Spring 2020
DD2372 Automata and Languages
Lab Assignment 2:
Model Checking Flow Graphs

H. Nemati O. Schwarz D. Gurov X. Yao

Published: 27 April 2020

1 Introduction

In automatic software verification such as model checking it is often required to check the control flow of a program against a given specification, that is, the allowed/desired behaviour (e.g., the initialization routine should always be called before the actual calculation; locks should be respected correctly). One way to perform such checks is with the help of two automata. One of them - a DFA - represents the specification of a safety property. Its language consists of the allowed terminating sequences of method calls (or *traces*). The other automaton is a PDA that is derived from the flow graph that represents the behaviour (i.e., possible traces) of the program to verify. Program correctness means that the set of possible words (read: traces) of the program is a subset of the traces from the specification language. This, however, can be reduced to testing emptiness of the product language of the PDA (or the corresponding CFG) and the complement of the specification DFA. At least for the terminating behaviours of the program, a positive result in this check assures compliance with the specification.

Note that the properties that can be checked with this method are of limited kind. Since only the control flow is analyzed, properties such as mathematical correctness of the program output are not covered. Furthermore, a DFA can not express properties such as "the pop function should not be called more often than the push function." Finally, non-terminating behaviour can not be checked this way. However, still many useful properties are automatically checkable with this method. Figure 1 provides an overview of the approach. The objective of this laboratory assignment is to design a tool that implements this approach and checks flow graphs for compliance with a given specification DFA. As a simplification, we skip the explicit translation of the flow graph into a context free grammar, and instead compute its product with the inverted DFA directly.

In more detail, the tool should proceed as follows:

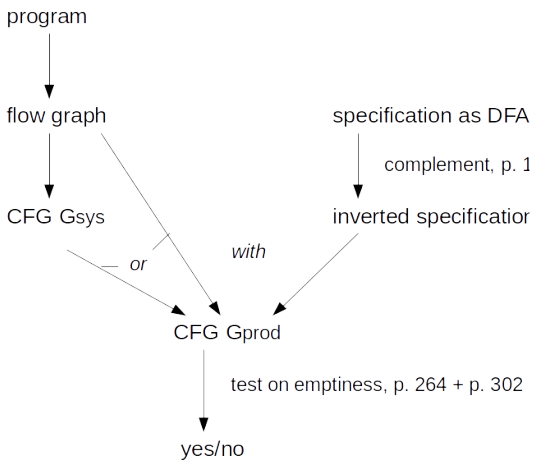


Figure 1: Overview of approach

1. Process an input file describing the flow graph FG, and store it in some suitable data structure.
2. Process an input file with the specification DFA, and store it in some suitable data structure.
3. Compute the complement of the DFA (see page 135 of the course book).
4. Compute the product of FG and the complement DFA, resulting in a context-free grammar Gprod.
5. Test Gprod for language emptiness. On fail, output a trace violating the specification.

You are free to choose the programming language in which to implement the tool. The assignment is carried out in teams of at most two persons.

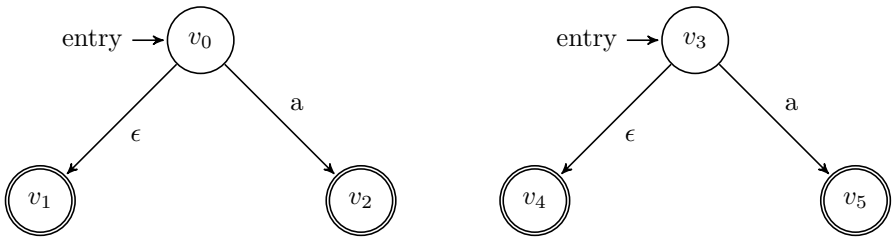
2 Implementation

2.1 Inputs to the Tool

We start by describing the two inputs to the tool, flow graphs and DFAs.

2.1.1 Flow Graphs

We use a (Control) Flow Graph (FG) to represent the input program. A flow graph consists of several disjoint graphs, one for each involved method of the program. *Nodes* in a graph represent straight-line blocks of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Each graph has an *entry node* and one or more *return nodes*. Moreover, the possible flows of the program are shown by *edges* connecting the nodes. These are either *transfer edges* labelled with ϵ indicating a silent action, or *call edges* labelled by the name of a program method, representing a function call to that method.



(a) method main (b) method a

Figure 2: Example for an Input Flow Graph; entry nodes are those without predecessor, return nodes are depicted by double circles

Figure 2 shows an example flow graph. The `main` function performs some check. Dependent on the outcome, it either ends execution or calls function `a`. Similarly, function `a` simply ends or calls itself recursively. In a concrete syntax, this flow graph can be encoded as follows:

```
node v0 meth(main) entry
node v1 meth(main) ret
node v2 meth(main) ret
node v3 meth(a) entry
node v4 meth(a) ret
node v5 meth(a) ret
edge v0 v1 eps
edge v0 v2 a
edge v3 v4 eps
edge v3 v5 a
```

Lines starting with “node” represent nodes in the graph, while lines starting with “edge” indicate connections between the nodes. Node lines also contain

a unique node identifier, the method the node belongs to, and the node type. Edge lines contain the identifier of the node they connect to, and the method name (for call edges) or "eps" (for transfer edges).

2.1.2 DFAs

We use a DFA-based specification to describe the desired behaviour of the system. The alphabet of the DFA is comprised of the method names. Therefore, the DFA reasons about sequences of methods calls. Accepting states relate to accepted behaviour (i.e., accepted method call sequences).

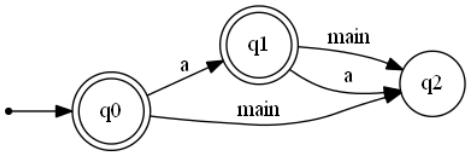


Figure 3: Example for a Specification DFA; edges from q_2 to itself are implicit

The DFA in Figure 3 specifies the safety property "at most one call is allowed, namely to method `a`", a property which is violated by the flow graph from Figure 2.

In a concrete syntax, this DFA can be encoded as:

```
=>(q0)-eps->(q0)
(q0)-a->(q1)
(q0)-main->[q2]
(q1)-main->[q2]
(q1)-a->[q2]
[q2]-a->[q2]
[q2]-main->[q2]
[q2]-eps->[q2]
(q1)-eps->(q1)
```

Each line in the specification represents one single labelled transition. An initial state is pointed to by $\Rightarrow$, and accepting states are surrounded by parentheses ().

The behaviour of the flow graph on Figure 2 violates the above safety property. The execution in which `main` calls method `a`, and then `a` calls itself, followed by returns, is a counterexample.

2.2 Product Construction

The $\mathsf{FG} \times \mathsf{DFA}$ product construction described here is due to Dilian Gurov, and has been developed specifically for this assignment. It combines the effect of the $\mathsf{FG} \to \mathsf{CFG}$ translation described in class with a $\mathsf{CFG} \times \mathsf{DFA}$ product construction yielding a CFG. The latter construction combines ideas from the $\mathsf{CFG} \to \mathsf{PDA}$

translation from page 244 of the course book, the $\mathsf{PDA} \times \mathsf{DFA}$ product construction described in the proof of Theorem 7.27 on page 292, and the $\mathsf{PDA} \to \mathsf{CFG}$ translation described in the proof of Theorem 6.14 on page 248. Please make sure that you have read and understood well all these constructions before continuing with the laboratory assignment.

Let F be a flow graph with nodes V, method names Σ (including a dedicated `main` symbol), entry nodes E, and return nodes R. Let furthermore $D = (Q, \Sigma, \delta, q_0, Q_F)$ be a DFA. Then the product CFG can be defined as $G_{prod} = (V_{G_{prod}}, \Sigma, P, S)$, where:

- $V_{G_{prod}} = \{S\} \cup \{[q_i\ X\ q_j] \mid X \in V \cup \Sigma \text{ and } q_i, q_j \in Q\}$
- P is a set of productions defined as follows:
  1. For every *final state* $q_i \in Q_F$, a production $S \to [q_0\ v_0\ q_i]$, where v_0 is the entry node of the main method.
  2. For every *transfer edge* $v_i \xrightarrow{\epsilon} v_j$ of F and every state sequence $q_a q_b \in Q^2$, a production $[q_a\ v_i\ q_b] \to [q_a\ v_j\ q_b]$.
  3. For every *call edge* $v_i \xrightarrow{m} v_j$ and every state sequence $q_a q_b q_c q_d \in Q^4$, a production $[q_a\ v_i\ q_d] \to [q_a\ m\ q_b][q_b\ v_k\ q_c][q_c\ v_j\ q_d]$, where v_k is the entry node of method m.
  4. For every *return node* $v_i \in R$ and every state $q_j \in Q$, a production $[q_j\ v_i\ q_j] \to \epsilon$.
  5. For every *transition* $\delta(q_i, a) = q_j$ of D, a production $[q_i\ a\ q_j] \to a$.

Note that the term "state sequence" used above does *not* refer to a path in the DFA, but to an arbitrary combination of states, possibly with repetitions.

Let us see an example. For the product of the flow graph in Figure 2 and the complement of the DFA in Figure 3, production rules would include:

1. the production for the final state q_2 of the complemented DFA, namely $S \to [q_0\ v_0\ q_2]$;
2. the productions for transfer edges, such as $[q_2\ v_3\ q_2] \to [q_2\ v_4\ q_2]$ for the transfer edge $v_3 \xrightarrow{\epsilon} v_4$ and the state sequence $q_2 q_2$;
3. the productions for call edges, such as $[q_0\ v_0\ q_2] \to [q_0\ a\ q_1][q_1\ v_3\ q_2][q_2\ v_2\ q_2]$ for the call edge $v_0 \xrightarrow{a} v_2$ and the state sequence $q_0 q_1 q_2 q_2$, and $[q_1\ v_3\ q_2] \to [q_1\ a\ q_2][q_2\ v_3\ q_2][q_2\ v_5\ q_2]$ for the call edge $v_3 \xrightarrow{a} v_5$ and the state sequence $q_1 q_2 q_2 q_2$;
4. the productions for return nodes, such as $[q_2\ v_2\ q_2] \to \epsilon$ for the return node v_2 and state q_2, $[q_2\ v_4\ q_2] \to \epsilon$ for the return node v_4 and state q_2, and $[q_2\ v_5\ q_2] \to \epsilon$ for the return node v_5 and state q_2; and
5. the productions for DFA transitions, such as $[q_0\ a\ q_1] \to a$ for the transition $\delta(q_0, a) = q_1$, and $[q_1\ a\ q_2] \to a$ for the transition $\delta(q_1, a) = q_2$.

For this example, here is a *left-most derivation* of the resulting CFG, using the productions shown above. The derivation models a co-execution of the program (i.e., the flow graph) with a monitor for the safety property (i.e., the complemented DFA), consisting of a call of method `a` by `main`, followed by a self-call of `a`:

$S \Rightarrow [q_0\ v_0\ q_2] \Rightarrow [q_0\ a\ q_1][q_1\ v_3\ q_2][q_2\ v_2\ q_2] \Rightarrow a[q_1\ v_3\ q_2][q_2\ v_2\ q_2] \Rightarrow a[q_1\ a\ q_2][q_2\ v_3\ q_2][q_2\ v_5\ q_2][q_2\ v_2\ q_2] \Rightarrow aa[q_2\ v_3\ q_2][q_2\ v_5\ q_2][q_2\ v_2\ q_2] \Rightarrow aa[q_2\ v_4\ q_2][q_2\ v_5\ q_2][q_2\ v_2\ q_2] \Rightarrow aa[q_2\ v_5\ q_2][q_2\ v_2\ q_2] \Rightarrow aa[q_2\ v_2\ q_2] \Rightarrow aa$

2.3 Testing Emptiness and Finding Counter-Examples

Given the product of the flow graph and the complemented specification automaton as a CFG, we test the grammar for language emptiness in the standard way, as described on pages 264 and 302 of the course book. If the language is empty, the program is correct; otherwise, the starting symbol is generating and allows the construction of a *counter-example*. To this end, consider a left-most derivation from the starting symbol. By removing q_i and q_j from all triples $[q_i X q_j]$, we obtain a derivation of an execution, based on flow graph nodes and method names. The desired counter-example is the final sentence of the derivation, and thus contains only method names (and possibly ϵ's).

3 Resources

Some test cases are available at: `https://gits-15.sys.kth.se/dd2372/lab2`. Each test case directory contains the flow graph (`xxx.cfg`), one or more specifications (`xxx.spec`), and a readme file.

Some examples include calls to *external library methods*, not part of the program itself. These can be treated atomically, similarly to transfer edges. Also, in the flow graphs (extracted by a tool like ConFlEx [1]) there may be information about *exceptional control flow*. You can choose to ignore this information.

4 Tasks

The present lab assignment consists of the following tasks:

1. Implement the tool as described above. Include the functionality of printing out the control flow graph and the specification DFA (graphically, or in table form); this will be useful for debugging and demonstrating the tool.

2. Construct and run a few small as well as larger test cases of your own.

[1] ConFlEx is a tool for extracting flow graphs from Java bytecode, developed by Pedro de Carvalho Gomes, available from `http://www.csc.kth.se/~pedrodcg/conflex/`.

3. Write a succinct report describing your solution, your experiments and your general observations from these. For instance, elaborate on the following questions:

   (a) Which data structures have you chosen for representing the control flow graph, the specification DFA and their product?

   (b) How do you handle calls to functions for which you do not have their definitions (control flow graph)?

   (c) What does it mean for the acceptance/rejection of a flow graph when the input DFA leaves some state transitions underspecified, i.e., does not specify the successor state for a certain pre-state and method invocation?

   (d) How do you implement the search for the counterexample?

   (e) If you decide to print the ϵ's in the counterexample, what information does that give you?

   (f) Is performance affected significantly by the sizes of the inputs?

   (g) How could you optimize your program?

5 Tips and Hints

1. When constructing test specifications, keep in mind that your DFAs need to be complete in the sense that each state has outgoing transitions for each known function name. In many cases this is resolved by introducing self-loops from the state to itself. You can choose to add those self-loops automatically.

2. For Java: a convenient way to read a file line by line in Java is through the standard input, using the following lines of code:

```
Scanner s = new Scanner(System.in);
while (s.hasNextLine())
  process(s.nextLine());
```

3. Graphical representations of the DFA and the flow graph may come in handy when debugging your tool. A graphical representation can be generated with the `dot` command, which is available on the lab computers through `module add csw/1.1`. The example DFA from above can be generated by executing `dot -o graph.png graph.gv`, assuming `graph.gv` contains the following code:

```
digraph finite_state_machine {
  rankdir=LR;
  node [shape = doublecircle]; q1; q0;
  node [shape = point]; point_q0;
  node [shape = circle];
```

```
  point_q0 -> q0
  q1 -> q2 [ label = "main"];
  q1 -> q2 [ label = "a"];
  q0 -> q2 [ label = "main"];
  q0 -> q1 [ label = "a"];
}
```