Algorithms for Synthesizing Reactive Systems: A Perspective

Pierre Wolper

Université de Liège

How it all Started

- Temporal logic was proposed in 1977 by Amir Pnueli as a language for specifying and verifying concurrent and reactive systems.
- Initially, first-order temporal logic was the focus.
- Later, interest in *propositional* temporal logic started to grow for several reasons:
  - It is already an interesting and expressive formalism,
  - It is decidable,
  - Its decision procedure, though of exponential complexity, is fairly simple and can be implemented.
- It was then seen that this decision procedure could be adapted to synthesize programs from propositional temporal logic specifications.

(Linear-Time) Temporal logic: A Brief Overview

- Formulas built from a set of propositions P, Boolean connectives $\neg$, $\wedge$, $\vee$, and the temporal operators $\bigcirc$, $\Box$, $\Diamond$, and U.

- Interpretations are infinite sequences $\sigma : \mathbb{N} \rightarrow 2^P$.

Temporal Operators

The temporal operators indicate in which state(s) of an infinite sequence a formula should be interpreted.

1. $\bigcirc$ which is read "at the next time" (in the next state of the sequence)

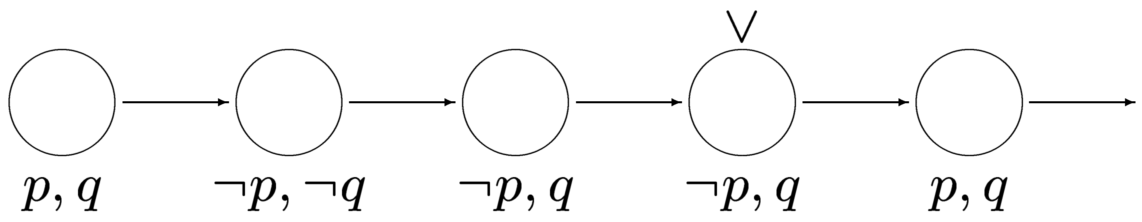
2. $\Box$ which is read "always in the future" (in all future states of the sequence)

3. $\Diamond$ which is read "eventually" (in some future state of the sequence)

4. U which is read "until" (binary operator)

Examples

In the marked state of the sequence



the following formulas are *true*:

$$\neg p \wedge q$$

$$\bigcirc(p \wedge q)$$

$$\Diamond\, p$$

$$q\, U\, p$$

the following formula is *false*:

$$\Box\, p$$

The Decision Procedure for Propositional Temporal Logic

- The idea is to attempt to build a model by separating what has to be true *now* from what has to be true later.
- This is done with the help of the following identities:

$$\begin{aligned} \Box p &\equiv (p \wedge \bigcirc \Box p) \\ \Diamond p &\equiv (p \vee \bigcirc \Diamond p), \\ p\, U\, q &\equiv (q \vee (p \wedge \bigcirc(p\, U\, q))). \end{aligned}$$

- A choice of "next" requirements is handled by exploring all alternatives.
- Since the "next" requirements are always combinations of subformulas of the initial formula, the procedure always terminates.
- The result of the procedure is a graph from which the models of the formula can be extracted (if any).

The Synthesis Procedure

[Manna & Wolper 81, Wolper 82, Manna & Wolper 84]

- Identify a finite set of "events" (e.g. synchronization events) in terms of which the system can be satisfied.
- Specify the required behavior in PTL, using these "events" as propositions.
- Build the graph used by the decision procedure and interpret this graph as a program.

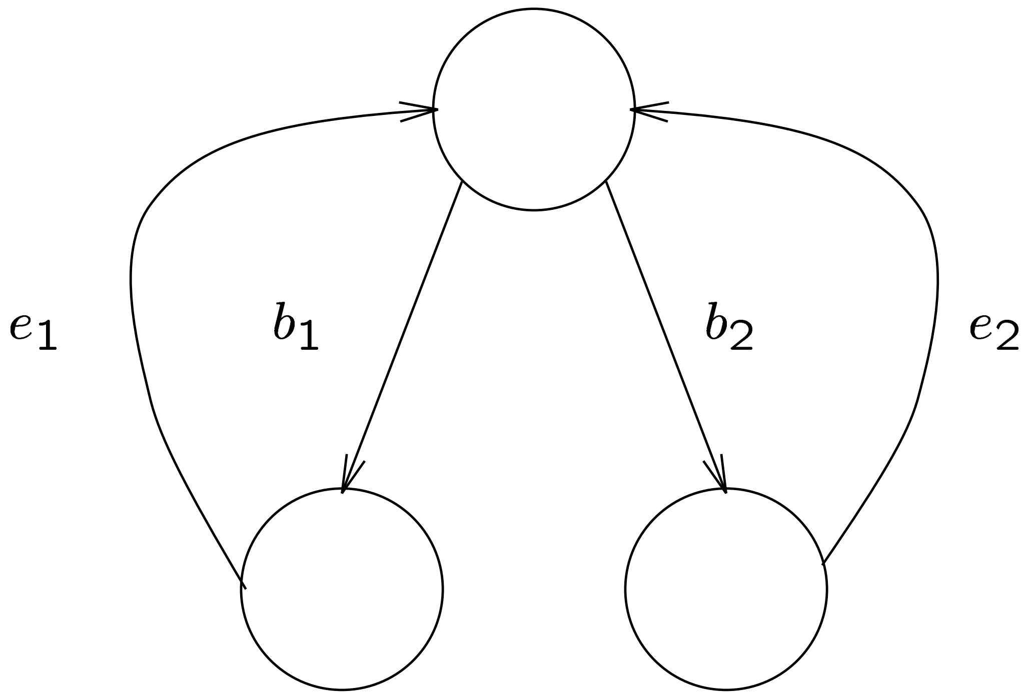
Synthesis: The Mutual Exclusion Example

The specification

$$
\begin{array}{l}
\neg inP_1 \, U \, begin_1 \\
\Box(begin_1 \supset \bigcirc(\neg inP_1 \, U \, end_1)) \\
\Box(end_1 \supset \bigcirc(\neg inP_1 \, U \, begin_1)) \\
\\
\neg inP_2 \, U \, begin_2 \\
\Box(begin_2 \supset \bigcirc(\neg inP_2 \, U \, end_2)) \\
\Box(end_2 \supset \bigcirc(\neg inP_2 \, U \, begin_2)) \\
\\
\Box(begin_1 \supset \neg begin_2 \, U \, end_1) \\
\Box(begin_2 \supset \neg begin_1 \, U \, end_2) \\
\\
\Box \Diamond inP_1 \\
\Box \Diamond inP_2
\end{array}
$$

with $inP_i \equiv begin_i \vee end_i$, $i = 1, 2$ and the assumption that only one action is performed at each time.

The result



Nice, but . . .

- The specification is more complicated than the result.
- The result is global: it assumes a single process handling all synchronization problems.
- The liveness requirements ($\Box \Diamond inP_1$ and $\Box \Diamond inP_2$) are not guaranteed to be satisfied, unless one assumes a "fair" scheduler or chooses a specific scheduling policy.

This synthesis algorithm does not really help us with the development of solutions to the mutual exclusion (or any other) synchronization problem.

Is “Branching-Time” Temporal Logic a Better Idea?

[Clarke & Emerson 81, Clarke & Emerson 82]

- Branching-time temporal logic is linear-time logic extended with *path quantifiers*: A (for all paths) and E for some path.
- It is interpreted over branching structures, or w.l.o.g. over infinite trees.
- It is interesting for synthesis since it can express not only requirements on all behaviors of the program, but also the existence of some possible behaviors.
- One can thus satisfy liveness requirements by choosing a specific scheduling policy while still maintaining some flexibility.

Branching Time: The Mutual Exclusion Example

The specification

Here, we assume that before attempting to enter its critical section, each process signals that it is ready to do so by the operation $ready_i$.

The specification is then the following.

$$
\begin{array}{l}
A \bigcirc (\neg inP_1 \, U \, ready_1) \wedge E \bigcirc (ready_1) \\
A \Box (ready_1 \supset \bigcirc (\neg inP_1 \, U \, begin_1)) \\
A \Box (begin_1 \supset \bigcirc (\neg inP_1 \, U \, end_1)) \\
A \Box (end_1 \supset \bigcirc (\neg inP_1 \, U \, ready_1)) \\
\\
A \Box (begin_1 \supset ((E \bigcirc inP_1) \, U \, inP_1)) \\
A \Box (end_1 \supset ((E \bigcirc inP_1) \, U \, inP_1)) \\
A \Box (ready_1 \supset \bigcirc \Diamond \, inP_1)
\end{array}
$$

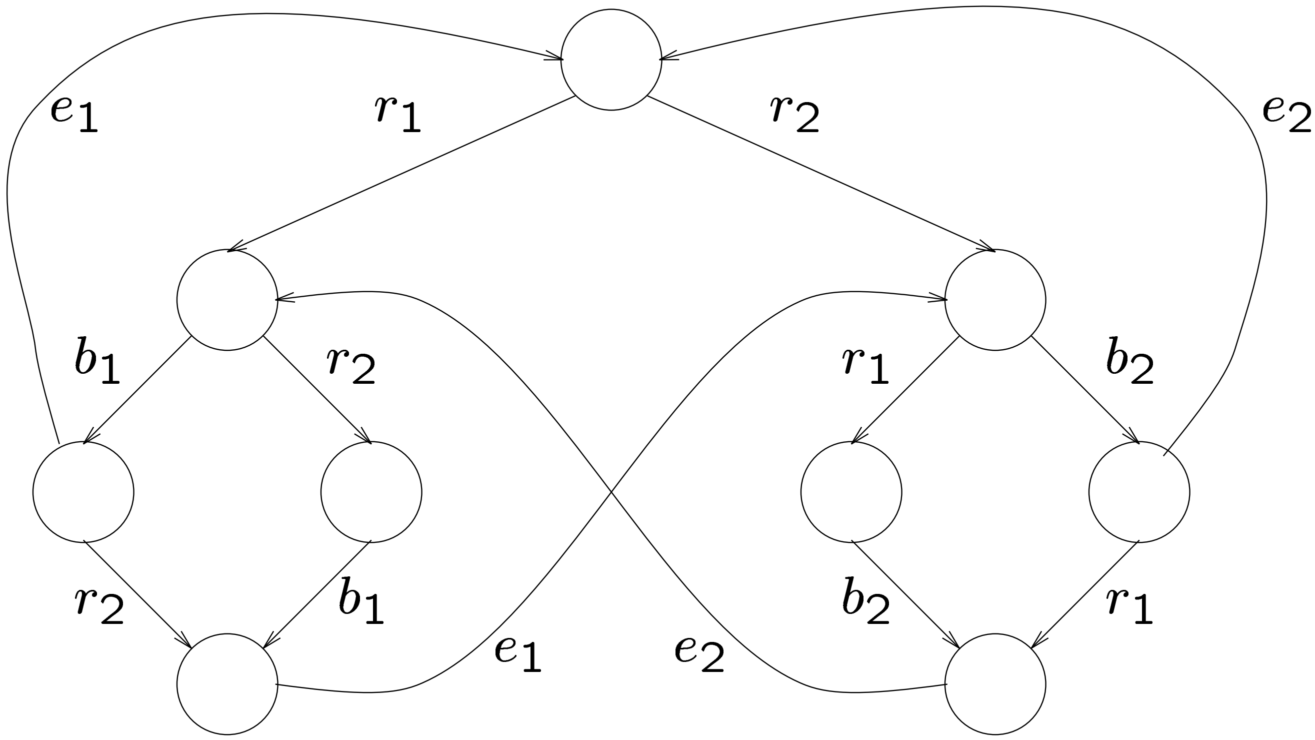
$$A \bigcirc(\neg inP_2 \, U \, ready_2) \wedge E \bigcirc(ready_2)$$
$$A \,\Box(ready_2 \supset \bigcirc(\neg inP_2 \, U \, begin_2))$$
$$A \,\Box(begin_2 \supset \bigcirc(\neg inP_2 \, U \, end_2))$$
$$A \,\Box(end_2 \supset \bigcirc(\neg inP_2 \, U \, ready_2))$$

$$A \,\Box(begin_2 \supset ((E \bigcirc inP_2) \, U \, inP_2))$$
$$A \,\Box(end_2 \supset ((E \bigcirc inP_2) \, U \, inP_2))$$
$$A \,\Box(ready_2 \supset \bigcirc \Diamond \, inP_2)$$

$$A \,\Box(begin_1 \supset \neg begin_2 \, U \, end_1)$$
$$A \,\Box(begin_2 \supset \neg begin_1 \, U \, end_2)$$

with $inP_i \equiv begin_i \vee end_i \vee ready_i$, $i = 1, 2$ and the assumption that only one action is done at each time.

The result



Looks Nicer, but ...

- The specification is even more complicated than in the linear case.
- The result is global: it assumes a single process handling all synchronization problems.
- Fairness among the processes is still needed at the level of the "ready" actions, except if they can be executed unconditionally, which requires the global monitoring of the process states.

This synthesis algorithm yields something that looks interesting but is not a usable solution to the mutual exclusion problem.

Both the linear and branching approaches synthesize a global system, not a distributed concurrent one.

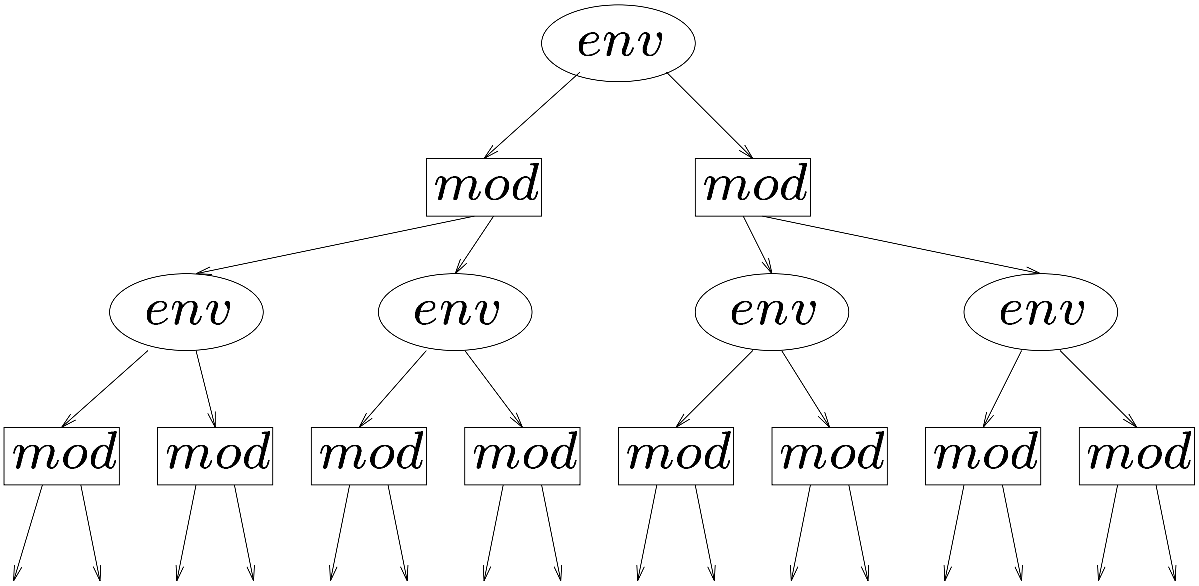
Synthesizing a Single Process

- Since the synthesis methods can only meaningfully produce a single entity, why not use them to synthesize a *single process* or *module*.
- Possible, but the problem is different: the process operates in an environment it does not control.
- One thus needs to assume that some events are controlled by the environment and some by the process.
- The problem is then to build a process that produces the correct behavior whatever the environment does.
- One can view this as a game between the environment and the process and the problem is to find a winning strategy for the process.

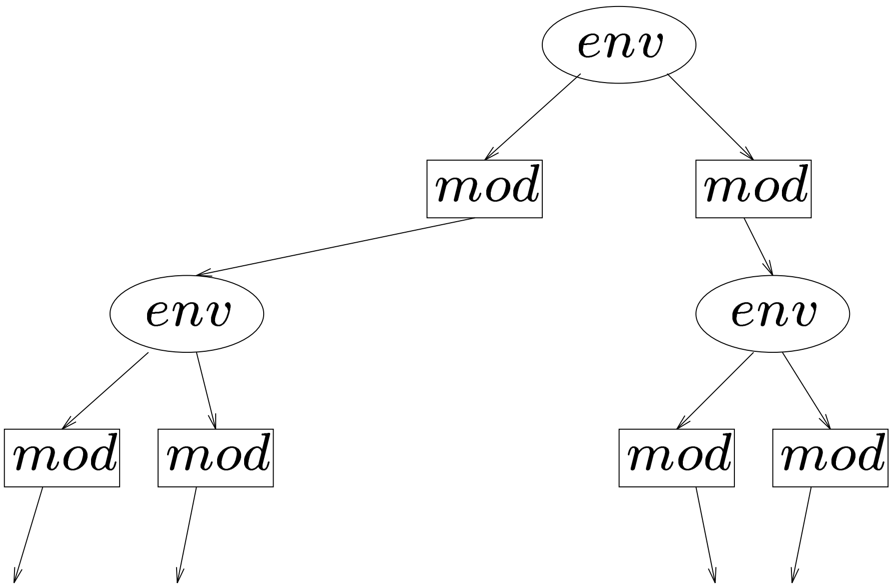
Solving Module Synthesis

This problem was considered in [Pnueli & Rosner 1985] as well as in [Abadi, Lamport & Wolper 85], but its solution goes back to [Büchi & Landweber 69].

The solution uses automata on trees and is the following. One considers the infinite tree representing the game between the environment and the module.



The specification is a *linear* requirement and the problem is to see if it can be met for all computations by choosing module moves.



env
mod
mod
env
env
mod
mod
mod
mod

To do this, one builds a *finite tree automaton* that checks the following:

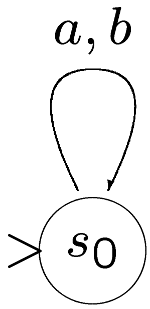
- The tree is obtained from the game tree by choosing module moves.
- All paths satisfy the given (linear) specification.

One then checks if the automaton accepts some tree. If it does such a tree can be finitely generated, which yields a program for the module.

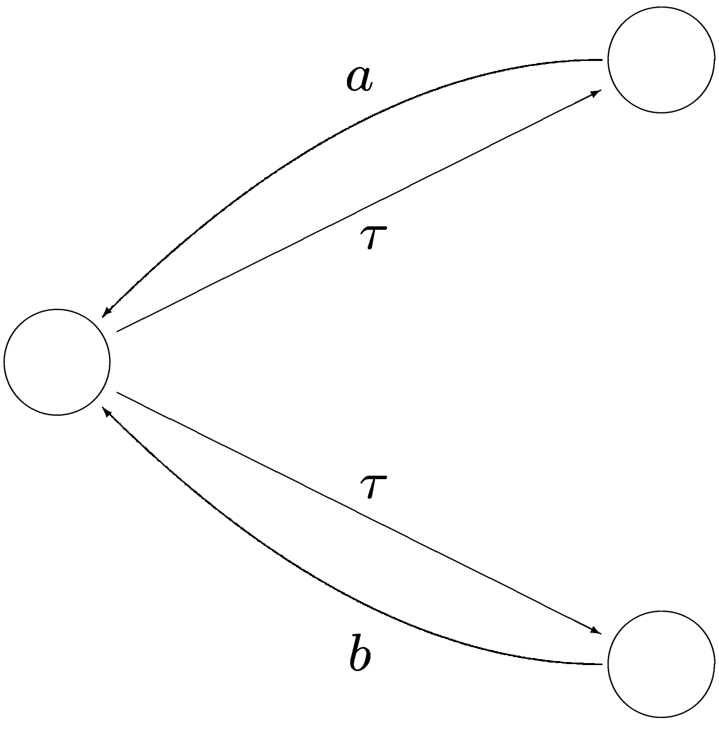
If the environment is given by a transition system and the specification by a linear temporal logic formula, the procedure is doubly exponential.

Module Synthesis
An example

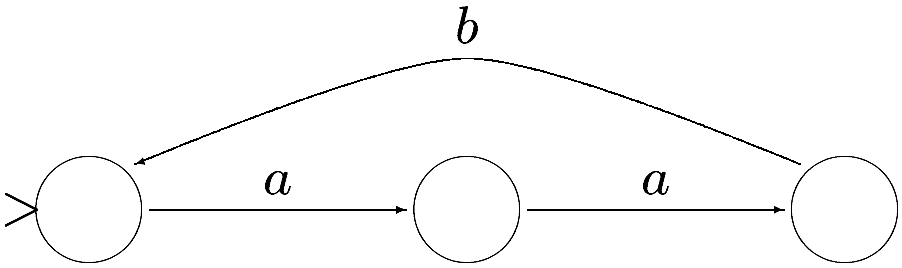
A cooperative environment (rendez-vous synchronization)



A less cooperative environment



A module for realizing $\Box \Diamond a \wedge \Box \Diamond b$ (cooperative environment).



For the noncooperative environment, the synthesis is impossible.

Much Nicer, but . . .

- The complexity of the procedure is such that it is quite problematic to use in practice.
- This solves the problem of building *one* module. What is usually needed is the ability to synthesize *several* interacting modules.
- One could synthesize a combined module and then decompose it, but decomposing is a hard problem with a huge solution space.
- There is nevertheless an application of the procedure that make sense: controller synthesis

This is a powerful procedure, but using it in practice is problematic.

What has Failed

- The hope was to synthesize meaningful concurrent algorithms. This has completely failed.
- Indeed, the methods we have outlined are not designed to cope with the huge solution space of distributed algorithms.
- It would actually be quite astonishing to find a general algorithmic method which could do so.

Challenge Problem: Find a method that can synthesize the alternating-bit protocol.

What has Succeeded

- Building transition systems from temporal logic formulas *is* useful, but for *verification*.
- Module or controller synthesis could yet be turned into something that is practically usable.

Conclusions on Synthesis in General

- Meaningful general synthesis is very hard to achieve.
- For more limited problems, interesting techniques can sometimes be found.
- Results developed with the goal of synthesis can have other valuable applications.