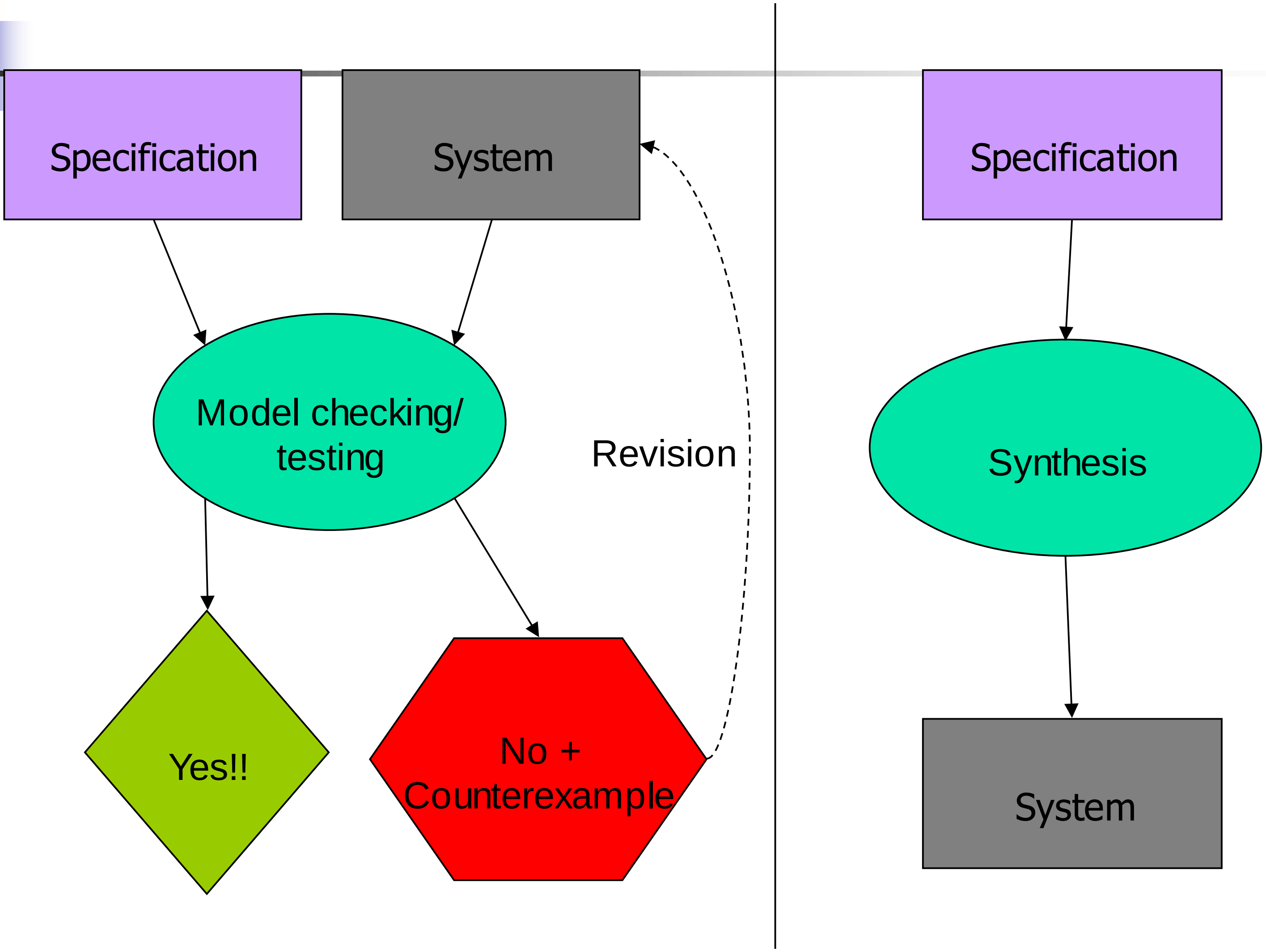
Automatic Synthesis of Code Using Genetic Programming

Doron A. Peled
Bar Ilan University, Israel

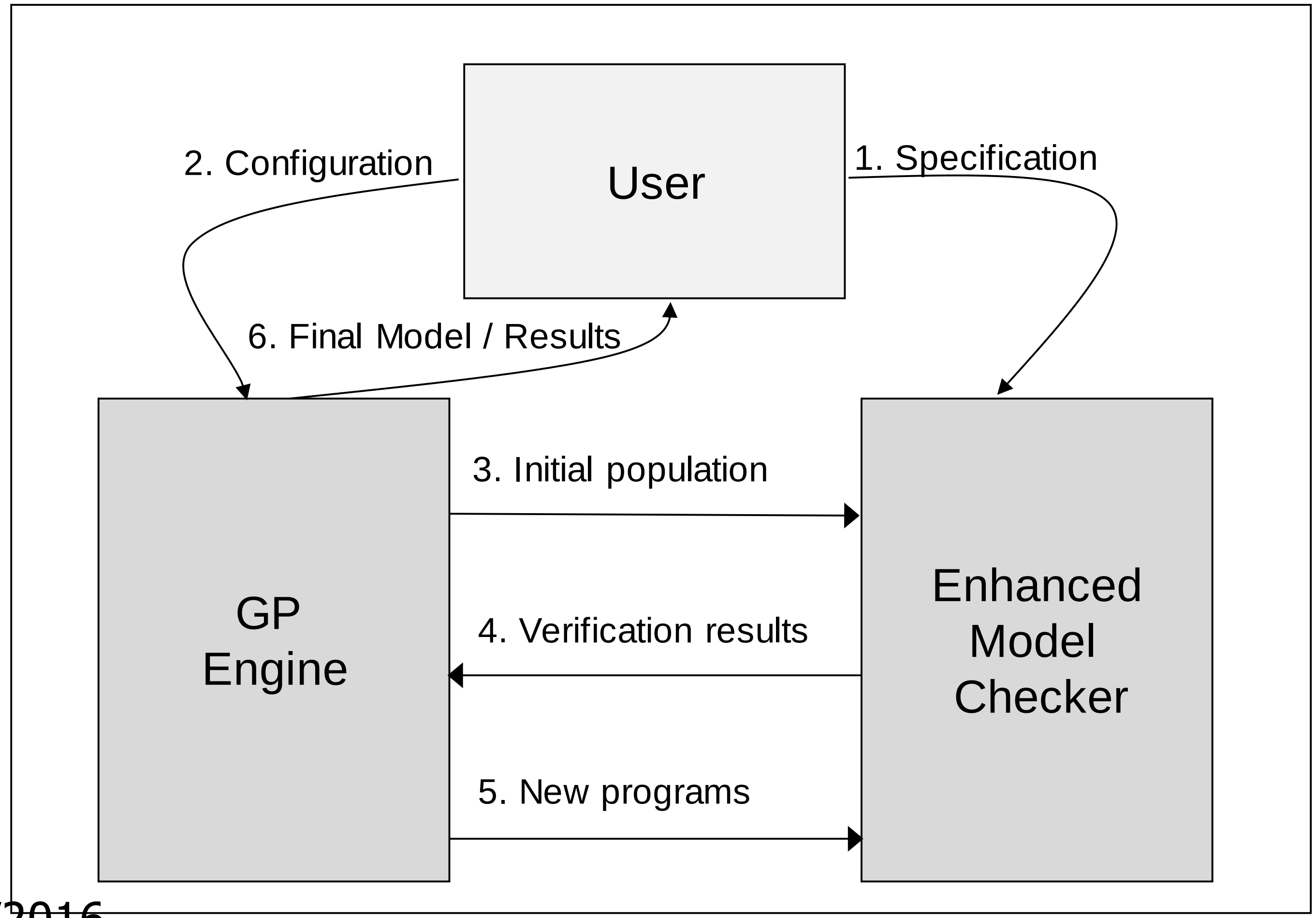
Why not synthesize the software directly from specification?



How to construct a model from the specification?

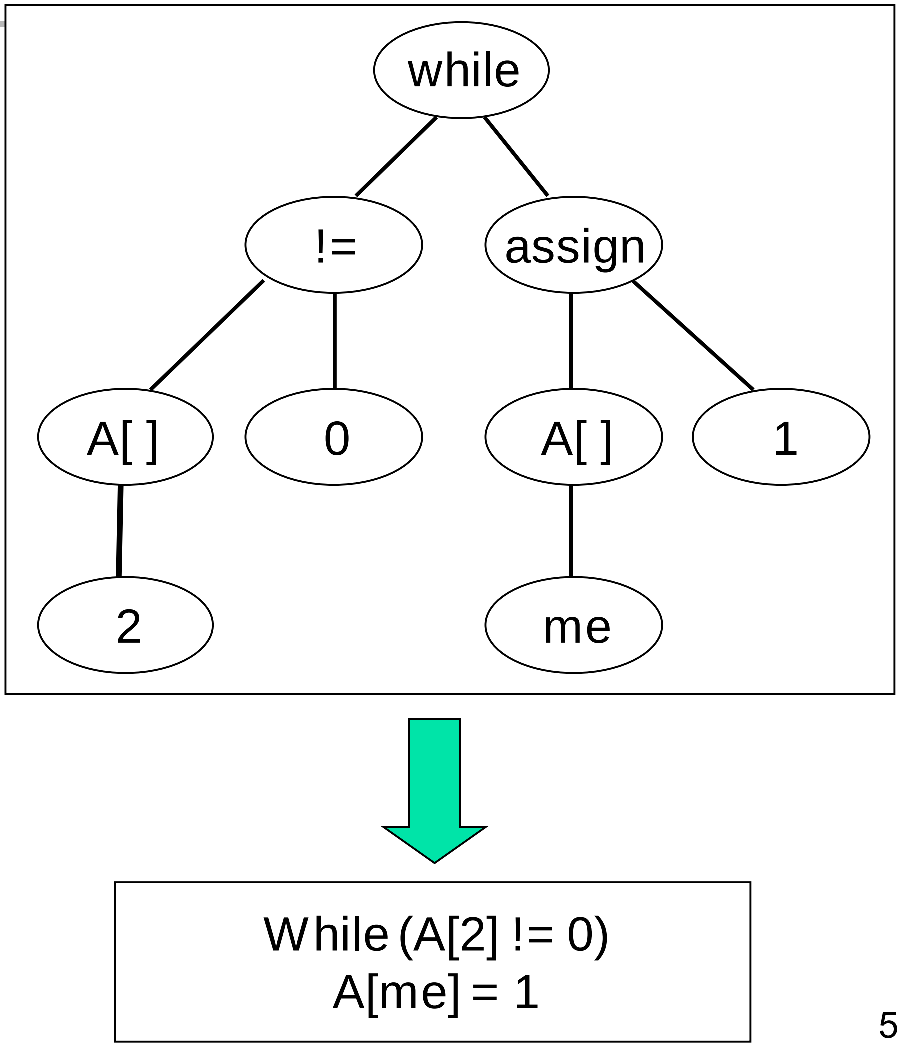
- Synthesis
  - Transforms spec. directly to a model that satisfies it.
  - Hard (complexitywise) and sometimes undecidable.
- Brute-force enumeration [Bar David, Taubenfeld]
  - All possible programs of a specific domain and size are generated and model-checked.
  - All existing solutions will eventually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

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].

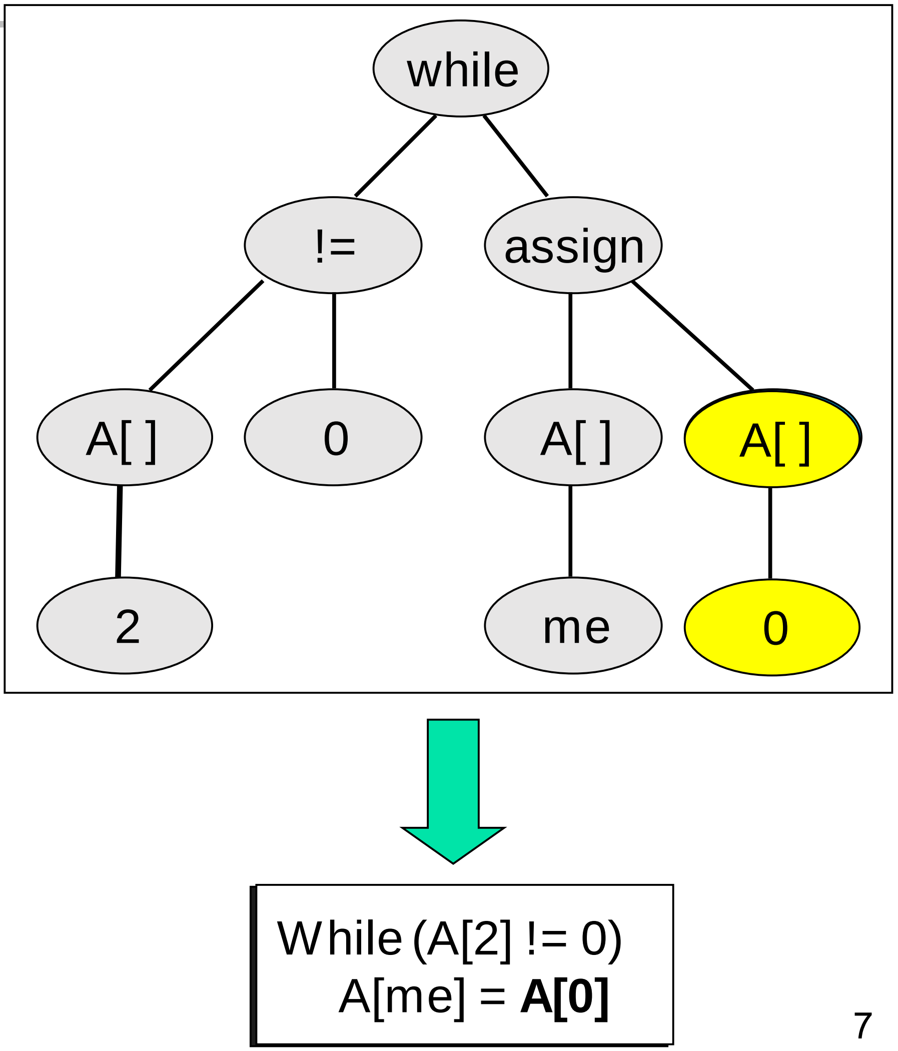


Mutation Operation

- 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)

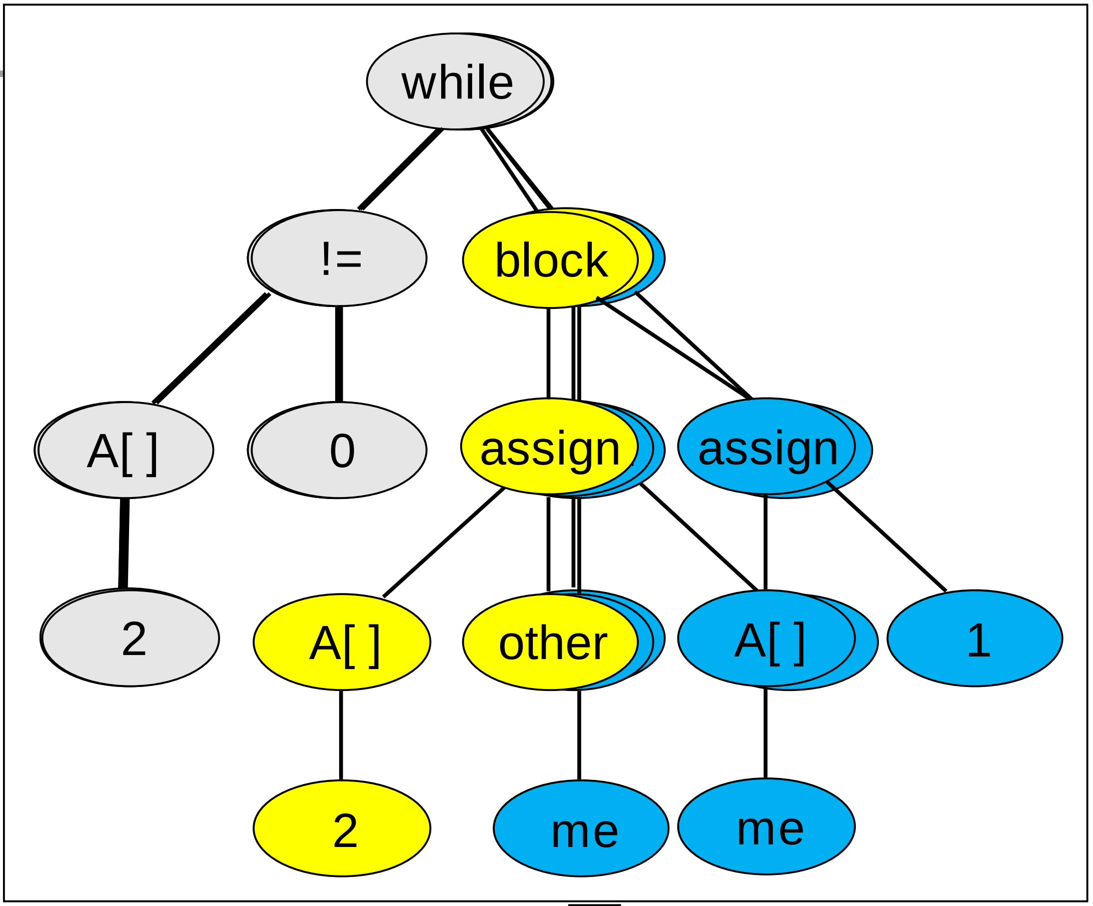
- Replace the sub-tree rooted by node with a new randomly generated sub-tree.
- Can change a single node or an entire sub-tree.



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.



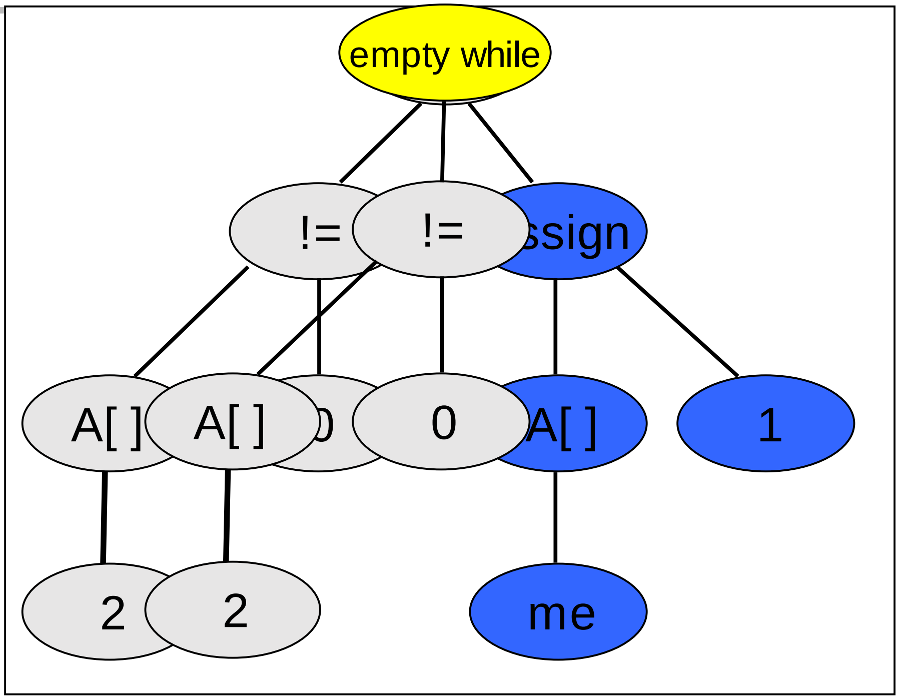
```
While  (A[2] != 0)
    A[2] = other
    A[me] = 1
```

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.

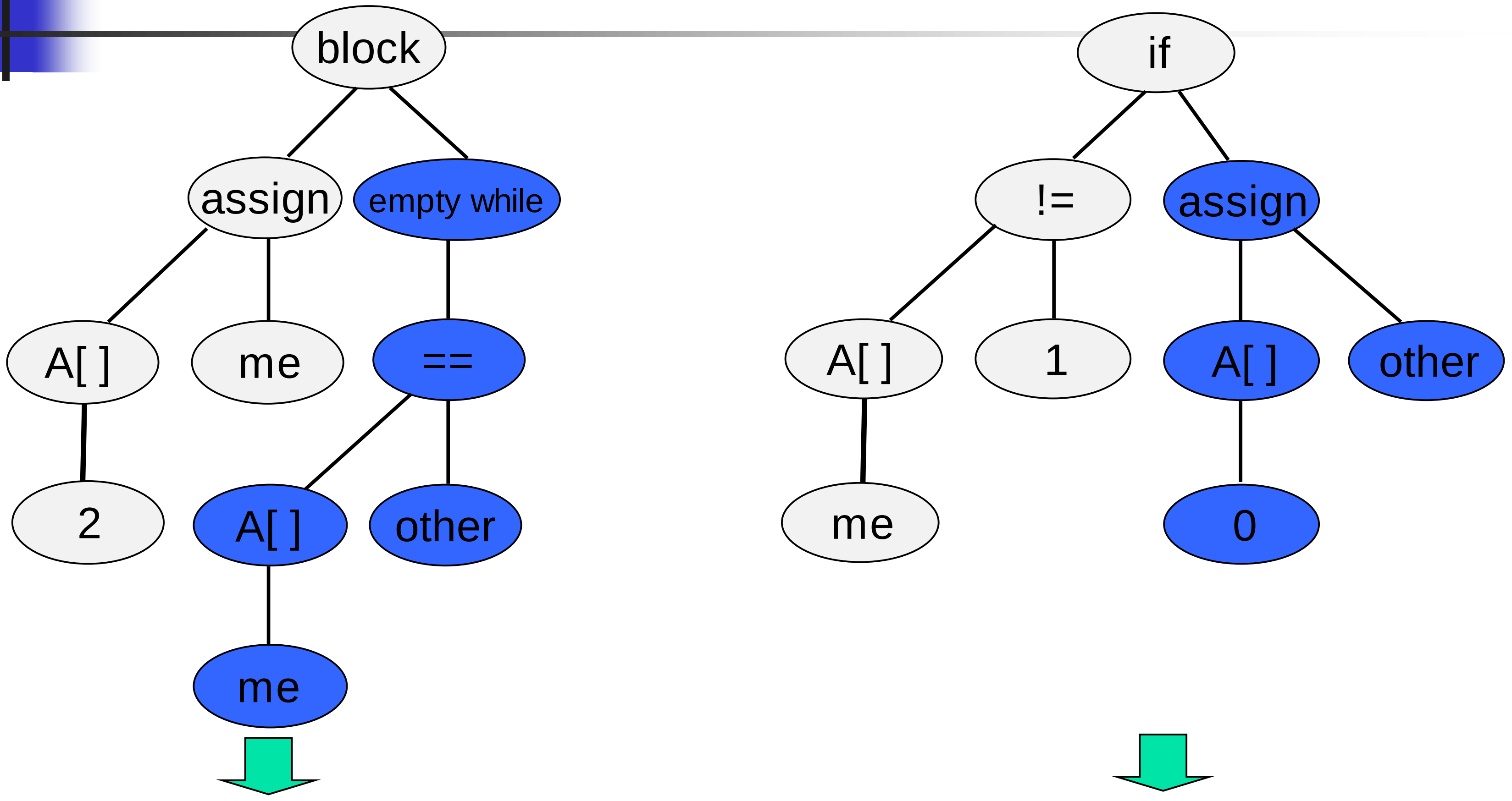
Deletion Mutation Type (d)

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



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

Crossover Example



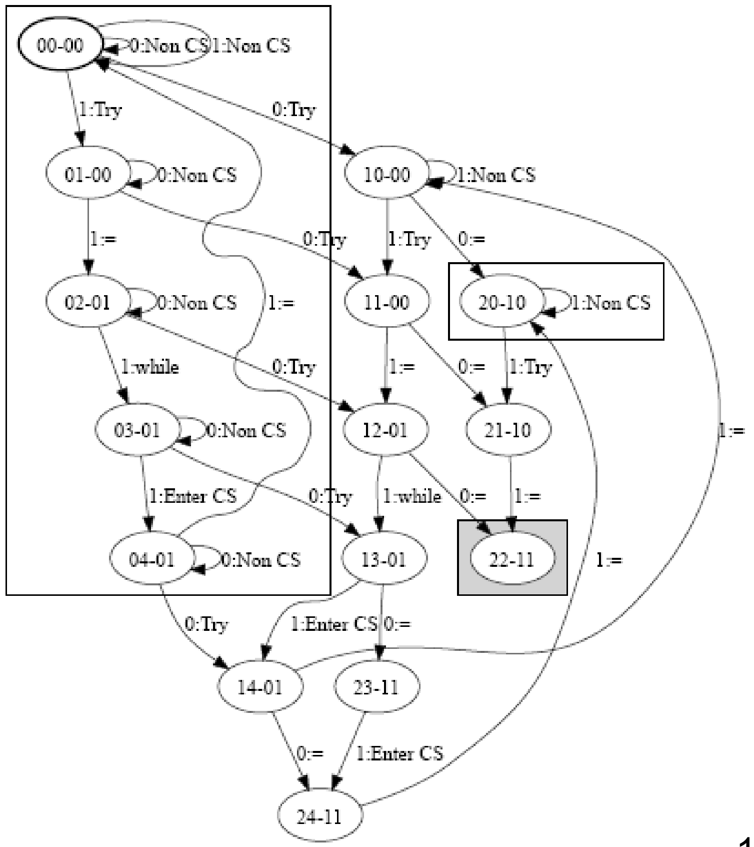
A[2] = me
a[0] = other

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

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 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.
    - $\Box(p)$ – p always occurs.
- A model M satisfies a formula φ ($M \models \varphi$) if every (fair) run of M satisfies φ.

Specification

- Safety: $\Box\neg(p_0 \text{ in } CS_0 \wedge p_1 \text{ in } CS_1)$
- Liveness: $\Box\ (p_i \text{ in } preCS_i \rightarrow p_i \text{ in } CS_i)$
- 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.

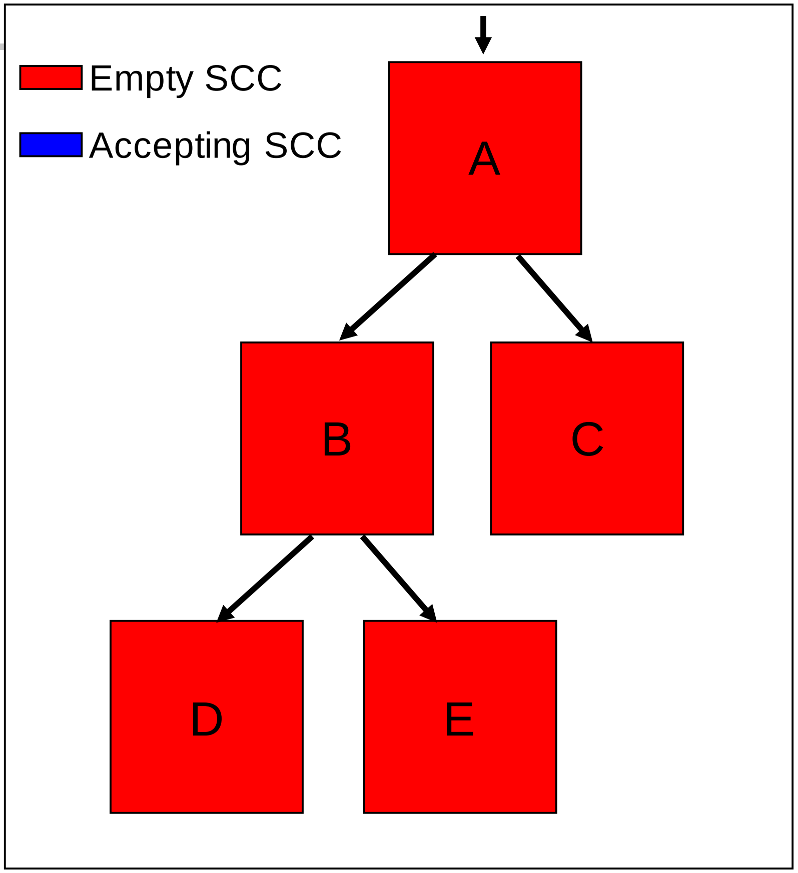
```
L0:While True do
  NC0:wait(Turn=0);
  CR0:Turn=1
endwhile ||
L1:While True do
  NC1:wait(Turn=1);
  CR1:Turn=0
endwhile
```

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.

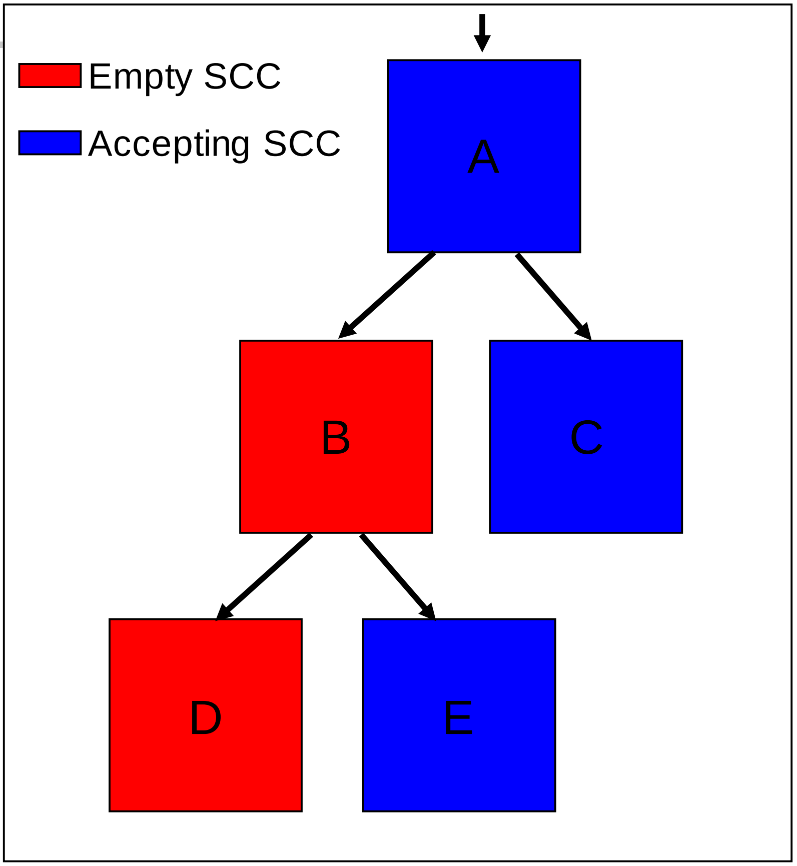
Fitness Level 0

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



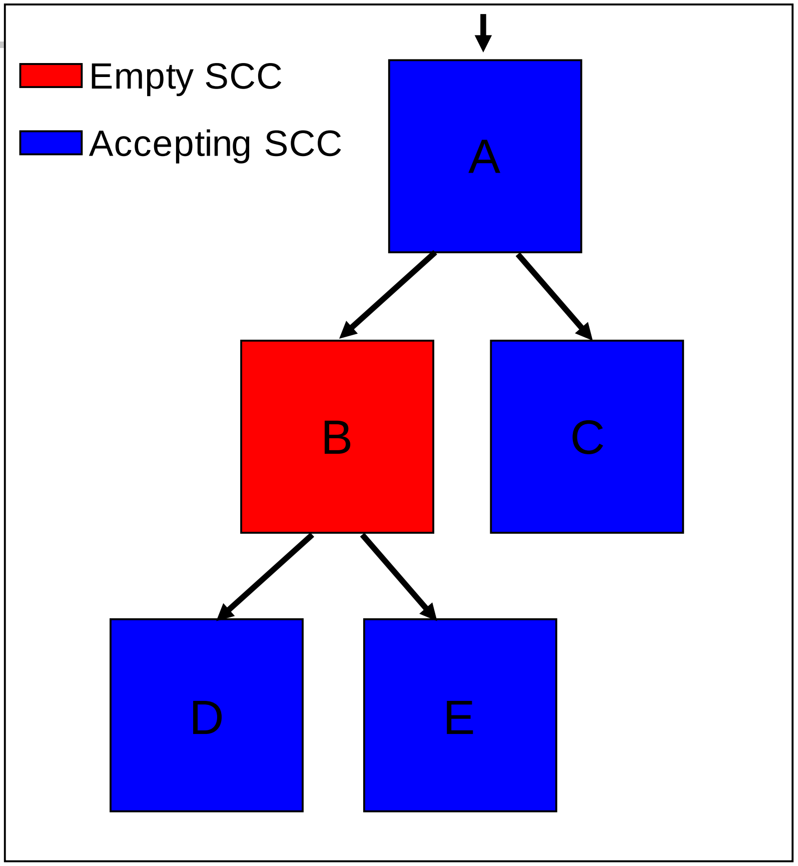
Fitness Level 1

- 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

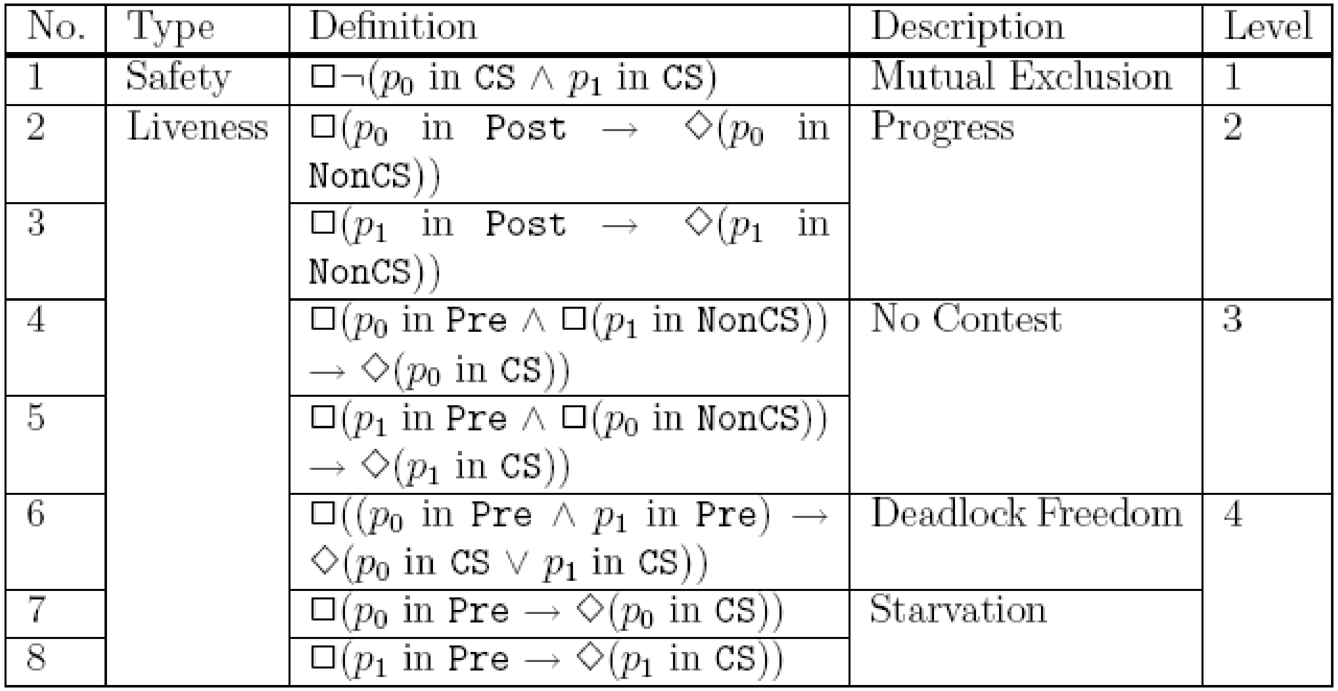
- GP programs tend to grow up over time to the maximal allowed tree size (“bloating”).
- To avoid that, we use parsimony as a secondary fitness measure.
- Number of program nodes * small factor is subtracted from the fitness score.
- The factor should be carefully chosen.
  - Should encourage programs 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.

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

- 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.

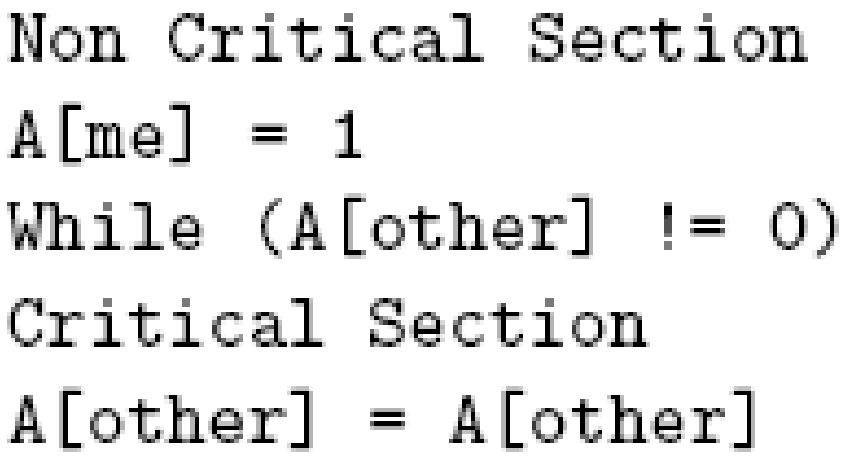
An Example of a Run (1st variant)

```
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)

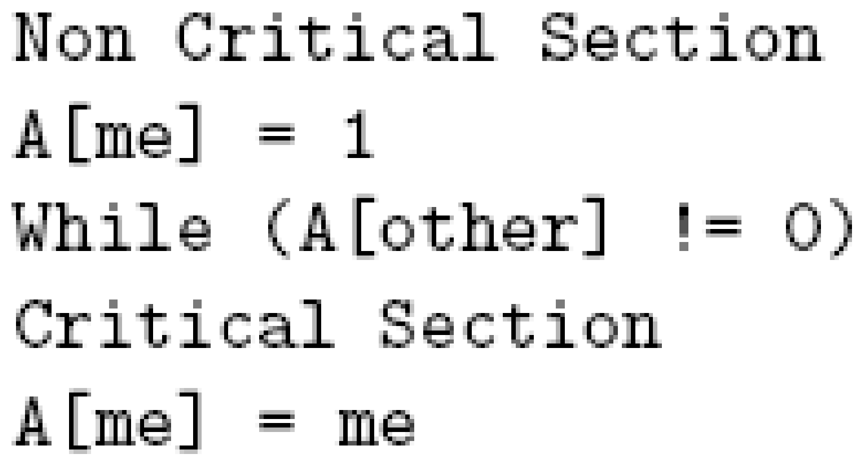


```
Non Critical Section
A[me] = 1
While (A[other] != 0)
Critical Section
A[other] = A[other]
```

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 (1st 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 (1st 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.

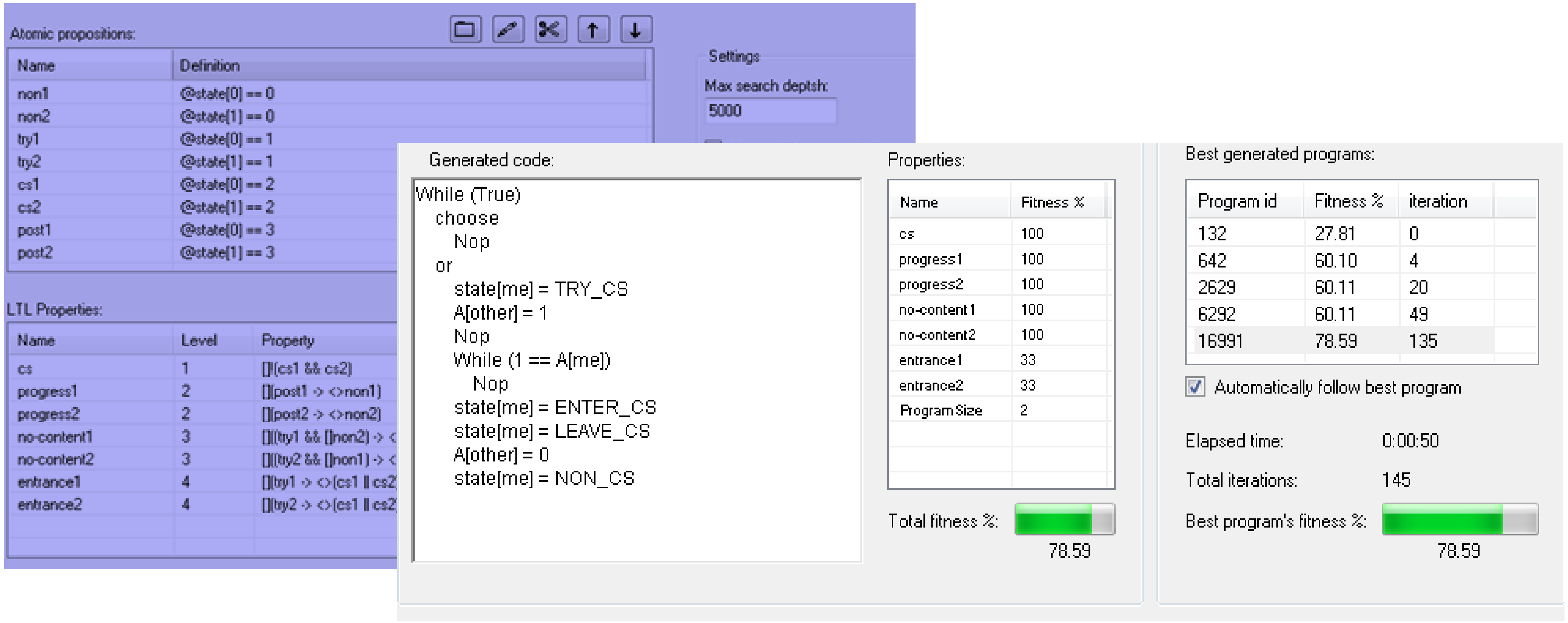
An Example of a Run (1st variant)

```
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].

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 depth: 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

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

Coevolution

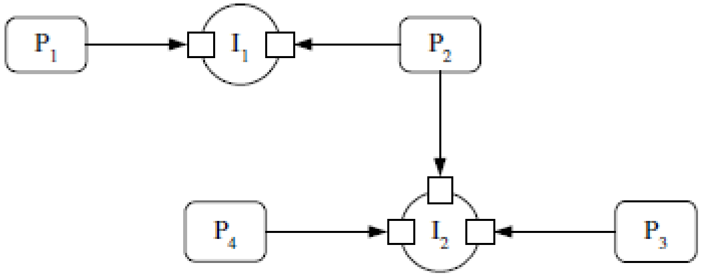
- Alternate between generating synthesis candidates and parameters for checking it.
- Different fitness functions for the two goals.
- Fitness for checking/testing parameters can increase with the number of candidates it manages to “destroy”.

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.).

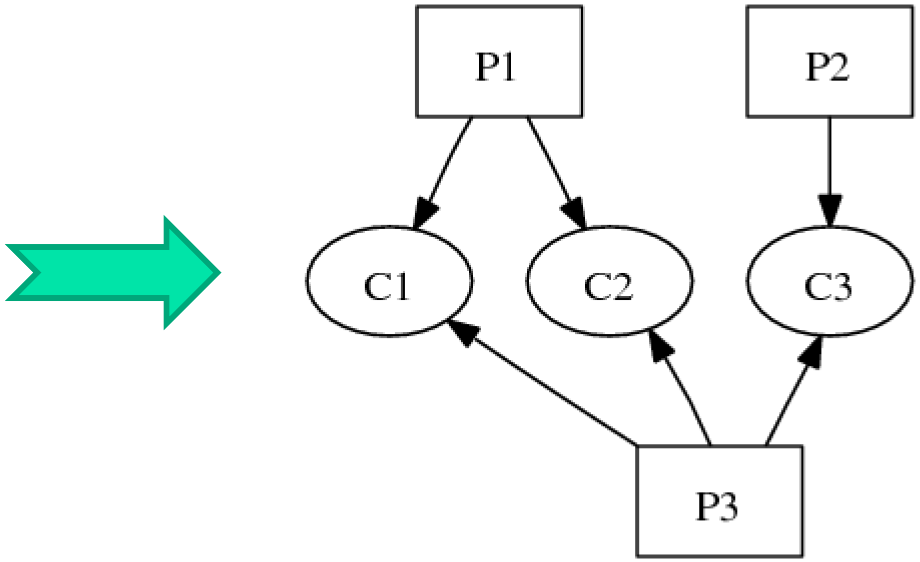
Initialization code for α-core Architectures

- 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)
```



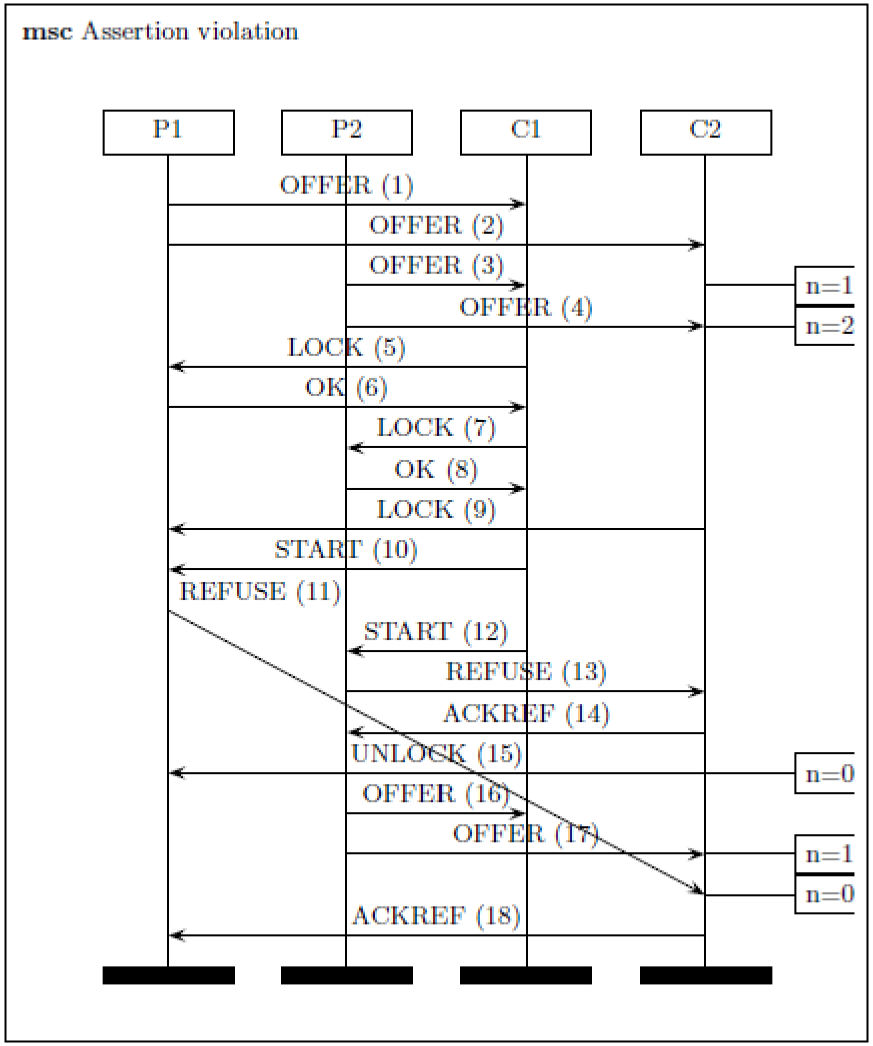
Coevolution: Evolving Violating Architectures

- 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.

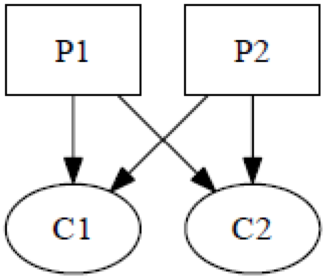
Finding the α-core Bug

- 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 ɑ-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:

$$\begin{tikzcd} \boxed{\begin{array}{c}\text{If } n > 0 \text{ then}\\ n := n - 1\end{array}} \arrow[d] \\ \boxed{\begin{array}{c}\text{If sender} \in \text{shared then}\\ n := n - 1\end{array}} \end{tikzcd}$$

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

- Can be used to synthesize concurrent code.
- Can be used to synthesize parametric code.
- Can be used to improve and correct code.
- For parametrized systems: use model checking as enhanced testing (for particular arguments/architectures).