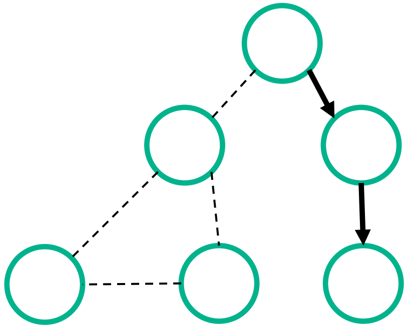
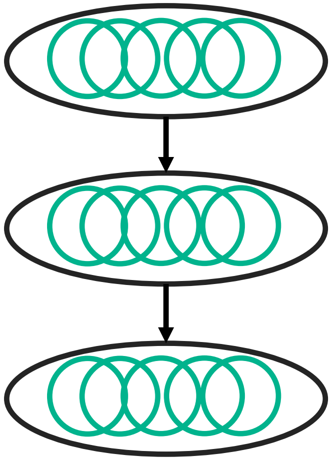
Using Genetic Programming for Software Reliability

Prof. Doron A. Peled
Bar Ilan University,
Israel

What is genetic algorithms [Holland71]?

- Heuristic search strategy.
- Beam search: progresses from one set of points ["generation" of "candidates"] to another, no backtracking.
- Uses ideas from genetic evolution: reproduction, mutation, probabilistic process.
- Parallelizable!





What can we do with a candidate?

Reproduction: candidate will *continue* to the next generation (possibly with the following changes).

Mutation: will make some probabilistic local changes.

Crossover: a pair of new candidates are formed by inheriting properties from a pair of parents.

Representation

- Each candidate is a string of fixed length corresponding to a chromosome.

1100111000111100100010111011111

Crossover

- Take two candidates and decide which position of letters to take from which parent.

01100101 10010111

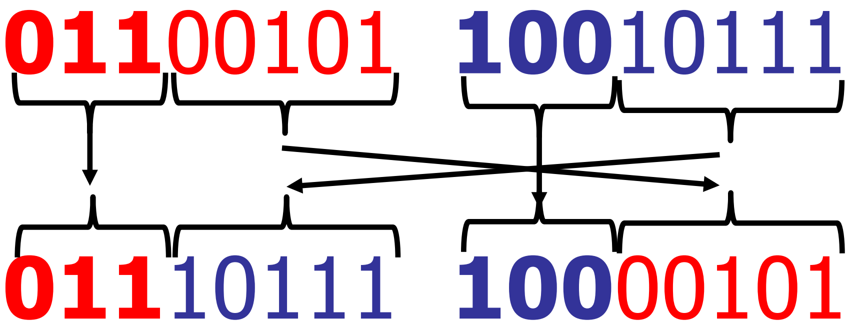
Crossover

- Take two candidates and decide which position of letters to take from which parent.

01100101 **100**10111

Crossover

- Take two candidates and decide which position of letters to take from which parent.



01100101 **100**10111

01110111 **100**00101

Mutation

- With some small probability *p*, decide whether to change each letter.

010101010

Mutation

- With some small probability *p*, decide whether to change each letter.

0**1**01**0**1010

Mutation

- With some small probability *p*, decide whether to change each letter.

0**1**01**0**1010

0**0**01**1**1010

Use fitness

- Fitness value (say, between 0 and 100) represents an estimate of how good is a candidate.
- It is important that fitess valus are dense (“smooth landscape”) to be able to distinguish between candidates.
- Candidates propagate from one generation to the next one proportional to the *ratio of their fitness and the average generation fitness*.

Use probability for:

- Generating initial candidates.
- Deciding which candidates will reproduce to the next generation. The probability is the relation between fitness value and average fitness of the generation.
- Deciding which candidates to apply crossover on, then the positions to select from each parent.
- Deciding whether to mutate a position in the string with some small probability *p*.

Combining it all

1. Generate at random the candidates of first generation.
2. Calculate fitness for candidates.
3. Stop if a “good” candidate was found.
4. Select candidates for reproduction based on fitness . Apply probabilistically mutation and crossover.
5. Repeat from Step 2 unless generation limit exceeded.
6. Can repeat process with a new random seed or change parameters.

Some math “schema theorem”.

- Consider only mutation (no crossover).
- We assume that a good solution is built from “good” building blocks (schemas) of the form e.g., 1*0*1, where 0 and 1 are constants, and * is a “wild card”.
- Thus, the scheme 1*0*1 has 4 candidates.
- There are 3^n schemes (but 2^{2^2} subsets).

Math (to show its not magic)…

- The expected number of times a candidate *x* will propagate to the next generation *t+1* is $f(x)/g(t)$: proportional to its fitness $f(x)$ divided by the average generation fitness $g(t)$.
- $N(s,t)$ – number of candidates of schema *s* in generation *t*. $u(s,t)$ – average fitness of candidates of schema *s* in generation *t*.
- Expected number of schema *s* candidates propagating to next generation:
-

$$\sum x \boxed{?} s,t \quad \uparrow ▦ f(x)/g(t) = u(s,t) \boxed{?} N(s,t)/g(t)$$

Math…

$$\sum x \boxed{?} s \boxed{?} ,t \boxed{?} \uparrow \text{▒} f(x)/g(t) = u(s,t) \boxed{?} N(s,t)/g(t)$$

- Order of scheme *s*: O(*s*) – number of non * elements.
- Probability of not ruining the scheme by mutation: $(1-p)^{O(s)}$

So, including the effect of mutation, we have

$$\mathrm{N}(s,t+1)=u(s,t) \boxed{?} N(s,t) \boxed{?} (1-p)\, O(s)/g(t)$$

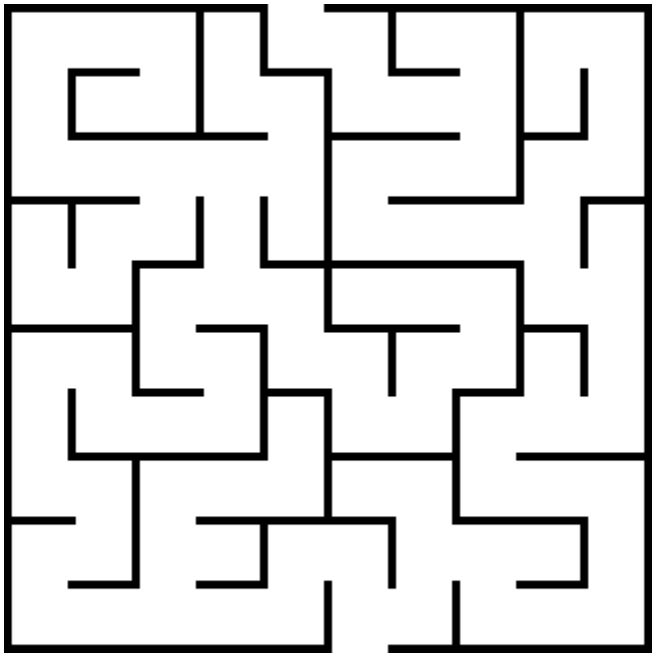
Can grow exponentially with the generations.

Some more good points

- Propagation can be parallelized.
- Propagation works on multiple schemes.
- If this did not convince you, well, some say its completely bullshit…

Classical example: solving a maze

- Candidates: string represents directions 00=left, 01=right, 10=down, 11=up.
- Fitness: follow a path. When cannot continue, use next move. Calculate the vertical+horizontal distance to end point. Fitness is reverse proportional to this value.



Use for testing [Godefroid,Khurshid]

- Test cases are represented as sequences of choices. [some concerns about fixed size representation].
- Fitness: shrinks with the number of enabled transitions along the test path; smaller number of transitions often lead to an error.
  Grows with the number of inline assertions along the path.
  Grows with the number of messages passed.
- Use *crossover* to generate new test cases.

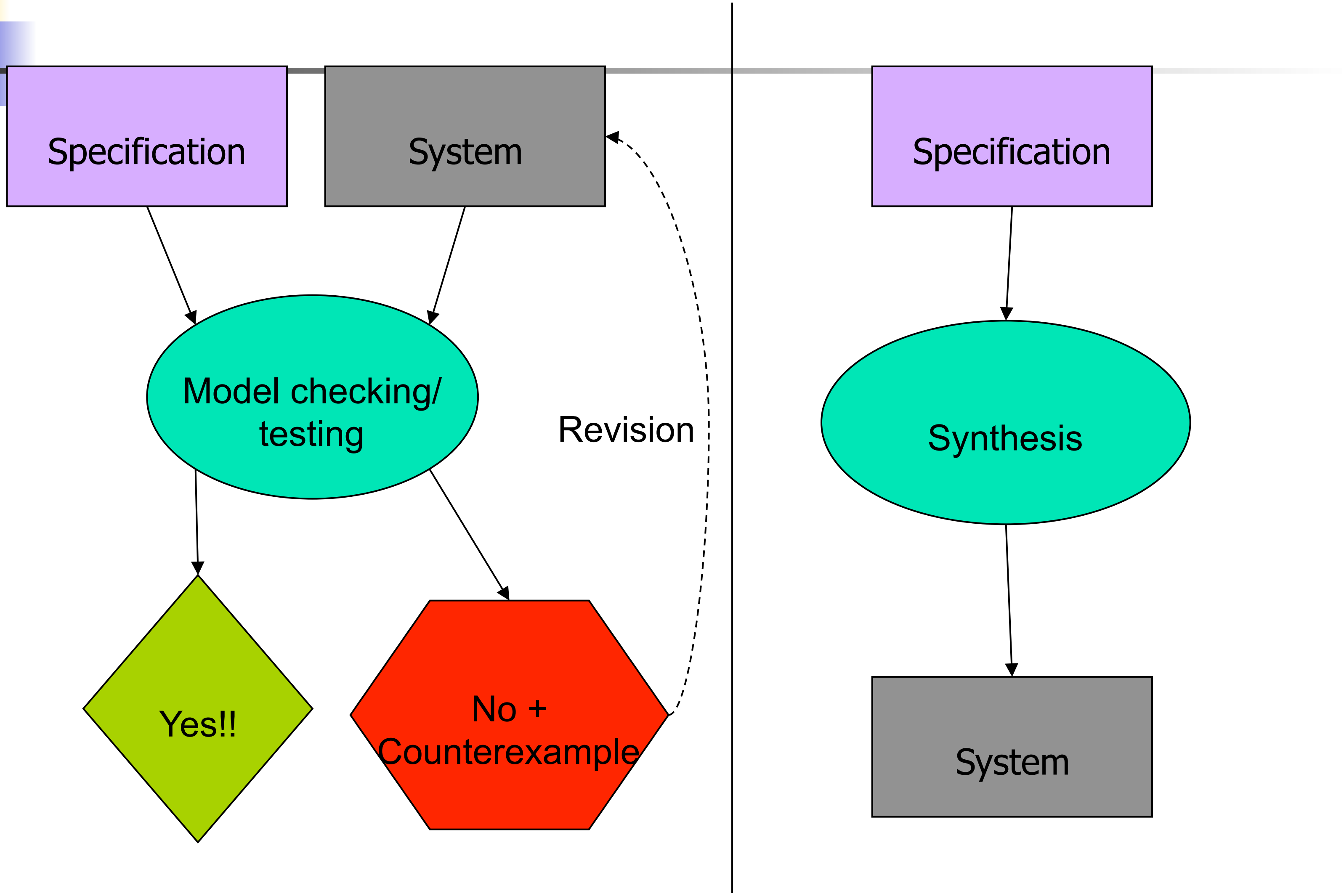
List of applications for genetic algorithms

- Airlines revenue management[1]
- Audio watermark insertion/detection
- Automated design = computer-automated design
- Automated design of mechatronic systems using bond graphs and genetic programming (NSF)
- Automated design of industrial equipment using catalogs of exemplar lever patterns
- Automated design of sophisticated trading systems in the financial sector
- Bayesian inference links to particle methods in Bayesian statistics and hidden Markov chain models[2][3]
- Code-breaking, using the GA to search large solution spaces of ciphers for the one correct decryption.[15]
- Computer architecture: using GA to find out weak links in approximate computing
- …….

My favorite application

- Find a strategy for Nash-style games, e.g.,
  - Rock-Paper-Scissors,
  - Prisoner’s dilemma,
  - Actually: financial market algorithms.

Why not synthesize the software directly from specification?



Complexity of sequential synthesis is high

- 2EXPTIME Complete for LTL specification.
- … But, there are provable systems where the number of states is doubly exponential.
- But must the size of a circuit that implements such a system be also doubly exponential?
- [Fearnly+Peled+Schewe]:
  If we knew, we could have decided whether EXPSPACE=2EXPTIME or not.

Concurrent synthesis

- Several processes, with some communication architecture. We want the system to satisfy some LTL property.
- [Pnueli+Rosner]: It is undecidable even to check whether there is a system with the given architecture that satisfies the LTL property.
- But under some strong assumptions (e.g., hierarchical systems) we can solve this [Pnueli+Rosner], [Finkbeiner+Schewe], [Thiagarajan+Madhusudan], [Kupferman+Vardi].

Mostly, negative results about synthesis of concurrent systems.

- Few positive results: …, it is decidable for some very limited architectures, mostly when there is a hierarchy between the processes.
- … in these cases, the complexity is very high …

A SHORT HISTORY OF MODERNIST PAINTING" (DETAIL) MARK TANSEY

How to construct a model from the specification?

- Synthesis
  - Transforms spec. directly to a model that satisfies it.
  - Hard (complexitywise) and sometimes undecidable.
- Brute-force enumeration [Bar David, Taubenfeld]
  - All possible programs of a specific domain and size are generated and model-checked.
  - All existing solutions will eventually be found.
  - Highly time-intensive. Not practical for programs with more than few lines of code.
- **Sketching** [Lazema]: small variants, resolved through SAT solving.

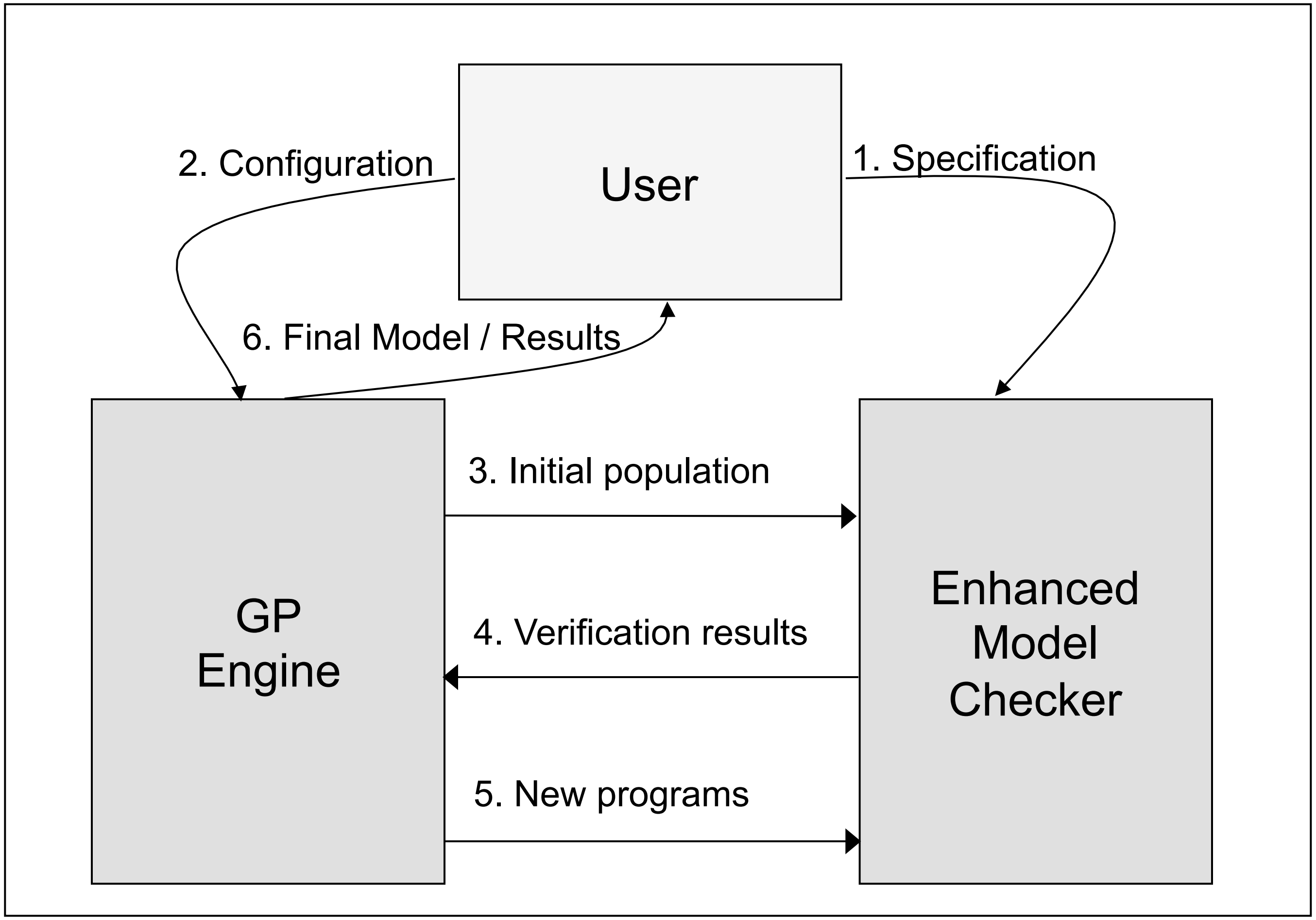
Genetic Programming

- A methodology for automatic programming inspired by Darwinian evolution [Koza 92].
- Used for automatic generation of programs in various fields.
- Mostly used for optimization related problems.
- Fitness is usually calculated by checking program performance against *test cases*.
- Less used for problems with a strict specification.
- There is no notion of fixed size chromosome. One usualy uses syntax trees.
- There are also schema theorems for genetic programming.

Main Steady-state GP Algorithm

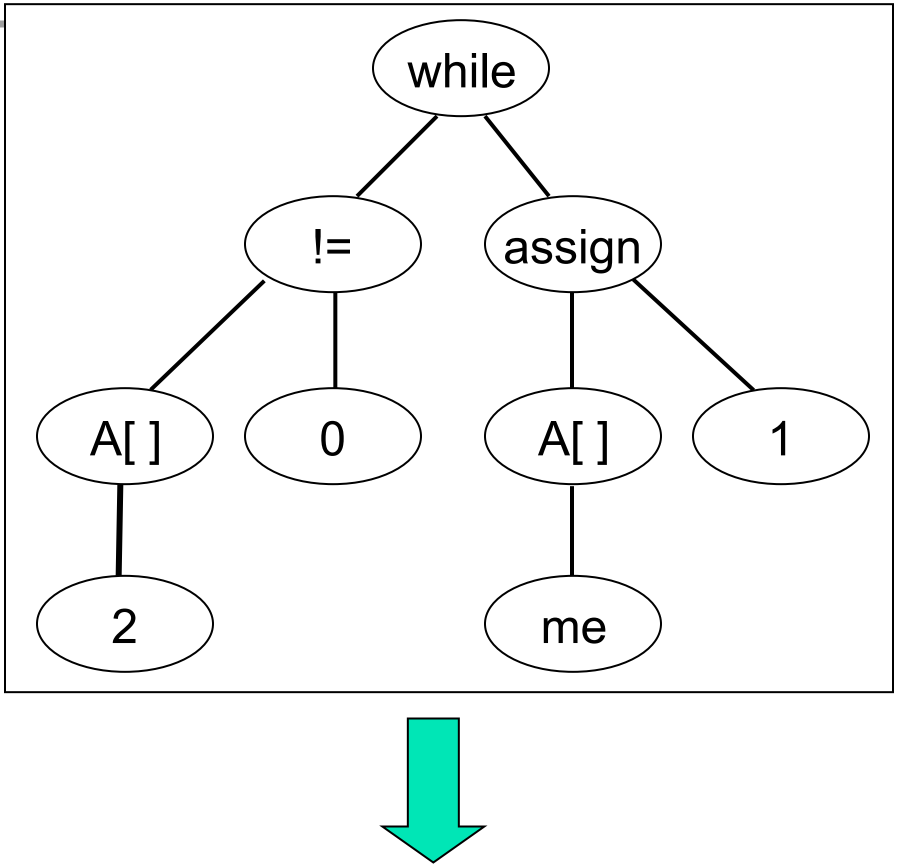
1. Create initial program population.
2. Randomly generate μ programs.
3. Create λ new programs by applying genetic operations to the above μ programs.
4. Calculate fitness function for μ + λ programs, and use it to select μ new programs.
5. Replace the old μ programs by the selected ones.
6. Repeat steps 2-5 until either:
    - a. a perfect solution is found, or
    - b. maximum allowed number of iterations is reached.

Combining GP & Model Checking



Program Representation

- Programs are represented as trees.
- Internal nodes represent expressions or instructions with parameters (assignment, while, if, block).
- Terminal nodes represent constants or expressions without any parameter (0, 1, 2, me, other).
- Strongly-typed GP is used [Montana 95].



While (A[2] != 0)
A[me] = 1

Initial Population Creation

- Population usually contains 100 – 1000 programs.
- Program are created recursively using the “grow” method [KOZA 92].
  - The root is randomly selected from instruction nodes.
  - Offspring are randomly selected from allowed node or terminals as long as rules are preserved.
  - If max tree depth is reached, a terminal must be chosen.

Genetic Operations

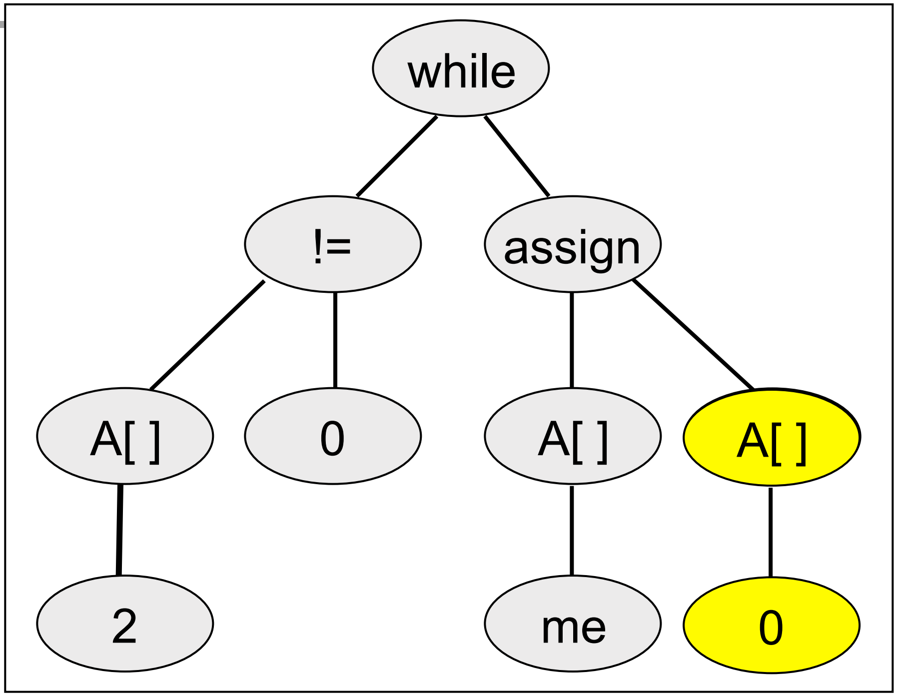
- At each iteration of the GP algorithm, the following genetic operations are applied to the selected programs:
  - Reproduction – programs are copied without any change
  - Mutation
  - Crossover

Mutation Operation

- The main operation we use.
- Allows performing small modifications to an existing program by the following method:
  - Randomly choose a program node (internal, or leaf).
  - According to the node type, apply one of the following operations with respect to the chosen node (strong typing must be kept):

Replacement Mutation type (a)

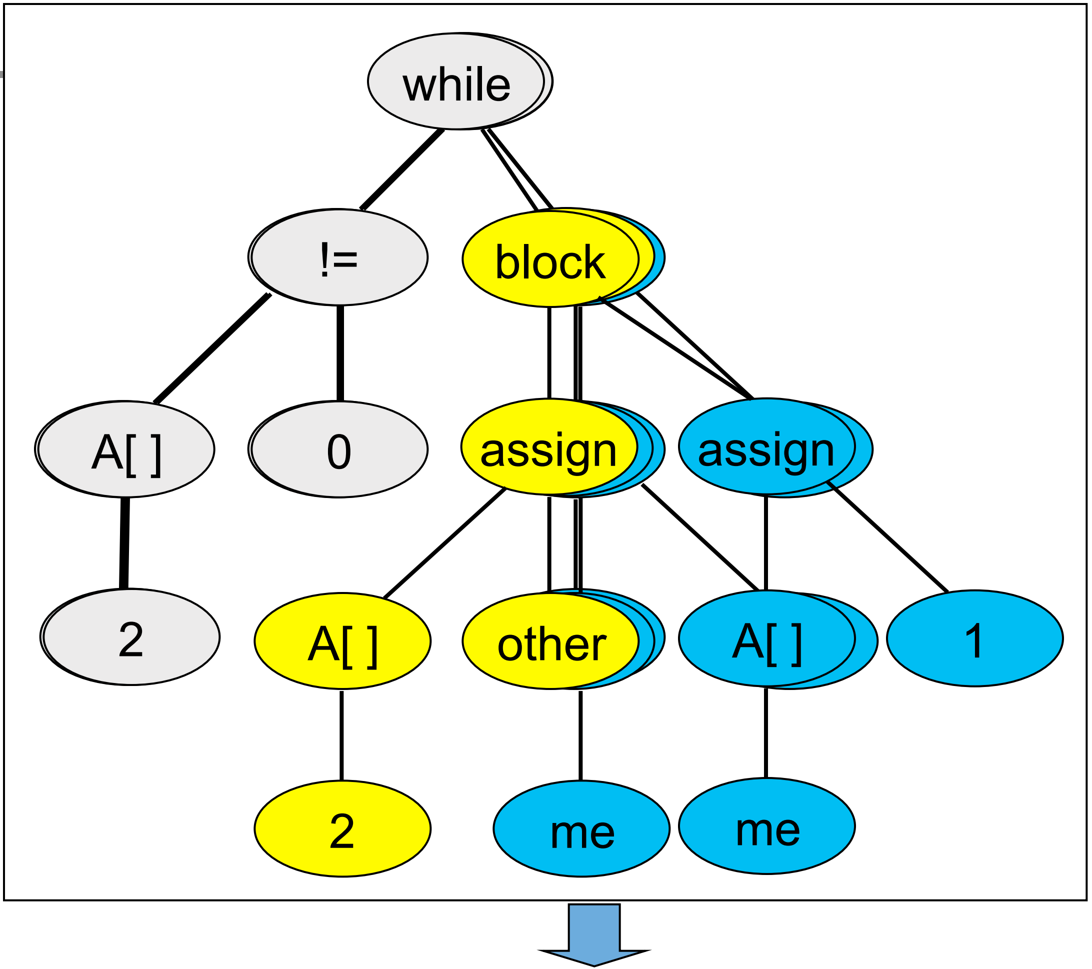
- Replace the sub-tree rooted by node with a new randomly generated sub-tree.
- Can change a single node or an entire sub-tree.



```
While (A[2] != 0)
    A[me] = A[0]
```

Insertion Mutation type (b)

- Add an immediate parent to the selected node.
- Randomly create other offspring to the new parent, if needed.
- According to the selected parent type, can cause:
  - Insertion of code,
  - Wrapping code with a while loop,
  - Extending Boolean expressions.



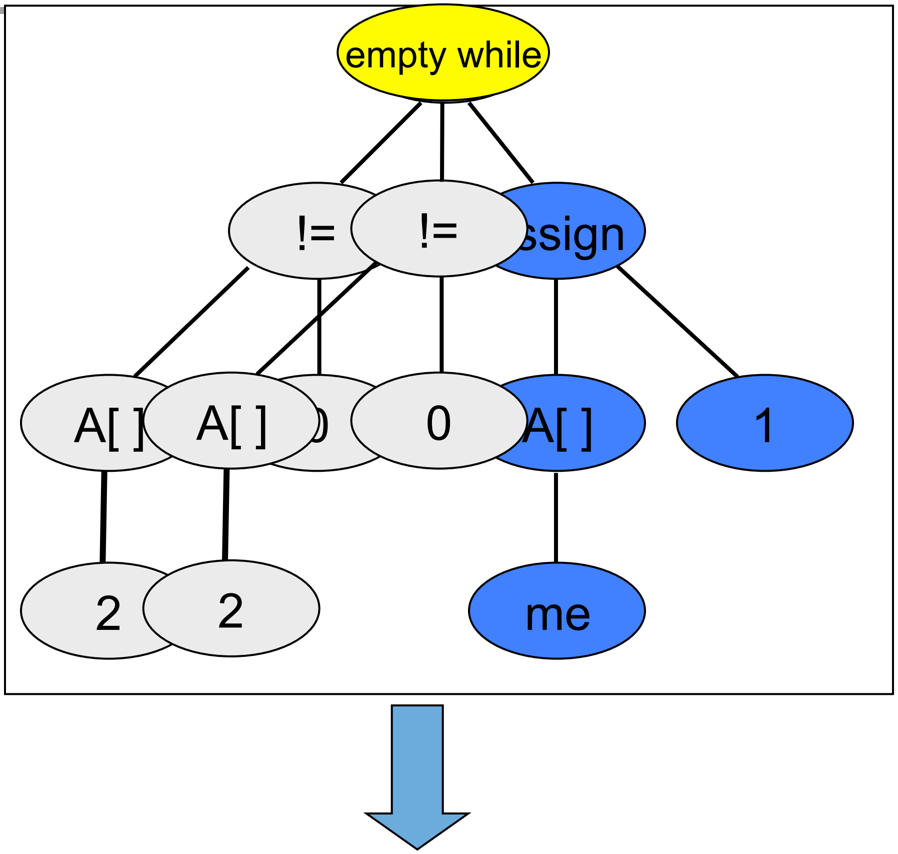
While (A[2] != 0)
A[2] = other
A[me] = 1

Reduction Mutation Type (c)

- Replace the selected node by one of its offspring.
- Delete the remaining offspring of the node.
- Has the opposite effect of the previous insertion mutation, and reduces the program size.

Deletion Mutation Type (d)

- Delete the sub-tree rooted by the node.
- Update ancestors recursively.



While (A[2] != 0)
A[me] = 1

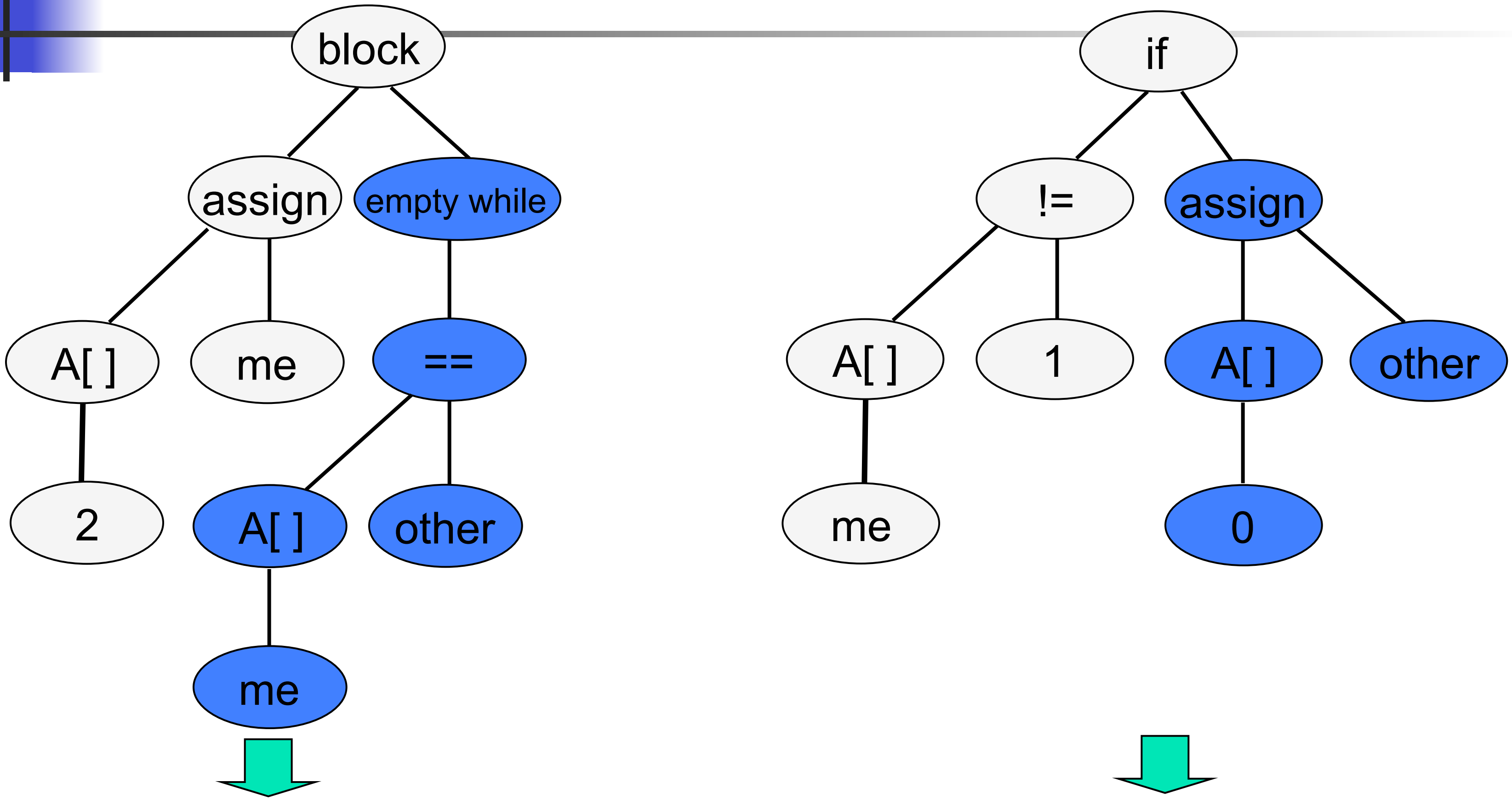
Mutation testing

- Mutation testing is used to check whether a test suite is good.
- Use mutations on the program, and check whether there is at least one test in the suite that can separate the behavior of the code with the mutation.
- Instead of providing fitness to the mutated code using a test suite, provide fitness to the test suite by mutating the code.
- Hypothesis: most programming errors are related to making a small error in switching something.
- If there is not, extend the test suite (e.g., based on path conditions, or GK genetic algorithm).

Crossover Operation

- Creates new programs by merging building blocks of two existing programs.
- Crossover steps are:
  - Randomly choose a node from the 1st program.
  - Randomly choose a node from the 2nd program, that has the same type as the 1st node.
  - Exchange between the sub-trees rooted by the two nodes, and use the two newly created programs.

Crossover Example



A[2] = me
a[0] = other

If (A[me] != 1)
while (a[me] == other)

Crossover (“excuses”)

- Heavily used by traditional GP [Koza].
- Tries to mimic biological process, but
- Unlike biology reproduction (and unlike GA), GP lacks the notion of “genes” [Banzhaf et al. 01].
- Often acts only as a macro-mutation.
- Various methods were developed in order to turn it into a more fruitful operation.
- Still, not a significant operation for small programs like those of Mutual Exclusion.
- Maybe my Phd student just did not want to implement it…

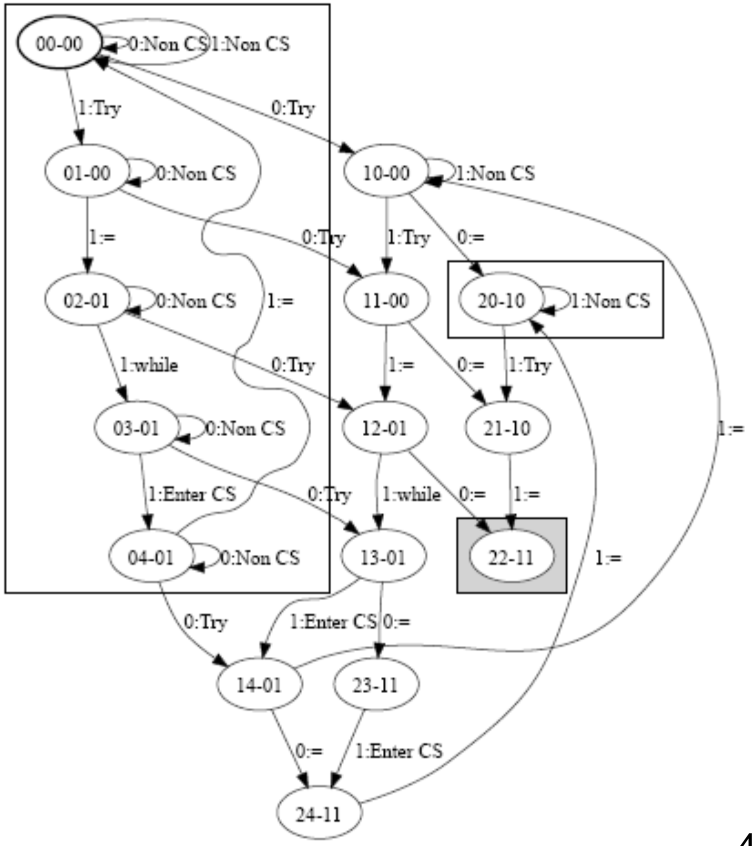
Selection

- At each iteration, selection is applied to all **$\mu + \lambda$** programs (over-production selection).
- Program are selected using a fitness-proportional (roulette) method [Holland 92].

Building Program's State-graph

- Each state consists of values of variables, program counters, buffers, etc.
- Edges represent atomic transitions caused by program instructions.

```
Non Critical Section
A[me] = 1
While (A[1] == A[other])
Critical Section
A[1] = other
```



- Can be built by a DFS algorithm.
- Can be decomposed into SCCs [Tarjan 72].

Example: The Mutual Exclusion Problem

- Originally described by [Dijkstra 65].
- Many variants and solutions exist.

```
while wi do
    Pre Protocol
    Critical Section
    Post Protocol
end while
```

- We want to automatically generate correct code for the pre and post protocol parts.

Specification

- We use Linear Temporal Logic (LTL) [Pnueli 77] to define specification properties.
- LTL formulas are interpreted over an infinite sequences of states, and consist of:
  - Propositional variables,
  - Logical connectives, such as $\neg$, $\wedge$, $\vee$, $\rightarrow$, and
  - Temporal operators, such as:
    - $\Diamond(p)$ – p will eventually occur.
    - �(p) – p always occurs.
- A model M satisfies a formula φ ($M \models \varphi$) if every (fair) run of M satisfies φ.

Specification

- Safety: $\neg(p_0$ in $CS_0 \wedge p_1$ in $CS_1)$
- Liveness: $(p_i$ in $preCS_i$ -> p_i in $CS_i)$
- Not enough: solution based on alternation requires always willing to enter critical section.
- That's why we added *wi* to control process' wishing to enter CS.

```
L0:While True do
  NC0:wait(Turn=0);
  CR0:Turn=1
endwhile ||
L1:While True do
  NC1:wait(Turn=1);
  CR1:Turn=0
endwhile
```

Instrument code as in RV to check LTL properties

- Use randomization for scheduling.
- Run k experiments where only one process wants to enter its critical section. In k_1 of them it succeeds.
- Run m experiments where both processes want. In m_1 of them only one succeeds, in m_2 both succeeds. $m_1 + m_2 \leq m$.
- Choose a, b, c, $a+c=100$, $b < c$.
- Fitness: $a \times k1/k + b \boxed{?} m1/m + c \boxed{?} m2/m$
- Further separate k_1, m_1, m_2 to cases where entering once or multiple times.

Model Checking and GP

- Can standard model checking results be used as a GP fitness function?
- Yes, but [Johnson 07]: a fitness function with just two values per proerpty is a poor one. Need more fitness levels.
  - No execution satisfies the property.
  - Some executions satisfy the property.
  - Every prefix of a bad execution can be continued to a good execution in the program (so, we made infinitely many “bad” choices”).
  - Statistically, at least/less than some portion of the executions satisfy the property.
  - All the executions satisfy the property.

Converting specification to ω-automaton

- Every LTL property can be converted into a Buchi automaton with a size exponential to the LTL formula size [Vardi & Wolper 94].
- For deterministic Streett automata, a determinization process is also required [Safra 88]. Expensive!! Avoid for probabilistic similar properties…
- May result in a doubly exponential blowup from LTL property.

The Model Checking Process [Vardi & Wolper 86]

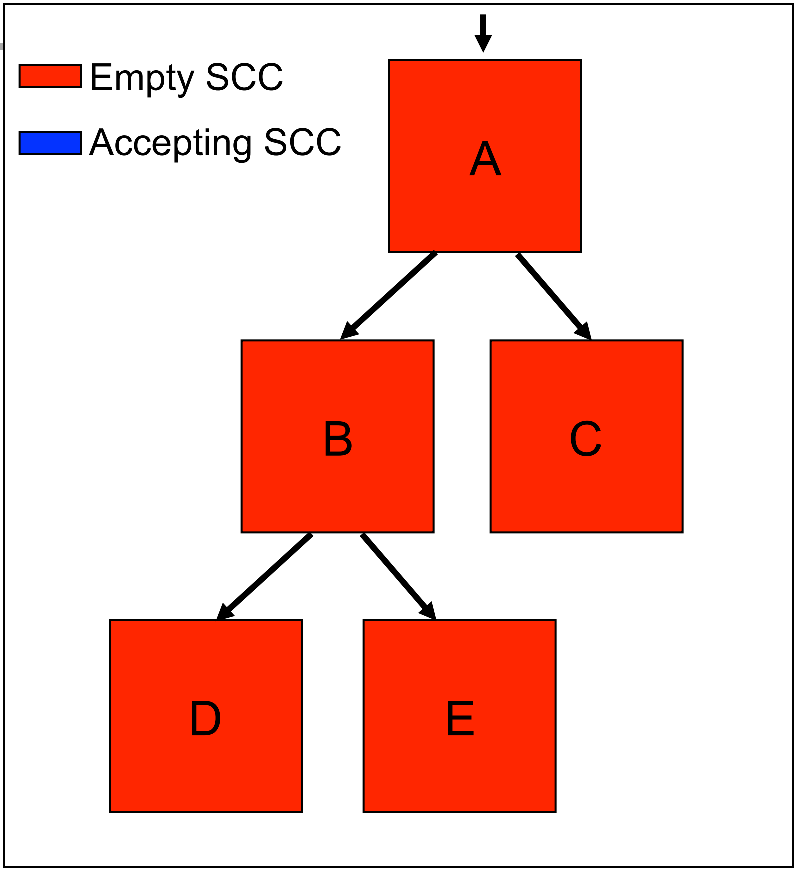
- Both model and speciation are converted to ω-automata over the same alphabet.
- The alphabet is 2^{AP}, where AP denotes a set of atomic propositions that may hold on the system states.
- Every word accepted by M (a fair run) should be accepted by the spec, therefore we have to check whether: $L(M) \subseteq L(\varphi($.

Model Checking and GP

- Can standard model checking results be used as a GP fitness function?
- Yes, but it was done so far with a limited success [Johnson 07].
- A fitness function with just two values is a poor one.
- We wish to analyze the model checking graph in order to quantify the level of satisfaction.

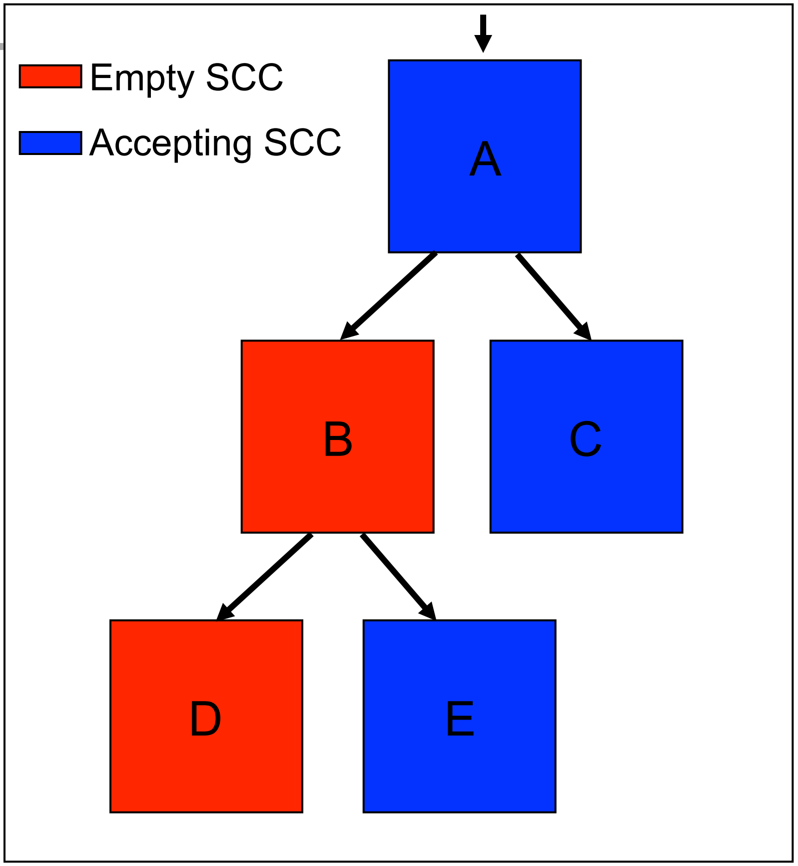
Fitness Level 0

- All SCCs are empty (not accepting).
- Property is never satisfied.
- No scheduler choices are needed.



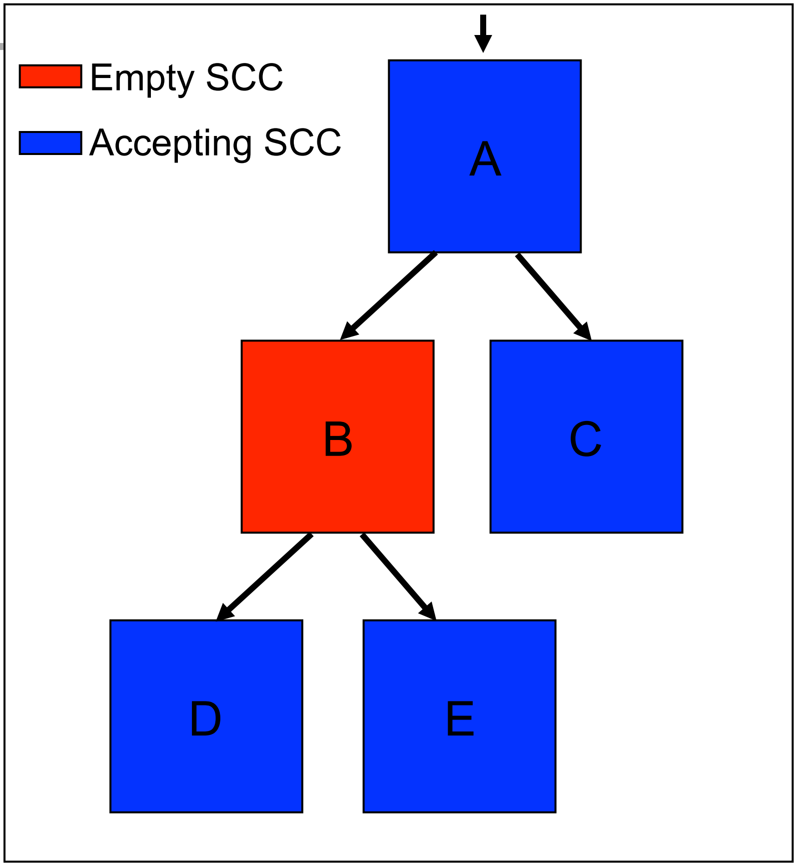
Fitness Level 1

- At least one accepting SCC.
- At least one empty bottom SCC.
- Finite number of scheduler choices can lead the execution into the empty BSCC (D in the example).
- The program will stay there forever.
- BSCC with only 1 node means a deadlock → gets worse score.



Fitness Level 2

- All BSCCs are accepting.
- At least one empty SCC.
- Infinite scheduler choices are needed for keeping the program inside the empty SCC (B in the example).



Fitness Level 3

- All executions are accepting.
- This can be checked by converting the negation of the property, and checking the emptiness of the intersection.

Overall Fitness Function

- Fitness levels & scores are calculated for each specification property.
- How to merge into a single fitness function?
- Naïve summing can bias the results, since some properties may be trivially satisfied when more basic properties are violated.
- Thus, spec. properties are divided into levels, starting from level 1 for most basic properties.
- As long as not all properties at level i are satisfied, properties at higher level gets fitness of 0.

Parsimony

- GP programs tend to grow up over time to the maximal allowed tree size (“bloating”).
- To avoid that, we use parsimony as a secondary fitness measure.
- Number of program nodes * small factor is subtracted from the fitness score.
- The factor should be carefully chosen.
  - Should encourage programs to reduce their size, but
  - Should not harm the evolutionary process.
- Therefore, programs cannot get a score of 100, but only get close to it. The run can be stopped when all properties are satisfied.
- Programs can be reduces either by mutations, or directly by detecting dead code by the model checking process, and then removing it.

“Vacuity”

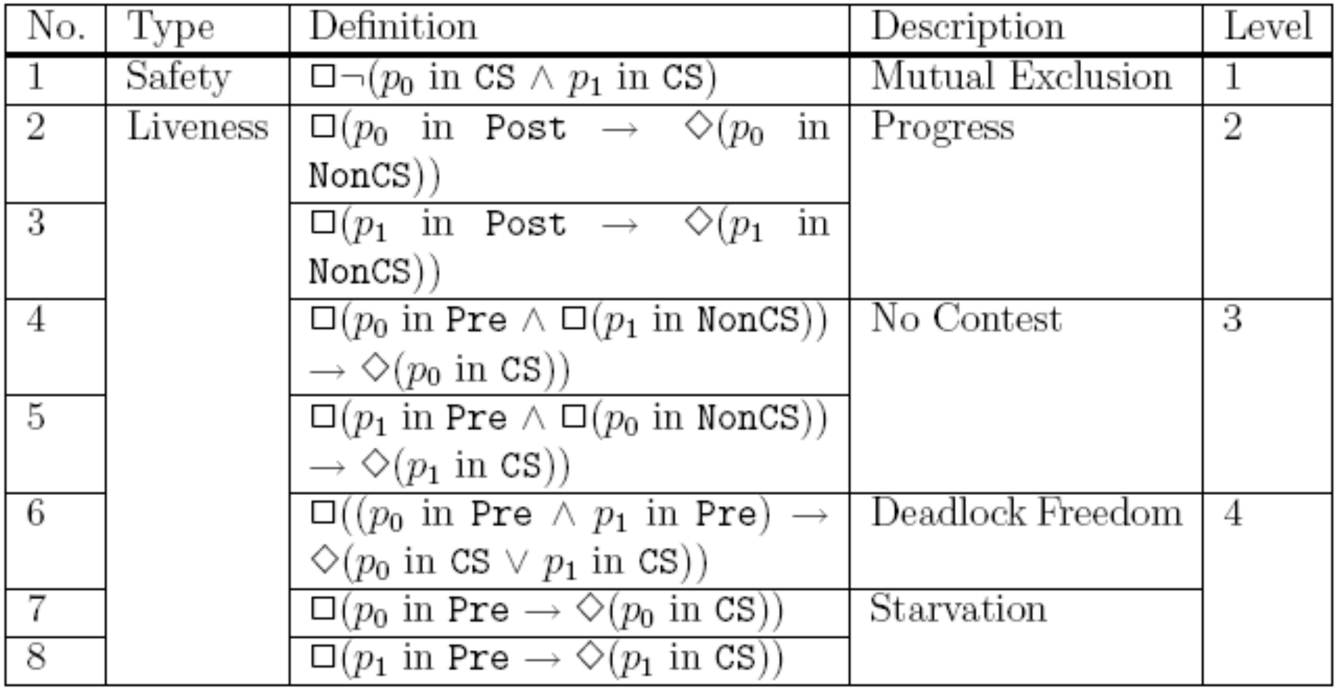
- A special care is needed for implication properties of the form �$(p \rightarrow \lozenge q)$.
- Some (or all) executions may be vacuously satisfied if p never happens.
- We are usually interested only on runs when p eventually occurs.
- Other runs are neither good nor bad. They are **irrelevant**.
- Thus, in these cases, the program automata is first intersected with the property $\lozenge p$.

The Mutual Exclusion Problem

- Many variants and solutions exist.
- Modeled using the following program parts inside a loop in each process:
  - Non Critical Section
  - Pre Protocol
  - Critical Section
  - Post Protocol
- We wish to automatically generate correct code for the pre and post protocol parts.

Spec. Properties

- The specification includes the following LTL properties:



No.	Type	Definition	Description	Level
1	Safety	$\Box\neg(p_0 \text{ in } \mathtt{CS} \wedge p_1 \text{ in } \mathtt{CS})$	Mutual Exclusion	1
2	Liveness	$\Box(p_0 \text{ in } \mathtt{Post} \rightarrow \Diamond(p_0 \text{ in } \mathtt{NonCS}))$	Progress	2
3		$\Box(p_1 \text{ in } \mathtt{Post} \rightarrow \Diamond(p_1 \text{ in } \mathtt{NonCS}))$		
4		$\Box(p_0 \text{ in } \mathtt{Pre} \wedge \Box(p_1 \text{ in } \mathtt{NonCS}) \rightarrow \Diamond(p_0 \text{ in } \mathtt{CS}))$	No Contest	3
5		$\Box(p_1 \text{ in } \mathtt{Pre} \wedge \Box(p_0 \text{ in } \mathtt{NonCS}) \rightarrow \Diamond(p_1 \text{ in } \mathtt{CS}))$		
6		$\Box((p_0 \text{ in } \mathtt{Pre} \wedge p_1 \text{ in } \mathtt{Pre}) \rightarrow \Diamond(p_0 \text{ in } \mathtt{CS} \vee p_1 \text{ in } \mathtt{CS}))$	Deadlock Freedom	4
7		$\Box(p_0 \text{ in } \mathtt{Pre} \rightarrow \Diamond(p_0 \text{ in } \mathtt{CS}))$	Starvation	
8		$\Box(p_1 \text{ in } \mathtt{Pre} \rightarrow \Diamond(p_1 \text{ in } \mathtt{CS}))$		

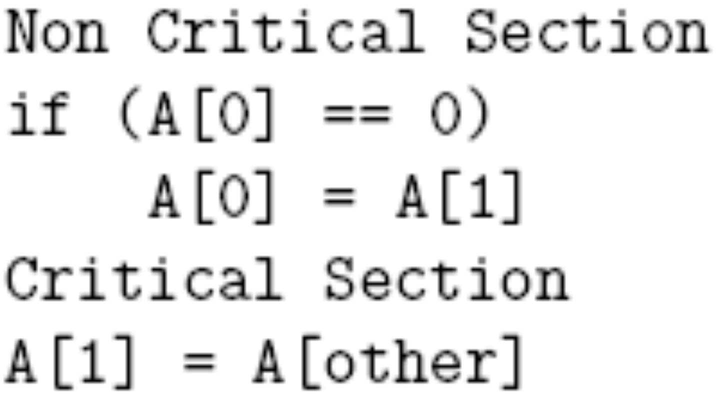
- Some properties are weaker/stronger than others, but they produce additional levels!

Runs Configuration

- The following parameters were used:
  - Population size: 150
  - Max number of iterations: 2000

In the following examples, we will show only the body of the while loop for one process (the other is symmetric).

An Example of a Run (1st variant)

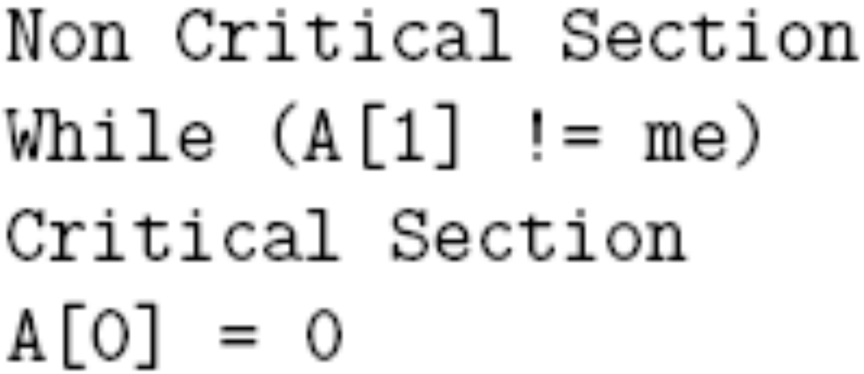


```
Non Critical Section
if (A[0] == 0)
     A[0] = A[1]
Critical Section
A[1] = A[other]
```

Score: 0.0

- Randomly created.
- Does not satisfy mutual exclusion property.
- Higher level properties are set to 0.

An Example of a Run (1st variant)

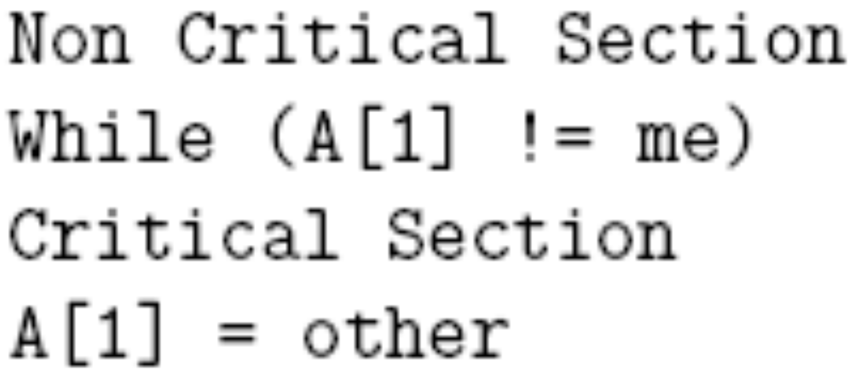


```
Non Critical Section
While (A[1] != me)
Critical Section
A[0] = 0
```

Score: **66.77**

- Randomly created.
- While loop guarantees mutual exclusion.
- Only process 0 can enter the critical section.

An Example of a Run (1st variant)

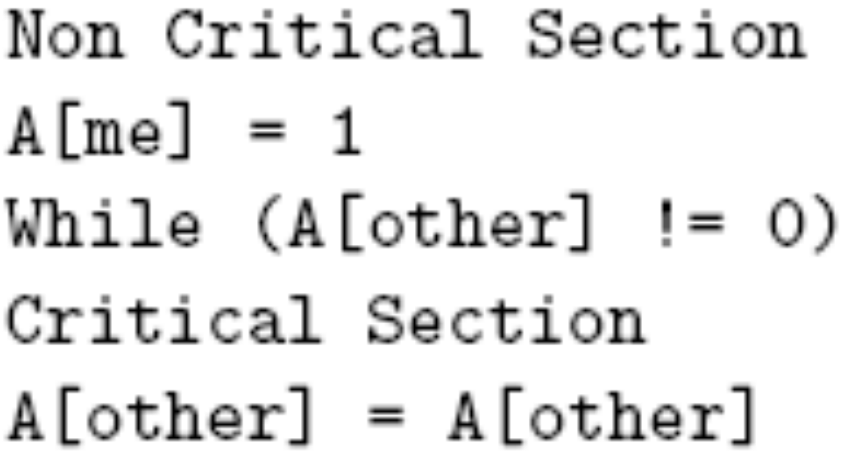


```
Non Critical Section
While (A[1] != me)
Critical Section
A[1] = other
```

Score: **75.77**

- Last line changed by a mutation.
- The naïve mutual exclusion algorithm.
- Processes uses a “turn” flag, but depend on each other.

An Example of a Run (1st variant)

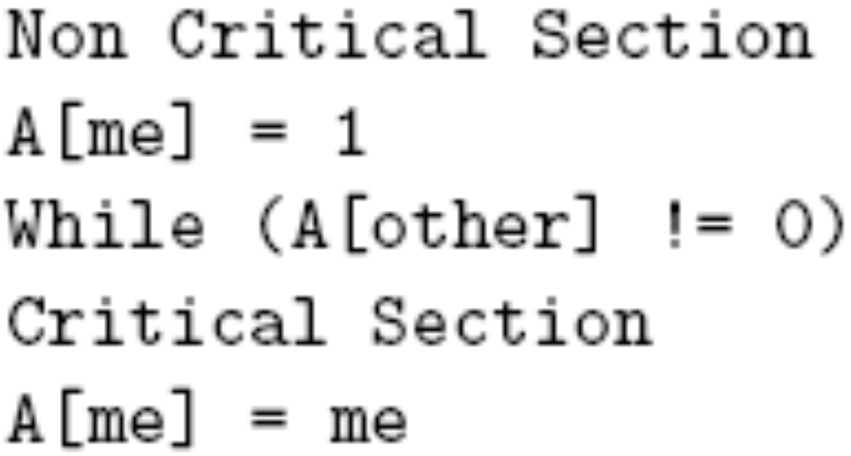


```
Non Critical Section
A[me] = 1
While (A[other] != 0)
Critical Section
A[other] = A[other]
```

Score: 70.17

- An important building block common to many algorithms.
- Each process set its own flag and wait for other’s flag, but
- The flag is not turned off correctly.
- Might eventually deadlock.

An Example of a Run (1st variant)

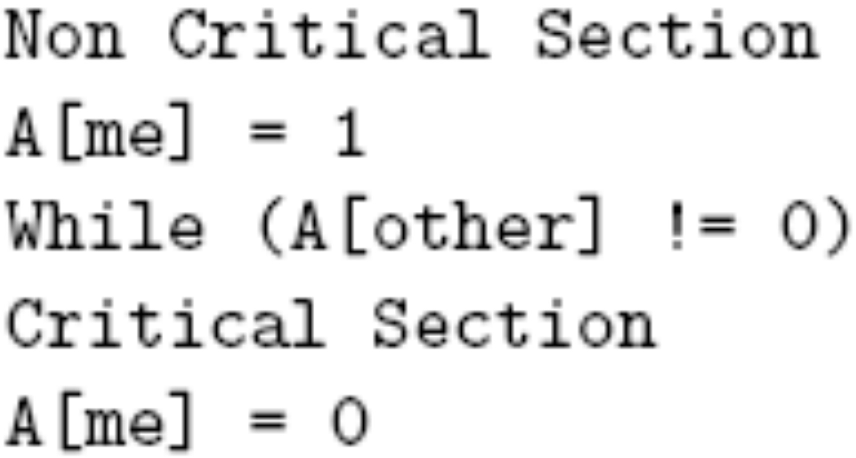


```
Non Critical Section
A[me] = 1
While (A[other] != 0)
Critical Section
A[me] = me
```

Score: 76.10

- Last line is replaced by a mutation.
- Now, process 0 correctly turns its flag off.
- Property 5 is fully satisfied

An Example of a Run (1^{st} variant)

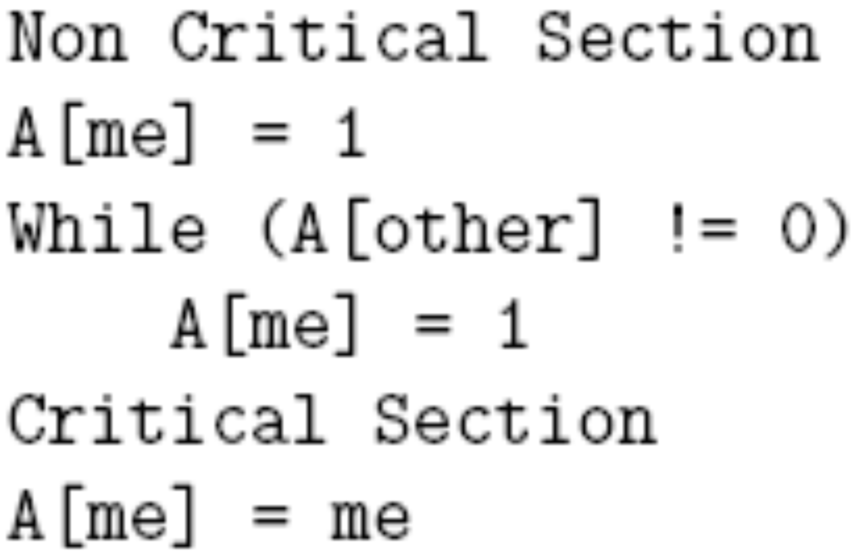


```
Non Critical Section
A[me] = 1
While (A[other] != 0)
Critical Section
A[me] = 0
```

Score: **92.77**

- A single node is changed by a mutation.
- Both processes turn off their flag.
- Properties 4 and 5 are fully satisfied.
- Still, deadlock occurs if both processes try to enter simultaneously.

An Example of a Run (1st variant)



```
Non Critical Section
A[me] = 1
While (A[other] != 0)
    A[me] = 1
Critical Section
A[me] = me
```

Score: **93.20**

- A mutation added a line to the empty while loop.
- This turns the deadlock into a livelock, and causes a slight fitness improvement.

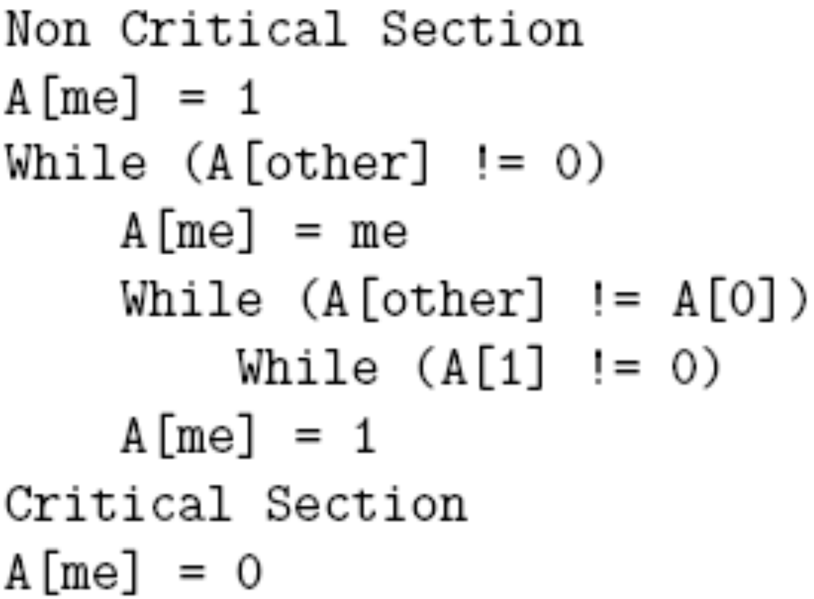
An Example of a Run (1^{st} variant)

```
Non Critical Section
A[me] = 1
While (A[other] != 0)
    A[me] = me
    A[me] = 1
Critical Section
A[me] = 0
```

Score: 94.37

- Another line is added to the while loop.
- No more dead or live locks, but property can still be violated by some infinite scheduler choices.

An Example of a Run (1st variant)

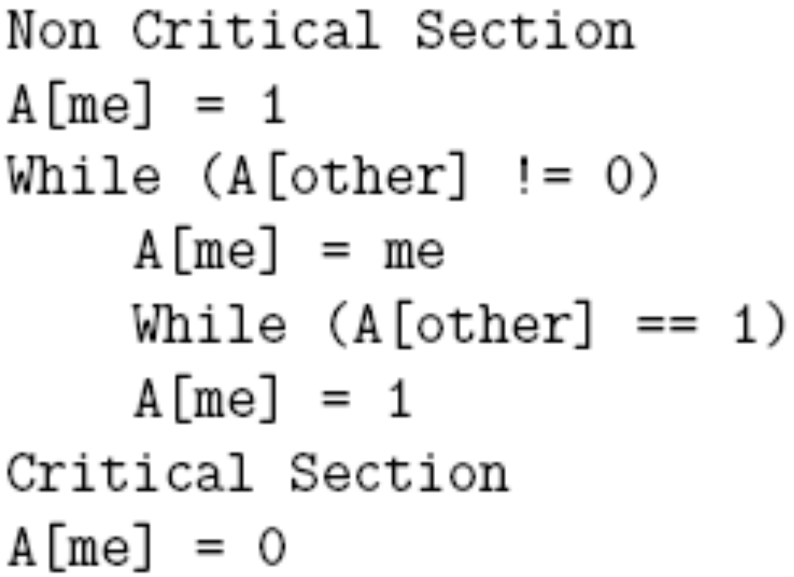


```
Non Critical Section
A[me] = 1
While (A[other] != 0)
    A[me] = me
    While (A[other] != A[0])
        While (A[1] != 0)
    A[me] = 1
Critical Section
A[me] = 0
```

Score: **96.50**

- Created by some random mutations.
- All properties are satisfied.
- Still, not the shortest solution.

An Example of a Run (1^{st} variant)



```
Non Critical Section
A[me] = 1
While (A[other] != 0)
    A[me] = me
    While (A[other] == 1)
    A[me] = 1
Critical Section
A[me] = 0
```

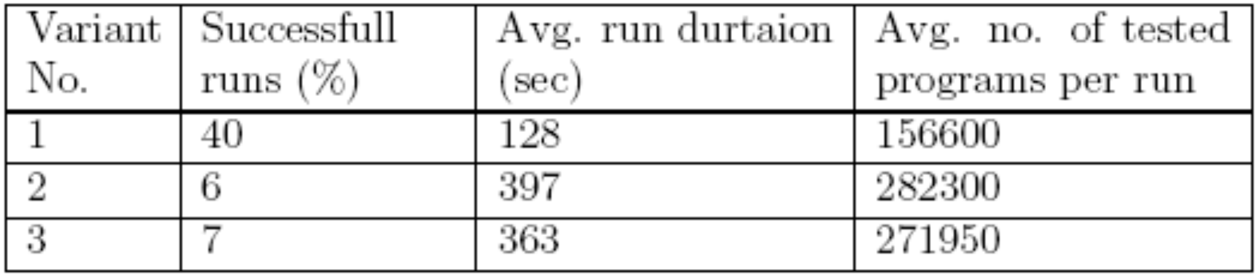
Score: **97.10**

- Created by more mutations.
- The shortest found algorithm.
- Identical to the known “One bit protocol” [Burns & Lynch 93].

More experiments

- Successfully found Dekker's algorithm. [Dijkstra 65].
- Successfully found Peterson’s algorithm. [Peterson & Fisher 77].
- Found a shorter algorithm than Dekker's.

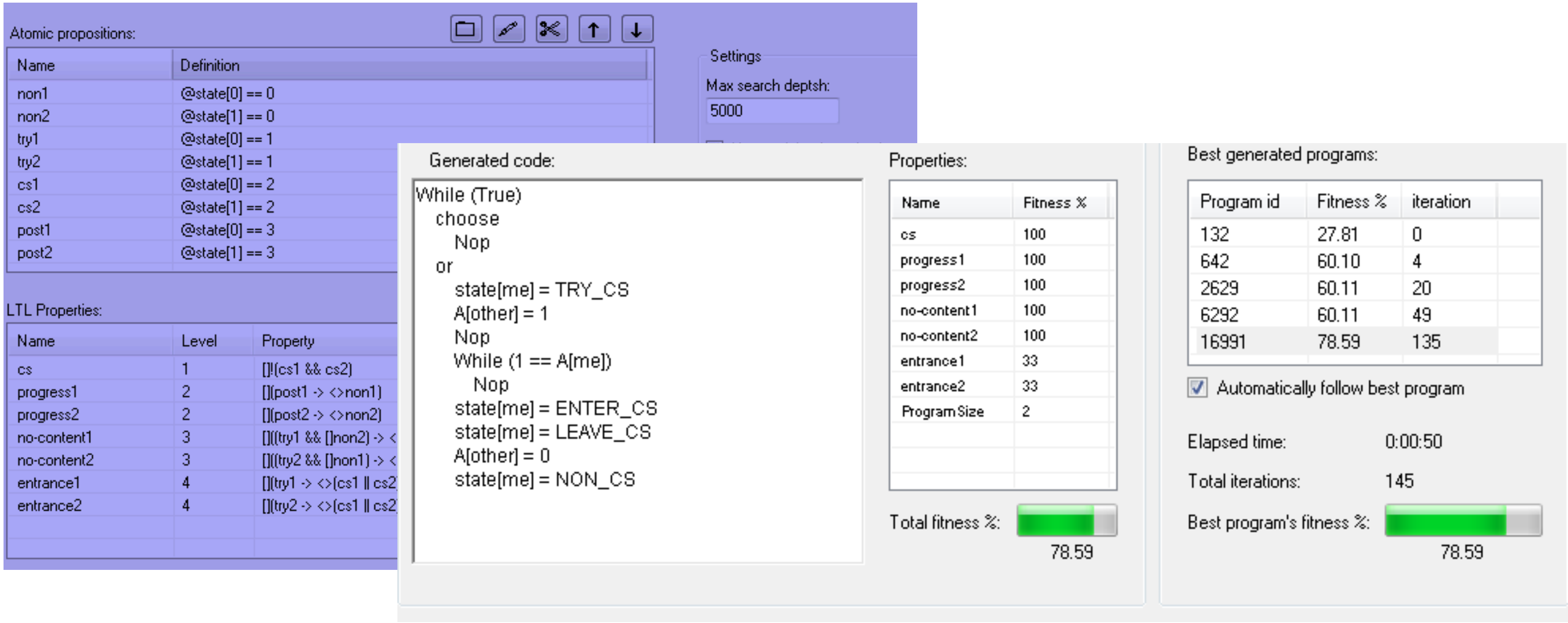
Performance



Variant No.	Successfull runs (%)	Avg. run durtaion (sec)	Avg. no. of tested programs per run
1	40	128	156600
2	6	397	282300
3	7	363	271950

- First variant was easiest to solve.
- Other variants are much harder to find.
- Still, much better than brute-force methods.
- Checked some more complicated requirement on efficiency (number of times of checking variables). Found improvied original algorithm!

MCGP – A Software Synthesis Tool Based on Model Checking and Genetic Programming



Atomic propositions:

Name	Definition
non1	@state[0] == 0
non2	@state[1] == 0
try1	@state[0] == 1
try2	@state[1] == 1
cs1	@state[0] == 2
cs2	@state[1] == 2
post1	@state[0] == 3
post2	@state[1] == 3

Settings

Max search deptsh: 5000

LTL Properties:

Name	Level	Property
cs	1	[]!(cs1 && cs2)
progress1	2	[](post1 -> <>non1)
progress2	2	[](post2 -> <>non2)
no-content1	3	[]((try1 && []non2) -> <
no-content2	3	[]((try2 && []non1) -> <
entrance1	4	[](try1 -> <>(cs1 || cs2
entrance2	4	[](try2 -> <>(cs1 || cs2

Generated code:

```
While (True)
  choose
    Nop
  or
    state[me] = TRY_CS
    A[other] = 1
    Nop
    While (1 == A[me])
      Nop
    state[me] = ENTER_CS
    state[me] = LEAVE_CS
    A[other] = 0
    state[me] = NON_CS
```

Properties:

Name	Fitness %
cs	100
progress1	100
progress2	100
no-content1	100
no-content2	100
entrance1	33
entrance2	33
Program Size	2

Total fitness %: 78.59

Best generated programs:

Program id	Fitness %	iteration
132	27.81	0
642	60.10	4
2629	60.11	20
6292	60.11	49
16991	78.59	135

☑ Automatically follow best program

Elapsed time: 0:00:50

Total iterations: 145

Best program's fitness %: 78.59

Tool Evaluation

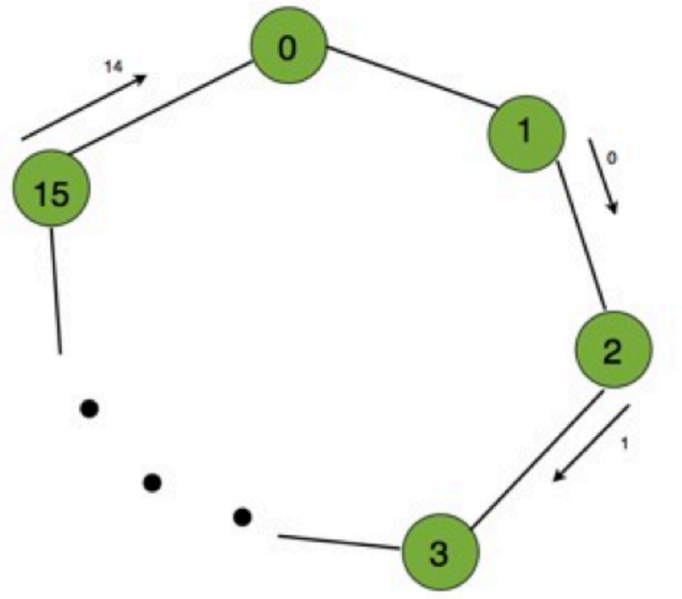
- Using the tool, solutions to a series of problems:
  - Classical mutual exclusion algorithms
  - Novel mutual exclusion algorithms
  - Parameterized leader election protocols
  - Discovering and correcting a bug in α-core protocol
- Synthesis takes between seconds to hours.
- Can benefit from modern multi-core machines

Synthesizing parametric programs (yes we can!)

- Dealing with parametric protocols running on various configurations and architectures:
    - Variable number of processes,
    - Various communication topologies.
- Undicidable [Apt,Kozen] for algorithms in a ring.
- Ah, then we synthesize exactly such an algorithm!!

Synthesizing parametric programs (yes we can!)

- First test case: leader in a unidirectional ring. Each process in a ring has a value and by exchanging messages in one direction. Find the process with highest value.
- Model checking is undecidable: *performs checks on specific values.*
- Succeeded to find n^2 protocol [Lellan,Chang,Roberts] but not $n \times \log n$ [Itai, Rodeh, Hirschberg, Sinclair].



Synthesizing parametric protocols

- Perform model checking for particular cases: in the leader election problem, with certain ring sizes.
- Coevolution: remember instances (sizes) that caused more candidates to fail, and recheck them.
- No complete guarantee: terminate if enough checks passed.
- Model checking as enhanced testing: comprehensive verification for specific values.

Process types

- Concurrent programs are built from process types
- Each process type
  - Has its own set of building blocks
  - Can have multiple running instances
  - Has a code skeleton, containing
    - Static parts defined by the user
    - Dynamic / empty part that have to be synthesized
- A special init process type is responsible for
  - Initialization of global variables
  - Creation of instances of the other process types

Various Synthesis Goals

- By setting program parts as static or dynamic, various goals can be achieved
- All parts are set to static
  - Nothing to synthesize. Just running the enhanced model checking algorithm
- Setting some processes as dynamic
  - The tool will try to synthesize dynamic parts
    - Can synthesize parts from scratch
    - Can synthesize only specific parts
    - Can replace and correct required parts if given

Model checking as enhanced testing

- For parametric programs, model checking is undecidable [Apt,Kozen].
- We can use testing but will have very little confidence.
- Perform model checking for specific instances (paraemters, architectures).
- Model checking as an “extended testing”: check comprehensively for particular parameters. Higher confidence than just testing.
- Use genetic programming to select good instances!

Coevolution

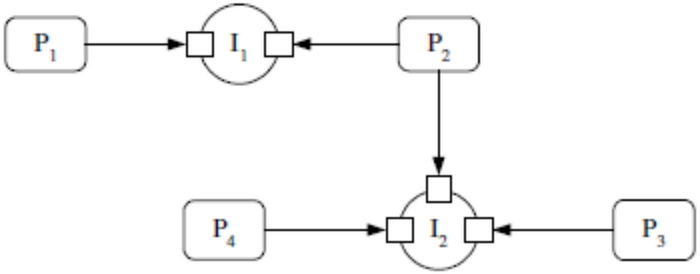
- Alternate between generating synthesis candidates and parameters for checking it.
- Different fitness functions for the two goals.
- Fitness for checking/testing parameters can increase with the number of candidates it manages to “destroy”.

Code Correction

- The goal is correcting existing protocols.
- The protocol’s code is divided by the user into:
  - Static parts that should remain unchanged,
  - Dynamic parts that can be improved or replaced by the synthesis process.

Motivating Example: The α-core Protocol

- Intended for allowing multiparty interactions between distributed processes.
- Published at COORDINATION 2002 conf., and Concurrency - Practice and Experience Journal.
- Two types of processes: Participants, Coordinators
- Multiple participants may perform a shared interaction, which is managed by a dedicated coordinator process.



The α-core Protocol

- Each process has its own state machine
- Processes communicate via asynchronous message passing
- The protocol should satisfy the following:
  - Exclusion between conflicting interactions.
  - If an interaction is committed, all of its participants must execute it.
  - Any enabled interaction is eventually committed or canceled.
    - **We showed that this requirement can be violated!**

Synthesizing Violating Architectures

- Main Idea:
  - Architectures can be generated by some initialization code. Thus, they can be synthesized similarly to normal code.
  - Define building blocks from which such code portions can be built.
  - Use genetic programming for the automatic generation and evolution of versions of the initialization code.
  - Define a fitness function that will guide us to the target architecture (violating the spec.).

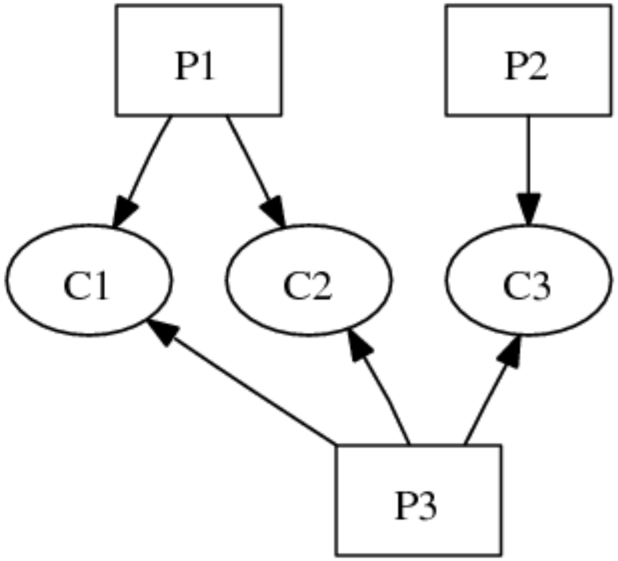
Initialization code for α-core Architectures

- We define the following building blocks:
  - Participant, Coordinator – constants of type proc_type
  - CreateProc(proc_type) – dynamically create new process of type proc_type
  - Connect(participant_id, coordinator_id) – connects between a particular participant and coordinator

Initialization code for α-core Architectures - Example

- The code on the left generates the architecture on the right:

```
CreateProc(Participant)
CreateProc(Participant)
CreateProc(Participant)
CreateProc(Coordinator)
CreateProc(Coordinator)
CreateProc(Coordinator)
Connect(1, 4)
Connect(1, 5)
Connect(2, 6)
Connect(3, 4)
Connect(3, 5)
Connect(3, 6)
```



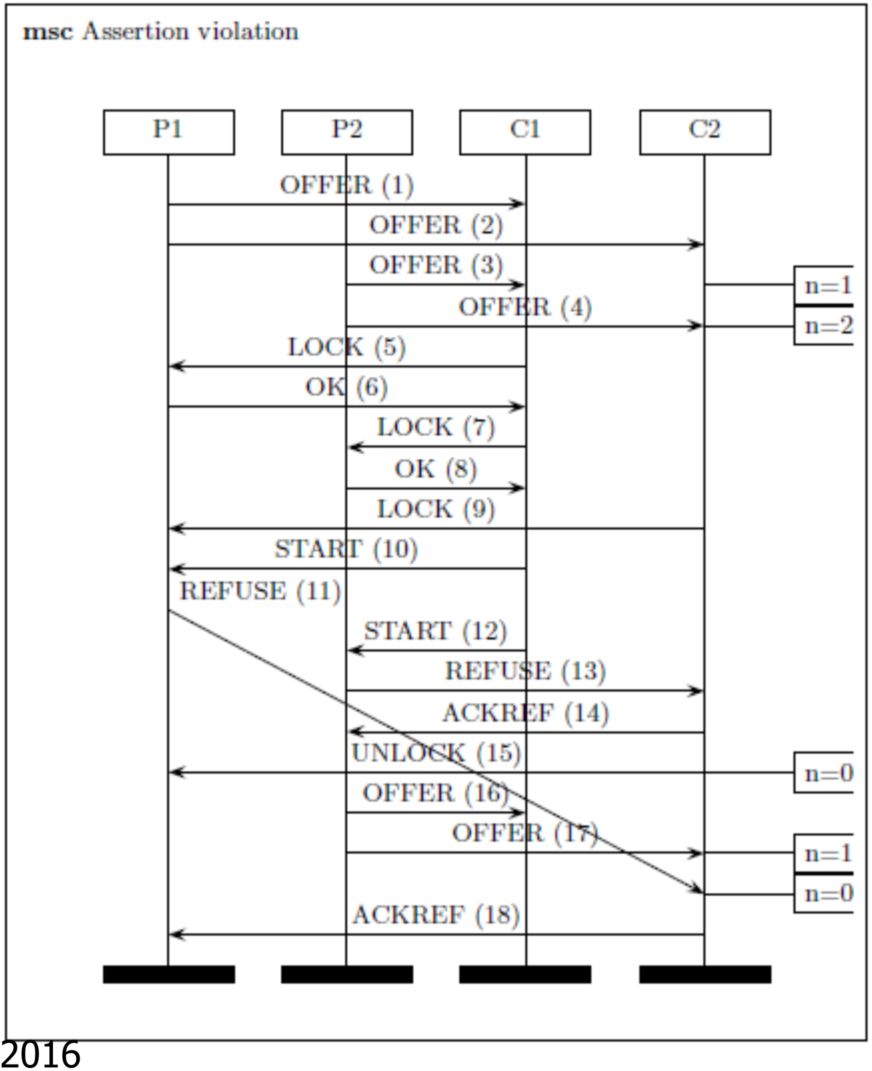
Coevolution: Evolving Violating Architectures

- Search of architectures is guided by a fitness function, assigning a score for each generated architecture.
- Based on model checking, but the goal is to falsify the specification.
- Highest score is given when at least one LTL property is violated
- Lower scores can be assigned to architectures which are “close” to violating a property.

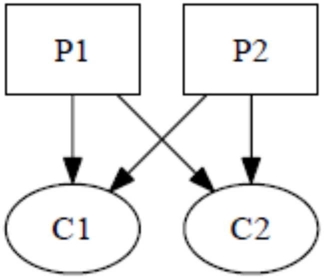
Finding the α-core Bug

- Each coordinator process uses a variable n counting its currently active offers.
- n should be decreased to 0 when an interaction is canceled.
- We suspected that this property might be violated in some rare cases, and fed the protocol and this property into our tool.
- The tool indeed discovered an architecture under which the property can be violated.
- The violation can lead to a livelocks and deadlocks in the algorithm.

The Found Architecture and Counterexample



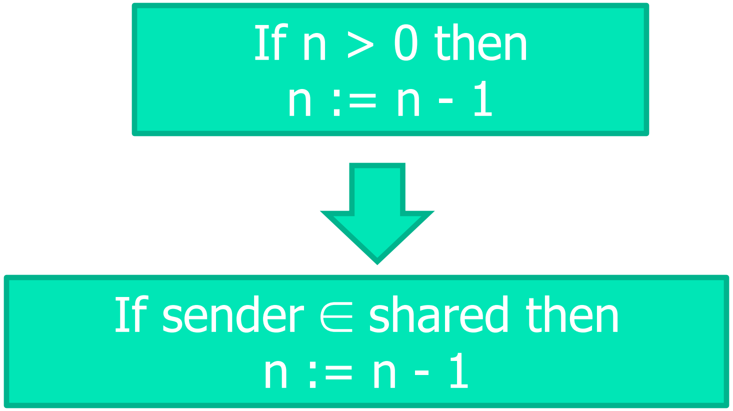
Found architecture



n is wrongly decreased twice

Correcting the α-core Bug

- The tool first found a correction for the above architecture.
- However, this correction was refuted by another discovered architecture.
- After a series of corrections and refutations, a final (and simple) solution was found, which could not be refuted.
- The solution includes the following code replacement:



If n > 0 then
n := n - 1

If sender ∈ shared then
n := n - 1

Conclusions

- Formal methods (Testing, RV, Model Checking) have severe limitations:
  - High complexity.
  - Decidable under some strict conditions.
- Synthesis is even more difficult!
- Use genetic programming to enhance the performance and these methods and alleviate restrictions.

More conclusions

- Genetic algorithms: heuristic beam search technique that combines ideas from evolution.
- Can be used to solve, e.g., optimization problems.
- Can be used to generate test cases.
- Genetic programming: similar ideas, but the objects are programs (represented as trees).

Even more conclusions

- Can be used to synthesize concurrent code.
- Can be used to synthesize parametric code.
- Can be used to improve and correct code.
- Model checking of genetically selected parameters as extended testing.
- Many other applications, e.g., Optimizing code [Harman]