BMCLua: Verification of Lua Programs in Digital TV Interactive Applications

Francisco A. Januario, **Lucas C. Cordeiro**, Vicente F. Lucena Jr., and Eddie B. de Lima Filho
lucascordeiro@ufam.edu.br

The Lua Language and its Applications

UFAM

- the Lua language is used in many areas, from games to digital TV applications
  - Adobe's Photoshop Lightroom
  - World of Warcraft e Angry Birds
  - Ginga Middleware (Digital TV)

extension language used in other programming languages

- C/C++
- JAVA
- NCL

interpreted, compact, and fast; it is used in **embedded devices**

- Mobile
- Set-Top Box

incorrect implicit conversion of variable types, returning null from functions with multiple values, and arithmetic overflow

Interactive TV applications are widely spread, but their verification becomes more difficult

UFAM

- functionality demands increased significantly in digital TV
  - **peer reviewing** and **testing**

```
local counter = 0
local dx, dy = canvas:attrSize()
function handler (evt)
    if evt.class=='ncl' then
        dy = dy + 1
        while dx ~= dy do
            counter = counter + 1
            canvas:drawText(10,10,'Progress: '..counter)
        end
    end
end
event.register(handler)
```

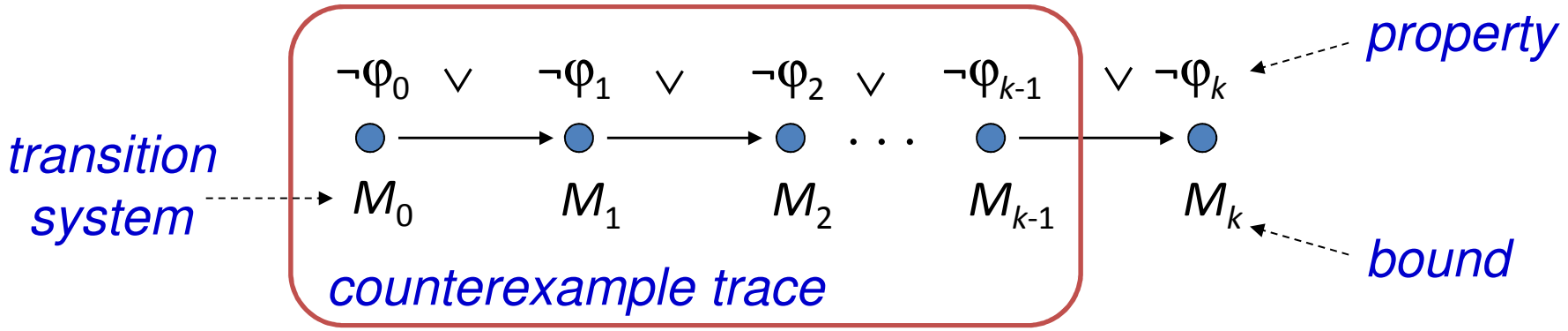
Arithmetic OverFlow

- negative impact on the performance of interactive TV applications (**presentation failures**)

Bounded Model Checking (BMC)

UFAM

Basic Idea: check negation of given property up to given depth



- transition system *M* unrolled *k* times
  - for programs: unroll loops, unfold arrays, …
- translated into verification condition ψ such that

  ψ satisfiable iff φ has counterexample of max. depth *k*
- has been applied successfully to verify (embedded) software since early 2000's

Objectives of this work

UFAM

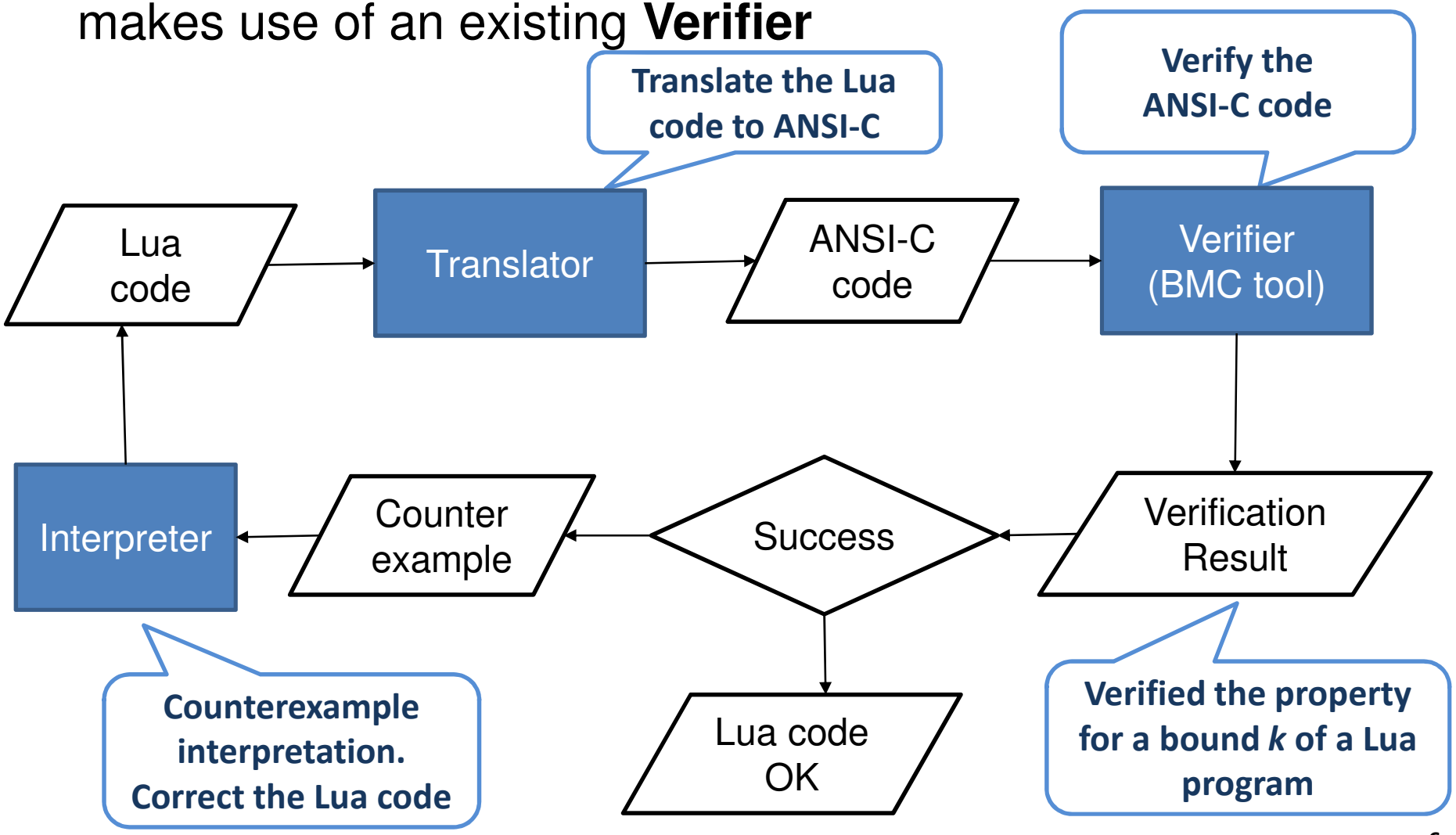
Apply BMC for Interactive TV software applications based on the Lua programming language

- develop a **verification platform** for Lua programs:
  - translate a Lua program to an intermediate representation
  - check for arithmetic overflow, division by zero, and user-specified assertions
  - interpret the counterexample
- exploit BMC tools to prune the **property and data dependent** search space and to exploit the **bit-accurate representation**
- implement this approach in BMCLua tool and evaluate it using Lua applications
  - consider time and correctness metrics

The BMCLua Verification Platform

UFAM

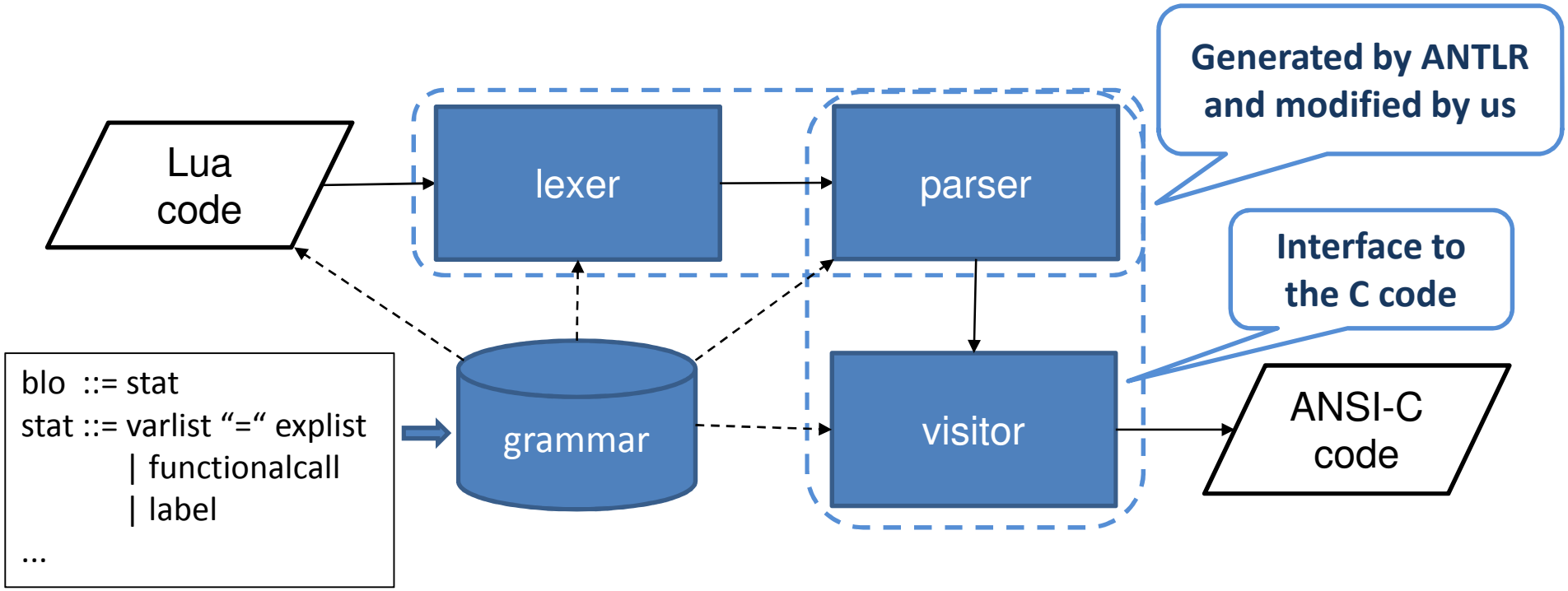
- BMCLua consists of a **Translator** and **Interpreter**, and makes use of an existing **Verifier**



The BMCLua Translator

UFAM

- the translator consists of the language **grammar**, the lexical analyzer (**lexer**), and the syntax analyzer (**parser**)

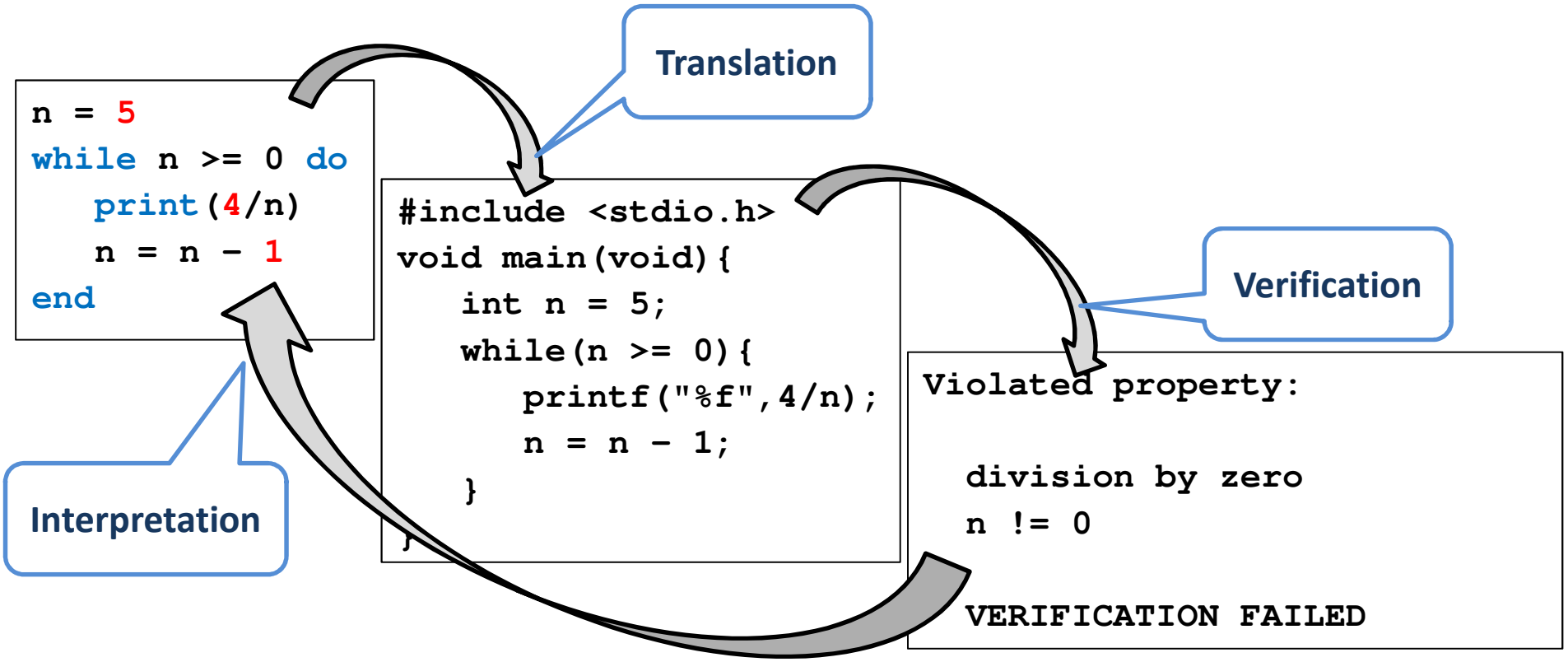


- the grammar consists of a set of **rules** describing the syntax
- the lexer generates **tokens** from a sequence of characters
- the parser checks the **syntax** of the input characters

Translation, Verification, and Interpretation

UFAM

- translates to an ANSI-C code (adds more code lines)
  - supports most primitive data types, relational and logical operators, decision and loops structures, and functions



- counterexample informs the code line and the violation

Experimental Evaluation

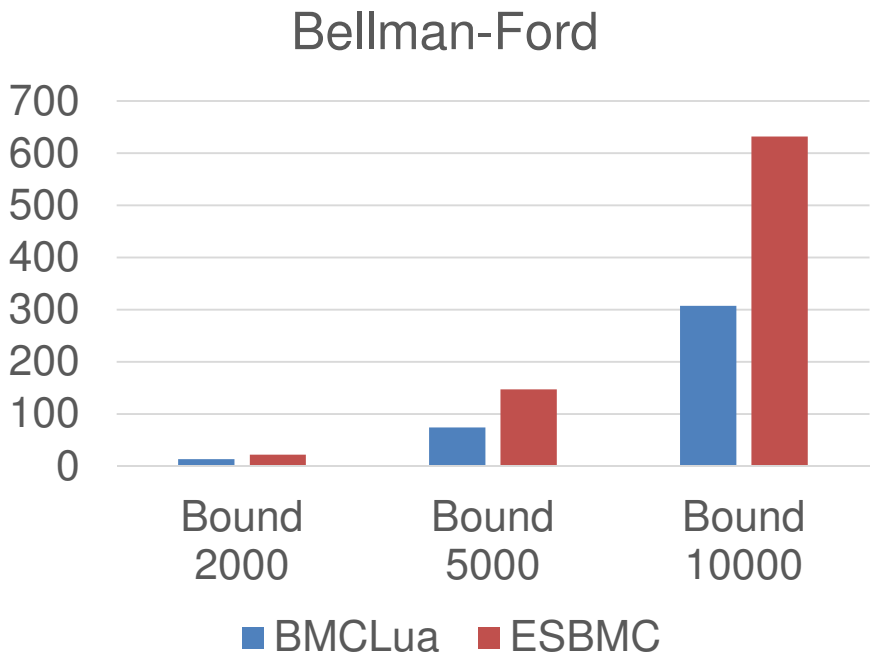
UFAM

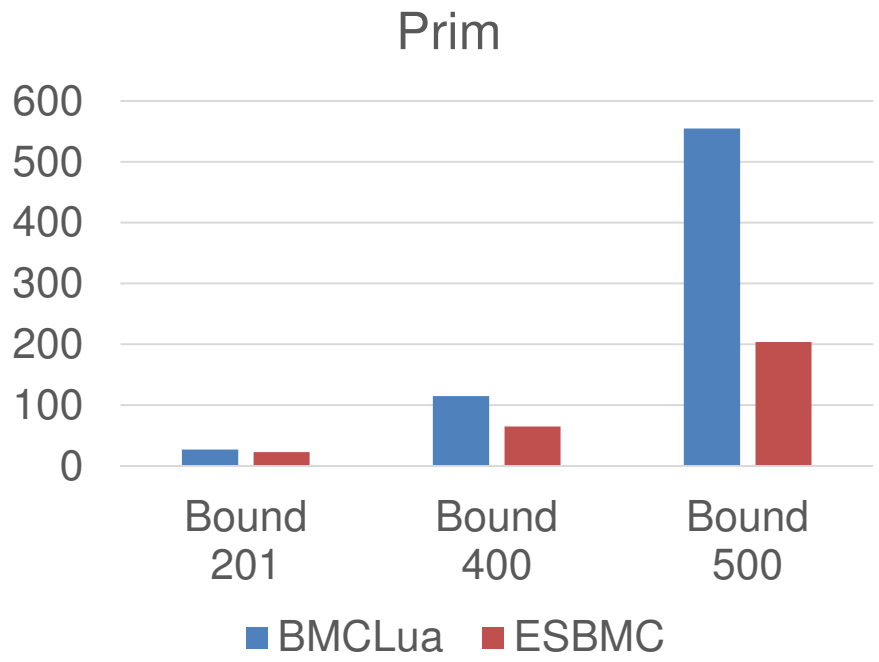
- Goal: evaluate the **performance** and **correctness** of BMCLua using standard benchmarks with a single user-specified property
  - compare to the verification time of ESBMC as a reference

- Experimental setup:
  - Intel Core i3 2.5 GHz with 2 GB of RAM running on Linux Ubuntu 32-bits
  - ESBMC v1.21 with SMT solver Z3 v3.2

Experimental Evaluation (Cont.)

- the **verification time** reported by BMCLua and ESBMC are **comparable** to each other for **smaller bounds**
  - the **translation time** is typically less than **one second**



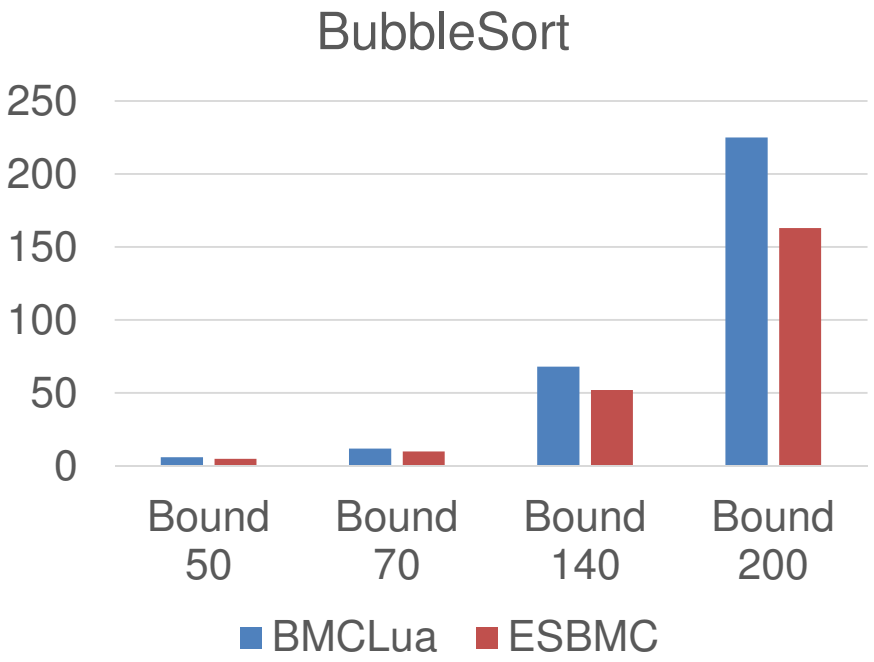


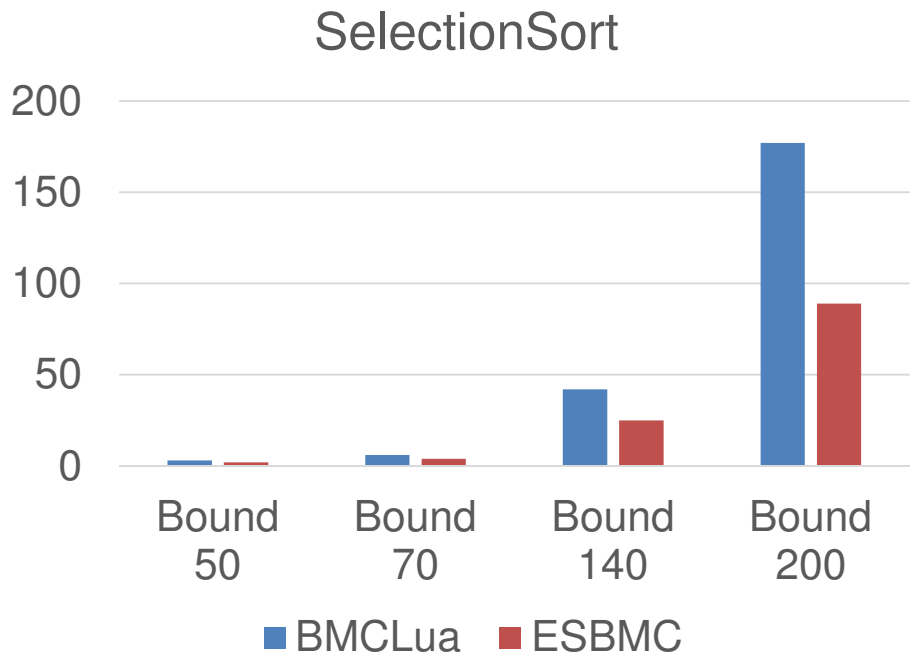
no false-positive / false-negative

Experimental Evaluation (Cont.)

UFAM

- the BMCLua **verification time** is **higher** due to the increase of **code lines** when translating into ANSI-C code
  - common subexpression elimination and constant propagation





no false-positive / false-negative

Conclusions

- proposed **first application** of BMC to Lua programs
- BMCLua checks for **arithmetic overflow**, **division by zero** and **user-specified assertions**
- the **verification time** of BMCLua is **comparable** to ESBMC
  - only 21% of the benchmarks present higher verification time
- BMCLua did not report **false-positive** or **false-negative**

Future Work

- support the remaining **Lua constructs** (typecasts, functional call, NCLua, and Lua library)
- convert Lua programs to **SMT formulas**
- integrate BMCLua into the **Eclipse** and **Ginga middleware**