Fault Localization in Multi-threaded C Software using Bounded Model Checking

Erickson H. da S. Alves, Lucas C. Cordeiro, and Eddie B. de Lima Filho
Multi-threaded Software and Difficulties

- Multi-threaded software are more common in embedded systems.
- Despite several advantages, they present difficulties related to asserting their correctness.

![Diagram of threads]

- Main thread
- Thread A
- Thread B
- Thread C
Multi-threaded Software and Difficulties

• Multi-threaded software are more common in embedded systems

• Despite several advantages, they present difficulties related to asserting their correctness

• Multi-threaded software difficulties are:
  – Concurrent bugs usually occur under specific thread interleavings
  – The number of interleavings grows exponentially with the number of threads and program statements
  – Context switches among threads increase the number of possible executions
  – However, concurrent bugs usually occur in few context switches [Qadeer&Rehof’05]
Multi-threaded Software and Difficulties

- Multi-threaded software are more common in embedded systems
- Despite several advantages, they present difficulties related to asserting their correctness
- Multi-threaded software difficulties are:
  - Concurrent bugs usually occur under specific thread interleavings
  - The number of interleavings grows exponentially with the number of threads and program statements
  - Context switches among threads increase the number of possible executions
  - However, concurrent bugs usually occur in few context switches [Qadeer&Rehof’05]
Multi-threaded Software and Difficulties

- Multi-threaded software are more common in embedded systems.
- Despite several advantages, they present difficulties related to asserting their correctness.
- Multi-threaded software difficulties are:
  - Concurrent bugs usually occur under specific thread interleavings.
  - The number of interleavings grows exponentially with the number of threads and program statements.
  - Context switches among threads increase the number of possible executions.
  - However, concurrent bugs usually occur in few context switches [Qadeer&Rehof’05].
Multi-threaded Software and Difficulties

- Multi-threaded software are more common in embedded systems
- Despite several advantages, they present difficulties related to asserting their correctness
- Multi-threaded software difficulties are:
  - Concurrent bugs usually occur under specific thread interleavings
  - The number of interleavings grows exponentially with the number of threads and program statements
  - Context switches among threads increase the number of possible executions
  - However, concurrent bugs usually occur in few context switches [Qadeer&Rehof’05]
Bounded Model Checking (BMC)

- Basic Idea: given a transition system $M$, check negation of a given property $\varphi$ up to given depth $k$

  ![Transition System Diagram]

- Translated into a VC $\psi$ such that: $\psi$ is satisfiable iff $\varphi$ has counterexample of max. depth $k$

- BMC has been applied successfully to verify multi-threaded software since 2005, but there are some limitations
Objectives

Provide a methodology to localize faults in multi-threaded C software using BMC techniques

- Expand existing fault localization approaches to sequential programs to handle multi-threaded software and provide a sequentialization method to translate multi-threaded C software into sequential software
- Design a grammar to model functions and variables related to multi-threaded programming in C
- Evaluate our proposed method using benchmarks from the Software Verification Competition (SV-Comp)
The Efficient SMT-Based Context-Bounded Model Checker (ESBMC)

- **C parser**: processes the ANSI-C file and builds an Abstract Syntax Tree (AST)
The Efficient SMT-Based Context-Bounded Model Checker (ESBMC)

GOTO Program: converts an ANSI-C program into a GOTO-Program (replacement of *switch* and *while* by *if* and *goto* expressions)
The Efficient SMT-Based Context-Bounded Model Checker (ESBMC)

GOTO Symex: performs a symbolic execution of the program and generates SMT equations for constraints ($C$) and properties ($P$)
The Efficient SMT-Based Context-Bounded Model Checker (ESBMC)

SMT Solver: evaluates the expression $C \land \neg P$, using the specified solver
Fault Localization using Model Checking

- [Griesmeyer’07] proposed a method to localize faults in sequential C code
  - It uses non-determinism to obtain values for a variable (diag), which represent faulty lines
  - Assignments are replaced by a non-deterministic version of them. If a counterexample is obtained, it also contains a value for diag

```c
int non_det();
int diag;
...
int main(void *args) {
    diag = non_det();
    ...
    assert(0);
}
```

```c
(17) i = 7;
    ...
    i = (diag == 17 ? non_det() : 7);
```
Our Proposed Methodology

**Step 1.A:** Check for deadlocks

- Deadlock property violation

**Step 1.B:** Check for other errors

- Other properties violation

**Step 2:** Define transformation rules

- No violation found
- Transform pthread statements
- Use a sequential framework

**Step 3:** Code transformation

- At least one violation found
- Use non-determinism
- Add specification to skip previous found line

**Step 4:** Verify using a BMC tool

- ESBMC

* CEX - counterexample
Our Proposed Methodology

**Motivation**

**Background**

**Methodology**

**Results**

**Conclusions**

---

**Step 1.A:** Check for deadlocks

- Deadlock property violation

**Step 1.B:** Check for other errors

- Other properties violation

**Set of faults**

- No violation found

**CEX?**

- No

**Safe**

- Transform pthread statements
  - Use a sequential framework

- At least one violation found

**Step 2:** Define transformation rules

**Step 3:** Code transformation

- Use non-determinism
- ESBMC

**Step 4:** Verify using a BMC tool

---

* CEX - counterexample
Our Proposed Methodology

Step 1.A: Check for deadlocks
- Deadlock property violation
- No violation found

Step 1.B: Check for other errors
- Other properties violation

Step 2: Define transformation rules
- At least one violation found

Step 3: Code transformation
- Add specification to skip previous found line

Step 4: Verify using a BMC tool
- ESBMC

* CEX - counterexample

Motivation
Background
Methodology
Results
Conclusions

Multi-threaded C software
multi-threaded_code.c

Set of faults

Safe

CEX?

No

No violation found

Yes

Transform pthread statements
- Use a sequential framework

No

Yes

At least one violation found

- Use non-determinism

- ESBMC
Our Proposed Methodology

**Motivation**

**Background**

**Methodology**

**Results**

**Conclusions**

---

**Step 1.A:** Check for deadlocks
- Deadlock property violation

**Step 1.B:** Check for other errors
- Other properties violation

**Step 2:** Define transformation rules
- At least one violation found

**Step 3:** Code transformation
- Add specification to skip previous found line

**Step 4:** Verify using a BMC tool
- ESBMC

---

**Multi-threaded C software**

multi-threaded_code.c

---

* CEX - counterexample
Our Proposed Methodology

Motivation

Background

Methodology

Results

Conclusions

- Deadlock property violation

Step 1.A: Check for deadlocks

- Other properties violation

Step 1.B: Check for other errors

- No violation found

Safe

CEX?

No

- Transform pthread statements
- Use a sequential framework

Step 2: Define transformation rules

- At least one violation found

Yes

Step 3: Code transformation

- Use non-determinism

Step 4: Verify using a BMC tool

- ESBMC

Set of faults

- No violation found

CEX?

Yes

- Add specification to skip previous found line

CEX?

- Transform pthread statements
- Use a sequential framework

Step 2: Define transformation rules

- At least one violation found

Yes

Step 3: Code transformation

- Use non-determinism

Step 4: Verify using a BMC tool

- ESBMC

18
Our Proposed Methodology

Step 1.A: Check for deadlocks
- Deadlock property violation

Step 1.B: Check for other errors
- Other properties violation

Set of faults

Step 2: Define transformation rules
- At least one violation found
- Use a sequential framework
- Transform pthread statements
- Use non-determinism

Step 3: Code transformation
- Add specification to skip previous found line
- ESBMC

Step 4: Verify using a BMC tool
- No violation found

* CEX - counterexample

Multi-threaded C software

multi-threaded_code.c
Our Proposed Methodology

Motivation

Background

Methodology

Results

Conclusions

Step 1.A: Check for deadlocks
- Deadlock property violation

Step 1.B: Check for other errors
- Other properties violation

Multi-threaded C software
multi-threaded_code.c

Step 2:
- Define transformation rules
- Transform `pthread` statements
- Use a sequential framework

Step 3:
- Code transformation
- Use non-determinism
- ESBMC

Step 4:
- Verify using a BMC tool

Safe

No

CEX?

Yes

No

CEX?

Yes

- At least one violation found
- Transform `pthread` statements
- Use a sequential framework

- Add specification to skip previous found line

Set of faults

* CEX - counterexample
Our Proposed Methodology

Motivation

Background

Methodology

Results

Conclusions

Step 1.A: Check for deadlocks
- Deadlock property violation

Step 1.B: Check for other errors
- Other properties violation

Set of faults

Step 2: Define transformation rules
- Transform *pthread* statements
- Use a sequential framework

Step 3: Code transformation
- Use non-determinism

Step 4: Verify using a BMC tool
- ESBMC

* CEX - counterexample
Our Proposed Methodology

1. **Multi-threaded C software**
   - multi-threaded_code.c

2. **Step 1.A:**
   - Check for deadlocks
   - Deadlock property violation
   - CEX?
   - No violation found

3. **Step 1.B:**
   - Check for other errors
   - Other properties violation

4. **Set of faults**
   - No
   - CEX?

5. **Step 2:**
   - Define transformation rules
   - At least one violation found
   - Transform *pthread* statements
   - Use a sequential framework

6. **Step 3:**
   - Code transformation
   - Use non-determinism
   - ESBMC

7. **Step 4:**
   - Verify using a BMC tool
   - ESBMC

* CEX - counterexample

- Yes
- No

- Safe
- No violation found
- Transform *pthread* statements
- Use a sequential framework

- CEX?
- Yes
- At least one violation found
- Transform *pthread* statements
- Use a sequential framework

- CEX?
- No
- Other properties violation
- Add specification to skip previous found line
- ESBMC

- CEX?
- Yes
- Other properties violation
- Add specification to skip previous found line
- ESBMC

- CEX?
- No
- Other properties violation
- Add specification to skip previous found line
- ESBMC
Our Proposed Methodology

**Motivation**

- Multi-threaded C software
- multi-threaded_code.c

**Background**

**Methodology**

**Results**

**Conclusions**

**Step 1.A:** Check for deadlocks
- Deadlock property violation

**Step 1.B:** Check for other errors
- Other properties violation

**Step 2:** Define transformation rules
- At least one violation found

**Step 3:** Code transformation
- Add specification to skip previous found line
- Use non-determinism

**Step 4:** Verify using a BMC tool
- ESBMC

Set of faults

CEX? - Counterexample

Safe
- No violation found

- Transform `pthread` statements
- Use a sequential framework

Yes

No
Our Proposed Methodology – Grammar

<table>
<thead>
<tr>
<th>Statement</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_t</td>
<td>ε</td>
</tr>
<tr>
<td>pthread_attr_t</td>
<td>ε</td>
</tr>
<tr>
<td>pthread_cond_attr_t</td>
<td>ε</td>
</tr>
<tr>
<td>pthread_create</td>
<td>ε</td>
</tr>
<tr>
<td>pthread_join</td>
<td>ε</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>ε</td>
</tr>
<tr>
<td>pthread_mutex_t</td>
<td>Integer variable is declared</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>1 is assigned to variable</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>0 is assigned to variable</td>
</tr>
<tr>
<td>pthread_cond_t</td>
<td>Integer variable is declared</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>1 is assigned to variable</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>0 is assigned to variable</td>
</tr>
</tbody>
</table>
Our Proposed Methodology – Rules

• Then, we have to use a framework that simulates multi-threaded programs execution

```c
#define NCS X
int cs[NCS] = {...};
int main(void *args) {
    for (int i = 0; i < NCS; i++) {
        switch (cs[i]) {
            ...  
        }
    }
    return 0;
}
```

```c
case 1: {
    case 11: { ... }
    case 12: { ... }
    ...
    case 20: { ... }
} break;

case 2: {
    case 21: { ... }
    case 22: { ... }
    ...
    case 30: { ... }
} break;
... 
Thread 0
```
Our Proposed Methodology – Rules

• Then, we have to use a framework that simulates multi-threaded programs execution

```c
#define NCS X
int cs[NCS] = {...};
int main(void *args) {
    for (int i = 0; i < NCS; i++) {
        switch (cs[i]) {
            ...  
            case 1: {
                case 11: { ... }
                case 12: { ... }
                ...  
                case 20: { ... }
            } break;

            case 2: {
                case 21: { ... }
                case 22: { ... }
                ...  
                case 30: { ... }
            } break;
            ...
        }
    }
    return 0;
}
```
Our Proposed Methodology – Rules

• Then, we have to use a framework that simulates multi-threaded programs execution

```c
#define NCS X
int cs[NCS] = {...};
int main(void *args) {
    for (int i = 0; i < NCS; i++) {
        switch (cs[i]) {
            ... 
        }
    }
    return 0;
}
```
Our Proposed Methodology – Rules

• Then, we have to use a framework that simulates multi-threaded programs execution

```c
#define NCS X
int cs[NCS] = {...};
int main(void *args) {
    for (int i = 0; i < NCS; i++) {
        switch (cs[i]) {
            ...
            case 20: { ... }
            break;

            case 2: {
                case 21: { ... }
                case 22: { ... }
                ...
                case 30: { ... }
            } break;
        } break;
    }
    return 0;
}
```

Context switch 10
Our Proposed Methodology – Rules

- Then, we have to use a framework that simulates multi-threaded programs execution

```c
#define NCS X
int cs[NCS] = {...};
int main(void *args) {
    for (int i = 0; i < NCS; i++) {
        switch (cs[i]) {
            ...
            case 1: {
                case 11: { ... }
                case 12: { ... }
                ...
                case 20: { ... }
            } break;
            case 2: {
                case 21: { ... }
                case 22: { ... }
                ...
                case 30: { ... }
            } break;
            ...
        }
    }
    return 0;
}
```
Our Proposed Methodology – Rules

• Then, we have to use a framework that simulates multi-threaded programs execution

```c
#define NCS X
int cs[NCS] = {...};
int main(void *args) {
    for (int i = 0; i < NCS; i++) {
        switch (cs[i]) {
            ...
        }
    }
    return 0;
}
```

Note: we can model up to 10 context switches for each thread
Running Example

```c
#include <pthread.h>
#include <assert.h>

pthread_mutex_t m;
int c = 0;

void *f1(void *arg) {
    pthread_mutex_lock(&m);
    c = c + 1;
    pthread_mutex_unlock(&m);
}

void *f2(void *arg) {
    pthread_mutex_lock(&m);
    c = c - 1;
    assert(c == 1);
    pthread_mutex_unlock(&m);
}

int main() {
    pthread_mutex_init(&m, NULL);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, f1, NULL);
    pthread_create(&t2, NULL, f2, NULL);
    return 0;
}
```
Running Example

```
esbmc --no-bounds-check --no-div-by-zero-check --no-pointer-check --deadlock-check --no-slice --boolector example.c
```

*** Thread interleavings 122 ***
Symex completed in: 0.001s
size of program expression: 110 assignments
Slicing time: 0.000s
Generated 0 VCC(s), 0 remaining after simplification
VERIFICATION SUCCESSFUL
BMC program time: 0.001s
Running Example

Multi-threaded software
Check for deadlocks
Check for other violations
Define transformation rules
Code transformation
Verifying using a BMC tool
Set of faults

```
esbmc --no-bounds-check --no-div-by-zero-check --no-pointer-check --no-slice --boolector example.c
```

Counterexample:
...
State 99 file example.c line 16 function f2 thread 2
c::f2 at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 67
----------------------------------------------------
Violated property:
  file example.c line 16 function f2
  assertion
  FALSE

VERIFICATION FAILED
Counterexample:
...
State 56 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 101
function pthread_create thread 0 c::pthread_create at example.c line 24 <main invocation>
--------------------------------------------------------------------------
State 75 file example.c line 11 function f1 thread 1 c::f1 at
/tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 67
--------------------------------------------------------------------------
State 99 file example.c line 16 function f2 thread 2 c::f2 at
/tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 67
--------------------------------------------------------------------------
Violated property:
file example.c line 16 function f2
assertion FALSE
VERIFICATION FAILED
Running Example

Counterexample:
...
State 56 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 101 function pthread_create thread 0 c::pthread_create at example.c line 24 <main invocation> .................................................................

State 75 file example.c line 11 function f1 thread 1 c::f1 at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 67 .................................................................

State 99 file example.c line 16 function f2 thread 2 c::f2 at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 67 .................................................................

Violated property:
file example.c line 16 function f2
assertion FALSE
VERIFICATION FAILED

cs[1]=21
Running Example

Counterexample:
...
State 56 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 101
function pthread_create thread 0 c::pthread_create at example.c line 24 <main invocation>
----------------------------------------------------
State 75 file example.c line 11 function f1 thread 1 c::f1 at /	mp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 67
----------------------------------------------------
State 99 file example.c line 16 function f2 thread 2 c::f2 at /	mp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c line 67
----------------------------------------------------
Violated property:
file example.c line 16 function f2
assertion FALSE
VERIFICATION FAILED

\[ cs[2]=31 \]
\[ NCS=3 \]
Running Example

```c
#include <pthread.h>
#include <assert.h>
#define NCS 3
int cs[NCS] = {11, 21, 31}, nondet(), diag, c = 0;

void f1_1() {
    int t = c + 1;
    c = (diag == 9 ? nondet() : t);
}

void f2_1() {
    int t = c - 1;
    c = (diag == 15 ? nondet() : t);
    __ESBMC_assume(c == 1);
}

int main() {
    int i; diag = nondet();
    for (i = 0; i < NCS; i++) {
        switch (cs[i]) {
            case 1: {
                case 11: {
                    if (cs[i] == 11) break;
                }
            }
            case 2: {
                case 21: {
                    f1_1();
                    if (cs[i] == 21) break;
                }
            }
            case 3: {
                case 31: {
                    f2_1();
                    if (cs[i] == 31) break;
                }
            }
        }
    }
    assert(0);
    return 0;
}
```
Running Example

Motivation

Background

Methodology

Results

Conclusions

Multi-threaded software

Check for deadlocks

Check for other violations

Define transformation rules

Code transformation

Verifying using a BMC tool

Set of faults

esbmc --no-bounds-check --no-div-by-zero-check --no-pointer-check --no-slice --boolector transformed_example.c

Counterexample:
...
State 9 file example-hawk.c line 5 thread 0
-----------------------------------------------
  diag=15 (15)
...
State 26 file example-hawk.c line 13 function f2_1 thread 0
c::f2_1 at example-hawk.c line 35
<main invocation>
-----------------------------------------------
  c::f2_1::$tmp::tmp$1=TRUE
...
State 27 thread 0
c::f2_1 at example-hawk.c line 35
<main invocation>
-----------------------------------------------
  c::f2_1::$tmp::tmp$1=TRUE
...
State 29 file example-hawk.c line 14 function f2_1 thread 0
c::f2_1 at example-hawk.c line 35
<main invocation>
-----------------------------------------------
  c=1 (1)
...
Running Example

Multi-threaded software → Check for deadlocks → Check for other violations → Define transformation rules → Code transformation → Verifying using a BMC tool → Set of faults

F = <Line=15, Correct Value=1>

esbmc --no-bounds-check --no-div-by-zero-check --no-pointer-check --no-slice --boollector repaired_example.c

*** Thread interleavings 197 ***
Symex completed in: 0.004s
size of program expression: 120 assignments
Slicing time: 0.000s
Generated 0 VCC(s), 0 remaining after simplification
VERIFICATION SUCCESSFUL
BMC program time: 0.004s

... (15) c = 1;
...
Experimental Objectives and Setup

- **Objectives**
  - Verify and validate our method using standard multi-threaded C software

- **Specs**
  - ESBMC v1.24.1 with SMT solver Boolector version 2.1.0
  - Core i7 4500 1.8 GHz
  - 8 GB of RAM
  - Fedora 21 64-bits

- **Benchmarks**
  - 11 benchmarks extracted from the Software Verification Competition, and the same used to evaluate ESBMC for multi-threaded C code
Experimental Results

- Timeouts in 27.27% of benchmarks
- No useful faulty lines in 18.18% of benchmarks
- Correct faulty lines in 54.55% of benchmarks
Experimental Results

• No useful faulty lines occur when ESBMC retrieves unreasonable diag in the counterexample for the translated software under verification
  – E.g. diag == 0, since there is no line 0 in the code, we cannot say anything about this fault

• Timeouts
  – If we run out of memory when first checking a benchmark using ESBMC, we denote that execution as a timeout
  – Thus, we cannot state it is a safe program, neither model it using our methodology
Experimental Results

• However, since we entirely rely on BMC tools to provide counterexamples to then translate the program under verification, timeouts are not due to our methodology

• This way, correct faulty lines are found in 6 out of 8 (75%) benchmarks
Experimental Results
Conclusions

• A novel method for localizing faults in multi-threaded C programs was proposed
  – It is based in BMC techniques and is also an extension to a sequential method to localize faults [Griesmeyer’07]
  – Useful for embedded systems

• Our methodology showed itself to be useful to assist in fault localization in standard multi-threaded software

Future Work

• Improve our code transformation
  – Use GOTO structure to model iterations
  – Model pthread statements more accurately

• Develop a tool to automate this process, such as an Eclipse plugin