IA159 Formal Verification Methods

Lecture Notes

Version: April 18, 2010

Syllabus

- Models of systems
- Formal specification of program properties (modal and temporal logics)
- Automatic verification - reachability analysis, symbolic and explicit model checking, equivalence checking
- Deductive verification methods (theorem proving)
- Software testing
- Program analysis, abstraction, abstract interpretation
- Counter-example guided abstraction refinement
- Combining formal methods, SW tools BLAST, Spec# etc.

---

1 Basic information about the course

1.1 What does "Formal Verification Methods" mean?

formal methods are a collection of notations and techniques for describing and analyzing systems. Methods are **formal** in the sense that they are based on some mathematical theories, such as logic, automata or graph theory. [Pel01]

verification is the process of applying a manual or an automatic technique that is supposed to establish whether the code either satisfies a given property or behaves in accordance with some higher-level description of it. [Pel01]

formal verificatin methods are techniques (usually based on mathematical theories) for analysing systems with the aim to improve their quality and reliability.

The course is focused on theoretical and algorithmic bases of verification methods. The software engineering aspects connected to verification methods are beyond the scope of this course.

1.2 Literature

There is no single reading material covering all the topics mentioned in this course. However, many these topics are covered by the books [Pel01] and [CGP99]. Recommended reading material for the other topics are some recent papers that will be referred in this text.

1.3 Connections to other courses

In the course, we assume that students are familiar with the content of the following courses (in particular with the mentioned notions).

- *IB005 Formal Languages and Automata I (aka FJA I)* - pushdown automata
- *IA006 Selected topics on automata theory (aka FJA II)* - infinite words, Büchi automata, bisimulation equivalence
- *IA040 Modal and Temporal Logics for Processes* - temporal logics, mainly LTL

- *IV113 Introduction to Validation and Verification* - automata based LTL model checking

Courses that are also relevant to our topic:

- *MA015 Graph Algorithms*
- *IV010 Communication and Parallelism*
- *IB002 Design of Algorithms I*
- *IV022 Design and Verification of Algorithms*
- *PA008 Compiler Construction*

Courses following (is some sense) our course:

- *IV115 Parallel and Distributed Laboratory Seminar*
- *IV074 Laboratory for Parallel and Distributed Systems*
- *IA072 Seminar on Concurrency*

1.4 Examination

There will be an oral exam at the end (no intrasemestral tests, no written exams, no homeworks).

2 Introduction

Verification methods can be loosely divided into the following basic cathegories:

- testing
  - simple, feasible, very good cost/performance ratio
  - very effective in early stages of debugging process
  - applicable directly to real systems
  - cannot guarantee that there are no errors
  - in practice: standard technique for enhancing the quality of systems, wide tool support
- deductive verification (with use of theorem provers)
  - applicable to models of real systems
  - needs a huge effort of an expert on both deductive verification and systems under verification
  - can guarantee that (a model of) a real system satisfies a given property
  - in practice: used rarely (e.g. partial correctness of FPU in AMD processors)
- equivalence checking
  - applicable to models of real systems
  - needs a detailed formal specification of a system under verification
  - there are no algorithms for reasonable equivalences and infinite-state systems
  - in practice: some specific applications (e.g. equivalence of different levels of hardware design)
- reachability and model checking
  - applicable to (usually finite-state) models of real systems

  - needs formal specification of a system under verification
  - fully automatic, but feasible only for relatively small finite-state systems
  - in practice: a standard technique for verification of simple hardware designs, used also for verification of small systems (e.g. communication protocols)
- static analysis and abstract interpretation
  - applicable directly to source code of real systems, feasible
  - can verify only a specific class of properties (including many interesting properties)
  - may produce false alarms (the number of false alarms grows with the ability to find real bugs)
  - automatic (verification of some properties may require provision of a suitable abstraction)
  - in practice: some static analysis is performed by almost every compiler, there are very efficient tools (e.g. *Coverity*, *Stanse*) able to work with big pieces of real software, for example a linux kernel
- combined methods
  - e.g. abstraction + model checking, model checking + counter-example guided abstraction refinement (CEGAR), abstract interpretation + counter-example guided abstraction refinement (CEGAR), testing + model checking, testing + symbolic execution (a case of abstract interpretation)
  - the aim is to develop methods which are applicable directly to (source code of) real systems and (more or less) automatic
  - may be incomplete and/or produce false alarms
  - in practice: already has some specific applications, e.g. verification of Windows drivers (by *Static Driver Verifier* [SDV]
  - definitely the most promising approach

3 Software testing

This section is based on Chapter 9 of [Pel01] and [MBT]. We mention only some *formal* parts of software testing area.

3.1 Basic terminology

There are two basic approaches to software testing:

white box testing (aka **transparent box testing**)- based on inspecting the source code of the system under test

black box testing - does not use the source code (which may be inaccessible or unknown)

Software testing methods can be also divided according to the level of the tested parts of the system

unit (module) testing - the lowest level of testing, where one tests small pieces of code separately

integration testing - testing that different pieces of code work well together

system testing - testing the system as a whole

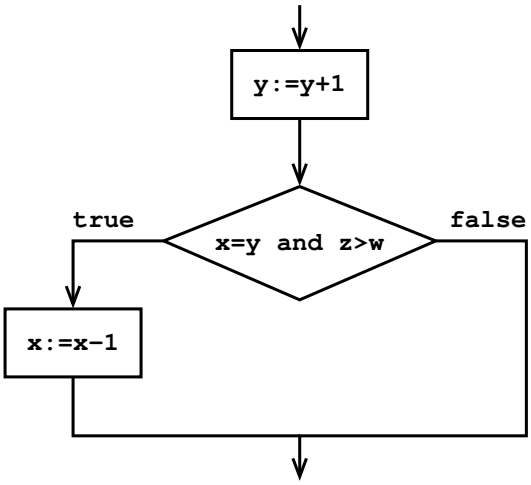


Figure 1: A simple flowchart.

White box testing is suitable for unit testing and integration testing, while black box testing is more appropriate for system testing.

We use the following terminology:

execution path is a path in the flowchart of the tested code, i.e., it is a sequence of control points and instructions appearing in the tested code

test case is a sequence of inputs, actions, and events accompanied with expected response of the system

test suite is a set of test cases

3.2 White box testing

A program typically has a very large, or even unbounded, number of execution paths. Hence, it is not feasible to examine all of them. Roughly speaking, we would like to have a reasonably small test suite which provides a high degree of probability of finding potential errors. Various *code coverage criteria* have been defined as a metrices saying whether (or to what extent) a given test suit covers a given code (the higher code coverage, the higher probability of finding potential error). The aim is to find the smallest test suit with the highest coverage.

3.2.1 Control flow coverage criteria

We will present the main coverage criteria and the differences between them using a small example described in Figure 1. For the sake of readability, the test cases for our example are described using the state of the variables just prior to the decision predicate $x = y \ \wedge \ z > w$.

statement coverage: *Each executable statement of the program (e.g. assignments, input, test, output) appears in at least one test case.*

One test case is enough to cover the flowchart according to statement coverage:

$$(x = 2, y = 2, z = 4, w = 3) \tag{1}$$

edge coverage: *Each execution edge of the flowchart appears in some test case.*

To cover the flowchart, we need to add to test case (1) another test case:

$$(x = 3, y = 3, z = 5, w = 7) \tag{2}$$

condition coverage: *Each decision predicate is a (possibly trivial) Boolean combination of element conditions (e.g. comparison between two expressions, application of a relation to some program variables). Each of these element conditions appears in some test case where it is calculated to TRUE and in another test case where it is calculated to FALSE, provided that it can have these truth values.*

Test case (2) together with the test case

$$(x = 3, y = 4, z = 7, w = 5) \tag{3}$$

cover the code. However, in both cases, the decision predicate is evaluated to FALSE...

edge/condition coverage: *This coverage criterion requires the executable edges as well as the conditions to be covered.*

Test cases (1), (2), and (3) together cover the code.

multiple condition coverage: *This is similar to condition coverage, but instead of taking care of each trivial condition, we require each Boolean combination that may appear in any decision predicate during some execution of the program must appear in some test case.*

To cover the code, we need to add one more test case to the three present test cases.

$$(x = 3, y = 4, z = 5, w = 6) \tag{4}$$

The main disadvantage of the multiple condition coverage is that it involves an explosion of the number of test cases.

path coverage: *This criterion requires that every executable path be covered by a test case.*

The number of paths for a given piece of code can be enormous. For example, loops may result in infinite or an unfeasible number of paths.

We say that one criterion *subsumes* another, if guaranteeing the former coverage also guarantees the latter. This relation appears in Figure 2, where a coverage criterion that subsumes another appears above it, with an arrow from the subsuming criterion to the subsumed one. Please note that it can happen due to a lucky selection of the test cases, a less comprehensive coverage will find errors that a more comprehensive approach will happen to miss.

The above criteria (except of path coverage) do not treat loops adequately: they do not care about number of iterations. There are several ad hoc strategies for testing loops

loop coverage:

- *Check the case where the loop is skipped.*
- *Check the case where the loop is executed once.*
- *Check the case where the loop is executed some typical number of times. (But what is typical?)*
- *If the bound n on the number of iterations of the loop is known, try executing it $n - 1$, n, and $n + 1$ times.*

Testing loops become even more difficult when nested loops are involved.

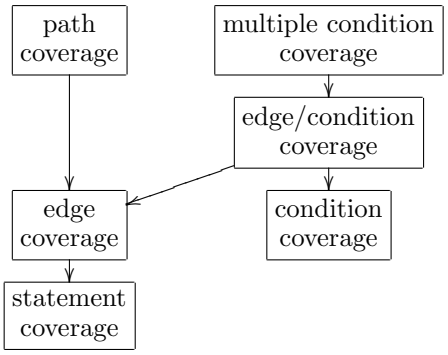


Figure 2: A hierarchy of control flow coverage criteria.

3.2.2 Dataflow coverage criteria

Using control flow coverage criteria, one may fail to include some execution path in which some variable is set to some value for a particular purpose, but later that value is misused. Hence, dataflow coverage criteria have been introduced [RW85].

Definition of dataflow coverage criteria employs the following sets of nodes. By nodes we mean nodes in the flowchart (corresponding to statements and conditions of the program). In the following, x ranges over program variables.

$def(x)$ = nodes where some value is assigned to x

$p{-}use(x)$ = nodes where x is used in a predicate (e.g. in *if* or *while* statements)

$c{-}use(x)$ = nodes where x is used in some expression other than a predicate

For each $s \in def(x)$ we define the following sets:

$dpu(s,x)$ = nodes $s' \in p{-}use(x)$ such that there is a path from s to s' going only through nodes not included in $def(x)$

$dcu(s,x)$ = nodes $s' \in c{-}use(x)$ such that there is a path from s to s' going only through nodes not included in $def(x)$

Each dataflow coverage criterion defines the paths that should be included in a test suite. For each program variable x and each node $s \in def(x)$, one needs to include at least the following paths starting in s and going only through nodes not included in $def(x)$, as subpaths in the test suite:

all-defs: include one path to some node in $dpu(s,x)$ or in $dcu(s,x)$.

all-p-uses: include one path to each node in $dpu(s,x)$.

all-c-uses/some-p-uses: include one path to each node in $dcu(s,x)$, but if $dcu(s,x)$ is empty, include at least one path to some node in $dpu(s,x)$.

all-p-uses/some-c-uses: include one path to each node in $dpu(s,x)$, but if $dpu(s,x)$ is empty, include at least one path to some node in $dcu(s,x)$.

all uses: include one path to each node in $dpu(s,x)$ and to each node in $dcu(s,x)$.

all-du-paths: include all the paths to each node in $dpu(s,x)$ and to each node in $dcu(s,x)$.

These paths should not contain cycles except for the first and the last nodes, which may be the same (for example, an assignment $x := x + 1$ is both in $def(x)$ and $c{-}use(x)$). The hierarchy of the dataflow coverage criteria appears in Figure 3.

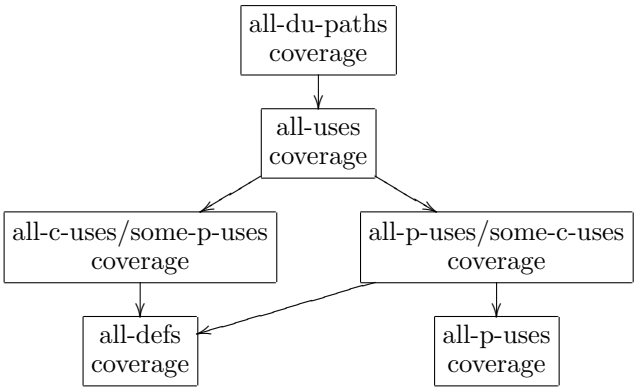


Figure 3: A hierarchy of dataflow coverage criteria.

3.2.3 Notes

In general, we cannot expect that our test suite achieves the full coverage (for example, some instructions may be unreachable). Moreover, we cannot even compute the maximal possible coverage as it is undecidable whether a given part of the code is reachable or not.

High code coverage is only one property of a quality test suite. There are also other ideas associated with quality of a test suite. For example, *mutation analysis* [BDLS80] is based on the following idea:

A test suite is unlikely to be comprehensive enough if it gives the same results to two different programs.

Given a test suite and a code, one generates several mutations of the program (based on code inspection or structural changes) and evaluates the test suite on these mutations. If some test case behaves differently on the original code and a mutation, then the mutation *dies*. At the end of the process, if a considerable number of mutations remain alive, the test suite is probably inappropriate.

Note that an addition of a code monitoring executions of test cases (a code measuring the coverage) can affect the behaviour of the system under test.

There are dedicated software packages for *test case generation*, *coverage evaluation*, *test execution*, and *test management* (maintaining different test suits, perform version control etc.)

3.3 Black box testing: model-based testing

The mission of black box testing is to check functionality of all features of the system under test. First we need to know the intended observable behaviour of the system. This behaviour can be described in many ways like simple text, message sequence charts, state machines, etc. Formal descriptions are called *models* of the system. Figure 4 provides an example of a model given as a finite state machine.

We use the model to generate test cases. Again, various coverage criteria can be chosen (some of them are similar to those mentioned before):

- covering all the edges
- covering all the nodes
- covering all the paths (this is usually impractical)
- covering each adjacent sequence of n nodes

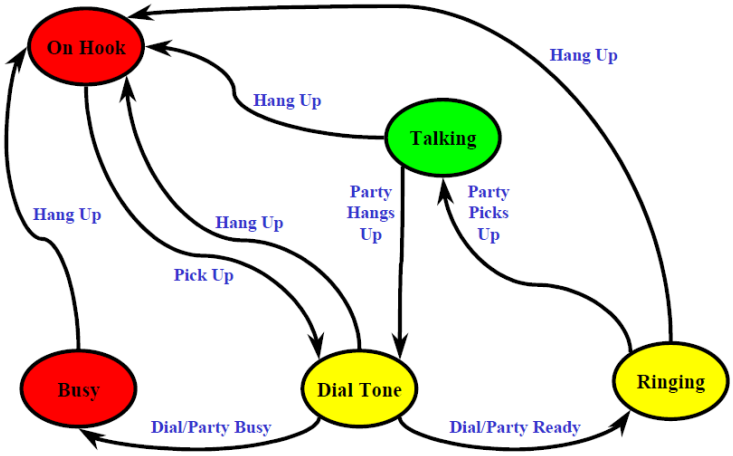


Figure 4: A model of a simple phone system.

- covering certain nodes at least/at most a given number of times in each sequence in the test suite
- (*switch coverage*) covering each pair of incoming and outcoming edge for all nodes

After execution of the tests and comparison of actual outputs with expected outputs, we decide on further actions (whether to modify the model, generate more tests, or stop testing).

3.3.1 Notes

Model checking can be applied to verify whether the model satisfies desired properties.

We may want to test primarily the typical executions of the system in order to maximize the *minimal time to failure (MTTF)*. In this case, we can employ *probabilistic testing*. The system is modelled as *Markov chain*. The test suite is then generated according to the probabilities of transitions.

4 Deductive software verification

This section is based on Chapter 7 of [Pel01]. Research in this area started in late 1960's by Floyd [Flo67] and Hoare [Hoa69].

Since deductive verification is often tedious, it is not performed frequently on the actual code. However, it can be performed on the basic algorithms or on abstractions of the code. The faithfulness of the translation of a program into an abstracted one can sometimes also be formally verified.

4.1 Two notions of correctness

To simplify the presentation, we adopt several assumptions on the programs we want to verify. More precisely, we assume that the initial values of the program are stored in input variables $x0, x1, \ldots$ and that these variables do not change their values during the execution of the program. We also consider only deterministic programs.

By *state* of a program we mean an assignment to the program variables. Let P be a program and a, b be its states. By $P(a, b)$ we denote the fact that executing P from

the state a, it terminates with a state b. Further, by $a \models \varphi$ we denote that the state a satisfies the formula φ.

A *specification* (or some desired property) of a program P is given by two first order formulae:

- *initial condition* φ is a formula with all its free variables among input variables of P
- *final assertion* ψ

We define two notions of correctness. The program P is

partially correct with respect to φ and ψ, written $\{\varphi\}P\{\psi\}$, iff for all states a, b the implication

$$P(a,b) \wedge a \models \varphi \implies b \models \psi$$

holds. In other words, if the program starts with a state satisfying φ and then terminates, then the terminal state satisfies ψ.

totally correct with respect to φ and ψ, written $\langle\varphi\rangle P\langle\psi\rangle$, iff $\{\varphi\}P\{\psi\}$ and for every state a satisfying φ the program terminates.

4.2 Verification of flowcharts

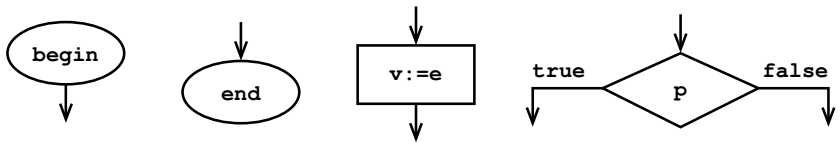


Figure 5: Nodes in a flowchart.

We identify each program with its flowchart. A flowchart has four kinds of nodes (see Figure 5):

begin - one outgoing edge, no incoming edges

end - one incoming edge, no outgoing edges

assignment has the form $v := e$, where v is a program variable and e is a first order term; one or more incoming edges, one outgoing edge

decision predicate p is an unquantified first order formula; one or more incoming edges, two outgoing edges marked with *true* and *false*

A *location* of a flowchart program is an edge connecting two flowchart nodes. An example of a simple flowchart program is given in Figure 6.

4.2.1 Partial correctness

To verify that a program is partially correct with respect to φ and ψ, it is sufficient to perform the following two steps.

1. We attach to each location of the flowchart a first order formula called *assertion*. To the location exiting from the *begin* node we attach φ and ψ is attached to the location entering *end* node. The idea is that these assertions should be satisfied by every state reachable in the corresponding location by an execution starting in a state satisfying φ. These assertions are also called *invariants*[1].

[1] In some programming languages assertions can be inserted into the code as additional runtime checks so that the program will break with a warning message whenever an invariant is violated.

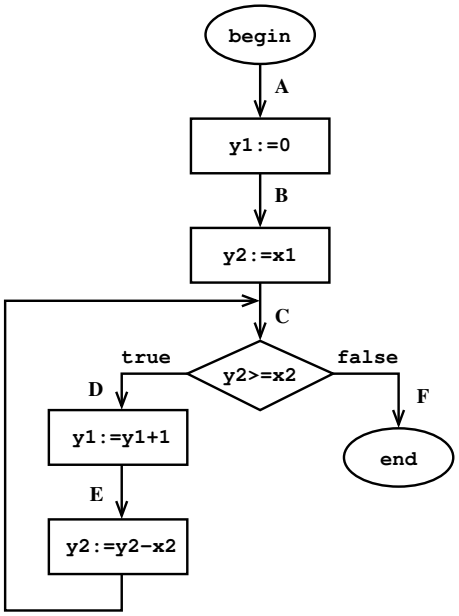


Figure 6: A flowchart program for integer division.

2. For every assignment or decision node c, every assumption $pre(c)$ on an incoming edge (called *precondition*), and every assumption $post(c)$ on an outgoing edge (called *postcondition*), we prove that

   > if the control of the program is just before c, with a state satisfying $pre(c)$ and c is executed such that the control moves to the location annotated with $post(c)$, then the state after the move satisfies $post(c)$.

   Every considered triple $pre(c), c, post(c)$ fits into one of the three cases:

   (a) c is a decision node with a predicate p and $post(c)$ is associated to the outgoing edge marked with *true*. Then we need to prove:

   $$pre(c) \wedge p \implies post(c)$$

   (b) c is a decision node with a predicate p and $post(c)$ is associated to the outgoing edge marked with *false*. Then we need to prove:

   $$pre(c) \wedge \neg p \implies post(c)$$

   (c) c is an assignment of the form $v := e$, where v is a variable and e an expression. This case is more difficult, as the states before and after the assignment are different (i.e. $pre(c)$ and $post(c)$ reason about different states). Therefore, we *relativize* the postcondition to assert about the states before the assignment. Hence, we have to prove

   $$pre(c) \implies post(c)[v/e]$$

   where $post(c)[v/e]$ is the assertion $post(c)$ where all occurences of v are replaced with e.

Proving the consistency between each precondition and postcondition of all nodes guarantess that $\{\varphi\}P\{\psi\}$. In fact, it guarantees even a stronger property:

In each execution that starts with a state satisfying the initial condition of the program, when the control of the program is at some location, the assumption attached to that location holds.

Finding assertions for the proof may be a difficult task. There are some heuristics and tools suggesting invariants. But there cannot be a fully automatic way of finding them (the problem is undecidable).

Example. In order to prove that the flowchart in Figure 6 computes an integer divison, we define the initial condition and final assertion as

$$\begin{array}{lcl} \varphi & \equiv & x1 \geq 0 \wedge x2 > 0 \\ \psi & \equiv & (x1 = y1 * x2 + y2) \wedge y2 \geq 0 \wedge y2 < x2 \end{array}$$

and the other invariants as

$$\begin{array}{lcl} \varphi(A) & \equiv & x1 \geq 0 \wedge x2 > 0 \\ \varphi(B) & \equiv & x1 \geq 0 \wedge x2 > 0 \wedge y1 = 0 \\ \varphi(C) & \equiv & (x1 = y1 * x2 + y2) \wedge y2 \geq 0 \\ \varphi(D) & \equiv & (x1 = y1 * x2 + y2) \wedge y2 \geq x2 \\ \varphi(E) & \equiv & (x1 = y1 * x2 + y2 - x2) \wedge y2 - x2 \geq 0 \\ \varphi(F) & \equiv & (x1 = y1 * x2 + y2) \wedge y2 \geq 0 \wedge y2 < x2 \end{array}$$

Please note that the decision node `y2>=x2` has in fact two ingoing edges, one leading from the node `y2:=x1`, the other leading from the node `y2:=y2-x2`. Both locations corresponding to these edges are associated with the invariant $\varphi(C)$.

Now the consistency can be checked using a mechanized theorem prover (at least in this case). □

4.2.2 Modification for array variables

To verify programs with array variables, the method has to be modified in one point: relativization of postconditions of assignment nodes.

The problem can be demonstarted by the following simple example. Consider the assignment $x[x[1]] := 2$ with precondition $pre(c) \equiv x[1] = 1 \wedge x[2] = 3$ and postcondition $post(c) \equiv x[x[1]] = 2$. It is easy to prove that

$$pre(c) \implies post(c)[x[x[1]]/2]$$

holds as $post(c)[x[x[1]]/2]$ is in fact the tautology $2 = 2$. But if $pre(c)$ holds and the assignment is performed, then $x[1] = 2$ and $x[x[1]] = 3$ and hence the $post(c)$ does not hold.

In order to fix this problem, we extend the syntax of terms with a new construct $(x; e1{:}e2)[e3]$, where x is an array variable (we assume that the set of array variables is disjoint from simple variables and we do not allow quantifying over array variables in our first order logic) and $e1, e2, e3$ are terms. Informally, $(x; e1{:}e2)$ denotes almost the same array as x: only the element with the index $e1$ has been set to $e2$.

The added construct does not increase the expressiveness of the logic. Consider a formula ρ containing an $(x; e1{:}e2)[e3]$ construct. This formula can be translated into an equivalent formula

$$(e1 = e3 \wedge \rho[(x; e1{:}e2)[e3]/e2]) \vee (\neg(e1 = e3) \wedge \rho[(x; e1{:}e2)[e3]/x[e3]).$$

This translation removes all occurrences of $(x; e1{:}e2)[e3]$. The process can be repeated until all added constructs are eliminated.

Let c be an array assignment $x[e1] := e2$. To guarantee that if $pre(c)$ holds and c is executed then $post(c)$ holds, we have to prove that

$$pre(c) \implies post(c)[x/(x; e1{:}e2)]$$

where $post(c)[x/(x; e1{:}e2)]$ denotes the $post(c)$ formula with x substituted by $(x; e1{:}e2)$.

Example. Let us return to the case where c is $x[x[1]] := 2$ and $post(c) \equiv x[x[1]] = 2$. The relativized postcondition $post(c)[(x; x[1]{:}2)/x]$ can be simplified in the following way:

$$\begin{array}{ll} & post(c)[x/(x; x[1]{:}2)] \quad \equiv \\ \equiv & (x[x[1]]) = 2)[x/(x; x[1]{:}2)] \\ \equiv & (x; x[1]{:}2)[(x; x[1]{:}2)[1]] = 2 \\ \equiv & (x[1] = 1 \ \wedge \ (x; x[1]{:}2)[2] = 2) \ \vee \ (\neg(x[1] = 1) \ \wedge \ (x; x[1]{:}2)[x[1]] = 2) \\ & \vdots \\ \equiv & x[1] = 1 \implies x[2] = 2 \end{array}$$

The resulting formula is the expected one. The simplification steps may be quite difficult and hard to follow. Fortunately, mechanized theorem provers can help. □

4.3 Proving termination

A *partially ordered domain* is a pair $(W, \prec)$ where W is a set and $\prec$ is a strict partial order relation over W (i.e. irreflexive, asymmetric, and transitive). We often write $u \succ v$ instead of $v \prec u$. We also write $u \succeq v$ when $u \succ v$ or $u = v$. A *well founded domain* is a partially ordered domain that contains no infinite sequence of the form $w_0 \succ w_1 \succ w_2 \succ \ldots$.

To prove the termination with respect to the initial condition φ, we need to follow the following steps:

1. Select a well founded domain $(W, \succ)$ such that W is a subset of the domain of program variables and $\succ$ is expressible using the signature of the program.
2. Attach to each location in the flowchart an invariant and an expression. The invariant attached to the outgoing edge of the *begin* node is the initial condition.
3. Show consistency for each triple $pre(c), c, post(c)$, as in the partial correctness proof.
4. We show that expressions associated with locations satisfy the following conditions:

   - When an expression e, attached to a flowchart location, is calculated in some state in the execution (when the program counter is in that location), it is within W. This means that for each location with an associated invariant ρ and expression e we have to prove

     $$\rho \implies (e \in W).$$

     Note that $e \in W$ is not, in general, a first order logic formula. However, in this context, it can often be translated into a first order formula.

   - In each execution of the program, when proceeding from one location to its successor location, the value of the associated expression does not increase. More precisely, for every node c, every invariant $pre(c)$ and expression $e1$ associated with an incoming edge, and every expression $e2$ associated with an outgoing edge, we have to prove that
     - if c is a decision node with predicate p and $e2$ is associated with the *true* edge, then
       $$pre(c) \wedge p \implies e1 \succeq e2,$$
     - if c is a decision node with predicate p and $e2$ is associated with the *false* edge, then
       $$pre(c) \wedge \neg p \implies e1 \succeq e2,$$

– if c is an assignment $v := e$ then

$$pre(c) \implies e1 \succeq e2[e/v].$$

- In each execution of the program, during a traversal of a cycle (a loop) in the flowchart there is some point where a decrease occurs in the value of the associated expression from one location to its successor. This means that for each cycle we have to find a node with an incoming and an outgoing edges such that the corresponding implication above holds even if $\succeq$ is replaced with $\succ$.

If we manage to do all these steps, the termination is proven. It may be difficult to find the right well founded domain, invariants and expressions. Clearly, termination and partial correctness can be proven simultaneously.

Example. Termination of the flowchart in Figure 6 with the initial condition $\varphi \equiv x1 \geq 0 \wedge x2 > 0$ can be proven with the following invariants and expressions:

$$\begin{array}{lcll} \varphi(A) & \equiv & x1 \geq 0 \wedge x2 > 0 & e(A) \equiv x1 \\ \varphi(B) & \equiv & x1 \geq 0 \wedge x2 > 0 & e(B) \equiv x1 \\ \varphi(C) & \equiv & x2 > 0 \wedge y2 \geq 0 & e(C) \equiv y2 \\ \varphi(D) & \equiv & x2 > 0 \wedge y2 \geq x2 & e(D) \equiv y2 \\ \varphi(E) & \equiv & x2 > 0 \wedge y2 \geq x2 & e(E) \equiv y2 \\ \varphi(F) & \equiv & y2 \geq 0 & e(F) \equiv y2 \end{array}$$

□

4.4 Axiomatic program verification

Hoare [Hoa69] developed a proof system that included both logic and pieces of code. This proof system allows us to verify code (rather than flowcharts) and to prove different sequential parts of the program separately (and combine the proofs later).

The logic in constructed on top of some first order deduction system. In addition to first order formulas, the logic allows assertions of the form $\{\varphi\}S\{\psi\}$, where φ, ψ are first order formulas and S is (a part of) a program with the following syntax:

$$S ::= v := e \mid \mathit{skip} \mid S; S \mid \mathit{if}\ p\ \mathit{then}\ S\ \mathit{else}\ S\ \mathit{fi} \mid \mathit{while}\ p\ \mathit{do}\ S\ \mathit{end} \mid \mathit{begin}\ S\ \mathit{end}$$

where v is a variable, e is a first order expression, and p is an unquantified first order formula. These assertions are called *Hoare triples*. A Hoare triple $\{\varphi\}S\{\psi\}$ means that if execution of S starts with a state satisfying φ and S terminates from that state, then a state satisfying ψ is reached. Clearly, if S is the entire program, then $\{\varphi\}S\{\psi\}$ claims that S is partially correct with respect to initial condition φ and final assertion ψ.

The axioms and proof rules of Hoare's logic are given below. The proof rules use the standard notation where premises are above the line while the consequent is below the line.

Assignment axiom

$$\{\varphi[v/e]\}v := e\{\varphi\}$$

Skip axiom

$$\{\varphi\}\mathit{skip}\{\varphi\}$$

Left strengthening rule

$$\frac{\varphi \implies \varphi' \quad \{\varphi'\}S\{\psi\}}{\{\varphi\}S\{\psi\}}$$

Right weakening rule

$$\frac{\{\varphi\}S\{\psi'\} \quad \psi' \Longrightarrow \psi}{\{\varphi\}S\{\psi\}}$$

Sequential composition rule

$$\frac{\{\varphi\}S_1\{\eta\} \quad \{\eta\}S_2\{\psi\}}{\{\varphi\}S_1; S_2\{\psi\}}$$

If-then-else rule

$$\frac{\{\varphi \wedge p\}S_1\{\psi\} \quad \{\varphi \wedge \neg p\}S_2\{\psi\}}{\{\varphi\}\mathit{if}\ p\ \mathit{then}\ S_1\ \mathit{else}\ S_2\ \mathit{fi}\{\psi\}}$$

While rule

$$\frac{\{\varphi \wedge p\}S\{\varphi\}}{\{\varphi\}\mathit{while}\ p\ \mathit{do}\ S\ \mathit{end}\{\varphi \wedge \neg p\}}$$

Begin-end rule

$$\frac{\{\varphi\}S\{\psi\}}{\{\varphi\}\mathit{begin}\ S\ \mathit{end}\{\psi\}}$$

Some other proof rules can be derived, like the following rules combining the assignment axiom with the left strengthening rule (this rule has actually just one premise, the axiom is written there just for explanation) and the sequential composition rule with the right weakening or left strengthening rule, respectively:

$$\frac{\varphi \Longrightarrow \psi[v/e] \quad (\mathit{axiom}\colon \{\psi[v/e]\}v := e\{\psi\})}{\{\varphi\}v := e\{\psi\}} \qquad \frac{\{\psi\}S_1\{\eta_1\} \quad \eta_1 \Longrightarrow \eta_2 \quad \{\eta_2\}S_2\{\psi\}}{\{\varphi\}S_1; S_2\{\psi\}}$$

The proof trees are constructed as usual...

There are several proof system that extend the Hoare proof system for verifying concurrent programs. Such proof systems provide axioms for dealing with shared variables, synchronous and asynchronous communication, procedure calls). They are usually tailored for a particular programming language, e.g. Pascal or CSP.

A generic proof system for handling concurrency was suggested by Manna and Pnueli [MP83]. The system is not connected to any particular syntax. Instead, it works with transition systems. It allows verifying temporal properties. Here is an example of proof rules (these two are called FCS and FPRS):

$$\frac{\begin{array}{l}\mathsf{G}(\varphi \Longrightarrow (\varphi_1 \vee \varphi_2)) \\ \mathsf{G}(\varphi_1 \Longrightarrow \mathsf{X}\psi) \\ \mathsf{G}(\varphi_2 \Longrightarrow \mathsf{X}\psi)\end{array}}{\mathsf{G}(\varphi \Longrightarrow \mathsf{X}\psi)} \qquad \frac{\mathsf{G}(\varphi \Longrightarrow \mathsf{X}\psi)}{\mathsf{G}(\varphi \Longrightarrow \mathsf{F}\psi)}$$

4.5 Notes

Deductive verification

- is not limited to finite state systems.
- can handle programs of various domains and datastructures (and even parametrized programs).
- can be applied directly to the code.
- can verify that the program is correct (but a bug can occur in compiler, in hardware, due to a wrong initial condition or difference between assumed semantics of code and the real one, etc.).

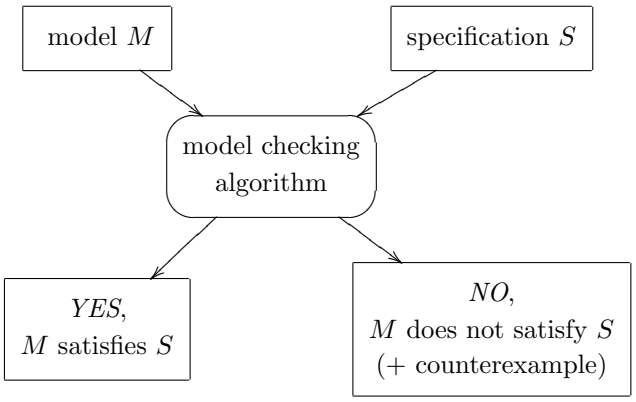


Figure 7: The model checking schema.

- needs a great mental effort as it is mostly manual (the result depends strongly on the ingenuity of the people performing verification). Hence it is very slow. Moreover, proofs constructed manually may contain errors. The chance to find an error is high.
- is not scalable.

The presented Hoare's proof system is sound. It is not complete thanks to incompletenes of first order logic with natural numbers and basic arithmetic operations over them (Gödel's incompletenes theorem). But it is *relatively complete*, i.e. any correct assertion can be proved under the following two (sometimes unrealistic) conditions:

- Every correct (first order) logic assertion that is needed in the proof is already included as an axiom in the proof system. (Alternatively: there is an *oracle* (e.g. a human) that is responsible to decide whether such an assertion is correct or not.)
- Every invariant or intermediate assertion that we need for the proof is expressive.

The second condition eliminates the cases when the proof needs some properties that cannot be expressed in first order logic.

The relative completeness of the Hoare's proof system implies that the system is complete for programs and first order logic with natural numbers and addition and subtraction as the only operators.

5 Model checking: an overview

In this section we define the model checking problem and we provide a basic taxonomy of system classes and temporal logics. In particular, we recall definitions of low-level formalisms for system description (*Kripke structure* and *labelled transition system*), *Linear Temporal Logic (LTL)*, and *Büchi automata (BA)*. We also provide borders of decidability of the model checking problem. Further, we recall the automata-based approach to LTL model checking of finite-state systems.

Informations presented in this section can be found e.g. in Chapters 1, 2, 3 and 9 of [CGP99] and in [May98] (everything specific to infinite-state systems).

Model checking problem is the problem to decide whether a given model (or system) satisfies a given specification (see Figure 7). The problem is further specified by the considered class of models and by the considered class of specifications.

5.1 Specifications

Specification is a formal description of some property that should be satisfied by the system. Note that in the context of model checking, specification usually does not describe the full functionality of the system. Specification is typically given as a formula of some *temporal logic*. There are two kinds of temporal logics:

linear-time logics Formulae are interpreted over sequences representing single runs. The most popular linear-time logic is *Linear Temporal Logic (LTL)*.

branching time logics Formulae are interpreted over trees, where a tree represents all possible runs starting in a given state. Typical examples of branching-time logics are *Computational Tree Logic (CTL)*, *CTL**, *Hennessy–Milner logic*, and *modal μ-calculus*.

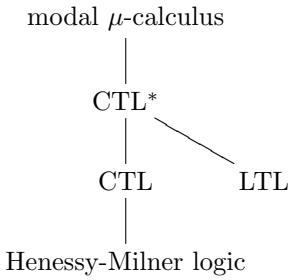


Figure 8: The hierarchy of basic temporal logics.

Figure 8 represents the relative expressive power of the mentioned logics: a line between two logics means that every property expressible in the lower logic can be also expresses in the upper logic, but not vice versa.

Temporal logics can be also divided into state-based and action-based.

state-based These logics talk about properties of states of the system. Properties of a single state are reflected by validity of *atomic propositions* in the state. These atomic propositions are then atomic formulaes of state-based logics.

action-based Every transition of a system is labelled with an action. Action-based logic are interpreted over behaviours of the system represented only by sequences (or trees) of actions.

In the following, we focus on LTL model checking problems. Now we provide definition of both action-based and state-based LTL.

Formulas of *state-based Linear Temporal Logic (LTL)* are defined by the abstract syntax equation

$$\varphi \ ::= \ \top \ \mid \ a \ \mid \ \neg\varphi \ \mid \ \varphi_1 \wedge \varphi_2 \ \mid \ \mathsf{X}\varphi \ \mid \ \varphi_1 \,\mathsf{U}\, \varphi_2,$$

where $\top$ stands for *true* and a ranges over a countable set AP of *atomic propositions*. We also use $\bot$ to abbreviate $\neg\top$, $\mathsf{F}\varphi$ to abbreviate $\top \,\mathsf{U}\, \varphi$, $\mathsf{G}\varphi$ to abbreviate $\neg\mathsf{F}\neg\varphi$, and $\varphi \,\mathsf{R}\, \psi$ to abbreviate $\neg(\neg\varphi \,\mathsf{U}\, \neg\psi)$. The temporal operators $\mathsf{X}, \mathsf{F}, \mathsf{U}, \mathsf{G}, \mathsf{R}$ are called *next*, *eventually*, *until*, *globally*, and *release*, respectively. The set of atomic propositions occurring in a formula φ is denoted by $AP(\varphi)$.

We define the semantics of LTL in terms of languages over infinite words. An *alphabet* is a set $\Sigma = 2^{AP'}$, where $AP' \subseteq AP$ is a finite subset. A *word* over alphabet Σ is an infinite sequence $w = w(0)w(1)w(2)\ldots \in \Sigma^\omega$ of letters from Σ. For every $i \in \mathbb{N}_0$, by w_i we denote the suffix of w of the form $w(i)w(i+1)w(i+2)\ldots$.

The *validity* of an LTL formula φ for $w \in \Sigma^\omega$, written $w \models \varphi$, is defined as follows:

$$
\begin{array}{lll}
w \models \top & & \\
w \models a & \text{iff} & a \in w(0) \\
w \models \neg\varphi & \text{iff} & w \not\models \varphi \\
w \models \varphi_1 \wedge \varphi_2 & \text{iff} & w \models \varphi_1 \wedge w \models \varphi_2 \\
w \models \mathsf{X}\varphi & \text{iff} & w_1 \models \varphi \\
w \models \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \exists i \in \mathbb{N}_0 : w_i \models \varphi_2 \wedge \forall\, 0 \leq j < i : w_j \models \varphi_1
\end{array}
$$

Given an alphabet Σ, an LTL formula φ defines the language

$$L^{\Sigma}(\varphi) = \{w \in \Sigma^{\omega} \mid w \models \varphi\}.$$

An action-based LTL is very similar to the state-based version. The only changes are the following. In the syntax, a ranges over countable set of *actions* Act. Formulae of action-based LTL are then interpreted over infinite sequences of actions from a finite subset $Act' \subseteq Act$. Semantics of formula a is defined as follows:

$$
\begin{array}{lll}
w \models a & \text{iff} & a = w(0)
\end{array}
$$

The specification can be formulated also without any logic. Linear-time properties can be defined for example with use of Büchi automata.

Definition 5.1. *A* Büchi automaton (BA) *is a tuple* $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$*, where*

- Σ *is a finite* alphabet*,*
- Q *is a finite set of* states*,*
- $\delta : Q \times \Sigma \to 2^Q$ *is a* transition function*,*
- $q_0 \in Q$ *is an* initial states*,*
- $F \subseteq Q$ *is a set of* accepting states*.*

A run *of* $\mathcal{A}$ *on inifnite word* $w = w(0)w(1)... \in \Sigma^{\omega}$ *is an infinite sequence of states* $\sigma = \sigma(0)\sigma(1)...$*, where* $\sigma(0) = q_0$ *and* $\sigma(i+1) \in \delta(\sigma(i), w(i))$ *holds for all* i*.*

A run σ *is* accepting *if* $Inf(\sigma) \cap F \neq \emptyset$*, where* $Inf(\sigma)$ *is a set of the states appearing in* σ *infinitely often. An automaton* $\mathcal{A}$ *accepts a word* w *if there is an accepting run of* $\mathcal{A}$ *on* w*. We set*

$$L(\mathcal{A}) = \{w \in \Sigma^{\omega} \mid \mathcal{A} \text{ accepts } w\}.$$

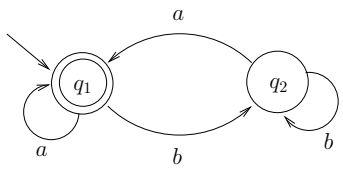


Figure 9: A Büchi automaton accepting words with infinitely many occurrences of a.

A graphical description of Büchi automata is identical to the graphical desciprion of nondeterministic finite automata (see Figure 9).

Later we show that every action-based LTL formula can be translated into an equivalent Büchi automaton.

```
byte cnt = 0;  // number of processes in critical sections
byte turn = 0; // token for entering a critical section

init {
   run(P0); run(P1);  // parallel execution of P0 a P1
}

proctype P0()
{
  // s0
  do
  // NC0 (noncritical section)
  :: do
     :: (turn == 0) -> break;
     :: else;
     od;
     // CS0 (critical section)
     cnt = cnt + 1;
     cnt = cnt - 1;
     turn = 1;
  od;
}
```

```
proctype P1()
{
  //s1
  do
  // NC1 (noncritical section)
  :: do
     :: (turn == 1) -> break;
     :: else;
     od;
     // CS1 (critical section)
     cnt = cnt + 1;
     cnt = cnt - 1;
     turn = 0;
  od;
}
```

Figure 10: Mutual exclusion program in ProMeLa.

5.2 Models

By model we mean a formal description of all possible behaviours of the system to be verified, where *behaviour* is a sequence (or a tree in the branching-time setting) of successive states or actions (depending on the setting) starting in an initial state of the system. A model can be described in a standard language (C, Java, VHDL, ...), or in some dedicated languge (for examle, SPIN [SPI] uses *Process* or *Protocol Meta Language ProMeLa*[2]), or with use of some process algebra (BPA, BPP, PA, *pushdown processes*, Petri nets, ... Here we introduce two low-level formalisms for representation of models: *Kripke structure* (used in context of state-based model checking) and *labelled transition systems* (used for the action-based approach).

Definition 5.2. *A* Kripke structure *is a tuple* $M = (S, R, S_0, L)$*, where*

- S *is a set of* states
- $R \subseteq S \times S$ *is* transitions relation
- $S_0 \subseteq S$ *is a set of* initial states
- $L : S \to 2^{AP}$ *is a* labelling function *associating to each state* $s \in S$ *the set of atomic propositions that are true in* s*.*

A path *in* M *starting in a state* s *is an infinite sequence* $\pi = s_0 s_1 s_2 ...$ *of states such that* $s_0 = s$ *and* $(s_i, s_{i+1}) \in R$ *holds for every* i*.*

Given a set $AP' \subseteq AP$*, we set*

$$L^{AP'}(M) = \{l_0 l_1 \ldots \mid \exists \text{ a path } s_0 s_1 \ldots \text{ of } M \,.\, \forall i \geq 0 : l_i = L(s_i) \cap AP'\}.$$

[2]An example of ProMeLa code is provided by Figure 10.

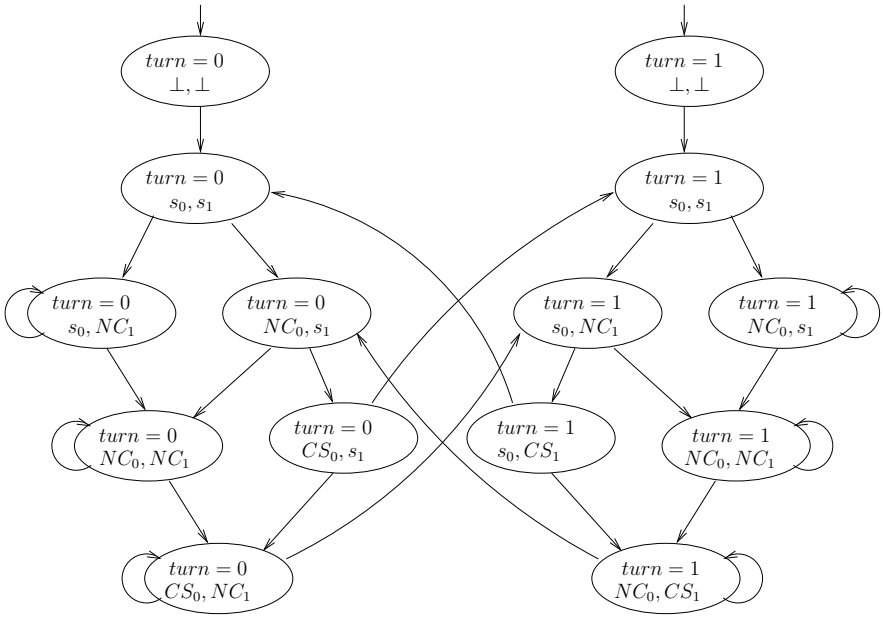


Figure 11: Kripke structure for a mutual exclusion program.

Figure 11 provides an example of a Kripke structure corresponding to the ProMeLa code of Figure 10. Each state is directly marked with the atomic propositions valid in the state.

Definition 5.3. *A* labelled transition system (LTS) *is a tuple* $M = (S, Act, \rightarrow, s_0)$, *where*

- S *is a set of* states
- Act *is a set of* atomic actions *or* labels
- $\rightarrow \subseteq S \times Act \times S$ *is* transitions relation *(we write* $s \xrightarrow{a} t$ *instead of* $(s, a, t) \in \rightarrow$*)*
- $s_0 \in S$ *is a distinguished* initial states

A run *of* M over $u = u_0 u_1 \ldots \in Act^\omega$ *is a seguence*

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots,$$

where s_0 *is the initial state. We set*

$$L(M) = \{u \in Act^\omega \mid \textit{there is a run of } M \textit{ over } u\}.$$

We often use some more concise formalism for description of models. In particular, infinite-state models cannot be finitely described directly by a Kripke structure or an LTS.

We introduce a formalism called *Process Rewrite Systems (PRS)* [May00], which subsumes many standard formalisms for description of infinite-state systems. More information about PRS and their properties can be found in [May98, Řeh07].

The concept of Process Rewrite Systems is based on rewriting of *process terms*.

Definition 5.4. *Let* $Const = \{X, \ldots\}$ *be a countably infinite set of* process constants. *The set* $\mathcal{T}$ *of* process terms *is defined by the abstract syntax*

$$t ::= \varepsilon \mid X \mid t_1.t_2 \mid t_1 \| t_2,$$

where

- ε *is the empty term,*
- $X \in Const$ *is a process constant (used as an atomic process),*
- *'*$\|$*' means a parallel composition, and*
- *'.' means a sequential composition.*

We always work with equivalence classes of terms modulo commutativity and associativity of '$\|$*' and modulo associativity of '.' We also define* $\varepsilon.t = t = t.\varepsilon$ *and* $t\|\varepsilon = t$. *By* $Const(t)$ *we denote the set of process constants occurring in* t.

We distinguish four *classes of process terms* as:

"1" terms consisting of a single process constant only (i.e. $\varepsilon \notin 1$), e.g. X.

"S" *sequential* terms without parallel composition, e.g. $X.Y.Z$.

"P" *parallel* terms without sequential composition. e.g. $X\|Y\|Z$.

"G" *general* terms with arbitrarily nested sequential and parallel compositions.

Definition 5.5. *Let* $Act = \{a, b, \cdots\}$ *be a countably infinite set of* atomic actions *and* $\alpha, \beta \in \{1, S, P, G\}$ *such that* $\alpha \subseteq \beta$. *An* (α, β)-PRS (process rewrite system) *is a pair* $\Delta = (R, t_0)$, *where*

- $R \subseteq ((\alpha \setminus \{\varepsilon\}) \times Act \times \beta)$ *is a finite set of* rewrite rules, *and*
- $t_0 \in \beta$ *is an* initial term.

We write $(t_1 \overset{a}{\hookrightarrow} t_2) \in R$ *instead of* $(t_1, a, t_2) \in R$.

We define $Const(\Delta)$ as the set of all constants occurring in the rewrite rules of Δ or in its initial state, and $Act(\Delta)$ as the set of all actions occurring in the rewrite rules of Δ.

Definition 5.6. *The semantics of an* (α, β)*-PRS* $\Delta = (R, t_0)$ *is given by the LTS* $(S, Act(\Delta), \longrightarrow_\Delta, t_0)$, *where*

- *the set of states* $S = \{t \in \beta \mid Const(t) \subseteq Const(\Delta)\}$,
- *the transition relation* $\longrightarrow_\Delta$ *is the least relation satisfying the following inference rules for all* t_1, t_2, $t \in S$ *and* $a \in Act(\Delta)$.

$$\frac{(t_1 \overset{a}{\hookrightarrow} t_2) \in R}{t_1 \overset{a}{\longrightarrow}_\Delta t_2} \qquad \frac{t_1 \overset{a}{\longrightarrow}_\Delta t_2}{t_1 \| t \overset{a}{\longrightarrow}_\Delta t_2 \| t} \qquad \frac{t_1 \overset{a}{\longrightarrow}_\Delta t_2}{t_1.t \overset{a}{\longrightarrow}_\Delta t_2.t}$$

Note that parallel composition is commutative and, thus, the inference rule for parallel composition also holds with t_1 *and* t *exchanged.*

Every pair $\alpha, \beta \in \{1, S, P, G\}$ induces a class of all labelled transition systems that can be expressed by an (α, β)-PRS. Some of the classes correspond to LTS classes of widely known models.

- $(1, 1)$-PRS are equivalent to *finite-state systems (FS)*. Every process constant corresponds to a state and the state space is bounded by $|Const(\Delta)|$.
- $(1, S)$-PRS are equivalent to *Basic Process Algebra processes (BPA)*.This class can model sequential programs with procedures. BPA can model unbounded nesting of procedure calls, but cannot model return values and global variables.

- $(1, P)$-PRS are equivalent to communication-free nets, the subclass of Petri nets where every transition has exactly one place in its preset. This class of Petri nets is equivalent to *Basic Parallel Processes (BPP)*.The class can model systems consisting of simple finite-state parallel processes. The number of parallel processes in unbounded. Processes can be dynamically created or terminated, but they cannot communicate.
- $(1, G)$-PRS are equivalent to *PA-processes*,process algebras with sequential and parallel composition, but no communication. This is the least common generalization of BPA and BPP.
- It is easy to see that pushdown automata can be encoded as a subclass of (S, S)-PRS (with at most two constants on the left-hand side of rules). Caucal showed that any unrestricted (S, S)-PRS can be presented as a pushdown automaton, in the sense that the transition systems are isomorphic up to the labelling of states. Thus (S, S)-PRS are equivalent to *pushdown processes (PDA)* (which are the processes described by pushdown automata). This formalism can model sequential programs with procedure call, return values, and global variables.
- (P, P)-PRS are equivalent to *Petri nets (PN)*. Every constant corresponds to a place in the net and the number of occurrences of a constant in a term corresponds to the number of tokens in this place. This is because we work with classes of terms modulo commutativity of parallel composition. Every rule in Δ corresponds to a transition in the net.
- (S, G)-PRS is the smallest common generalisation of pushdown processes and PA-processes. They are called *PAD* (PA + PDA).
- (P, G)-PRS are called *PAN-processes*.It is the smallest common generalisation of Petri nets and PA-processes and it strictly subsumes both of them (e.g. PAN can describe all Chomsky-2 languages while Petri nets cannot).
- The most general case (G, G)-PRS is simply called *PRS*.

Figure 12 describes a hierarchy of (α, β)-PRS classes with respect to strong bisimulation equivalence (bisimilarity). We call this hierarchy the *PRS-hierarchy*. More precisely, the classes of PRS systems are interpreted as the sets of their underlying labelled transition systems. A line connecting X and Y with Y placed higher than X means that every transition systems definable in X can be (up to bisimulation equivalence) defined in Y while the reverse does not hold – we write $X \subsetneq Y$. Moreover, the classes that are not connected by any sequence of upward going lines are incomparable.

5.3 Model checking problems and decidability

At the beginning, we have said that the model checking problem is to decide whether a given model satisfies a given specification. We have to define when a system satisfies a specification. In general, it means that all behaviours of the systems satisfy the specification. In state-based LTL setting, it means that a given model M and a given LTL formula φ satisfy

$$L^{AP(\varphi)}(M) \subseteq L^{2^{AP(\varphi)}}(\varphi).$$

Similarly, in action-based LTL setting, it means that a given model M and a given LTL formula φ satisfy

$$L(M) \subseteq L^{Act}(\varphi).$$

Recall that the model checking problem is parametrized by a logic and a class of systems. Hence, the model checking problem for a logic L and a class of systems C is to decide whether a given model of C satisfies a given formula of L. The combination of L and C determines whether the problem is decidable or not. For example, Figure 13

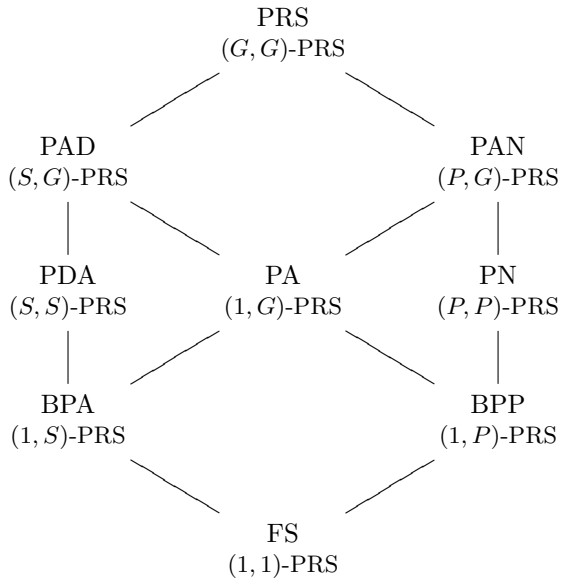


Figure 12: The PRS-hierarchy

presents the decidability boundary of action-based LTL model checking problem for classes of PRS hierarchy,

The situation in state-based LTL model checking is more complicated as the decidability depends also on the set of atomic proposition and their relation to states. The decidability border may differ from the one for action-based LTL model checking mentioned. For example, the state-based LTL model checking is undecidable for the class of Petri nets with atomic propositions saying whether a particular places are non-empty.

Later we will show that state-based LTL model checking remains decidable for PDA where validity of atomic propositions depends only on control state and the topmost stack symbol.

5.4 Automata-based LTL model checking of finite systems

This subsection is devoted to a prominent model checking problem: state-based LTL model checking of finite state systems.

Figure 14 represents the automata-based approach to LTL model checking of finite-state systems. For details see [CGP99]. Time and space complexity of this Lalgorithm is $\mathcal{O}(|M| \cdot 2^{\mathcal{O}(|\varphi|)})$, where $|M|$ is the number of states and transitions in the Kripke structure M. The LTL model checking problem is PSPACE-complete.

The biggest disadvantage of this approach originates in translations of a model into the corresponding Kripke structure: the Kripke structure is extremely large even for systems with relatively short description in some higher-level formalism (these formalisms represent the state space implicitly, while a Kripke structure explicitely represents each state of the model). This phenomenon is called *state explosion problem*. The main sources of the explosion are large domains of variable values, parallelism, dynamically allocated memory, etc. Many techniques fighting this problem have been

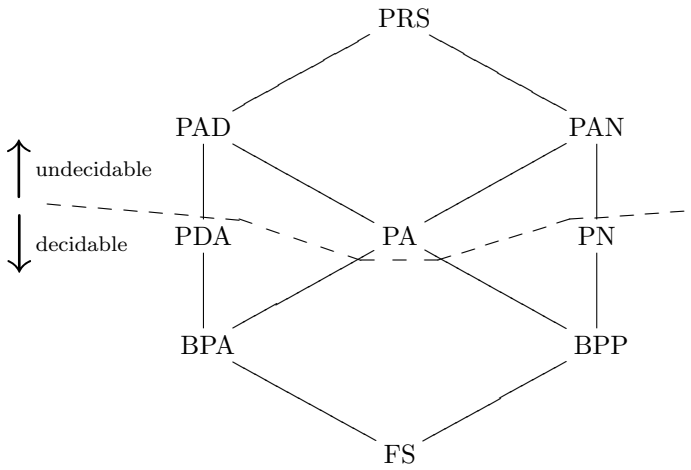


Figure 13: The decidability boundaries of the action-based LTL model checking.

suggested. Some of them aim to minimize the Kripke structure, for example

- abstraction,
- partial order reduction,
- symmetry reduction.

Other techniques try to optimize the memory consumption of model checking algorithms or enable the algorithms to use more hardware sources (e.g. by running on more computers simultaneously):

- on-the-fly algorithms
- symbolic model checking
- distributed algorithms

Some of these methods are mentioned in the following sections.

6 Translation LTL→BA via alternating automata

There are two popular translations of LTL formulae into equivalent Büchi automata. Both of them proceed in two steps:

- an LTL formula is translated into an intermediate formalism
- this formalism is then translated into Büchi automata.

One translation uses *generalized Büchi automata* (see e.g. Chapter 9 of [CGP99]), while the other employs *alternating 1-weak Büchi automata* [Var95]. The latter translation is more frequent nowadays as there exist some optimization reducing the size of alternating 1-weak automata and hence the Büchi automata produced by the whole translation are in some cases smaller than the automata produced by the other translation. We present the translation using alternating automata.

6.1 Alternating 1-weak Büchi automata (A1W)

The transition function of an alternating automaton assigns to each state and letter a positive boolean formula over states. The set of *positive boolean formulae* over set

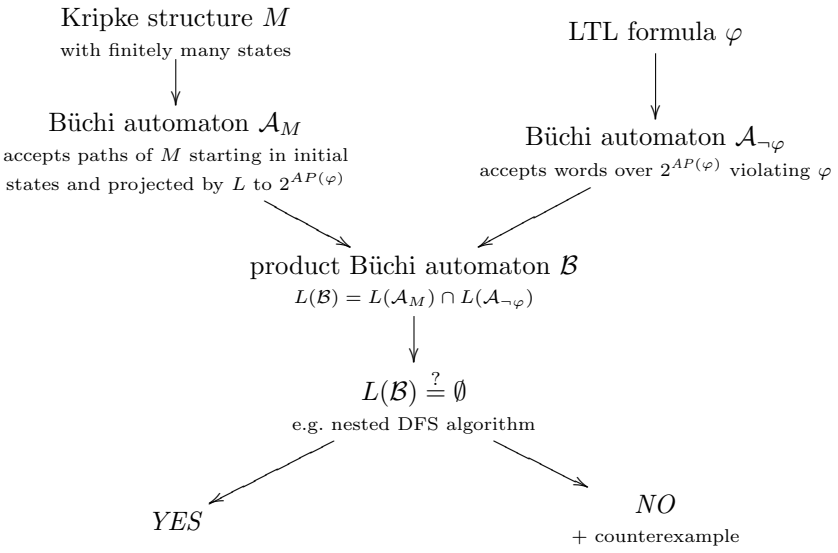


Figure 14: Automata-based LTL model checking of finite system.

Q (denoted $\mathcal{B}^+(Q)$) consists of formulae $\top$ (true), $\bot$ (false), all elements of Q, and boolean combinations over Q built with $\wedge$ and $\vee$. A subset S of Q is a *model* of $\varphi \in \mathcal{B}^+(Q)$ iff φ is satisfied by the valuation assigning true just to states in S. A set S is a *minimal model* of φ (denoted $S \models \varphi$) iff S is a model of φ and no proper subset of S is a model of φ. If we transform a positive boolean formula into a disjunctive normal form (DNF), then every minimal model corresponds to some clause and vice versa. Let us note that there is no model of $\bot$, while every subset of Q is a model of $\top$. The minimal model of $\top$ is the empty set.

Definition 6.1. *An* alternating Büchi automaton *is a tuple* $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$*, where*

- Σ *is a finite* alphabet*,*
- Q *is a finite set of* states*,*
- $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ *is a* transition function*,*
- $q_0 \in Q$ *is an* initial state*,*
- $F \subseteq Q$ *is a set of* accepting states*.*

By $\mathcal{A}(p)$ *we denote the automaton* $\mathcal{A}$ *with initial state* $p \in Q$ *instead of* q_0*.*

A run of an alternating automaton is a (potentially infinite) tree. A tree *is a set* $T \subseteq \mathbb{N}_0^*$ *such that if* $xc \in T$*, where* $x \in \mathbb{N}_0^*$ *and* $c \in \mathbb{N}_0$*, then also* $x \in T$ *and* $xc' \in T$ *for all* $0 \leq c' < c$*. A* Q-labeled tree *is a pair* (T, r) *where* T *is a tree and* $r : T \to Q$ *is a labeling function. A* run *of an automaton* $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ *over word* $w = w(0)w(1)\ldots \in \Sigma^\omega$ *is a* Q*-labeled tree* (T, r) *such that* $r(\varepsilon) = q_0$ *and for each* $x \in T$ *the set* $S = \{r(xc) \mid c \in \mathbb{N}_0, xc \in T\}$ *satisfies* $S \models \delta(r(x), w(|x|))$*. A run* (T, r) *is* accepting *iff for each infinite path* π *in* T *it holds that* $Inf(\pi) \cap F \neq \emptyset$*, where* $Inf(\pi)$ *is the set of all labels (i.e. states) appearing infinitely often on* π*. An automaton* $\mathcal{A}$ accepts *a word* $w \in \Sigma^\omega$ *iff there exists an accepting run of* $\mathcal{A}$ *over* w*. A language of all words accepted by an automaton* $\mathcal{A}$ *is denoted by* $L(\mathcal{A})$*.*

A (standard) *Büchi automaton (BA)* corresponds to an alternating Büchi automaton where, for each $q \in Q$ and $l \in \Sigma$, $\delta(q, l)$ is a disjunction of states (or $\bot$). The

difference is only in notation: in the case of (standard) Büchi automata we usually write $\delta(q,l) = \{q_1, \ldots, q_n\}$ instead of $\delta(q,l) = q_1 \vee \ldots \vee q_n$ and $\delta(q,l) = \emptyset$ instead of $\delta(q,l) = \bot$.

Definition 6.2. *Let $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ be an alternating Büchi automaton. By $Succ(p)$ we denote the set $Succ(p) = \{q \mid \exists l \in \Sigma, S \subseteq Q : S \cup \{q\} \models \delta(p,l)\}$ of all possible successors of p. We also set $Succ'(p) = Succ(p) \setminus \{p\}$. An automaton $\mathcal{A}$ is called* 1-weak *(or* linear *or* very weak*) if there exists an ordering $<$ on the set of states Q such that $q \in Succ'(p)$ implies $q < p$.*

In the following we use *A1W automaton* meaning 'alternating 1-weak Büchi automaton'. Further, instead of $S \models \delta(l,p)$ we write $p \xrightarrow{l} S$ and say that there is a transition leading from p to S under l. We also say that a state p has a *loop* whenever $p \in Succ(p)$.

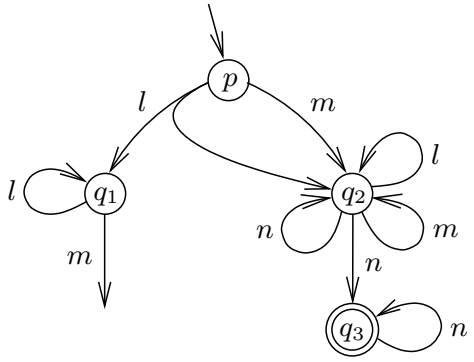


Figure 15: An A1W automaton accepting the language $l^*m(l+m+n)^*n^\omega$.

An A1W automaton $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ can be drawn as a graph; nodes are the states and every transition $p \xrightarrow{l} S$ is depicted as a branching edge labelled with l and leading from node p to the nodes in S. Edges that are not leading to any node correspond to the cases when S is the empty set, i.e. $\delta(p,l) = \top$. Initial and accepting states are indicated in the standard way. For example, Figure 15 depicts an automaton accepting the language $l^*m\{l,m,n\}^*n^\omega$.

6.2 LTL→A1W translation [MSS88, Var95]

Let φ be an LTL formula and Σ be an alphabet. The formula can be translated into an automaton $\mathcal{A}$ satisfying $L(\mathcal{A}) = L^\Sigma(\varphi)$, where $\mathcal{A} = (\Sigma, Q, \delta, q_\varphi, F)$ and

- the states $Q = \{q_\psi, q_{\neg\psi} \mid \psi \text{ is a subformula of } \varphi\}$ correspond to the subformulae of φ and their negations,
- the transition function δ is defined inductively as:

$$\begin{array}{lcl}
\delta(q_\top, l) & = & \top \\
\delta(q_a, l) & = & \top \text{ if } a \in l,\ \delta(q_a,l) = \bot \text{ otherwise} \\
\delta(q_{\neg\psi}, l) & = & \overline{\delta(q_\psi, l)} \\
\delta(q_{\psi\wedge\rho}, l) & = & \delta(q_\psi, l) \wedge \delta(q_\rho, l) \\
\delta(q_{\mathsf{X}\psi}, l) & = & q_\psi \\
\delta(q_{\psi\mathsf{U}\rho}, l) & = & \delta(q_\rho, l) \vee (\delta(q_\psi, l) \wedge q_{\psi\mathsf{U}\rho})
\end{array}$$

where $\overline{\alpha}$ denotes the positive boolean formula dual to α defined by induction on the structure of α as:

$$\begin{array}{lll} \overline{\top} = \bot & \overline{q_{\neg\psi}} = q_\psi & \overline{\beta \wedge \gamma} = \overline{\beta} \vee \overline{\gamma} \\ \overline{\bot} = \top & \overline{q_\psi} = q_{\neg\psi} & \overline{\beta \vee \gamma} = \overline{\beta} \wedge \overline{\gamma} \end{array}$$

- the set of accepting states is $F = \{q_{\neg(\psi \mathsf{U} \rho)} \mid \psi \mathsf{U} \rho \text{ is a subformula of } \varphi\}$.

One can readily confirm that the resulting automaton is always A1W automaton. Moreover, the number of its states is linear in the length of φ.

6.3 A1W→BA translation [Var95]

Let $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ be an alternating BA. A Büchi automaton accepting the language $L(\mathcal{A})$ can be constructed as $\mathcal{A}' = (\Sigma, Q', \delta', q_0', F')$, where

- $Q' = 2^Q \times 2^Q$,
- $q_0' = (\{q_0\}, \emptyset)$,
- $F' = \{\emptyset\} \times 2^Q$,
- $\delta'((U,V), l)$ is defined as:
  - if $U \neq \emptyset$ then

$$\begin{aligned} \delta'((U,V),l) = \{(U',V') \mid\ & \exists X, Y \subseteq Q \text{ such that} \\ & X \models \textstyle\bigwedge_{q \in U} \delta(q,l) \text{ and} \\ & Y \models \textstyle\bigwedge_{q \in V} \delta(q,l) \text{ and} \\ & U' = X \setminus F \text{ and } V' = Y \cup (X \cap F)\} \end{aligned}$$

  - if $U = \emptyset$ then

$$\begin{aligned} \delta'((\emptyset,V),l) = \{(U',V') \mid\ & \exists Y \subseteq Q \text{ such that} \\ & Y \models \textstyle\bigwedge_{q \in V} \delta(q,l) \text{ and} \\ & U' = Y \setminus F \text{ and } V' = Y \cap F)\} \end{aligned}$$

Intuitively, $\mathcal{A}'$ guesses labeling of each level of the computation tree of $\mathcal{A}$. Moreover, $\mathcal{A}'$ has to divide the set of states into two sets: states labeling paths with recent occurrence of an accepting states and the other states.

7 Partial order reduction

Let $\mathrm{LTL}_{-\mathsf{X}}$ denote the fragment of all LTL formalas without X operator. The partial order reduction method can be used in model checking of finite systems against $\mathrm{LTL}_{-\mathsf{X}}$ formulas to reduce the systems to be checked. The size of the reduced system is usually 3–99% of the original size [Pel06]. Hence, the model checking process is faster and consumes less memory. The method is best suited for asynchronous systems. The method is also called *model checking using representatives*. This section is based on Chapter 10 of [CGP99].

In this section we use a slightly different definition of a Kripke structure:

Definition 7.1. Kripke structure *is a tuple* (S, T, S_0, L)*, where*

- S *is a set of* states,
- T *is a set of* transitions *(for each* $\alpha \in T$*,* $\alpha \subseteq S \times S$*),*
- $S_0 \subseteq S$ *is a set of* initial states,
- $L : S \to 2^{AP}$ *is a* labelling function *associating to each state a set of atomic propositions that are true in the state.*

We also consider finite and deterministic systems only. Hence every $\alpha \in T$ is seen as a partial function $\alpha : S \to S$. A transition α is *enabled* in a state s if $\alpha(s)$ is defined. Otherwise, α is disabled in s. The set of transitions enabled in s is denoted by $enabled(s)$.

A *path* in a Kripke structure K starting from a state $s \in S$ is an infinite sequence $\pi = s_0, s_1, \ldots$ of states such that $s_0 = s$ and for every i there is a transition $\alpha_i \in T$ satisfying $\alpha_i(s_i) = s_{i+1}$. A path starting in a fixed state can be also identified with a sequence of transitions.

Let φ be an LTL formula and let $AP(\varphi)$ denote the finite set of atomic propositions occurring in φ. A path $\pi = s_0, s_1, \ldots$ of a Kripke structure (S, T, S_0, L) *satisfies* φ, written $\pi \models \varphi$, if $w \models \varphi$, where the word $w = w(0)w(1)\ldots$ is defined as $w(i) = L(s_i) \cap AP(\varphi)$ for all $i \geq 0$. A Kripke structure K *satisfies* φ, written $K \models \varphi$, if all paths starting from initial states of K satisfy φ.

The *model checking problem* is the problem to decide whether for a given Kripke structure K and a given specification formula φ it holds that $K \models \varphi$.

Let us consider a fixed Kripke structure $K = (S, T, S_0, L)$ and a fixed LTL$_{-\mathsf{X}}$ formula φ. The idea of partial order reduction method is to disable some transitions in some states of K in such a way that the resulting structure K' is equivalent to K in the sense that $K' \models \varphi$ if and only if $K \models \varphi$. More precisely, for every path π of the original system starting from an initial state there has to be a path π' in the reduced system starting from an initial state such that $\pi \models \varphi$ iff $\pi' \models \varphi$.

The reduced system is defined by so-called *ample* sets. For each state s, $ample(s) \subseteq enabled(s)$ is the set of transitions that are enabled in s in the reduced system. Calculation of ample sets needs to satisfy three goals:

1. The reduced system given by ample sets has to be correct, i.e. it has to satisfy φ iff the original system satisfies φ.
2. The reduced system should be substantially smaller than the original.
3. The overhead in calculating ample sets must be reasonably small.

First we formulate some conditions on ample sets. Then we prove that ample sets matching these conditions define a correct reduced system. Finally we discuss some heuristics for calculating such ample sets and we argue that these sets can be calculated on-the-fly.

7.1 Conditions on ample sets

First we recall the cornerstone of partial order reduction: the concept of *stuttering*. Let w be an infinite word. A letter $w(i)$ is called *redundant* iff $w(i) = w(i+1)$ and there is $j > i$ such that $w(i) \neq w(j)$. The *canonical form* of w is the infinite word obtained by deleting all redundant letters from w. Two infinite words w_1, w_2 are *stutter equivalent*, written $w_1 \sim w_2$, iff they have the same canonical form.

Theorem 7.2 ([Lam83])**.** *Any* LTL$_{-\mathsf{X}}$ *formula cannot distinguish between words that are stutter equivalent, i.e. the formula either satifies all such words or none of them.*

The stuttering equivalence can be extended to paths and Kripke structures. Paths $\pi = s_0, s_1, \ldots$ and $\pi' = s'_0, s'_1, \ldots$ are stutter equivalent with respect to a set $AP' \subseteq AP$, written $\pi \sim_{AP'} \pi'$, iff $w \sim w'$, where w, w' are defined as $w(i) = L(s_i) \cap AP'$ and $w'(i) = L(s'_i) \cap AP'$ for all $i \geq 0$. Two Kripke structures K, K' are stutter equivalent with respect to AP', written $K \sim_{AP'} K'$, iff

- K and K' have the same set of initial states,
- for each path π of K starting in an initial state s there exists a path π' of K' starting in the same initial state such that $\pi \sim_{AP'} \pi'$ and vice versa.

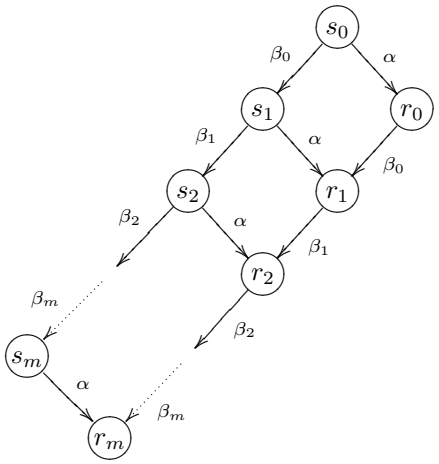


Figure 16: Transition α commuting with $\beta_0\beta_1 \dots \beta_m$.

From Theorem 7.2 we immediately get:

Corollary 7.3. *Let φ be an* $\mathrm{LTL}_{-\mathsf{X}}$ *formula and K, K' be Kripke structures such that $K \sim_{AP(\varphi)} K'$. Then $K \models \varphi$ iff $K' \models \varphi$.*

Hence, given a fixed $\mathrm{LTL}_{-\mathsf{X}}$ formula φ, for every set of stutter equivalent paths (with respect to $AP(\varphi)$) of the original Kripke structure it is sufficient to keep at least one representant of these paths in the reduced structure.

A transition $\alpha \in T$ is *invisible* with respect to a set of propositions $AP' \subseteq AP$ if for each pair of states $s, s' \in S$ such that $\alpha(s) = s'$ it holds that $L(s) \cap AP' = L(s') \cap AP'$. We always consider invisibility with respect to the set $AP(\varphi)$. A transition is *visible* if it is not invisible.

Definition 7.4. *An* independence *relation $I \subseteq T \times T$ is a symmetric and antireflexive relation satisfying the following two conditions for each state $s \in S$ and for each $(\alpha, \beta) \in I$:*

- *Enabledness: if $\alpha, \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$*
- *Commutativity: if $\alpha, \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$*

The dependency *relation D is the complement of I, namely $D = (T \times T) \setminus I$.*

When it is hard to check whether two transitions are independent or not, it is safe to assume that they are dependent.

We say that state s is *fully expanded* when $ample(s) = enabled(s)$.

The following four conditions ensure that the reduced system is stutter equivalent to the original one (with respect to $AP(\varphi)$).

C0 $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$

C1 Along every path in the original structure that starts in s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occuring first.

The condition C1 implies the following lemma.

Lemma 7.5. *The transitions in $enabled(s) \setminus ample(s)$ are all independent of those in $ample(s)$.*

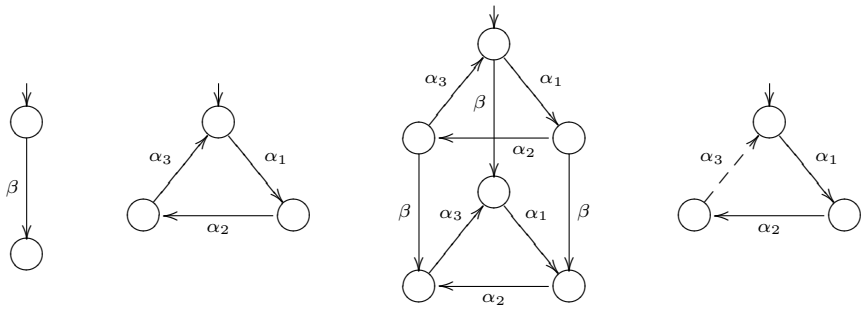


Figure 17: Two concurrent processes, full and (an invalid) reduced structures (β is visible, $\alpha_1, \alpha_2, \alpha_3$ are invisible, β is independent of $\alpha_1, \alpha_2, \alpha_3$, and $\alpha_1, \alpha_2, \alpha_3$ are interdependent).

Thanks to C1, all paths of the original structure starting in a state s and not included in the reduced structure have one of the following two forms:

- the path has a prefix $\beta_0\beta_1 \ldots \beta_m\alpha$, where $\alpha \in \mathit{ample}(s)$ and each β_i is independent of all transitions in $\mathit{ample}(s)$ including α.
- the path is an infinite sequence of transitions $\beta_0\beta_1 \ldots$ where each β_i is independent of all transitions in $\mathit{ample}(s)$.

Due to C1, after execution of a finite sequence of transitions $\beta_0\beta_1 \ldots \beta_m$ not in $\mathit{ample}(s)$ from s, all the transitions in $\mathit{ample}(s)$ remain enabled. Further, C1 implies that the sequence of transitions $\beta_0\beta_1 \ldots \beta_m\alpha$ executed from s leads to the same state as the sequence $\alpha\beta_0\beta_1 \ldots \beta_m$ (see Figure 16). As the sequence $\beta_0\beta_1 \ldots \beta_m\alpha$ is not included in the reduced system, we want $\beta_0\beta_1 \ldots \beta_m\alpha$ and $\alpha\beta_0\beta_1 \ldots \beta_m$ to be prefixes of stutter equivalent paths. This is quaranteed if α is invisible. Therefore we formulate condition C2:

C2 (invisibility) If s is not fully expanded, then every $\alpha \in \mathit{ample}(s)$ is invisible.

Conditions C0, C1 and C2 are not yet sufficient to guarantee that the reduced structure is stutter equivalent to the original one. There is a possibility that some transition will be delayed forever because of a cycle (see Figure 17). To remedy this problem, we add the following condition:

C3 (cycle condition) A cycle in reduced structure is not allowed if it contains a state in which some transition is enabled, but is never included in $\mathit{ample}(s)$ for any state s on the cycle.

Condition C0, C1, C2, and C3 are sufficient.

7.2 Example

Consider the following program for *mutual exclusion*:

P :: m : **cobegin** $P_0 \| P_1$ **coend**

P_0 :: l_0 : **while** *True* **do**
NC_0 : **wait**$(turn = 0)$;
CS_0 : $turn := 1$;
endwhile;

P_1 :: l_1 : **while** *True* **do**
NC_1 : **wait**$(turn = 1)$;

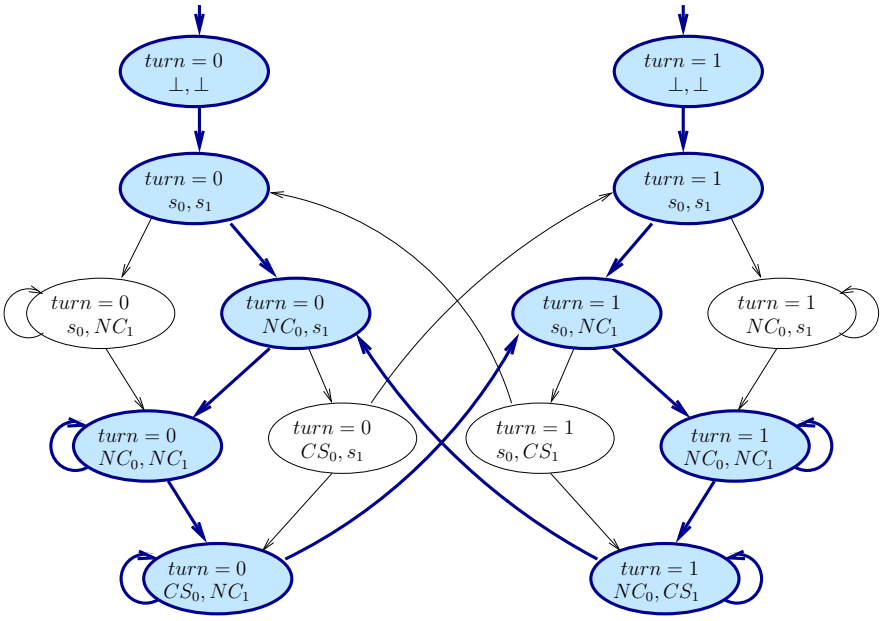


Figure 18: Reduced Kripke structure for the mutual exclusion program.

CS_1 : *turn* := 0;
endwhile;

Each process P_i has a *noncritical region* (when *program counter* pc_i has value NC_i) and a *critical region* (when $pc_i = CS_i$). The processes share the boolean variable *turn*. We would like to prove that, regardless the initial value of *turn*, the two processes cannot be in their critical regions at the same time. This desired property can be described by LTL$_{-\mathsf{X}}$ formula $\varphi = \mathsf{G}\neg((pc_0 = CS_0) \wedge (pc_1 = CS_1))$.

Figure 18 presents the Kripke structure corresponding to the program. Each state is labelled by values of *turn*, pc_0 and pc_1 (value of pc_0 and pc_1 is $\perp$ before the processes P_0, P_1 are started). The transition α corresponds to the parallel start of processes P_0, P_1 and transitions $\beta_i, \epsilon_i, \gamma_i, \delta_i$ to execution of the commands at program locations l_i, NC_i when *turn* $\neq i$, NC_i when *turn* $= i$, CS_i, respectively.

The system can be reduced before model checking against the property φ. The reduced system has to be stutter equivalent to the original one with respect to $AP(\varphi) = \{pc_0 = CS_0, pc_1 = CS_1\}$. Only the actions $\gamma_0, \gamma_1, \delta_0, \delta_1$ are visible with respect to $AP(\varphi)$.

The dependency relation is calculated according to the following rules.

- All of the transitions are dependent on α as it must be executed before any other transition in the program.
- Each transition is dependent on itself (since dependency is reflexive).
- Two transitions that change the same variable (including program counters) are dependent.
- If one transition sets a variable and the other checks that variable, then the transitions are dependent.

Thus, all transitions of the same process are interdependent. Also, pairs (γ_1, δ_0), (γ_0, δ_1), (ϵ_1, δ_0), (ϵ_0, δ_1), (δ_0, δ_1) are dependent.

The reduced system satisfying conditions C0–C3 is also given in Figure 18.

7.3 Correctness

Let φ be a fixed specification formula. The considered reduction is done with respect to the set $AP(\varphi)$. Let K be a full structure and K' be its reduced version satisfying the conditions C0–C3. We prove that $K \sim_{AP(\varphi)} K'$.

Given a (finite or infinite) sequence v of transitions, $vis(v)$ denotes the projection of v onto the visible transitions. Given two finite sequences v, w of transitions, we write $v \sqsubset w$ if v can be obtained from w by erasing one or more transitions. We write $v \sqsubseteq w$ whenever $v \sqsubset w$ or $v = w$.

For the rest of this subsection we extend the definition of *path* allowing both finite and infinite paths. By $\sigma \circ \eta$ we denote the path constructed by concatenation of a finite path σ and a (finite or infinite) path η ($\circ$ is applicable only if the last state $last(\sigma)$ of σ is the same as the first state of η). By $|\sigma|$ we denote the number of transitions in σ. Let $tr(\pi)$ denote the sequence of transitions on a path π.

Let π be an infinite path of K starting in some initial state. We construct an infinite sequence of infinite paths $\pi_0, \pi_1, \pi_2, \ldots$ where $\pi = \pi_0$. Each π_i is defined as $\sigma_i \circ \eta_i$ such that $|\sigma_i| = i$. Paths π_i are defined by induction. Clearly, $\pi_0 = \pi = \sigma_0 \circ \eta_0$ where σ_0 is just the first state of π and $\eta_0 = \pi$. The path $\pi_{i+1} = \sigma_{i+1} \circ \eta_{i+1}$ is constructed from $\pi_i = \sigma_i \circ \eta_i$ in the following way. Let $s_0 = last(\sigma_i)$ and

$$\eta_i = s_0 \stackrel{\alpha_0}{\rightarrow} s_1 \stackrel{\alpha_1}{\rightarrow} s_2 \stackrel{\alpha_2}{\rightarrow} \ldots.$$

There are two cases:

A $\alpha_0 \in ample(s_0)$. Then $\sigma_{i+1} = \sigma_i \circ (s_0 \stackrel{\alpha_0}{\rightarrow} s_1)$ and $\eta_{i+1} = s_1 \stackrel{\alpha_1}{\rightarrow} s_2 \stackrel{\alpha_2}{\rightarrow} \ldots$.

B $\alpha_0 \notin ample(s_0)$. By C2, all transitions in $ample(s_0)$ must be invisible. There are two cases:

B1 Some $\beta \in ample(s_0)$ appears on η_i after some sequence of independent transitions $\alpha_0 \alpha_1 \ldots \alpha_{k-1}$ (i.e. $\beta = \alpha_k$). Then there is a path

$$\xi = s_0 \stackrel{\beta}{\rightarrow} \beta(s_0) \stackrel{\alpha_0}{\rightarrow} \beta(s_1) \stackrel{\alpha_1}{\rightarrow} \ldots \stackrel{\alpha_{k-1}}{\rightarrow} \beta(s_k) \stackrel{\alpha_{k+1}}{\rightarrow} s_{k+2} \stackrel{\alpha_{k+2}}{\rightarrow} \ldots.$$

B2 Some $\beta \in ample(s_0)$ is independent of all the transitions in η_i. Then there is a path

$$\xi = s_0 \stackrel{\beta}{\rightarrow} \beta(s_0) \stackrel{\alpha_0}{\rightarrow} \beta(s_1) \stackrel{\alpha_1}{\rightarrow} \beta(s_2) \stackrel{\alpha_2}{\rightarrow} \ldots.$$

In both cases $\sigma_{i+1} = \sigma_i \circ (s_0 \stackrel{\beta}{\rightarrow} \beta(s_0))$ and η_{i+1} is obtained from ξ by removing the first transition $s_0 \stackrel{\beta}{\rightarrow} \beta(s_0)$.

Please note that the above construction uses conditions C0 and C1.

Lemma 7.6. *For every $0 \leq i \leq j$ it holds that:*

1. $\pi_i \sim_{AP(\varphi)} \pi_j$.
2. $vis(tr(\pi_i)) = vis(tr(\pi_j))$.
3. *Let ξ_i, ξ_j be prefixes of π_i, π_j such that $vis(tr(\xi_i)) = vis(tr(\xi_j))$. Then*

$$L(last(\xi_i)) \cap AP(\varphi) = L(last(\xi_j)) \cap AP(\varphi).$$

Proof. It is sufficient to consider the case where $j = i + 1$. The proof is trivial if π_{i+1} was constructed according to the case A as $\pi_i = \pi_{i+1}$. The proof for the cases B1 and B2 are straightforward. □

The following lemma is obvious.

Lemma 7.7. *Let σ be the infinite path constructed as the limit of the finite paths σ_i. Then σ belongs to the reduced structure K'.*

Now it remains to prove that the path σ is stutter equivalent to π. First we show that σ contains all the visible transitions of π, and in the same order.

Lemma 7.8. *Let α be the first transition of η_i. There exists $j > i$ such that α is the last transition of σ_j and, for all $i \leq k < j$, α is the first transition of η_k.*

Proof. If α is the first transition of η_k then it is clearly the last transition of σ_{k+1} (case A) or the first transition of η_{k+1} (case B). The condition C3 implies that the second case cannot hold for all $k \geq i$. □

Lemma 7.9. *Let γ be the first visible transition on η_i and $prefix_\gamma(\eta_i)$ be the maximal prefix of $tr(\eta_i)$ that does not contain γ. Then one of the following holds:*

- *γ is the first action of η_i and the last transition of σ_{i+1}, or*
- *γ is the first visible transition of η_{i+1}, the last transition of σ_{i+1} is invisible, and $prefix_\gamma(\eta_{i+1}) \sqsubseteq prefix_\gamma(\eta_i)$.*

Proof. If π_{i+1} was constructed from π_i by the case A, then the first case of the lemma holds: if the first action of η_i is visible, then $ample(s_0) = enabled(s_0)$ due to C2 and hence $\gamma \in ample(s_0)$.

Now assume that π_{i+i} was constructed according to B, i.e. there exists another action β that is appended to σ_i to form σ_{i+1}. Due to C2, β is invisible. There are three cases:

1. β appears in η_i before γ (case B1)
2. β appears in η_i after γ (case B1)
3. β is independent of all transitions of η_i (case B2)

In case (1), $prefix_\gamma(\eta_{i+1}) \sqsubset prefix_\gamma(\eta_i)$ while $prefix_\gamma(\eta_{i+1}) = prefix_\gamma(\eta_i)$ in cases (2) and (3). □

Lemma 7.10. *Let v be a prefix of $vis(tr(\pi))$. Then there exists a path σ_i such that $v = vis(tr(\sigma_i))$.*

Proof. By induction on the length of v. The base case ($|v| = 0$) is trivial. Assume that $v\gamma$ is a prefix of $vis(tr(\pi))$ and there is a path σ_i such that $vis(tr(\sigma_i)) = v$. We show that there is a path σ_j where $j > i$ and $vis(tr(\sigma_i)) = v\gamma$, i.e. we show that γ is eventually added to σ_j for some $j > i$ and that no other visible transition is added to σ_k for $i < k < j$. The construction implies that a visible transition can be added to the end of σ_k to form σ_{k+1} only if it is the first transition of η_k. Lemma 7.9 says that γ remains the first visible transition in paths η_k for $k > i$ unless it is being added to some σ_j. Moreover the sequence of transitions before γ can only shrink. Lemma 7.8 says that the sequence will eventually shrink. Thus, γ is eventually added to some σ_j. □

Theorem 7.11. *The structures K and K' are stutter equivalent with respect to $AP(\varphi)$.*

Proof. It is sufficient to show that, for every path π of K starting from an initial state, the path σ produced by the described construction is stutter equivalent to π with respect to $AP(\varphi)$. Note that σ starts from the same state as π, this state is also an initial state in K', and that σ is a path in K' due to Lemma 7.7.

Lemma 7.10 implies that $vis(tr(\pi)) = vis(tr(\sigma))$. Let v_i denote the prefix of $vis(tr(\pi)) = vis(tr(\sigma))$ of length i. Let $\pi|_{v_i}$ be the shortest prefix of π such that $vis(tr(\pi|_{v_i})) = v_i$ and let $\sigma|_{v_i}$ be the shortest prefix of σ such that $vis(tr(\sigma|_{v_i})) = v_i$. The definition of σ implies that $\sigma|_{v_i}$ is a prefix of some π_k. Lemma 7.6 (3) says that $L(last(\pi|_{v_i})) \cap AP(\varphi) = L(last(\sigma|_{v_i})) \cap AP(\varphi)$. Further, for every state s reachable from $last(\pi|_{v_i})$ or $last(\sigma|_{v_i})$ by a sequence of invisible actions it holds that $L(s) \cap AP(\varphi) = L(last(\pi|_{v_i})) \cap AP(\varphi) = L(last(\sigma|_{v_i})) \cap AP(\varphi)$. Hence, $\sigma \sim_{AP(\varphi)} \pi$. □

7.4 Calculating ample sets

First we discuss the complexity of checking conditions C0 to C3. Conditions C0 and C2 are *local* in the sense that their satisafaction in a state s depends just on sets $enabled(s)$ and $ample(s)$. C0 can be checked in constant time while C2 can be checked in linear time (with respect to the size of $ample(s)$). We focus on more complex non-local conditions C1 and C3.

Theorem 7.12. *Checking condition C1 for a state s and a set of transitions $T \subseteq enabled(s)$ is at least as hard as checking reachability for the original structure.*

Proof. It is easy to show that the problem whether a given state is reachable in a given structure can be reduced to checking condition C1. □

To avoid checking condition C1 for arbitrary subset of enabled transitions, we will give a procedure computing a set of transitions that is guaranteed to satisfy C1. The computed sets do not have to be optimal – there is a tradeoff between efficiency of computation and amount of reduction.

Condition C3 is also non-local. In contrast to C1, C3 refers only to the reduced structure. Instead of checking C3, we formulate a stronger condition which is easier to check.

Lemma 7.13. *Assume that C1 holds for all ample sets along a cycle in a reduced structure. If at least one state along the cycle is fully expanded, then C3 hold for this cycle.*

Proof. Lemma 7.5 says that transitions in $enabled(s) \setminus ample(s)$ are all independent of those in $ample(s)$. Hence, every transition in $enabled(s) \setminus ample(s)$ for some state s on a cycle is also enabled in the next state on the cycle. If the cycle contains a fully expanded state, then it surely satisfies C3. □

If we use depth-first search strategy to generate (and verify) a reduced structure, we can use the fact that every cycle in the reduced structure has to contain a *back edge* (i.e. an edge going to a state on the search stack) to replace C3 by the following stronger condition.

C3' If s is not fully expanded, then no transition in $ample(s)$ may reach a state that is on the search stack.

Now we give some heuristics for calculating ample sets. The algorithm depends on the model of computation. We consider processes with shared variables and message passing with queues. By $pc_i(s)$ we denote the program counter of process P_i in a state s. We use the following notation:

- $pre(\alpha)$ is a set including all transitions β such that there exists a state s for which $\alpha \notin enabled(s)$ and $\alpha \in enabled(\beta(s))$.
- $dep(\alpha) = \{\beta \mid (\beta, \alpha) \in D\}$ is the set of all transitions that are dependent on α.
- T_i is the set of transitions of process P_i.
- $T_i(s) = T_i \cap enabled(s)$.
- $current_i(s)$ is the set of all transitions of P_i that are enabled in some s' such that $pc_i(s) = pc_i(s')$. Hence, $T_i(s) \subseteq current_i(s)$.

In fact, we do not need to compute the sets $pre(\alpha)$ and $dep(\alpha)$ precisely. We preffer to efficiently compute over-approximations of these sets.

We construct $pre(\alpha)$ as follows:

- $pre(\alpha)$ includes the transitions of the processes that contain α and that can change a program counter to a value from which α can execute.

- If the enabling condition for α involves shared variables, then $pre(\alpha)$ includes all other transitions that can change these shared variables.
- If α sends or receives messages on some queue q, then $pre(\alpha)$ includes transitions of other processes that receive or send data through q, respectively.

We construct $dep(\alpha)$ as follows:

- Pairs of transitions that share a variable, which is changed by at least one of them, are dependent.
- Pairs of transitions belonging to the same process are dependent.
- Two send transitions that use the same message queue are dependent (sending a message may cause the queue to fill). Two receive transitions are also dependent.

Note that a pair of send and receive transitions in different processes are independent as they can potentially enable each other, but not disable.

An obvious candidate for $ample(s)$ is $T_i(s)$ (as transitions in $T_i(s)$ are interdependent, $ample(s)$ must include either all of them or none of them – see Lemma 7.5). To compute $ample(s)$, we check whether some $T_i(s) \neq \emptyset$ satisfies the conditions C1, C2, and C3'. If there is no such $T_i(s)$, we set $ample(s) = enabled(s)$.

Assume that condition C1 is violated for $ample(s) = T_i(s)$. This means that some transitions independent of those in $T_i(s)$ may be successively executed from s, eventually enabling a transition $\alpha \notin T_i(s)$ dependent on $T_i(s)$. The independent transitions preceding α cannot be in T_i since all transitions of P_i are considered as interdependent. There are two cases.

1. $\alpha \in T_j$ for some $i \neq j$. Then $dep(T_i(s)) \cap T_j \neq \emptyset$.
2. $\alpha \in T_i$. Let s' be the state where α is enabled. The transitions executed on the path from s to s' are independent of $T_i(s)$ and hence, are from other processes. Therefore $pc_i(s) = pc_i(s')$ and this implies $\alpha \in current_i(s)$. As $\alpha \notin T_i(s)$, we get $\alpha \in current_i(s) \setminus T_i(s)$. As $\alpha \notin T_i(s)$, there has to be a transition of $pre(\alpha)$ included in the sequence from s to s'. Hence, $pre(current_i(s) \setminus T_i(s))$ includes transitions of processes other than P_i.

The following function checks whether $ample(s) = T_i(s)$ satisfies C1. More precisely, if the function returns *true*, then C1 holds. Unfortunately, it may return *false* even if $T_i(s)$ satisfies C1.

```
function checkC1(s, P_i)
    forall P_i ≠ P_j do
        if dep(T_i(s)) ∩ T_j ≠ ∅ or pre(current_i(s) \ T_i(s)) ∩ T_j ≠ ∅ then
            return false
    return true
end function
```

Functions for checking C2 and C3'and for calculating $ample(s)$ are straightforward.

```
function checkC2(X)
    forall α ∈ X do
        if visible(α) then return false
    return true
end function
```

```
function checkC3'(s, X)
    forall α ∈ X do
        if onStack(α(s)) then return false
    return true
end function
```

```
function ample(s)
    forall P_i such that T_i(s) ≠ ∅ do
        if checkC1(s, P_i) and checkC2(T_i(s)) and checkC3'(s, T_i(s)) then return T_i(s)
    return enabled(s)
end function
```

8 LTL model checking of pushdown systems

In this section we demostrante the decidability of (state-based) LTL model checking problem for pushdown systems. Let us note that these systems can be used to precisely model sequential programs with procedure calls, recursion, and both local and global variables. This section is based on [EHRS00, Sch02].

Definition 8.1. *A* Pushdown system *is a triple* $\mathcal{P} = (P, \Gamma, \Delta)$*, where*

- P *is a finite set of* control locations*,*
- Γ *is a finite* stack alphabet*,*
- $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ *is a finite set of* transition rules*.*

We write $\langle q, \gamma \rangle \hookrightarrow \langle q', w \rangle$ instead of $((q, \gamma), (q', w)) \in \Delta$. Notice that we do not consider any input alphabet as we do not use pushdown systems as language acceptors.

A *configuration* of $\mathcal{P}$ is a pair $\langle p, w \rangle \in P \times \Gamma^*$, where w is a *stack content* (the topmost symbol is on the left). The set of all configurations is denoted by $\mathcal{C}$. An immediate successor relation on configurations is defined in standard way. *Reachability relation* $\Rightarrow \subseteq \mathcal{C} \times \mathcal{C}$ is the reflexive and transitive closure of the immediate successor relation, while $\stackrel{+}{\Rightarrow} \subseteq \mathcal{C} \times \mathcal{C}$ is the transitive closure of the immediate successor relation. Given a set $C \subseteq \mathcal{C}$ of configurations, we define the set of their predecessors as $pre^*(C) = \{c \in \mathcal{C} \mid \exists c' \in C \,.\, c \Rightarrow c'\}$.

We use a sort of finite automata to represent sets of configurations. These automata use Γ as an alphabet and P as a set of initial states (there is one initial state for every control location of the pushdown system). Formal definition follows.

Definition 8.2. *Given a pushdown system* $\mathcal{P} = (P, \Gamma, \Delta)$*, a* $\mathcal{P}$-automaton *(or simply* automaton*) is a tuple* $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ *where*

- Q *is a finite set of* states *such that* $P \subseteq Q$*,*
- $\delta \subseteq Q \times \Gamma \times Q$ *is a set of* transitions*,*
- $F \subseteq Q$ *is a set of* final states*.*

A (reflexive and transitive) *transition relation* $\rightarrow \subseteq Q \times \Gamma^* \times Q$ is again defined in a standard way. $\mathcal{P}$-automaton $\mathcal{A}$ represents the set of configurations

$$Conf(\mathcal{A}) = \{\langle p, w \rangle \mid \exists q \in F \,.\, p \xrightarrow{w} q\}.$$

A set of configurations of $\mathcal{P}$ is called *regular* if it is recognized by some $\mathcal{P}$-automaton.

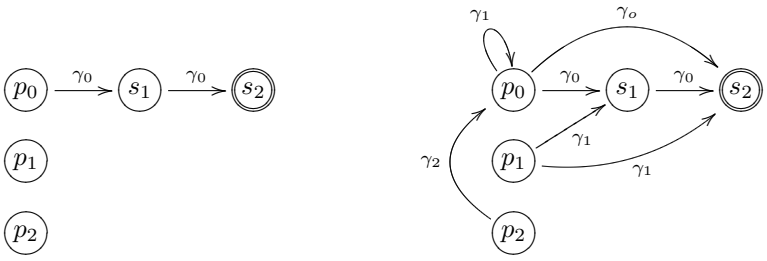
In the rest of this section, we use symbols $p, p', p'', \ldots$ to denote initial states of an automaton (i.e. elements of P). Non-initial states are denoted by $s, s', s'', \ldots$, and arbitrary states (initial or not) by $q, q', q'', \ldots$.

8.1 Computing $pre^*(C)$ for a regular set C

We show that, given a $\mathcal{P}$ and a $\mathcal{P}$-automaton $\mathcal{A}$ defining a regular set C of configurations, then the set $pre^*(C)$ is again regular and the corresponding automaton $\mathcal{A}_{pre^*}$ is effectively constructible.

Without loss of generality we assume that $\mathcal{A}$ has no transition leading to an initial state. The automaton $\mathcal{A}_{pre^*}$ is obtained from $\mathcal{A}$ by addition of new transitions according to the following *saturation rule*:

If $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ and $p' \xrightarrow{w} q$ in the current automaton, add a transition (p, γ, q).



$$\Delta = \{ \quad \langle p_0, \gamma_0 \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma_0 \rangle, \qquad \langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_0, \gamma_1 \rangle,$$
$$\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2 \gamma_0 \rangle, \qquad \langle p_0, \gamma_1 \rangle \hookrightarrow \langle p_0, \varepsilon \rangle \quad \}$$

Figure 19: Automata $\mathcal{A}$ (left) and $\mathcal{A}_{pre^*}$.

We apply this rule repeatedly until we reach a fixpoint (there is a fixpoint as the number of possible new transitions is finite). The resulting $\mathcal{P}$-automaton is $\mathcal{A}_{pre^*}$. Figure 19 provides an example of a $\mathcal{P}$-automaton $\mathcal{A}$ and the resulting automaton $\mathcal{A}_{pre^*}$.

A pushdown system is in *normal form* if every rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle$ satisfies $|w| \leq 2$. The efficient algorithm given in Figure 20 works only with pushdown systems in normal form. This is not a real restiction as any pushdown system can be put into this form with only linear size increase.

The algorithm computes just transitions of $\mathcal{A}_{pre^*}$. The rest of the automaton is identical to $\mathcal{A}$. The algorithm uses sets *rel* and *trans* containing the transitions that are known to belong to $\mathcal{A}_{pre^*}$: the set *rel* contains transitions that have already been examined. No transition is examined more than once. When we have a rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle$ and transitions $t_1 = (p', \gamma', q')$ and $t_2 = (q', \gamma'', q'')$ (where q, q' are arbitrary states), we have to add transition (p, γ, q''). We do it in such a way that

---

Input: a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ in normal form
a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ without transitions into P
Output: the set of transitions of $\mathcal{A}_{pre^*}$

```
1  rel := ∅; trans := δ; Δ' := ∅;
2  forall ⟨p,γ⟩ ↪ ⟨p',ε⟩ ∈ Δ do trans := trans ∪ {(p,γ,p')};
3  while trans ≠ ∅ do
4      pop t = (q,γ,q') from trans;
5      if t ∉ rel then
6          rel := rel ∪ {t};
7          forall ⟨p1,γ1⟩ ↪ ⟨q,γ⟩ ∈ (Δ ∪ Δ') do
8              trans := trans ∪ {(p1,γ1,q')};
9          forall ⟨p1,γ1⟩ ↪ ⟨q,γγ2⟩ ∈ Δ do
10             Δ' := Δ' ∪ {⟨p1,γ1⟩ ↪ ⟨q',γ2⟩};
11             forall (q',γ2,q'') ∈ rel do
12                 trans := trans ∪ {(p1,γ1,q'')};
13 return rel
```

Figure 20: Algorithm for computing $pre^*(C)$.

whenever we examine t_1, we check whether there is a corresponding $t_2 \in rel$ and we add an extra rule $\langle p, \gamma \rangle \hookrightarrow \langle q', \gamma'' \rangle$ to a set of such extra rules Δ'. This extra rule guarantees that if a suitable t_2 will be examined in the future, the transition (p, γ, q'') will be added.

Theorem 8.3. *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system and $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ be a $\mathcal{P}$-automaton. There exists an automaton $\mathcal{A}_{pre^*}$ recognizing $pre^*(Conf(\mathcal{A}))$. Moreover, $\mathcal{A}_{pre^*}$ can be constructed in $\mathcal{O}(|Q|^2 \cdot |\Delta|)$ time and $\mathcal{O}(|Q| \cdot |\Delta| + |\delta|)$ space.*

Proof. We can assume that every transition is added to *trans* at most once. This can be done (without asymptotic loss of time) by storing all transitions which are ever added to *trans* in an additional hash table.

Further, we assume that there is at least one rule in Δ for every $\gamma \in \Gamma$ (transitions of $\mathcal{A}$ under some γ not satisfying this assumption can be moved directly to *rel*).

The number of transitions in δ as well as the number of iterations of the **while**-loop is bounded by $|Q|^2 \cdot |\Delta|$.

Line 10 is executed for each combination of a rule $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle$ and a transition $(q, \gamma, q') \in trans$, i.e. at most $|Q| \cdot |\Delta|$ times. Hence, $|\Delta'| \leq |Q| \cdot |\Delta|$. For the loop starting at line 11, q' and γ_2 are fixed. Thus, line 12 is executed at most $|Q|^2 \cdot |\Delta|$ times.

Line 8 is executed for each combination of a rule $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle \in (\Delta \cup \Delta')$ and a transition $(q, \gamma, q') \in trans$. We already know that $|\Delta'| \leq |Q| \cdot |\Delta|$, hence line 8 is executed at most $\mathcal{O}(|Q|^2 \cdot |\Delta|)$ times.

As a conclusion, the algorithm takes $\mathcal{O}(|Q|^2 \cdot |\Delta|)$ time.

Memory is needed for storing *rel*, *trans*, and Δ'. The size of Δ' is in $\mathcal{O}(|Q| \cdot |\Delta|)$. Line 1 adds $|\delta|$ transitions to *trans*. Line 2 adds at most $|\Delta|$ transitions to *trans*. In lines 8 and 12, p_1 and γ_1 are given by the head of a rule in Δ (note that every rule in Δ' have the same head as some rule in Δ). Hence, lines 8 and 12 add at most $|Q| \cdot |\Delta|$ different transitions.

We directly get that the algorithm needs $\mathcal{O}(|Q| \cdot |\Delta| + |\delta|)$ space. As this is also the size of the result *rel*, the algorithm is optimal with respect to the memory usage. □

8.2 LTL model checking

This subsection deals with the *global state-based model checking problem* for LTL and pushdown processes, i.e.

> to compute the set of all configurations of a given pushdown system $\mathcal{P}$ that violate a given LTL formula φ (where a configuration c violates φ if there is a path starting from c and not satifying φ).

To talk about LTL properties, we need to enrich the formalism of pushdown systems with a *labelling function* $L : (P \times \Gamma) \to 2^{AP}$ assigning to each pair (p, γ) of a control location and a stack symbol a set of atomic propositions that are true of it. The labelling function can be directly extended to configurations such that $L(\langle p, \gamma w \rangle) = L(p, \gamma)$. In other words, the set of atomic propositions valid in a configuration is determined by the control location and the topmost stack symbol. Definition of a Kripke structure induced by a pushdown system enriched with a labelling function is now straightforward (the set of initial states is not important as we are interested in global model checking problem).

To show that the problem is decidable, we reduce it to *accepting run problem* first. As in the automata-based approach to LTL model checking of finite-state systems, we first translate $\neg\varphi$ into a corresponding Büchi automaton $\mathcal{B} = (2^{AP(\varphi)}, Q, \delta, q_0, F)$ and then we make a product of this automaton and the pushdown system. The result of the product is called *Büchi pushdown system*, which is basically a pushdown system extended with a set of accepting control locations. The Büchi pushdown system given

as product of a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ with a labelling function L, and a Büchi automaton $\mathcal{B} = (2^{AP(\varphi)}, Q, \delta, q_0, F)$, is defined as $\mathcal{BP} = ((P \times Q), \Gamma, \Delta', G)$, where

$$\langle (p,q), \gamma \rangle \hookrightarrow \langle (p',q'), w \rangle \in \Delta' \text{ if } \langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta \text{ and } q' \in \delta(q, L(p,\gamma) \cap AP(\varphi))$$

and $G = P \times F$. An *accepting run* of a $\mathcal{BP}$ is a path passing through some accepting control location infinitely often. Clearly, a configuration $\langle p, w \rangle$ of $\mathcal{P}$ violates φ if $\mathcal{BP}$ has an accepting run starting from $\langle (p, q_0), w \rangle$.

Hence, it is sufficient to solve the following *accepting run problem*:

> Compute the set $\mathcal{C}_a$ of configurations c of $\mathcal{BP}$ such that $\mathcal{BP}$ has an accepting run starting from c (i.e. a run which visits infinitely often configurations with control locations in G).

Now we define some terms useful for characterization of the configurations from which there are accepting runs.

Definition 8.4. *Let $\mathcal{BP} = (P, \Gamma, \Delta, G)$ be a Büchi pushdown system. The relation $\overset{r}{\Rightarrow}$ between configurations of $\mathcal{BP}$ is defined as follows: $c \overset{r}{\Rightarrow} c'$ if $c \Rightarrow \langle g, u \rangle \overset{+}{\Rightarrow} c'$ for some configuration $\langle g, u \rangle$ with $g \in G$.*

The head *of a transition rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ is the configuration $\langle p, \gamma \rangle$. A head $\langle p, \gamma \rangle$ is* repeating *if there exists $v \in \Gamma^*$ such that $\langle p, \gamma \rangle \overset{r}{\Rightarrow} \langle p, \gamma v \rangle$. The set of repeating heads of $\mathcal{BP}$ is denoted by R.*

Lemma 8.5. *Let c be a configuration of a Büchi pushdown system $\mathcal{BP} = (P, \Gamma, \Delta, G)$. $\mathcal{BP}$ has an accepting run starting from c if and only if there exists a repeating head $\langle p, \gamma \rangle$ such that $c \Rightarrow \langle p, \gamma w \rangle$ for some $w \in \Gamma^*$.*

Proof. The implication "$\Longleftarrow$" is obvious. Let us now assume that $\mathcal{BP}$ has an accepting run starting from from c and going through configurations $\langle p_0, w_0 \rangle, \langle p_1, w_1 \rangle, \langle p_2, w_2 \rangle, \ldots$. We construct a strictly increasing sequence of indices $i_0, i_1, \ldots$ as follows:

- $|w_{i_0}| = \min\{|w_j| \mid j \geq 0\}$
- $|w_{i_k}| = \min\{|w_j| \mid j > i_{k-1}\}$ for $k > 0$

In other words, once a configuration $\langle p_{i_k}, w_{i_k} \rangle$ is reached, the rest of the run never looks at or changes the bottom $|w_{i_k}| - 1$ stack symbols. Let γ_{i_k} be the topmost symbol of w_{i_k} for each $k \geq 0$. As the number of pairs (p_{i_k}, γ_{i_k}) is bounded by $|P \times \Gamma|$, there has to be a pair (p, γ) repeated ininitely many times. Moreover, since some $g \in G$ becomes a control location infinitely often, we can select two indeces $j_1 < j_2$ out of $i_0, i_1, \ldots$ such that

$$\langle p_{j_1}, w_{j_1} \rangle = \langle p, \gamma w \rangle \quad \overset{r}{\Rightarrow} \quad \langle p_{j_2}, w_{j_2} \rangle = \langle p, \gamma v w \rangle$$

for some $w, v \in \Gamma^*$. As w is never looked at or changed in the rest of the run, we have that $\langle p, \gamma \rangle \overset{r}{\Rightarrow} \langle p, \gamma v \rangle$. This proves the implication "$\Longrightarrow$". □

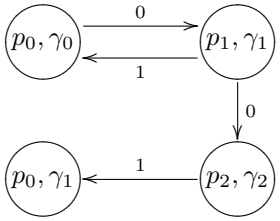
The lemma directly implies that the set of all configurations violating the considered formula φ can be computed as $pre^*(R\Gamma^*)$, where $R\Gamma^* = \{\langle p, \gamma w \rangle \mid \langle p, \gamma \rangle \in R, w \in \Gamma^*\}$. As the set R of repeating heads is finite, the set $R\Gamma^*$ is clearly regular. As we already have an algorithm computing $pre^*(C)$ for a given regular set C, the only remaining step to solve the global model checking problem is the algorithm computing the set R.

The problem of finding the repeating heads is reduced to a graph-theoretic problem. Given a $\mathcal{BP} = (P, \Gamma, \Delta, G)$, we construct a head reachability graph $\mathcal{G} = (P \times \Gamma, E)$ whose nodes are the heads of $\mathcal{BP}$. The set of edges $E \subseteq (P \times \Gamma) \times \{0, 1\} \times (P \times \Gamma)$ corresponds to the reachability relation between heads. We define $G(p) = 1$ if $p \in G$ and $G(p) = 0$ otherwise. E consists of the following edges:

If $\langle p, \gamma \rangle \hookrightarrow \langle p'', v_1 \gamma' v_2 \rangle$ and $\langle p'', v_1 \rangle \Rightarrow \langle p', \varepsilon \rangle$ then $((p, \gamma), G(p), (p', \gamma')) \in E$.
Moreover, if $\langle p'', v_1 \rangle \overset{r}{\Rightarrow} \langle p', \varepsilon \rangle$ then $((p, \gamma), 1, (p', \gamma')) \in E$.

The instances satisfying $\langle p'', v_1 \rangle \Rightarrow \langle p', \varepsilon \rangle$ or $\langle p'', v_1 \rangle \overset{r}{\Rightarrow} \langle p', \varepsilon \rangle$ can be found with the algorithm for $pre^*(\{\langle p', \varepsilon \rangle\})$ or its small modification, respectively.

Once the graph $\mathcal{G}$ is constructed, R can be computed by exploiting the fact that some head $\langle p, \gamma \rangle$ is repeating if and only if (p, γ) is part of a strongly connected component of $\mathcal{G}$ which has an internal edge labelled with 1.



$$\Delta = \{ \quad \langle p_0, \gamma_0 \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma_0 \rangle, \qquad \langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_0, \gamma_1 \rangle,$$
$$\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2 \gamma_0 \rangle, \qquad \langle p_0, \gamma_1 \rangle \hookrightarrow \langle p_0, \varepsilon \rangle \quad \}$$

Figure 21: The graph $\mathcal{G}$ for $\mathcal{BP} = (\{p_0, p_1, p_2\}, \{\gamma_0, \gamma_1, \gamma_2\}, \Delta, \{p_2\})$.

Figure 21 provides a graph $\mathcal{G}$ constructed for the Büchi pushdown system $\mathcal{BP} = (\{p_0, p_1, p_2\}, \{\gamma_0, \gamma_1, \gamma_2\}, \Delta, \{p_2\})$. The repeating heads are $\langle p_0, \gamma_0 \rangle$ and $\langle p_1, \gamma_1 \rangle$.

An efficient algorithm for computing the set R of repeating head is formulated in Figure 22. This algorithm assumes that the $\mathcal{BP}$ on input is in normal form.

The algorithm runs in two phases. During the first phase, it computes the automaton $\mathcal{A}_{pre^*}$ recognizing the set $pre^*(\{\langle p, \varepsilon \rangle \mid p \in P\})$. Every transition (p, γ, p') of $\mathcal{A}_{pre^*}$ signifies that $\langle p, \gamma \rangle \Rightarrow \langle p', \varepsilon \rangle$. As we also need the information whether $\langle p, \gamma \rangle \overset{r}{\Rightarrow} \langle p', \varepsilon \rangle$, we enrich the alphabet of $\mathcal{A}_{pre^*}$: transitions of the form (p, γ, p') are replaced by $(p, [\gamma, b], p')$ where b is a boolean. The meaning of a transition $(p, [\gamma, 1], p')$ should be that $\langle p, \gamma \rangle \overset{r}{\Rightarrow} \langle p', \varepsilon \rangle$.

The second phase of the algorithm constructs the graph $\mathcal{G}$, identifies its strongly conected components (e.g. using Tarjan's algorithm [Tar72]), and determines the set of repeating heads.

Theorem 8.6. *Let $\mathcal{BP} = (P, \Gamma, \Delta, G)$ be a Büchi pushdown system. The set of repeating heads R can be computed in $\mathcal{O}(|P|^2 \cdot |\Delta|)$ time and $\mathcal{O}(|P| \cdot |\Delta|)$ space.*

Proof. The first part of the algorithm is essentially the same as the algorithm computing $\mathcal{A}_{pre^*}$. The size of $\mathcal{G}$ is in $\mathcal{O}(|P| \cdot |\Delta|)$. Determining the strongly connected components takes linear time in the size of the graph [Tar72]. The same holds for searching each component for an internal 1-edge. □

Now we have all necessary components to solve the global model checking problem.

Theorem 8.7. *Let $\mathcal{P}$ be a pushdown system and φ be an LTL formula. The global model checking problem can be solved in $\mathcal{O}(|\mathcal{P}|^3 \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|\mathcal{P}|^2 \cdot |\mathcal{B}|^2)$ space, where $\mathcal{B}$ is a Büchi automaton corresponding to $\neg\varphi$.*

Proof. For proof see [Sch02]. □

Input: a Büchi pushdown system $\mathcal{BP} = (P, \Gamma, \Delta, G)$ in normal form
Output: the set of repeating heads in $\mathcal{BP}$

$rel := \emptyset$; $trans := \emptyset$; $\Delta' := \emptyset$;
forall $\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta$ **do**
  $trans := trans \cup \{(p, [\gamma, G(p)], p')\}$;
while $trans \neq \emptyset$ **do**
  pop $t = (p, [\gamma, b], p')$ from $trans$;
  if $t \notin rel$ **then**
    $rel := rel \cup \{t\}$;
    forall $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p, \gamma\rangle \in \Delta$ **do**
      $trans := trans \cup \{(p_1, [\gamma_1, b \vee G(p_1)], p')\}$;
    forall $\langle p_1, \gamma_1\rangle \overset{b'}{\longrightarrow} \langle p, \gamma\rangle \in \Delta'$ **do**
      $trans := trans \cup \{(p_1, [\gamma_1, b \vee b'], p')\}$;
    forall $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p, \gamma\gamma_2\rangle \in \Delta$ **do**
      $\Delta' := \Delta' \cup \{\langle p_1, \gamma_1\rangle \overset{b \vee G(p_1)}{\longrightarrow} \langle p', \gamma_2\rangle\}$;
      forall $(p', [\gamma_2, b'], p'') \in rel$ **do**
        $trans := trans \cup \{(p_1, [\gamma_1, b \vee b' \vee G(p_1)], p'')\}$;

$R := \emptyset$; $E := \emptyset$;
forall $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ **do** $E := E \cup \{((p, \gamma), G(p), (p', \gamma'))\}$;
forall $\langle p, \gamma\rangle \overset{b}{\longrightarrow} \langle p', \gamma'\rangle \in \Delta'$ **do** $E := E \cup \{((p, \gamma), b, (p', \gamma'))\}$;
forall $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta$ **do** $E := E \cup \{((p, \gamma), G(p), (p', \gamma'))\}$;
find all strongly connected components in $\mathcal{G} = ((P \times \Gamma), E)$;
forall components C **do**
  if C has a 1-edge **then** $R := R \cup C$;
return R

Figure 22: Algorithm for computing the set of repeating heads.

8.3 Notes

Similarly to the definition of $pre^*(C)$, we can also define the set $post^*(C)$ of successor of configurations in C. If C is regular, then $post^*(C)$ is regular as well.

Theorem 8.8. *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system and $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ be a $\mathcal{P}$-automaton. There exists an automaton $\mathcal{A}_{post^*}$ recognising $post^*(Conf(\mathcal{A}))$. Moreover, $\mathcal{A}_{post^*}$ can be constructed in $\mathcal{O}(|P| \cdot |\Delta| \cdot (|Q| + |\Delta|) + |P| \cdot |\delta|)$ time and space.*

9 Abstraction

Please use slides as the study material for this topic.

References

[BDLS80] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to

test the functional correctness of programs. In *Conference Record of the Seventh ACM Symposium on Principles of Programming Languages (POPL '80)*, pages 220–233. ACM Press, 1980.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[Lam83] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Proceedings of the IFIP Congress on Information Processing*, pages 657–667, Amsterdam, 1983. North-Holland.

[May98] Richard Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, Technische Universität München, 1998.

[May00] Richard Mayr. Process rewrite systems. *Information and Computation*, 156(1):264–286, 2000.

[MBT] Model-based testing. https://www.goldpractices.com/practices/mbt/.

[MP83] Zohar Manna and Amir Pnueli. How to cook a temporal proof system for your pet language. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 141–154. ACM Press, 1983.

[MSS88] David E. Muller, Ahmed Saoudi, and Paul E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science (LICS 1988)*, pages 422–427. IEEE Computer Society Press, 1988.

[Pel01] Doron Peled. *Software Reliability Methods*. Springer, 2001.

[Pel06] R. Pelánek. *Reduction and Abstraction Techniques for Model Checking*. PhD thesis, Faculty of Informatics, Masaryk University, Brno, 2006.

[Řeh07] Vojtěch Řehák. *On Extensions of Process Rewrite Systems*. PhD thesis, Masaryk University, Brno, 2007.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[Sch02] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[SDV] Static driver verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx.

[SPI] Spin model checker. http://www.spinroot.com.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput*, 1(2):146–160, 1972.

[Var95] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1995.