Fault Localization in Multi-Threaded C Programs using Bounded Model Checking

Erickson H. da S. Alves, Lucas C. Cordeiro, Eddie B. de Lima Filho
Federal University of Amazonas, Brazil
E-mails: erickson.alves@indt.org.br, lucascordeiro@ufam.edu.br, eddie@ctpim.org.br

***Abstract*—Software debugging is a very time-consuming process, which is even worse for multi-threaded programs, due to the non-deterministic behavior of thread-scheduling algorithms. However, the debugging time may be greatly reduced, if automatic methods are used for localizing faults. In this study, a new method for fault localization, in multi-threaded C programs, is proposed. It transforms a multi-threaded program into a corresponding sequential one and then uses a fault-diagnosis method suitable for this type of program, in order to localize faults. The code transformation is implemented with rules and context switch information from counterexamples, which are typically generated by bounded model checkers. Experimental results show that the proposed method is effective, in such a way that sequential fault-localization methods can be extended to multi-threaded programs.**

***Keywords*—Multi-threaded Software, Bounded Model Checking, Fault Localization.**

I. Introduction

Recently, it has become more and more common for technology to handle various tasks in everyday life, each one with an associated complexity. Ensuring that systems work properly imply in cost reduction and even, in some areas, that lives are safe. This is what makes program debugging so worthy of attention in computer-based systems. Program debugging is a very important but time-consuming task, in software development processes, which can be divided into three steps: fault detection, fault localization, and fault correction. However, the associated debugging time can be largely reduced, if automatic methods are used for localizing faults, especially in multi-threaded programs, which are widely used in embedded system products.

A number of different approaches have been introduced, in order to provide automated methods for localizing faults in applications, based on the generation of a program model that is extracted from its source code [1]. Those include slicing [2], mutation testing [3], trace-based analysis [4], delta-debugging [5], model-based debugging [6], and model checking [7]. In this study, a fault localization method is proposed, which relies entirely on model-checking techniques. In particular, Bounded Model Checking (BMC) based on Satisfiability Modulo Theories (SMT) is used to automatically refute a safety property and consequently produce a counterexample, if the (multi-threaded) program does not satisfy a given specification. It is worth noticing that the generated counterexamples may be regarded as error traces, which contain useful information about faults, so that one can localize and correct them.

The C Bounded Model Checker (CBMC) [8] and also the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) [9] are both well known BMC tools, which are suitable for verifying multi-threaded C programs. Since the majority of the initiatives, in multi-threaded software verification, have focused on Java, as shown by Park *et al.* [10], this study suggests an automated fault localization method for multi-threaded C programs, using SMT-based BMC techniques. In particular, in this present work, ESBMC [9] is adopted, since it is one of the most efficient BMC tools, as reported by Beyer [11], and it also supports both single- and multi-threaded programs, using different SMT solvers to check for the generated verification conditions (VCs) [12].

The two major contributions of this work are: the evaluation of the method proposed by Griesmeyer *et al.* [13] and the improvement/extension of this same method, in order to support multi-threaded applications. The basic concept of extending the mentioned work [13] consists in transforming a multi-threaded program into a corresponding sequential one, by carrying out evaluation and transformation steps, and then using that work for localizing faults. As a consequence, the found violations, in the sequential code, can show the location of the initial faults, in the original multi-threaded program.

II. Preliminaries

A. Bounded Model Checking to Multi-threaded Software

The basic idea of BMC applied to multi-threaded software is to check for the negation of a given property, at a given depth [16]. Given a reachability tree $\Upsilon = \{\nu_1, \ldots, \nu_N\}$, which represents the program unfolding for a context bound C and a bound k, and a property ϕ, BMC derives a VC ψ_k^π for a given interleaving (or computation path) $\pi = \{\nu_1, \ldots, \nu_k\}$, such that ψ_k^π is satisfiable if and only if ϕ has a counterexample of depth *k*, which is exhibited by π. The VC 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 [19]. The model checking problem associated with SMT-based BMC, of a given π, is formulated by constructing the logical formula [12]

$$\psi_k^\pi = \overbrace{I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{k-1}, s_k)}^{constraints} \wedge \overbrace{\neg\phi_k}^{property} \quad . \quad (1)$$

Here, ϕ_k represents a safety property ϕ, in step k, I is the set of initial states, and $R(s_i, s_{i+1})$ is the transition relation at time steps i and $i+1$, as described by the states in π nodes. In order to check if (1) is satisfiable or unsatisfiable, the SMT solver constrains some symbols, by a given background theory (*e.g.*, arithmetic restricts the interpretation of symbols such as $+$, $\leq$, 0, and 1) [19]. If (1) is satisfiable, then ϕ is violated and the SMT solver provides a satisfying assignment, from which one can extract values of program variables to construct a counterexample, *i.e.*, a sequence of states $s_0, s_1, \ldots, s_k$, with $s_0 \in S_0$, and transition relations $R(s_i, s_{i+1})$, for $0 \leq i < k$. If (1) is unsatisfiable, one can conclude that no error state is reachable, in length k along π.

B. Fault Localization using Model Checking

The most basic task regarding fault localization, in model checking, is to generate a counterexample, which is provided when a program does not satisfy a given specification. According to Clarke *et al.* [20], [21], a counterexample does not solely provide information about the cause-effect relation of

a given violation, but also about fault localization. However, since an enormous amount of information is presented in a counterexample, actual fault lines may not be easily identified.

Several methods have been proposed, in order to localize possible fault causes, by means of counterexamples. An approach proposed by Ball *et al.* [22] tries isolating possible causes of counterexamples, which are generated by the SLAM model checker [23]. In summary, potential fault lines can be isolated by comparing transitions among obtained counterexamples and successful traces, since transitions not included in correct traces are possible causes of errors. Groce and Visser [24] state that if a counterexample exists, a similar but successful trace can also be found, using BMC techniques. Program elements related to a given violation are implicated by the minimal differences between that counterexample and a successful trace, which can be detected by the Java Pathfinder [25], thus providing execution paths that lead to error states, with respect to multi-threaded programs (*e.g.*, data race). The essence of the approach described by Groce *et al.* [26] is similar to the latter and uses alignment constraints to associate states, in a counterexample, with corresponding states in a successful trace, which was 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 employed for representing program executions, and non-matching constraints that represent concrete states might lead to faults. Additionally, if a distance metric property is not satisfied, a counterexample is generated by the BMC tool [26].

In contrast to the transition-based and difference-based methods mentioned above, a method can directly identify possible faults by combining instrumented programs and BMC, as shown by Griesmeyer *et al.* [13], [27], [28], which will be further demonstrated. The approach proposed in the present paper is based on that method and consists in an extension to multi-threaded programs, that is, it tries to identify fault lines in multi-threaded programs, using BMC techniques.

C. Method demonstration

The method proposed by Griesmeyer *et al.* [13] is based on the BMC technique, which can directly identify possible faults in programs. In particular, this method adds additional numerical variables (*e.g.*, $\texttt{diag}_1, \ldots, \texttt{diag}_n$) to identify a fault in a given program. Each line of a program, representing a statement `S`, is changed to a logic version of that statement. As a consequence, the value held by `S` will be either non-deterministically chosen by the BMC tool, if the value of `diag` is the same as the one representing the line related to statement `S`, or the one originally specified.

If the BMC tool identifies a `diag` value, by correcting this line in the original program, the fault can be avoided. In the case of multiple `diag` values, correcting those lines lead to a successful code execution. In order to find the full set of lines that cause a faulty behavior in a program, a new line[1] can be added to its source code, which is then rerun in the BMC tool. This process is repeatedly executed, until no more values of `diag` are obtained (*i.e.*, the run succeeds) [13].

As an example, a simple program slightly modified from Griesmeyer *et al.* [13], is presented in Fig. 1. Its modified version, using the mentioned method [13] and ready to be run by a BMC tool, is shown in Fig. 2. The diagnosis informed by a BMC tool is `diag == 4` and `diag == 3`, which means that changing line 4 (to "`a = 6`") or line 3 (to `if(0)`), in the original program, can result in source code that is able to successfully execute, therefore, one can note that the example below contains a single fault.

[1] `assume(diag != a)` (`a` is the line number obtained in the last run)

```
1  void main() {
2    int a, b, c, d;
3    if (a) {
4      a = 5;
5      b = 2;
6      c = a + b;
7      if (a % 2 == 0) {
8      int d;
9      a = d;
10     }
11     assert (c == 8);
12   }
13 }
```

Fig. 1. A simple ANSI-C program with a single fault.

```
1  int nondet();
2  void main() {
3    int a, b, c, d;
4    int diag;
5    diag = nondet();
6    a = 5; b = 2; c = 7;
7    if (diag == 3? nondet(): a) {
8      a = (diag == 4? nondet(): 5);
9      b = (diag == 5? nondet(): 2);
10     c = (diag == 6? nondet(): (a + b));
11     if (diag == 7? nondet(): (a%2==0)) {
12       int d;
13       a = (diag == 9? nondet(): d);
14     }
15     assume(c == 8);
16   }
17   assert(false);
18 }
```

Fig. 2. The diagnosis model of the example shown in Fig. 1, where *nondet()* represents a non-deterministic function.

III. Fault Localization in Multi-Threaded C Programs using BMC

The proposed method, which has the goal of localizing faults in multi-threaded C programs, is based on Griesmeyer's method [13] and counterexamples generated by BMC tools, such as ESBMC. Its key concept is to transform a multi-threaded program into a corresponding sequential one and then apply instrumentation for identifying faults [13].

A. Transformations from Multi- to Single-threaded Programs

The transformation from multi-threaded programs into sequential ones can be split into four distinct steps. First, counterexamples are obtained from a BMC tool, which contain useful pieces of information related to faults. Then, the framework described below is applied, which consists in code used as fixed structure for a new sequential version, together with the use of some rules (defined later in section III-B2). Following that, an original (multi-threaded) program is converted into its sequential version and, finally, order control is included into the latter, specifying the order in which threads are executed. These steps are summarized in Fig. 3.

A framework provides the same execution sequence as in the original program. It consists basically in writing each thread code inside a $case$ statement, and their execution sequence is specified in the $order$ array. Such a framework

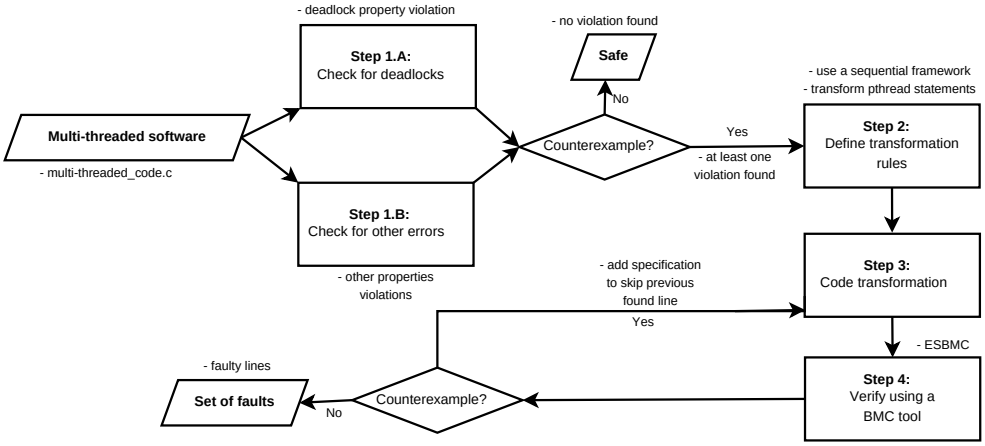

Fig. 3. Proposed method for fault localization in multi-threaded software.

```
1  int order[1] = {1};
2  int main(int argc, char *argv[]) {
3    int order_index;
4    for(order_index= 0; order_index < 1;
5        order_index++) {
6      switch(order[order_index]) {
7        case 1:
8            case 11: { ... }
9            ...
10           case 20: { ... }
11       break;
12       case 2:
13           case 21: { ... }
14           ...
15           case 30: { ... }
16       break;
17       case 3:
18           case 31: { ... }
19           ...
20           case 40: { ... }
21       break;
22       ...
23       default:
24       break;
25     }
26   }
27   return 1;
28 }
```

Fig. 4. The standard framework to localize faults in sequential code.

is used as the basic structure for new sequential versions of multi-threaded programs, and Fig. 4 shows how it is encoded.

As one can note, the mentioned framework provides new fixed positions, for each part of the original code, and Table I shows the relation between new positions and code-fragment types, that is, 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 $main$ function. The body of its $main$ function, in the original code, is placed between the $case$ 1 statement and its respective $break$ command, the 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 $main$ function are all passed to the sequential version $main$ one. In cases where threads are partially executed, a context switch occurs, another thread is executed, or a previous thread continues to execute from where it stopped, the respective pieces of code are inserted into each $case$ inside the N^{th} $case$ (the N^{th} case represents the N^{th} thread), in such a way that the execution order remains the same.

TABLE I. 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, switch order control is required. A fixed context switch order, from a counterexample of a multi-threaded program, can be copied to a new sequential one by controlling “$case$” and conditional statements[2], in the framework $switch$ statement. In general, adding context switch order control to the new sequential program can be divided into two steps. In order to show a simple situation for illustrating that, 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 tool, has N context switches, and from those N context switches, 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 is to get 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 conditional statements[2] 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 the original code.

One can note that if there are iteration statements in the original multi-threaded program, for every iteration statement, a global variable named “$loopcounter$” is added. 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, then the value held by $loopcounter$ must also be used in the

[2] $if(order[order\_index]) == X)$ $break$;, where X represents the number of the context switch

respective $break$ statement, in order to maintain the original program execution sequence.

The second step consists in modifying values related to the $order$ array, in such a way that the execution order is kept, in a new sequential program. By changing lines 1 and 4, in Fig. 4, according to the specific number of occurred context switches and their execution order, it is possible to guarantee the original execution order, since a $switch$ statement (line 6) selects which piece of code (representing threads from the original program) is executed, based on $order[order\_index]$. For instance, if the execution order of the original code is thread 0, thread 2, and thread 1 (note that this information was previously extracted from the counterexample), the $order$ array will hold 11, 31, and 21, meaning that the first $case$ will be executed, then the third and, finally, the second one.

B. Code Transformation

1) Grammar: Transformation rules, regarding code fragments, are needed, when code fragments are added to corresponding positions in the mentioned framework. Given that the most common faults, in multi-threaded programs, are related to data races and deadlocks [29], a simple grammar, for code fragments, can be defined, w.r.t. these two faults types.

Threads in C are typically implemented through the POSIX Pthreads [17] standard, which defines an application programming interface for creating and handling threads. POSIX threads are available in a library, called $pthread$, which is used in UNIX operating systems. Therefore, two groups are created: one regarding *pthread* non-related code fragments, which is group $non-pthread$, and another for *pthread* related code fragments, called group $pthread$.

2) Rules: The rules used to transform code fragments, in the original multi-threaded program, are shown in Table II. Note that such transformations rely on counterexamples generated by a BMC tool. Additionally, different threads are simulated by different `case` statements, since the $main$ function is in the first $case$ statement, the first executed thread is in the second $case$ statement, and so on, as already mentioned.

TABLE II. RULES TO TRANSFORM MULTI-THREADED PROGRAMS

Group	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	Integer variable is declared
	pthread_mutex_lock	ϵ	1 is assigned to variable
	pthread_mutex_unlock	ϵ	0 is assigned to variable
	pthread_cond_t	ϵ	Integer variable is declared
	pthread_cond_init	ϵ	0 is assigned to variable
	pthread_cond_wait	ϵ	1 is assigned to variable
	pthread_cond_signal	ϵ	0 is assigned to variable

In Table II, ϵ stands for the removal of the respective statement in the new sequential version. When a deadlock is returned, by the BMC tool, one needs to add an integer variable for simulating the *pthread_mutex_t* and/or *pthread_cond_t* variables. Finally, the $unwind$ process for expressions, in Table II, consists in removing the original function and directly writing this piece of code, *e.g.*, if the value returned by function f is assigned to variable i, when it is called with argument a ($i = f(a)$), such call is removed and replaced by the actual calculation. This process is described in Figures 5 and 6.

```
1 int f(int m) {
2   int b;
3   b = m;
4   return m;
5 }
6 ...
7 int i;
8 i = f(a);
```

Fig. 5. Original code fragment.

```
1 int i;
2 {
3   int m = a;
4   int b;
5   b = m;
6   i = m;
7 }
```

Fig. 6. Transformed code fragment from Fig. 5.

In addition, if an error detected by the chosen BMC tool is a deadlock, then rules in groups $non-pthread$ and $pthread$ are applied to create the sequential version of the original program; otherwise, only rules in group $non-pthread$ are used.

IV. EXPERIMENTAL EVALUATION

A. Experimental Setup

In order to verify and validate the proposed method, ESBMC v1.24.1 with SMT solver Boolector [30] was used. All experiments were conducted on an otherwise idle Intel Core i7 – 4500 1.8Ghz processor, with 8 GB of RAM and running Fedora 21 64-bits operating system.

The benchmarks in Table III are the same used when evaluating ESBMC for multi-threaded C programs [16]. $account\_bad.c$ 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. $lazy01\_bad.c$ uses a mutex to control summation operations over a shared variable and then checks its value. $queue\_bad.c$ is a program simulating a data-queue structure. $sync01\_bad.c$ and $sync02\_bad.c$ 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 shared variables and checks whether they are equivalent, through different threads. $twostage\_bad.c$ simulates a great number of threads running simultaneously and, finally, $wronglock\_bad.c$ simulates a large number of producer threads and the propagation of their respective values, to other threads.

The experimental evaluation procedure can be split into three different steps. First, it is necessary to identify which group (see section III-B) a given benchmark belongs to and, in order to have this information, one needs to execute a specific command line[3], in ESBMC. If the result given by ESBMC is *verification failed*, then the benchmark belongs to group $pthread$; otherwise, it belongs to group $non-pthread$. In

the second step, it is necessary to add context-switch numbers through the method presented in III-A, which is achieved by removing the `--deadlock-check` option in the issued command line[3]. 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 III-B2 and the method proposed by Griesmeyer *et al.* [13].

Finally, the sequential version of the program can be verified in ESBMC, using a command line[3] without the `--deadlock-check` option, changing the specified file, and applying the same strategy demonstrated in section II-C.

B. Experimental Results

Table III summarizes the experimental results. **F** describes the name of the benchmark, **L** represents the number of lines in the code, **T** is the number of threads in the code, **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, **AE** is the number of actual errors, **R** stands for the actual result (`1` if the information retrieved by ESBMC is helpful), and, finally, **VT** is the time that ESBMC took to verify the benchmark. The question mark is used to identify tests from which no information was retrieved, due to system limitations.

TABLE III. 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	?	?	∞	?
circular_buffer_bad.c	109	2	0	7/7	0.227	1
lazy01_bad.c	48	3	1	4/4	0.125	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	1/0	0.101	0
twostage_bad.c	128	9	?	?	∞	?
wronglock_bad.c	111	7	?	?	∞	?

The verification of *account_bad.c* presented 3 different `diag` values, which are in different parts of the code; however, they ultimately identified the actual fault in the original code, 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 both *lazy01_bad.c* and *queue_bad.c* presented 4 errors. In the former, ESBMC indicated that the faults lie on the code part, where its shared variable is used, which led to a bad assertion. In the latter, 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.

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

Although *sync01_bad.c* and *token_ring_bad.c* presented no errors, both were 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 assert, ESBMC still diagnoses 0. Indeed, both have synchronization problems and the proposed method was unable to provide useful information.

The proposed methodology was not able to verify benchmarks *carter_bad.c*, *twostage_bad.c*, and *wronglock_bad.c*, since there was not enough memory while ESBMC checked for deadlocks. This probably occurred due to the great number of threads (in case of *twostage_bad.c*, and *wronglock_bad.c*) or due to a very large set of data variables (*carter01_bad.c*).

According to the results shown in Table III, one can note that the proposed methodology was able to find faults (useful information) in 6 out of 11 benchmarks, which amounts to 54.55%. Note that benchmarks whose verification failed and, consequently, from which no counterexample was extracted, 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, and loop transformations also need a significant work, so that threads interleaving inside loops can be better represented.

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 multi-threaded C programs, since ESBMC is able to provide helpful diagnosis information regarding potential faults.

V. RELATED WORK

Jones *et al.* [14] show how test information visualization can assist in fault localization. By coloring program statements that participate in the outcome of a program execution, with a test suite, it is possible to assist users to inspect code, evaluate statements involved in failures, and identify possible faults. Despite being useful, this approach is not fully automated, since it still needs users to execute most of its steps.

Cleve *et al.* [5] show how *cause transitions*, which are moments where a variable replaces another as failure cause, can locate defects in programs. Such an approach is possible due to a comparison regarding program states of failing and passing runs; however, given that such state differences can occur all over the program run, the focus is *on space*, with a subset of variables that is relevant to the failure occurrence, and also *on time*, where *cause transitions* occur. Even though this approach works, high code-coverage test cases and precise choices in space and time are needed.

Birch *et al.* [15] describe a method for fast model-based fault localization, which, given a test suite, automatically identifies a small subset of program locations, where faults exist, by using symbolic execution methods. In summary, the mentioned algorithm tries to find counterexamples that are capable of localizing faults, based on failing test cases from a test suite. The key factor to its speed is that if an execution takes longer than expected, it is pushed into a queue, for later handling, and then another execution is chosen to be run. This approach is indeed effective, but as a downside it relies entirely on a test suite to be accurate.

Cordeiro *et al.* [16] describe three approaches (*lazy*, *schedule recording*, and *underapproximation and widening*) to model check multi-threaded software using ESBMC [9]. By

modelling synchronization fundamentals of the POSIX thread library [17], it creates an instrumented program, with respect to the original one, and model checks this new version, trying to find a defect or explore all possible interleavings. On one hand, this work shows itself to be effective to verify multi-threaded software; on the other hand, it can only state whether a defect exists or not, and if so, it cannot directly point the location of such.

Griesmeyer *et al.* [13] propose a method for localizing faults, in ANSI-C programs, by instrumenting the original code and running that new version on a model checker. Such an approach is very helpful, given that model checkers are able to identify the exact fault line; however, this work only presented a method for sequential programs [13]. A good point in this approach is that the counterexamples generated by a model checker indeed indicate faulty lines; a drawback of this approach is that it only works for standard ANSI-C programs, *i.e.*, procedural/sequential software.

Jose *et al.* [18] report a method to localize faults in programs, using a reduction to the Maximal Satisfiability Problem, which informs the maximum number of clauses, of a Boolean formula, that can simultaneously be satisfied by an assignment. The potential error is given by finding the maximal set of clauses that can be satisfied, in a formula generated by combining a failing program execution and a Boolean trace formula, and outputting the complement set. Although this approach is useful for locating faulty lines, it still depends on a failing program execution and a correctness specification.

The closest related work is that of Park *et al.* [10], which describe a dynamic fault localization method to localize the root causes of concurrency bugs in Java programs, based on dynamic pattern detection and statistical fault localization. In contrast to Park *et al.*, this present paper marks the first application of a fault-localization method, based on BMC techniques, to a broader range of multi-threaded C programs. Despite of being effective to concurrent programs, the approach proposed by Park *et al.* works only for Java programs.

To the best of our knowledge, this work presents a new method to localize faults in multi-threaded C programs by instrumenting a sequential version of the original code. However, to obtain this sequential version of the multi-threaded program, some rules and grammar are needed in order to maintain its original execution.

VI. Conclusion

A novel method for localizing faults in multi-threaded C programs, using code transformation and BMC techniques, was proposed. It is based on the approach introduced by Griesmeyer *et al.* [13] and an extension specific to handle multi-threaded programs, which is useful for embedded systems.

The experimental results revealed the performance of the proposed methodology, when localizing faults in standard multi-threaded C benchmarks. In particular, it was able to identify potential faults in multi-threaded software, in 54.55% of the chosen benchmarks. Besides, this number may change to 75%, if only the ones able to be verified are considered, *i.e.*, those where counterexamples are provided by the BMC tool (see column **VT** in Table III).

As future work, new rules for code transformation and also an improved grammar will be developed, in order to increase the methodology accuracy. Additionally, an *Eclipse* plug-in will be developed for automating the fault diagnosis process, during development.

Acknowledgements. Part of the results presented in this paper were obtained with the project for research and human resources qualification, for under- and post-graduate levels, in the areas of industrial automation, mobile devices software, and Digital TV, sponsored by Samsung Eletrônica da Amazônia Ltda, under the terms of Brazilian Federal Law number $8.387/91$. This research was also supported by CNPq $475647/2013\text{-}0$ and FAPEAM $062.01722/2014$ grants.

References

[1] W. Mayer and M. Stumptner., *Evaluating Models for Model-Based Debugging*. *ASE*, pp. 128–137, 2008.

[2] F. Tip, *"A Survey of Program Slicing Techniques"*. In Journal of Programming Languages, v. 3, no. 3, pp. 121–189, 1995.

[3] A. J. Offutt *et al.*, *An Experimental Determination of Sufficient Mutant Operators*. *TOSEM*, v. 5, no. 2, pp. 99–118, 1996.

[4] H. He and N. Gupta, *Automated Debugging using Path-based Weakest preconditions*. *FASE*, *LNCS* 2984, pp. 267–280, 2004.

[5] H. Cleve and A. Zeller, *Locating Causes of Program Failures*. *ICSE*, pp. 342–351, 2005.

[6] G. Friedrich *et al.*, *Model-based Diagnosis of Hardware Designs*. *ECAI*, pp.491–495, 1996.

[7] S. Chaki *et al.*, *Explaining Abstract Counterexamples*. In SIGSOFT FSE, pp.73–82, 2004.

[8] E. Clarke *et al.*, *A Tool for Checking ANSI-C Programs*. *TACAS*, *LNCS* 2988, pp. 168–176, 2004.

[9] L. Cordeiro *et al.*, *SMT-Based Bounded Model Checking for Embedded ANSI-C Software*. *TSE*, v. 38, n. 4, pp. 957–974, 2012.

[10] S. M. Park, *Effective Fault Localization Techniques for Concurrent Software*. PhD dissertation, 2014.

[11] D. Beyer, *Software Verification and Verifiable Witnesses (Report on SV-COMP 2015)*. *TACAS*, *LNCS* 9035, pp. 401-416, 2015.

[12] J. Morse *et al.*, *ESBMC 1.22 (Competition Contribution)*. *TACAS*, *LNCS* 8413, pp. 405-407, 2014.

[13] A. Griesmeyer *et al.*, *Automated Fault Localization for C Programs*. *ENTCS*, v. 174, pp. 95–111, 2007.

[14] J. A. Jones *et al.*, *Visualization of Test Information to Assist Fault Localization*. *ICSE*, pp. 467–477, 2002.

[15] G. Birch *et al.*, *Fast Model-Based Fault Localisation with Test Suites*. To appear in *TAP*, 2015.

[16] L. Cordeiro and B. Fischer. *Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking*. *ICSE*, pp. 331-340, 2011.

[17] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[18] M. Jose and R. Majumdar, *Cause clue clauses: error localization using maximum satisfiability*. *PLDI*, pp. 437–446, 2011.

[19] L. M. de Moura and N. Bjørner. *Z3: An efficient SMT solver. TACAS*, *LNCS* 4963, pp. 337–340, 2008.

[20] E. Clarke *et al.*, *Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking*. *DAC*, pp. 95–111, 1995.

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

[22] T. Ball *et al.*, *From Symptom to Cause: Localizing Erros in Counterexample Traces*. *POPL*, pp. 97–105, 2003.

[23] T. Ball and S. Rajamani, *Automatically Validating Temporal Safety Properties of Interfaces*. *SPIN*, *LNCS* 2057, pp. 103–122, 2001.

[24] A. Groce and W. Visser, *What Went Wrong: Explaining Counterexamples*. *SPIN*, *LNCS* 2648, pp. 121–135, 2003.

[25] Java Pathfinder: framework for verification of Java programs. *http://babelfish.arc.nasa.gov/trac/jpf*.

[26] A. Groce *et al.*, *Error Explanation with Distance Metrics*. *STTT*, v. 8, no. 3, pp. 229–247, 2006.

[27] A. Griesmeyer, *Debugging Software: From Verification to Repair*. PhD dissertation, 2007.

[28] A. Griesmeyer *et al.*, *Fault Localization using a Model Checker*. *STVR*, v. 20, pp. 149–173, 2010.

[29] A. Mühlenfeld and F. Wotawa, *Fault Detection in Multi-Threaded C++ Server Applications*. *ENTCS*, v. 174, n. 9, pp. 5-22, 2007.

[30] R. Brummayer and A. Biere, *Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays*. *TACAS*, *LNCS* 5505, 2009.