Dependence Guided Model Checking

Sabrina Sayasith[1], Callie Griffin[2], Rodion Podorozhny[3]

1. Smith College

2 Appalachian State University

3. Texas State University

Problem and Background Information

•**Verification** is the process of ensuring that a system will function as intended under all circumstances.

•This practice is especially difficult for **cyber-physical systems(CPS).** CPS have an extra layer of physical constraints and environmental factors to account for.

•**Static** verification checks all possible execution paths of a program. While thorough, is immensely time consuming, and it cannot realistically be used on larger systems.

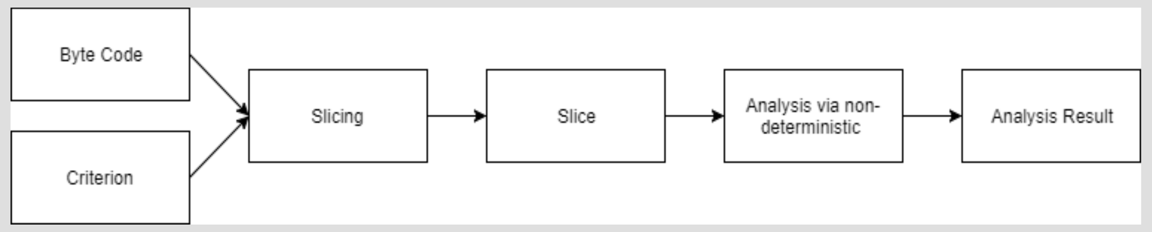
•**Dynamic** verification allows for testing under real and simulated conditions, but it cannot account for all possible scenarios.

Approach

•**Current monitoring systems** that operate while the CPS functions only carry out testing. They cannot give high assurance results for properties that require analysis of all possible paths and interleavings such as lack of deadlocks, race conditions and problem domain specific safety properties.

•We designed a monitoring system that combines **slicing with non deterministic execution** in one pass of the control flow graph. We use a slicing algorithm to gather information about the data and control dependencies in the system. While the CPS carries out dynamic testing, our system performs bounded non deterministic traversal of the possible execution paths in manageable slices of code.

Fig 1. Overview of Our Approach



Implementation

•We use the **Apache Commons BCEL** library to produce control flow graphs directly from byte code.

•Our implementation of **Weiser's slicing algorithm** on the control flow graph limits the number of paths we need to traverse. Weiser's algorithm chooses the paths based on a stopping point and variables of interest (criterion).

Fig. 2: (a) Is an example code that computes a sum and product. (b) is the example code after the statements related to calculating the sum have been sliced away according to Weiser's Algorithm

(a)

```
(1)   read(n);
(2)   i := 1;
(3)   sum := 0;
(4)   product := 1;
(5)   while i <= n do
      begin
(6)     sum := sum + i;
(7)     product := product * i;
(8)     i := i + 1
      end;
(9)   write(sum);
(10)  write(product)
```

(b)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

Evaluation

•We created **three control flow graphs** created based on three simple programs: one with no branch statements, one with an if statement, and one with a loop.

•We measure the **time taken** for non deterministic execution of depth first search for each control flow graph before and after slicing.

•We calculated the **ratio of nodes or statements** that remain once sliced

Fig 3.

Timing Data in Nanoseconds of Depth-First Traversal of Control Flow Graph

Type of Program	Before Slicing	After Slicing	% Change in Total Time
Simple Sequential Program	2443039	132287	94.6%
Program with If statement	1442616	233214	83.8%
Program with a while loop	1560648	228652	85.3%

Fig 4.

Size Reduction of Code due to Slicing

Type of Program	# of Statements in Original	# of Statements in Sliced Code	Size of Slice Compared to Original Code
Sequential	9	3	33%
With If Statements	10	7	70%
With Loop	10	7	70%

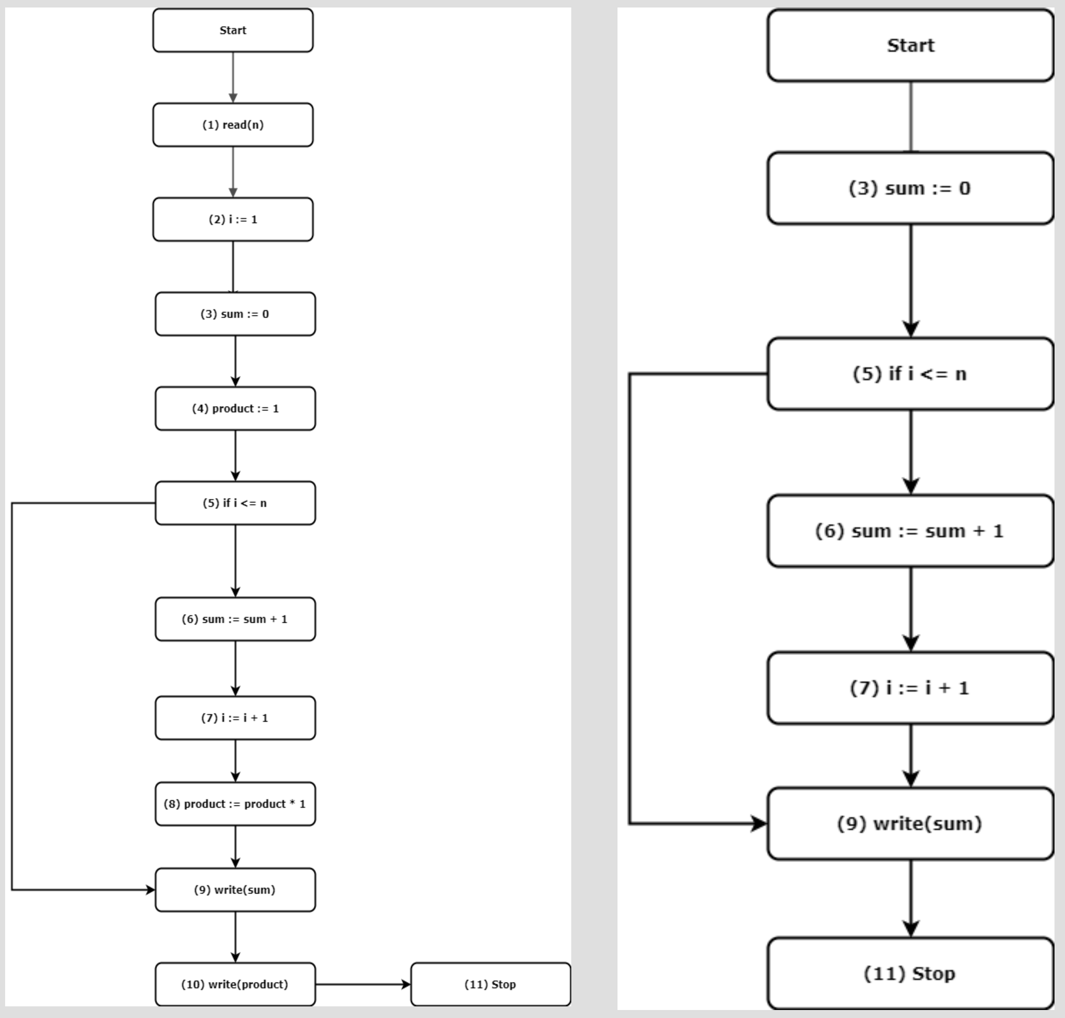
Future Work

- Implement a forward slicing algorithm in addition to Weiser's slicing algorithm for more dependency sets and information to be used in the depth-first search traversal.
- Combine the slicing and verification algorithms into one forward pass of the control flow graph.
- Get the algorithm working on realistic sized systems

Conclusions

- Dependency information evaluated for each statement can be used to alter the traversal of the control flow graph and decrease running time of the verification process.
- It is unviable to merge a backwards slicing algorithm with forward non-deterministic execution to achieve one single pass of the control flow graph.
- It is clear that slicing decreases the time for nondeterministic execution

Fig 5. Left pictures a control flow graph for a program that computes a sum and product with an if statement. Right pictures the control flow graph of the program on the left after it has been sliced to only include statements related to the computation of the sum.



Citations

•"Formal Verification of Cyber-Physical Automation Systems Modelled with Timed Block Diagrams." 2016 IEEE 25th International Symposium on Industrial Electronics (ISIE), Industrial Electronics (ISIE), 2016 IEEE 25th International Symposium on (2016): 316. Print.

•Korel, Bogdan, et al. "Forward Computation of Dynamic Program Slices." Master Essay, Department of Computer Science Wayne State University, 1994. 66-79. Print.

•Larsen, Loren, et al. "Slicing Object-Oriented Software." Proceedings of ICSE-18. Department of Computer Science Clemson University. 495-505. Print.

•Matsubara, Masahiro, et al. "Model Checking with Program Slicing Based on Variable Dependence Graphs." Electronic Proceedings in Theoretical Computer Science, Vol 105, Iss Proc.FTSCS 2012, Pp 56-68 (2012) (2012): 56. Print.

•Su, T. (. 1.)., et al. Combining Symbolic Execution and Model Checking for Data Flow Testing. 1 Vol. IEEE Computer Society, 2015. Print.

•TIP, F. A Survey of Program Slicing Techniques. 3 Vol. , 1995. Print.

Acknowledgements

•We thank the National Science Foundation for funding the research under the Research Experiences for Undergraduates site programs

•(CNS-1358939, CNS-1659807) at Texas State University to perform this piece of work and the infrastructure provided by a NSF-CRI 1305302 award.