Concurrent control synthesis based on knowledge (plus Genetic Synthesis, if we have time...)

Doron Peled
Bar Ilan University
Formal methods

Specification → System

Model checking

Yes!! → No + Counterexample

Revision

Specification → Synthesis → System

Synthesis
Synthesizing software

- Given a specification, say in LTL, can we automatically construct a program that satisfies it?
- **Sequential and closed system** ➔ one can translate the specification to an automaton [MW81,CE81].
- What about the case where the program has to interact with uncontrolled inputs? or consists of several processes, with limited view of each other (and some uncontrolled choices)?
- What about controlling (restricting) existing programs to admit additional properties?
Classical software synthesis

- Synthesizing **sequential program** (automata) of an open (interactive) system is in 2EXPTIME [PR88]. One converts the LTL formula into a deterministic tree automaton that checks that finite memory is sufficient to enforce a strategy that will guarantee to satisfy the specification.

- Synthesizing **concurrent programs** is **undecidable** [PR89]. Same for synthesizing a **controller** that will restrict the system [T].
Synthesizing controller

1. **Specification** → **Code Synthesis** → **System**
2. **Additional Specification** → **Controller Synthesis** → **System**
3. **Supervisor**
3 trains: TGV, regular, freight, compete on tracks to enter the station.

We want to implement the system such that each track decides “locally”, without a centralized control.
Main problem

- Controlling a concurrent system that needs to satisfy some global restrictions.
- In general this is an **undecidable** problem \([T]\).  
  *Decidable alternative*: allow synchronizing actions as needed by the analysis (at the limit it becomes a sequential control problem).
- Actions are controlled based on the **knowledge** of being in a certain collection of global states.
- This “knowledge” is calculated in advance (**model checking**), and then the program is transformed to exploit it.
For the train tracks system

- Trains, when approaching tracks, may not decide independently: allow trains (or tracks) to synchronize in order to make a joint decision.

- Minimize synchronization: synchronizing (costly!) may block trains that want to progress.
Imposed invariant

- We are given a transition system, e.g., a Petri Net.
- We want to impose that each pair \((s, \tau)\) of a state and transition fired from it must be from a given allowed set \(\Psi\).
- **Running example:**
  - *Priorities* - a partial order between the transitions.
  - At each global state, we can execute a transition that has a maximal priority among the enabled transitions.
A (safe) Petri Net

t₁ – a transition. With input places p₁ and p₇, and output place p₃.

A transition *fires* (i.e., executes) if the input transition have tokens and the output transitions do not have tokens (is empty).

A Petri Net is *safe* if when the input places to a transition have tokens, the output transitions do not have tokens.
A (safe) Petri Net

A state is a set of places, e.g., \{p_1, p_2, p_7\}.
Can be written as $p_1 \land p_2 \land \neg p_3 \land \neg p_4 \land \neg p_5 \land \neg p_6 \land p_7$. 
A (safe) Petri Net
A (safe) Petri Net

\[ p_1 \xrightarrow{t_1} p_3 \xrightarrow{t_3} p_5 \]

\[ p_2 \xrightarrow{t_2} p_4 \xrightarrow{t_4} p_6 \]

\[ p_7 \]

\[ p_7 \text{ is a} \]

“semaphore”!
A (safe) Petri Net
A (safe) Petri Net

Transitions are independent (concurrent) if they do not share input and out places.
A (safe) Petri Net with priorities:
Partial order between transitions.
A (safe) Petri Net with priorities.
Partial order between transitions.
A (safe) Petri Net with priorities. Partial order between transitions.

Transitions are independent (concurrent) if they do not share input and output places.

\[ t_1 << t_4, \quad t_2 << t_3 \]
Problem: how to guarantee execution according to priorities?

- Execute only *maximally* enabled transitions.
- Some transitions may be *uncontrollable*, i.e., if enabled, one cannot or must not stop them.
- Make a distributed (local) decision.
- Is it even possible?
- Can we check when it is possible? (Decidable?)
- What if we cannot make the distributed decision?
Each process has its own controller.

Based on knowledge calculated in advance, transitions can be permitted or blocked (uncontrolled transitions cannot be blocked!).

Each transition must be “supported” in this way by at least one of the controllers of the participating processes (disjunctive controller).
So, what can we do?

- What do we **know** in each local state?
- Use **logic of knowledge** to express (and check) if a process knows that its enabled transition has maximal priority.
- Transform the program to **act** based on the knowledge.
- There are variations of knowledge (based on the ability to recall our local history); tradeoff on price of transformation.
Some definitions

- A **process** is a set of transitions. [No two transitions of a process can execute independently at any moment]. The processes cover all the transitions (there can be additional constraints).
- The **neighborhood** of a process is the set of all input and output places of its transitions.

The **local information** of a process in a state, is the state limited to the neighborhood of the process. E.g., for the left process, in state \{ p_1, p_2, p_7 \}, it is \{p_1,p_7\}. (=“local state”).
We are ready to define “knowledge”

- Given a “local information” as a process, a process knows everything that is consistent with it.
- Easy to calculate what a process knows.
- That is: it knows of any property that holds in all global states with the same “local information”.
- This is a standard definitions (e.g., [FMHV]).
- Later: “knowledge with perfect recall”.

When left process is given that \( p_1 \land p_7 \), it knows that \( \neg p_4 \).

\[(p_1 \land p_7) \rightarrow K_1 \neg p_4\]
Calculating what a process knows given some local information

- We can write (and calculate) a formula that represents “all reachable states”: $\phi_{\text{reach}}$
- We can write a formula that represents “a given local information $x$”: $\phi_x$
- Intersect the above two formulas, and obtain “the states given some local information”:
  $\phi_{\text{reach}} \land \phi_x$.
- We *know*, when $x$ holds, whatever is implied from this intersection $(\phi_{\text{reach}} \land \phi_x) \rightarrow \psi$.
- The standard notation is $K\psi$ and we can embed this within temporal logic.
Combined knowledge of several processes

- The **joint knowledge** of several processes is the properties that hold in all the global states consistent with the combined local information of these processes.
- Since we combine the local information, there are fewer consistent states. So, together, the processes know more.
- However, to utilize this combined knowledge, the processes need to coordinate with each other.
What is known, during execution can be model checked:

$\varphi_1$ Each process knows \textit{which} of its enabled transitions have maximal priorities.

$\varphi_2$ Each process knows about \textit{at least one} of its enabled transitions that has a maximal priority.

$\varphi_3$ In each non-deadlock state, \textit{at least one process knows} about at least one enabled maximal priority transition.

Maybe even $\varphi_3$ does not hold...
The **supporting process policy**

- A transition can belong to multiple processes (e.g., communication).
- If enabled, **and** at least one process knows of it having maximal priority, then it can fire.
- Related to a problem in **control theory**: we are seeking a distributed disjunctive controller with limited observability (this is, in general: undecideable).
- Build for each process a “support table”: for each local information, which transition, if any, is supported.
Some important points:

- The knowledge is calculated based on the states of the original (priorityless) program.
- Then transitions are being controlled, which generates a different program.
- The transformation adds constraints to the enabledness conditions, based on the support table.
- Monotonicity: Knowledge about the original program remains known in the transformed program, since the reachable states can only reduce reachable states and executions!
- Approximation: maybe there are programs with more knowledge…
And what if neither of these invariants hold?

- **Combine** knowledge of processes. The “knowledge of several processes” is bigger than of single processes. Synchronize to obtain more knowledge.

- **Compromise**, and allow transitions that are not maximal, to complete support table, such that in any reachable state, at least one transition is enabled. Experimentally measure how far is a compromised execution from a perfectly prioritized one.

- **Switch to** “knowledge of perfect recall”
Motivation: given local information \{p_3\} for left process: 
t_3 is enabled, can it be fired?
Motivation: when $t_3$ is enabled, can it be fired?

Now it has a maximal priority.
Motivation: when $t_3$ is enabled, can it be fired?

Continuing the execution...
Motivation: when $t_3$ is enabled, can it be fired?

Continuing the execution…
Motivation: when $t_3$ is enabled, can it be fired?

Continuing the execution…
Motivation: when $t_3$ is enabled, can it be fired?

Same local information as before for left process: \{p_3\}.

Now $t_5$ has maximal priority!
Based on this, the left process does not know if $t_3$ has the highest priority at both times. But if we remember how many times $t_1$ fired, we can distinguish between these two cases!
Knowledge of perfect recall

- Knowledge is based on the history that a process can sense, up to any point in its computation.
- A process can sense changes in its neighborhood, including changes to the neighborhood by other processes.
- This way we can distinguish between two local states (local information) with different histories. So, we know more!
- But don’t we need infinite amount of information?
Knowledge of perfect recall: finite information.

- Ron Van der Meyden showed that finite amount of information is sufficient.
- Given some history of computation, the rest of the system can be in a finite number of states consistent with it (a "subset construction").
- If a process makes a single transition, or some process changes the neighborhood, the system can be in any global state that is obtained by making that transition, plus any sequence of (independent) transitions that do not change the neighborhood.
- For the transformation, we can update the knowledge as an automaton. This can become a gigantic automaton (with $2^{\text{EXP}}$ states).
An aside: Causal knowledge

- [Genest, Peled, Schewe: FOSSACS 2015]
  Another kind of knowledge.

- Idea: when processes communicate, then they are allowed to pass information.

- Construction built on top of “Gossip automata”
  [Mukund, Suhoni] keeps most recent local information between processes.

- Knowledge is deeper than total recall knowledge. Complexity is exponentially better (so where is the catch? 😊)
Another solution

- When there are states where no one knows which transition as higher priority: offer interactions between processes that together have the knowledge.
- Interaction is offered through a coordinating algorithm such as $\alpha$-core.
- The problem becomes decidable, since at the limit we get a full product that is controlled globally.
Each process has its own controller. Based on knowledge calculated in advance, transitions can be permitted or blocked (uncontrolled transitions cannot be blocked!). Each transition must be “supported” in this way by at least one of the controllers of the participating processes (disjunctive controller). Controllers are allowed to coordinate to temporary synchronize to obtain joint knowledge.
The $\alpha$-core algorithm


- There are several kinds of coordinations, each between a set of processes. Each coordination is controlled by a separate process.

- A process may have several choices between coordinations.

- Program processes and coordination processes exchange messages until coordination is established.

P$_4$ has multiple choice at some state.
We can mark each local state of a process $p$ as follows:

1. $K_1$: $p$ knows he has a transition with maximal priority.
2. $K_2$: $p$ does not know ..., but knows that some other process knows ... (not necessarily the same process for each state consistent with the local state).
3. $K_3$: Both above cases do not hold.

$\varphi_3$: for each reachable global state, there is a local state where $K_1$ holds.
Calculating $K_2$

- We first mark local states by $K_1$ according to the previous algorithm.
- Then $K_2$ holds in a local state, if all the global states containing it have a local state where $K_1$ holds.
- If a local state of some process satisfies $K_2$, then the process “shouldn’t worry” – “someone else will do the job”.
What if $\varphi_3$ does not hold?

- We set up a table that tells us about combined knowledge.
- Take some tuples of them (start with pairs), and check whether they have a combined knowledge of maximal priority.
- Make a table of such tuples.
- Each tuple is a coordination!
- Add tuples until each global state includes such a tuple.
- Each tuple suggests a coordination where combined knowledge about who has maximal priority exists.
Minimize the number of interactions
Minimize the number of interactions (NP-Complete)
More minimizations: Calculate the support table according to the executions satisfying the constraint property!

Based on the knowledge of the original system:
The right process would not execute $a$ since it has a lower priority than $c$ and $d$.
The left process would not execute $c$ since it has a lower priority than $b$.
But the execution $(cd)^\omega$ respects the priorities, and based on its reachable states $\{p_1,p_4\}$ and $\{p_1,p_5\}$, (and $\{p_2,p_4\}$ is not reachable then) the right process has knowledge to support this execution.

$a\ll\{c,d\}\ll b$
The rule:

- It is safe to calculate the knowledge based on the states reachable in executions that satisfy the needed restriction $\Psi$.

- Intuition: by induction on the sequence so far, the knowledge calculation includes in the possibilities the actual state and so far the new constraint was enforced. Now, a transition is only allowed from a state so that as a pair they satisfy $\Psi$. So it is safe.

- One also needs to check that no new deadlocks are introduced.
Strengthening the safety constraint

- It is possible that the safety constraint that we want to impose includes already some unavoidable new deadlocks: reachable states from which no transition can be fired, even if we synchronize all the processes.

- We may be in a previous state where all enabled transitions, or at least one uncontrolled transition, will lead to such deadlocks.

- We may be further in previous states that leads to ... that leads to ...
Strengthening the property $\Psi$

- Backwards strategy search:
- Start from (new) deadlock states and eliminate them.
- Eliminate every state with an uncontrolled enabled transition to a state that was eliminated.
- Eliminate every state where all the controlled enabled transitions lead to eliminated states.
- We obtain a reduced state space. Calculate knowledge and synthesize w.r.t. it.
- Preprocessing to find strengthen invariant: in EXPTIME complete (PSPACE, if no uncontrolled states).
Semi-global tie breaking

- Coordinating many processes takes a lot of overhead.
- Suppose there is a tie-breaking process (or several processes) that can help with acquiring knowledge.
- When a process does not know what to do, it logs into that process, and provides its local information.
- When the tie breaker has enough joint knowledge, it decides which transition can be fired.
Each process has its own controller.

Based on knowledge calculated in advance, transitions can be permitted or blocked (uncontrolled transitions cannot be blocked!).

Each transition must be “supported” in this way by at least one of the controllers of the participating processes (disjunctive controller).

Controllers are allowed to coordinate to temporary synchronize to obtain joint knowledge.
Conclusions

- We can employ **model checking** of knowledge properties to **implement priorities in a distributed way**.
- Solves a distributed **controllability with limited observability** problem but relaxes requirement on architecture.
- Problem is **decideable** since we allow additional synchronization.
- Does not provide a perfect solution (not always the least constrained solution)!
- Space/performance tradeoff in using different kinds of knowledge.
Synthesis: Our Method
Combining GP & Model Checking

1. Specification
2. Configuration
3. Initial population
4. Verification results
5. New programs
6. Final Model / Results
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.
- Programs 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 allowed tree depth is reached, a terminal must be chosen.
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[me] = 1
```

Deletion Mutation Type (d)

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

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

\[
\text{if } (A[\text{me}] \neq 1) \Rightarrow \text{assign } A[\text{me}] \text{ other}
\]

\[
\text{empty while } = (A[\text{me}] = \text{other})
\]

\[
\text{block}
\]

\[
\text{assign} \quad A[\text{me}] = \text{me}
\]

\[
A[2] = \text{me} \quad a[0] = \text{other}
\]
The Mutual Exclusion Problem

- Originally described by [Dijkstra 65].
- Many variants and solutions exist.
- Modeled using the following program parts:
  - 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:

<table>
<thead>
<tr>
<th>No.</th>
<th>Type</th>
<th>Definition</th>
<th>Description</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Safety</td>
<td>$\square \neg (p_0 \text{ in } CS \land p_1 \text{ in } CS)$</td>
<td>Mutual Exclusion</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Liveness</td>
<td>$\square (p_0 \text{ in } Post \rightarrow \Diamond (p_0 \text{ in NonCS}))$</td>
<td>Progress</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>$\square (p_1 \text{ in } Post \rightarrow \Diamond (p_1 \text{ in NonCS}))$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>$\square (p_0 \text{ in } Pre \land \square (p_1 \text{ in NonCS})) \rightarrow \Diamond (p_0 \text{ in CS})$</td>
<td>No Contest</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>$\square (p_1 \text{ in } Pre \land \square (p_0 \text{ in NonCS})) \rightarrow \Diamond (p_1 \text{ in CS})$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>$\square ((p_0 \text{ in } Pre \land p_1 \text{ in Pre}) \rightarrow \Diamond (p_0 \text{ in CS} \lor p_1 \text{ in CS}))$</td>
<td>Deadlock Freedom</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>$\square (p_0 \text{ in } Pre \rightarrow \Diamond (p_0 \text{ in CS}))$</td>
<td>Starvation</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>$\square (p_1 \text{ in } Pre \rightarrow \Diamond (p_1 \text{ in CS}))$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The properties are converted into Streett automata.
Runs Configuration

- 3 different sets of runs:

<table>
<thead>
<tr>
<th>Variant No.</th>
<th>Number of bits</th>
<th>Conditions</th>
<th>Requirement</th>
<th>Relevant properties</th>
<th>Known algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>Simple</td>
<td>Deadlock Freedom</td>
<td>1,2,3,4,5,6</td>
<td>One bit protocol</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Simple</td>
<td>Starvation Freedom</td>
<td>1,2,3,4,5,7,8</td>
<td>Dekker</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Complex</td>
<td>Starvation Freedom</td>
<td>1,2,3,4,5,7,8</td>
<td>Peterson</td>
</tr>
</tbody>
</table>

- The following parameters were used:
  - Population size: 150
  - Max number of iterations: 2000
  - \( \mu \): 5
  - \( \lambda \): 150
An Example of a Run (1st variant)

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

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

Score: 0.0
An Example of a Run (1st variant)

- Randomly created.
- While loop guarantees mutual exclusion.
- Only process 0 can enter the critical section.
An Example of a Run (1\textsuperscript{st} 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.
- A local maximum point in the search space.
An Example of a Run (1st variant)

- An important building block common to many algorithms.
- Each process sets its own flag and waits for other's flag, but
- The flag is not turned off correctly.
- Might eventually deadlock, thus, properties 4 and 5 get fitness level of 1.

Score: 70.17

```
Non Critical Section
A[me] = 1
While (A[other] != 0)
Critical Section
A[other] = A[other]
```
An Example of a Run (1st variant)

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

```
Non Critical Section
A[me] = 1
While (A[other] != 0)
Critical Section
A[me] = me
```
An Example of a Run (1st variant)

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 enter simultaneously.
An Example of a Run (1st variant)

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

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

Score: 93.20
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)

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

Score: 96.50

```plaintext
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
```
An Example of a Run (1\textsuperscript{st} variant)

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

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
What else can we do? (we actually did!)

- Model checking parametric systems, even with complicated topologies: use mutation on constructs to build processes and assign communication.

- Correct programs: start with some incorrect version, which will have some low fitness value. We corrected the \( \alpha \)-core algorithm.

- Improve programs: start with known version but fitness will reflect that it is not optimal. We found new solutions to mutual exclusion.