach-call.c	56	2	0	2/2	0.264	1
qw2004_false-unreach-call.c	60	1	0	1/1	0.250	1
half_sync.c	22	2	0	1/0	0.322	0
no_sync.c	21	2	0	1/0	0.320	0

assertion and, finally, the forth referred benchmark contains a bad formulated assertion, which the methodology was able to pinpoint.

The methodology was able to inform 1 different faulty line for *race-1_1-join_true-unreach-call.c*, which led to a bad formulated assertion in the program, and changing the value in the comparison would produce a successful execution of the program.

Benchmarks *bigshot_p_false-unreach-call.c*, *fib_bench_false-unreach-call.c*, and *fib_bench_longer_false-unreach-call.c* were diagnosed with 2 errors. In the first one, the obtained faulty lines refer to an assertion, meaning that a boolean expression is not true under a specified interleaving. The other two benchmarks are essentially the same, differing only in a value evaluated in an assertion. The associated faulty lines are also related to such, since the values used in the mentioned comparisons might be different from the expected ones.

lazy01_false-unreach-call.c and *qw2004_false-unreach-call.c* presented 1 error each. In both, those errors are related to assertions, meaning that the associated expressions should be evaluated with different values.

stateful01_false-unreach-call.c was diagnosed with 2 faults that lead to expressions in assertions. Thus, in order to produce a successful execution of the program, we must change the adopted values.

Although *sync01_bad.c* presented no errors, it was diagnosed with one fault. Indeed, ESBMC found a `diag` with value 0, which is particularly odd, since there is no line 0. Besides, even after adding an assertion, ESBMC still diagnoses 0. Indeed, it has synchronization problems, but the proposed method was unable to provide useful information. Furthermore, this benchmarks presents a deadlock related to its conditionals, which our methodology does not currently model.

The main cases where our methodology failed were *half_sync.c* and *no_sync.c*. Even though ESBMC provided reasonable `diag` values, they do not lead to the actual faults, since applying the proposed repair does not fix the input program and such a procedure instead causes different error paths. The faults in these programs are related to thread synchronization, *i.e.*, the property is validated by the model checker before other threads finish execution. Moreover, the lack of proper `pthread_join` calls leads to these synchronization faults, which are currently not supported by the proposed approach.

According to the results shown in Table 3, one can note that the proposed methodology was able to find faults (useful information) in 16 out of 19 benchmarks, which amounts to 84.21%. Note that benchmarks whose verification failed and, consequently, from which no counterexample was able to be obtained are also included into this evaluation. The methodology itself showed to be useful in diagnosing data race violations, since most of the used benchmarks presented a fault related to that problem; however, the proposed method needs to be improved, in order to verify deadlocks in a more efficient way, loop transformations also need a significant work, so that threads interleaving inside loops can be better represented, and also a proper model of joining threads still needs to be proposed, so that synchronization problems are fully covered.

With respect to the verification time of instrumented sequential programs, in the majority of benchmarks it is less than 2 seconds. That occurs because our methodology provides a pruned and sequentialized version of the concurrent program, and also with few sources of non-determinism, *i.e.*, basically the `diag` values, which leads to a small overhead in the model checker. In 3

benchmarks, however, we obtained significantly higher verification times, 5 to 6 times more than usual. The execution of the model checker was analyzed and, in those cases, the SMT solver took longer than expected to reason about the SMT formula for the program, possibly due to the non-determinism in `diag`.

Regarding benchmarks in which no useful information was obtained, that leads to the conclusion that improved grammar and rules are needed, in order to localize faults. Apart from that, the experimental results showed the feasibility of the proposed methodology for localizing violations, in concurrent C programs, since ESBMC and Lazy-CSeq are able to provide helpful diagnosis information regarding potential faults.

4.4. Comparison With Other Fault Localization Techniques

Spectrum-based fault localization (SBFL) approaches, such as the one proposed by Jones *et al.* [35], try to identify suspicious code blocks, *i.e.*, the ones that are executed when a failed test case is run and lead to program faults. On the one hand, SBFL techniques do not perform complex semantic analysis on a program, since suspiciousness scores are calculated using only information provided from the test suite execution. On the other hand, the accuracy of SBFL approaches is often below desired, when addressing large real-world programs [36].

Mutant-based fault localization (MBFL) approaches, such as the one that Hong *et al.* [36] presented, also try to find suspicious code blocks, but they generate mutants for such blocks and re-execute test cases: if a mutant code block is killed, then the related code block has a higher suspiciousness score. This mutation schema usually outperforms SBFL approaches [36], but it also takes longer to provide useful results, since there are a variety of possible mutants for each code expression.

Given the current knowledge in fault localization, there is no other fault localization technique that addresses the problem using model checking counterexamples and works on concurrent programs. Therefore, we were unable to compare our method directly with the SBFL and MBFL approaches since we would need to re-implement them to handle C programs based on the Pthread-library, which could introduce errors and inconclusive results.

4.5. Scalability

Our methodology relies entirely on model checking techniques and a failing program execution, *i.e.*, a counterexample provided by a model checker,

in order to localize faults in concurrent programs. Thus, we state that our methodology is scalable as long as the model checker used to apply it is scalable and can successfully provide a counterexample for the program under verification.

Currently, model checkers are scalable to handle large concurrent programs [37]. Primarily, model checkers tried to model concurrent programs in their natural states, by modeling concurrent execution and statements, but this approach usually faced the space-state explosion problem. Then, researches came up with different approaches, in order to scale model checking w.r.t. concurrent programs. As one example, Lazy-CSeq [17] is able to find violations in extensive test suites, such as the SV-Comp concurrency suite, containing errors of different natures, and it has also been rewarded as one of the most efficient model checkers to address concurrency, in SV-Comp editions.

4.6. Applying the Proposed Approach in Practice

In order to evaluate how our methodology improves the overall debugging time, we have conducted a simple experiment with 6 developers, with different experience levels. We asked them to debug *account_bad.c* the way they would normally perform such task. Then, we explained our approach steps, asked them to debug the same program using our methodology, and annotated the elapsed time for each one. It is worth noticing that the volunteers did not have any prior knowledge about our approach and each one was independently contacted and evaluated.

A summary of the obtained results is available in Table 4, where **Job description** represents the level of experience of each developer, **Non-assisted debugging time** represents the time, in minutes, that each developer took to find an error in the input program, and, finally, **Assisted debugging time** represents the time, in minutes, that each developer took to find an error, using our methodology.

Although the group of volunteers was not large and no previous familiarity with the proposed error was evaluated, as well as their practical experience in debugging, it seems that there is a trend towards a reduction in the associated elapsed time needed for identifying the existing error, when verifying the chosen program, along with a smaller difference regarding the experience level of each developer. The main cause is that intuition and experience are replaced by our methodology, which consists of providing a failing trace for the given program and then applying the sequentialization framework, in

Table 4: Measuring the benefits of our methodology in practice.

Job description	**Non-assisted debbuging time**	**Assisted debugging time**
Senior Developer A	13.20	2.50
Senior Developer B	14.00	2.75
Developer A	19.50	3.20
Developer B	20.00	3.00
Junior Developer A	38.30	5.10
Junior Developer B	41.20	4.25

order to detect which lines need repair. As a consequence, the fault localization task became a systematic process composed by the following steps: obtainment of a counterexample, creation of an equivalent non-deterministic instrumented sequential code, and extraction of faulty lines from the new counterexample.

4.7. Final Considerations

The employed benchmarks basically present one type of error: deadlock, assertion, lock acquisition, division by zero, pointer safety, arithmetic overflow, or out-of-bounds array. Therefore, we applied different rules, depending on the detected fault. Fig. 17 shows a summary of all obtained results, which show that the proposed methodology gives useful information about failing traces, w.r.t. the error present in the benchmark, in 84.21% of the adopted benchmarks, besides identifying what lines are related to the associated fault and what values can be assigned to produce a successful execution. Regarding all the adopted benchmarks, our methodology was not able to diagnose fault lines in three of them, which was possibly due to our modeling of the C concurrent library, since those benchmarks presented deadlocks and synchronization problems.

Fig. 18 shows a summary of the verification time needed by each benchmark. Thirteen programs took less than 2 seconds, in order to reason about the existing faults and the associated assignments for producing a successful execution, three programs were not correctly verified, with associated verification times lower than 2 seconds, and, finally, three benchmarks took about 4 to 5 times longer to produce a useful result. Regarding the latter, by analyzing the trace provided by the model checker, we found that the bottleneck was the SMT solver, possibly due to the amount of sources

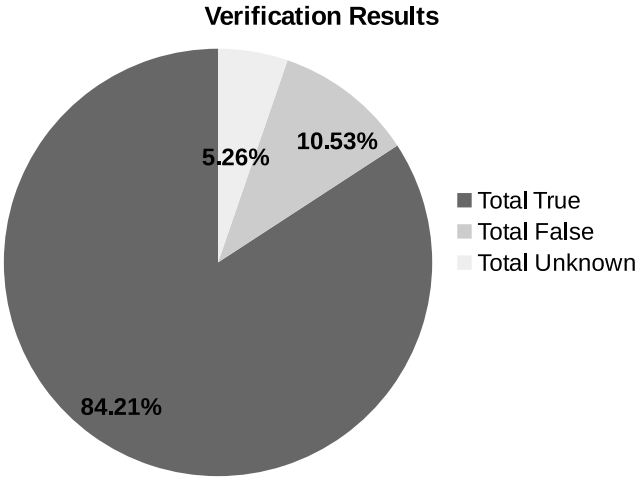

Figure 17: Summary of verification results

of non-determinism present in those benchmarks. Nonetheless, the overall verification time is still short, if compared with what can be obtained with non-assisted program debugging [3].

In summary, the presented results show that the proposed methodology is suitable for localizing faults, in concurrent C programs, and that the related verification time necessary to produce a successful program execution is short. Therefore, our methodology can be useful in assisting developers when debugging programs.

5. Related Work

Tomasco *et al.* [38] reported an approach that uses a technique called memory unwinding (MU), which means that operations are written to a shared memory, in order to symbolically verify concurrent programs that use shared memory and dynamic thread creation. First, a sequence of writes, *i.e.*, a possible MU, is arbitrarily defined and then all program executions compatible with this definition are simulated. In each attempt, the main idea is to sequentialize a concurrent program, w.r.t. MU rules, and then use model checking in the new code, bounded by the total number of write operations to the shared memory and using an existing sequential verification tool. If an error is not found in the defined MU, a new one is generated and the simulation process is performed again, until an error is found or all possible

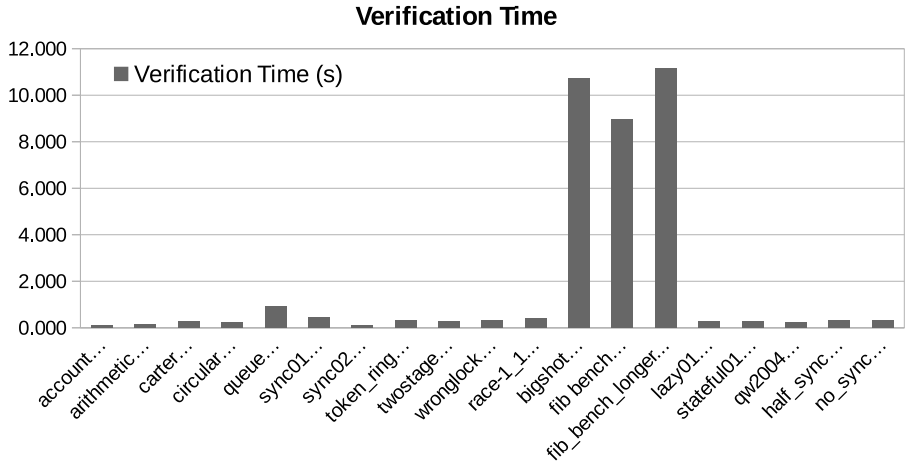

Figure 18: Summary of the results of all benchmarks

interleavings are explored. The approach described by the authors is effective to verify concurrent programs, using a sequentialization algorithm that operates in a greedy approach [39] and uses the notion of memory unwinding. The modeling of Pthread library [16] primitives is fully done; however, memory dynamic allocation is still limited. Even though the results of that paper show that the implemented algorithm in the MU-CSeq tool was able to find all defects in the concurrency suite of the International Competition on Software Verification 2015, it can only assure whether an error exists or not, but it is not able to easily pinpoint lines that need repairs.

Cordeiro *et al.* [19] describe three approaches (lazy, schedule recording, and underapproximation and widening) to verify concurrent programs, using the ESBMC [15] model checker, which is based on SMT. The first approach generates all possible interleavings and calls the SMT solver, in each one of them, the second encodes all possible interleavings, in a single formula, and explores the solver's speed, and the third one reduces the state space, by abstracting the number of interleavings from the unsatisfiability proofs generated by an SMT solver. By modeling the synchronization primitives of the Pthread library [16], ESBMC creates an instrumented program, w.r.t. the original one, and uses model checking, bounded by the number of context switches in the new program version, which aims to find an error or explore all possible interleavings. According to the experimental results, this

work showed itself useful to handle concurrent programs, by finding not only atomicity and order violation errors, but also global and local deadlocks. Among the three proposed approaches, the lazy approach proved itself to be the most efficient, given that it was able to verify all the proposed programs. Nonetheless, a model checker can state whether an error exists or not and, but it cannot directly pinpoint where such a fault is located.

Jones *et al.* [35] presented an approach to assist in fault localization, using the visualization of test information. The idea is to run a test suite and color program statements related to successful and failed test cases, in such a way that developers are able to inspect program executions, reason about program statements related to faulty behaviors, and, possibly, identify faults. The authors also describe the developed technique, named Tarantula, which uses the visualization technique, and show that their technique is useful to assist in program debugging, by enabling a global perspective instead of only a local one, as a traditional debugger offers. However, as one may notice, such an approach is not completely automatic, since an iterative execution of a program and its related test cases is still required. Faulty lines can be marked as safe, with respect to a test case, and the provided results are not so useful when a program presents multiple faults.

Cleve *et al.* [7] discuss an approach to localize faults in programs, by using search in space (*i.e.*, program states) and in time (*i.e.*, cause transitions, which are moments in time where a variable is no longer the cause of a fault and another one starts to be). The comparison among program states in successful executions and failed ones was essential to find locations where cause transitions occur, since they are not only good repair locations, but also point to defects that cause failures. Besides, it is possible to identify code pieces that result in problems. Such a technique was applied to an open-source debugging tool, AskIgor [40] and, according to the results presented by the mentioned authors, it showed itself to be more useful than other methods introduced at that time. Even though those points are in fact helpful, a test suite with a high code coverage rate is still needed, since that technique relies on those inputs to localize faults and it is still necessary to precisely choose the state search in space and time, so that a fault can be indeed found.

Birch *et al.* [41] presented an algorithm to fast model-based fault localization, which executes a test suite and, with the use of symbolic execution methods, automatically identifies a small subset of program locations where repairs are needed, based on failed test cases. The algorithm uses time limits

to improve its speed, in such a way that if a test case takes longer than excepted, the current execution is postponed and another one takes its place. Such a process improves the overall performance of this technique, because a list of possible locations where repairs are needed is kept and the postponement of the analysis of a given test case can be eventually productive, since more information about a program can be obtained through the analysis of other test cases. The algorithm presented in that paper showed itself to be efficient and optimized for handling extensive unit test suites, since it is able to delay the search for faults based on a given test case, if it takes longer than expected. Regarding the symbolic execution of the original program model, the authors used KLEE [42] and ESBMC [15]. Even though the mentioned approach narrows the state space search, it still depends entirely on a well-formed test suite, that is, if the input suite does not cover the faulty program location, the algorithm will not find such faults.

Griesmayer *et al.* [11] proposed a method to localize faults in ANSI-C programs. Given an application, a specification, and a counterexample, which is used to show that the specification is not satisfied, *i.e.*, that a fault exists, the authors use them to create an extended version of such program. The associated inputs are fixed w.r.t. the values in the counterexample and introduce abnormal predicates for each program component, which generates an instrumented version of the original code. Program variables are non-deterministically modeled, in such a way that they satisfy the original specification of a program. The instrumented program counterexample holds lines that lead to a fault and whose values are necessary, in order to produce a successful program execution. A good aspect of such a method is the fact that the counterexamples generated by the model checker not only point to faulty lines in a program, but also to the assignment of program inputs, in order to correct such fault. A downside of it is that the method only works in standard ANSI-C programs, *i.e.*, procedural/sequential programs and the conversion, unwinding, and program internal representation generation time lead to greater fault localization time.

Park *et al.* [43] present a dynamic fault localization method to localize root causes of concurrency defects and a prototype implementation of that technique, named FALCON. Using dynamic pattern detection and statistic fault localization, FALCON is able to show the existence of defects in concurrent programs, from atomicity to order violation bugs, and assists developers to fix faults in code. This technique uses data provided by test cases applied to the program under verification and tries to find predefined shared memory

access patterns, which are later statistically organized, in such a way that it prioritizes which are the possible existing faults, in a program. According to the empirical study conducted by the authors, that technique seemed to be effective in handling atomicity and order violations, in concurrent programs, and showed itself to be efficient, in terms of used space and time. Nonetheless, that approach was developed to handle only Java programs and depends on test cases to search for faulty patterns, and, in the referred work, the authors only evaluated the tool with one test input and multiple program executions.

Jose *et al.* [44] discuss an algorithm to localize error causes, considering a reduction to the maximum satisfiability (SAT) problem (MAX-SAT), which shows the maximum number of clauses in a boolean formula that an assignment can simultaneously satisfy. The key idea is to combine a boolean trace formula and an unsatisfiable formula, both w.r.t. the program unwinding and a failed program execution, and then use MAX-SAT to find the maximum set of clauses that can be satisfied at the same time, in that formula. The complement of the set returned by the MAX-SAT solver holds program locations that lead to an error; this way, by correcting those locations, it is possible to obtain a non-failing program execution w.r.t. the given test case. The presented algorithm is able to localize faulty lines and the authors also conducted experiments to suggest repairs to arithmetic assignments and change comparison operators, in the input program. Even though this approach is useful to localize faulty lines, it still relies on a failing test case and it only works in procedural/sequential ANSI-C software.

Hong *et al.* [36] proposed a mutation-based fault localization technique for multilingual programs, *i.e.*, programs written in more than one language that provide interfaces for communication, named Museum. The authors have developed a new set of mutation operators that, along with conventional ones, are able to locate faulty statements in all employed benchmarks, which improves the accuracy when fiding multilingual faults. Thus, by using this extended set of mutant operators and appropriate suspiciousness metrics, Museum is able to identify bugs in JNI programs. The empirical evaluation shows that Museum is more precise and accurate than state-of-the-art spectrum-based fault localization techniques. Nevertheless, the proposed technique needs at least one failing test case and it has not been evaluated in concurrent programs yet.

Papadakis *et al.* [45] present a fault localization approach based on mutation analysis, named Metallaxis. The approach consists in using mutants

and associating them with potential faulty program locations, and as mutants are killed (mainly by failing test cases), such are essentially good indications about faulty locations in a program. Moreover, the presented approach takes advantage of a higher number of test cases to rank mutated statements, w.r.t. their suspiciousness score, which leads to program faults. According to experimental data, Metallaxis is substantially more effective than statement-based approaches, even when mutation cost-reduction techniques (*e.g.*, mutation sampling) are used. Metallaxis locates faults effectively and, since it works on a test suite, it can be used while testing to locate potential program faults. The approach shows itself superior to statement-based approaches; however, it still relies on an extensive test suite and it is only evaluated in procedural/sequential ANSI-C software.

The main differences between our approach to the ones discussed in this section are:

- Our method only requires a program's source code, where, in other methods, more information needs to be provided by the user, such as a failing test case, since in our method we are able to extract the needed counterexample from the model checker;
- It works with concurrent C programs, which are widely used in embedded systems, since their faulty behavior is reproduced by the sequentialization scheme;
- It provides a sequentialization method to reproduce the faulty behavior in a original program, which leads to fast diagnosis regarding faults;
- We are able to provide assignments to obtain a successful execution of a program, which assists not only in fault localization, but also in repairs;
- Our methodology extracts additional information, *i.e.*, a counterexample from the program's source code, whereas other approaches require additional data;
- And, we can pinpoint faulty lines, whereas most approaches only verify the program safety.

6. Conclusion

A novel method for localizing faults in concurrent C programs, using code transformation and BMC techniques, was proposed. It is based on the approach introduced by Griesmayer *et al.* [11] and an specific extension to handle concurrent programs, which is useful for embedded systems.

Regarding the experimental results, the proposed methodology was able to identify potential faults in concurrent software, 84.21% of the chosen benchmarks, while failed to obtain useful faulty lines for three of the adopted benchmarks. Indeed, one of them presented a deadlock and the other two failed due to thread synchronization problems, which claims for a deeper investigation, in order to provide improvements regarding modeling approaches for C concurrent synchronization primitives. Nonetheless, our method is useful to reason about concurrent programs that present faults related to bad formulated assertions and deadlocks, which makes it interesting for assisting developers to find assignments that lead to successful program executions and, consequently, minimizes program debugging effort.

When comparing with other approaches, our method is able to localize faults in concurrent programs using only the program's source code, instead of a failing trace and a test suite. It is also able to not only pinpoint faulty lines, but also provide possible repairs to produce a successful program execution.

We provided a well-structured methodology that can be included in other frameworks or tuned to specific applications. Also, even though we evaluated it using only two model checkers, it can be applied to any model checker that handles concurrent programs. Thus, it is flexible to reason about concurrent programs, in different scenarios.

As future work, new rules for code transformation and also an improved grammar will be developed, in order to increase the methodology accuracy. A more sophisticated fault repair mechanism will be developed, in particular regarding fixing deadlocks and synchronization problems, in order to provide even more useful correction suggestions. Additionally, an *Eclipse* plug-in will be developed for automating the fault diagnosis process, during development processes.

References

[1] C. Baraniuk, The Number Glitch That Can Lead to Catastrophe, [Online; posted 5-May-2015] (May 2015).

URL http://goo.gl/qabuJF

[2] G. Myers, T. Badgett, C. Sandler, The Art of Software Testing, 3rd Edition, Wiley, 2011.

[3] W. Mayer, M. Stumptner, Evaluating Models for Model-Based Debugging, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 128–137.

[4] F. Tip, A Survey of Program Slicing Techniques, Journal of Programming Languages (1995) 121–189.

[5] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, An Experimental Determination of Sufficient Mutant Operators, ACM Transactions on Software Engineering and Methodology (2) (1996) 99–118.

[6] H. He, N. Gupta, Automated Debugging using Path-Based Weakest Preconditions, in: Fundamental Approaches to Software Engineering, Springer, LNCS, 2004, pp. 267–280.

[7] H. Cleve, A. Zeller, Locating Causes of Program Failures, in: Proceedings of the 27th International Conference on Software Engineering, 2005, pp. 342–351.

[8] G. Friedrich, M. Stumptner, F. Wotawa, Model-Based Diagnosis of Hardware Designs, Artificial Intelligence (1996) 3–39.

[9] S. Chaki, A. Groce, O. Strichman, Explaining Abstract Counterexamples, in: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, pp. 73–82.

[10] A. Zeller, Why Programs Fail: A Guide to Systematic Debugging, 2nd Edition, Morgan Kaufmann, 2009.

[11] A. Griesmayer, S. Staber, R. Bloem, Automated Fault Localization for C Programs, Electronic Notes in Theoretical Computer Science (2007) 95–111.

[12] S. Qadeer, D. Wu, KISS: Keep It Simple and Sequential, in: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, 2004, pp. 14–24.

[13] A. Biere, A. Cimatti, E. Clarke, O. Strichman, Y. Zhu, Bounded Model Checking, Advances in Computers (2003) 117–148.

[14] E. Alves, L. Cordeiro, E. Lima Filho, Fault Localization in Multi-Threaded C Programs Using Bounded Model Checking, in: 2015 Brazilian Symposium on Computing Systems Engineering (SBESC), 2015, pp. 96–101.

[15] L. Cordeiro, B. Fischer, J. Marques-Silva, Smt-Based Bounded Model Checking for Embedded ANSI-C Software, IEEE Transactions on Software Engineering (2012) 957–974.

[16] D. R. Butenhof, Programming with POSIX Threads, Addison-Wesley Longman Publishing Co., Inc., 1997.

[17] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, G. Parlato, Bounded Model Checking of Multi-Threaded C Programs via Lazy Sequentialization, in: Proceedings of the 16th International Conference on Computer Aided Verification, 2014, pp. 585–602.

[18] P. Godefroid, N. Nagappan, Concurrency at Microsoft: An Exploratory Survey, in: CAV Workshop on Exploiting Concurrency Efficiently and Correctly, 2008.

[19] L. Cordeiro, B. Fischer, Verifying Multi-Threaded Software Using SMT-Based Context-Bounded Model Checking, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 331–340.

[20] J. Morse, L. C. Cordeiro, D. Nicole, B. Fischer, Context-bounded model checking of LTL properties for ANSI-C software, in: 9th International Conference Software Engineering and Formal Methods SEFM, Vol. 7041 of LNCS, 2011, pp. 302–317.

[21] J. Morse, L. C. Cordeiro, D. Nicole, B. Fischer, Model checking LTL properties over ANSI-C programs with bounded traces, Software and System Modeling 14 (1) (2015) 65–81.

[22] L. C. Cordeiro, SMT-based Bounded Model Checking for Multi-Threaded Software in Embedded Systems, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 2010, pp. 373–376.

[23] S. La Torre, P. Madhusudan, G. Parlato, Reducing Context-Bounded Concurrent Reachability to Sequential Reachability, in: Proceedings of the 21st International Conference on Computer Aided Verification, 2009, pp. 477–492.

[24] E. Clarke, D. Kroening, F. Lerda, A Tool for Checking ANSI-C Programs, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2004, pp. 168–176.

[25] E. M. Clarke, H. Veith, Counterexamples Revisited: Principles, Algorithms, Applications, in: Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday, 2003, pp. 208–224.

[26] E. M. Clarke, O. Grumberg, K. L. McMillan, X. Zhao, Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking, in: Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference, 1995, pp. 427–432.

[27] T. Ball, M. Naik, S. K. Rajamani, From Symptom to Cause: Localizing Errors in Counterexample Traces, in: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2003, pp. 97–105.

[28] T. Ball, S. K. Rajamani, Automatically Validating Temporal Safety Properties of Interfaces, in: Proceedings of the 8th International SPIN Workshop on Model Checking of Software, 2001, pp. 103–122.

[29] A. Groce, W. Visser, What Went Wrong: Explaining Counterexamples, in: Proceedings of the 10th International Conference on Model Checking Software, 2003, pp. 121–136.

[30] Java Pathfinder: Framework for Verification of Java Programs. URL `http://babelfish.arc.nasa.gov/trac/jpf`

[31] A. Groce, S. Chaki, D. Kroening, O. Strichman, Error Explanation with Distance Metrics, International Journal on Software Tools for Technology Transfer (2006) 229–247.

[32] A. Griesmayer, Dissertation Debugging Software: From Verification to Repair (2007).

[33] H. Ohanian, J. Markert, Physics for Engineers and Scientists, 3rd Edition, W. W. Norton & Company, 2006.

[34] R. Brummayer, A. Biere, Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays, in: Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, 2009, pp. 174–177.

[35] J. A. Jones, M. J. Harrold, J. Stasko, Visualization of Test Information to Assist Fault Localization, in: Proceedings of the 24th International Conference on Software Engineering, 2002, pp. 467–477.

[36] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, M. Kim, Mutation-Based Fault Localization for Real-World Multilingual Programs, in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 464–475.

[37] D. Beyer, Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016), 2016, pp. 887–904.

[38] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, G. Parlato, Verifying Concurrent Programs by Memory Unwinding, in: 21st International Conference Tools and Algorithms for the Construction and Analysis of Systems, 2015, pp. 551–565.

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, The MIT Press, 2009.

[40] A. Zeller, AskIgor: Automated Debugging (2006).
URL `http://www.st.cs.uni-saarland.de/askigor/`

[41] G. Birch, B. Fischer, M. R. Poppleton, Fast Model-Based Fault Localisation with Test Suites, in: 9th International Conference Tests and Proofs, 2015, pp. 38–57.

[42] C. Cadar, D. Dunbar, D. Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008, pp. 209–224.

[43] S. M. Park, Effective Fault Localization Techniques for Concurrent Software (2014).

[44] M. Jose, R. Majumdar, Cause Clue Clauses: Error Localization Using Maximum Satisfiability, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011, pp. 437–446.

[45] M. Papadakis, Y. Le Traon, Metallaxis-FL: Mutation-Based Fault Localization, Software Testing, Verification and Reliability (2015) 605–628.