Synthesis of Programs from Temporal Property Specifications

Amir Pnueli

New York University and Weizmann Institute of Sciences (Emeritus)

Seminar on Verification, Distributed Computing, and Computing Sciences
Ben Gurion University, June, 2009

Based on Joint work with

Uri Klein, Nir Piterman, Roni Rosner, Yaniv Sa'ar,

Research Supported in part by SRC grant 2004-TJ-1256 and the European Union project Prosyd.

Motivation

Why verify, if we can automatically synthesize a program which is correct by construction?

The Synthesis Problem

Given an interface specification (identification of input and output variables) and a behavioral specification, e.g. an LTL formula φ for a desired reactive system.

- Determine if there exists an implementation that realizes the specification. That is, maintain the specification φ against all possible behaviors of the environment.
- If the specification is realizable, construct an implementation.

Example of a Specification

Behavioral Specification:

$$\square\,(\bigcirc x = (y \oplus \bigcirc y))$$

Is this specification realizable?

The essence of synthesis is the conversion

From relations to Functions.

Historically

The synthesis problem has been first formulated by Church in 1963. In fact, he already discussed the problem in a Summer Institute in 1957.

In 1969, Büchi and Landweber provided a first solution to Church's problem. Solution was based on infinite games.

In 1972, M. Rabin provided a second solution. Solution was based on the theory of automata on Infinite Trees developed by him in 1969.

These two techniques (Games and Trees) are still the main techniques for performing synthesis.

In 1981 Wolper and Emerson in their PH.D. theses, reconsidered the problem from a CS perspective. They both concluded that φ is realizable iff it is satisfiable.

Realizability $\sqsubset$ Satisfiability

There are two different reasons why a specification may fail to be realizable.

Inconsistency

$$\diamond g \quad \wedge \quad \square \neg g$$

Non-Causality For a system

Realizing the specification

$$g \quad \longleftrightarrow \quad \diamond r$$

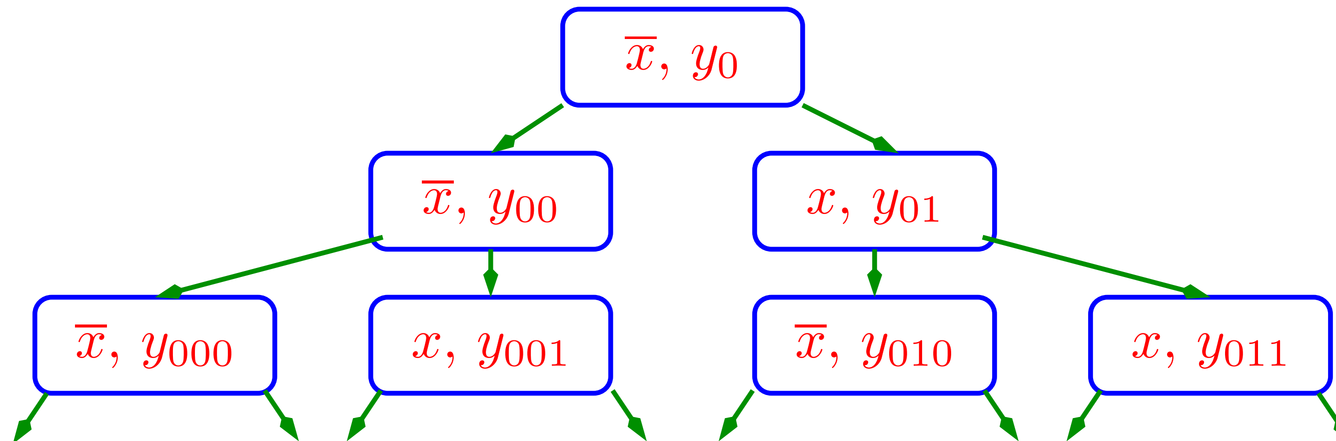
requires clairvoyance.

The essence of reactive systems synthesis is the conversion from relations to causal functions.

Maintaining Specification Against Adversarial Environments

In 1988, Rosner claimed that realizability should guarantee the LTL specification φ against all possible (including adversarial) environments.

To solve the problem one must find a satisfying tree where the branching represents all possible inputs:



Can be formalized by stating that [PR-POPL89]

The specification $\varphi(x, y)$ is realizable iff the CTL* formula $\forall x \, \exists y$ **A** $\varphi(x, y)$ is valid (over all trees).

The operator **A** is the CTL "for-all-paths" path quantifier.

The Bad News

The same paper [PR-POPL89] showed that the synthesis process has worst case complexity which is doubly exponential. In the upper bound, the first exponent comes from the translation of φ into a non-deterministic Büchi automaton. The second exponent is due to the determinization of the automaton.

This result doomed synthesis to be considered highly intractable, and discouraged further research on the subject for a long time.

It Needs an Outsider to Brave a Double Exponent Lower Bound

In 1989, Ramadge and Wonham introduced the notion of controller synthesis and showed that for a specification of the form $\square\, p$, the controller can be synthesized in linear time.

In 1995, Asarin, Maler, P, and Sifakis, extended controller synthesis to timed systems, and showed that for specifications of the form $\square\, p$ and $\diamond\, q$, the problem can be solved by symbolic methods in linear time.

Property-Based System Design

While the rest of the world seems to be moving in the direction of model-based design (see UML) as the prevalent development methodology, some of us persisted with the vision of property-based approach.

Specification is stated declaratively as a set of properties, from which a design can be extracted.

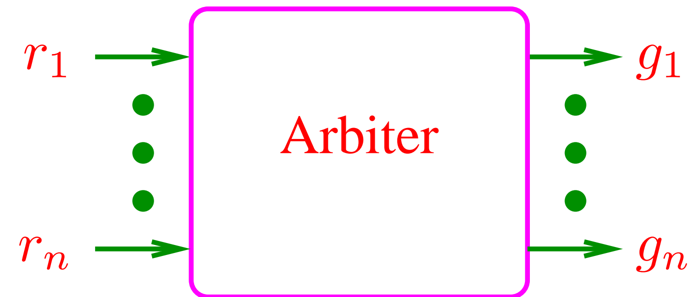
This has been investigated in the hardware-oriented European project PROSYD.

Design synthesis is needed in two places in the development flow:

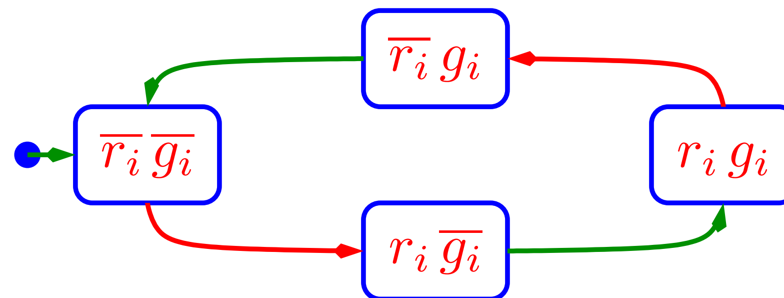
- Automatic synthesis of small blocks whose time and space efficiency are not critical.
- As part of the specification analysis phase, ascertaining that the specification is realizable.

Example Specification: An Arbiter

Consider a specification for an arbiter.



The protocol for each client:



Required to satisfy

$$\bigwedge_i \square\diamond\neg(r_i \wedge g_i) \quad\rightarrow\quad \bigwedge_{i\neq j} \square\neg(g_i \wedge g_j) \;\wedge\; \bigwedge_i \square\diamond(g_i = r_i)$$

Solving Games for Generalized Reactivity[1] (Streett[1])

Following [KPP03], the work in [PPS06] developed an N^3 algorithm for solving games whose winning condition is given by the (generalized) Reactivity[1] condition

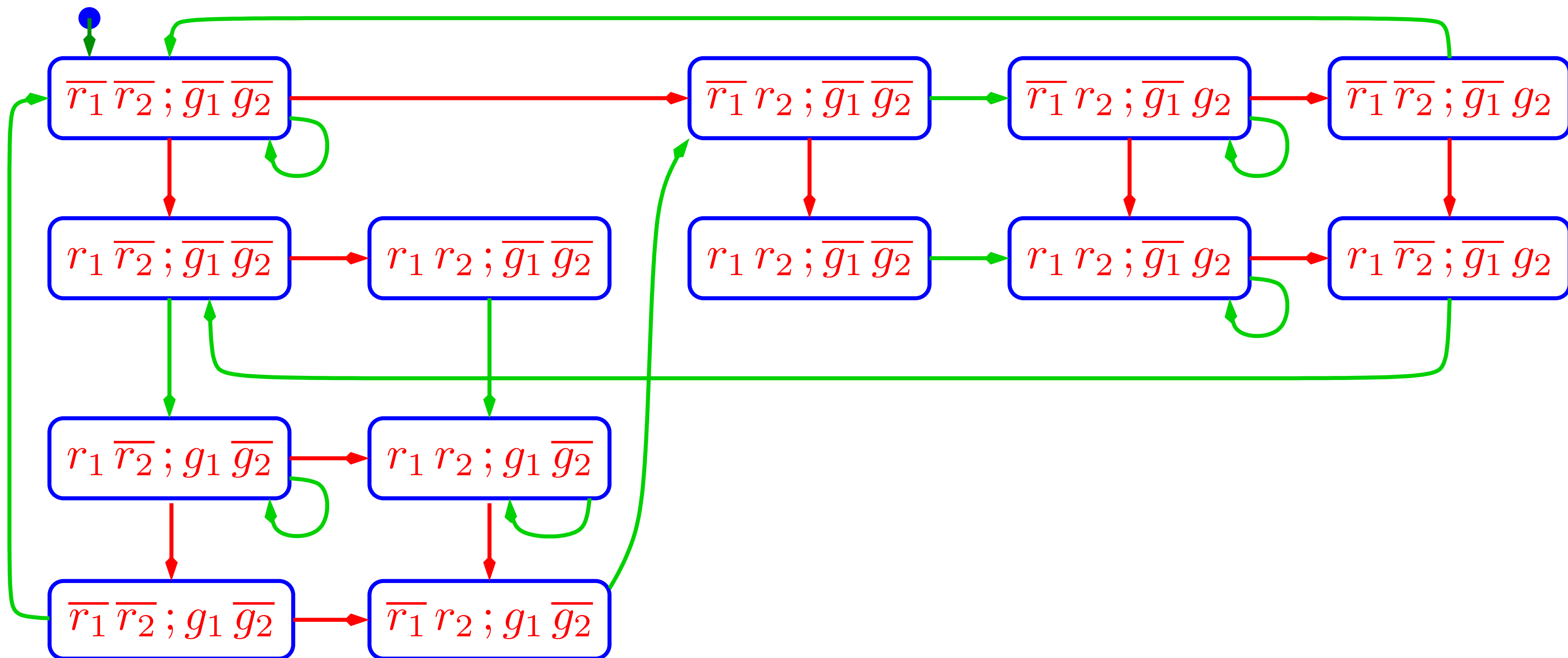
$$(\Box\Diamond p_1 \land \Box\Diamond p_2 \land \cdots \land \Box\Diamond p_m) \quad \rightarrow \quad \Box\Diamond q_1 \land \Box\Diamond q_2 \land \cdots \land \Box\Diamond q_n$$

This class of properties is bigger than the properties specifiable by deterministic Büchi automata. It covers a great majority of the properties we have seen so far. In fact, it covered all the sample specifications that were considered within the Prosyd project.

For example, it covers the realizable version of the specification for the Arbiter design.

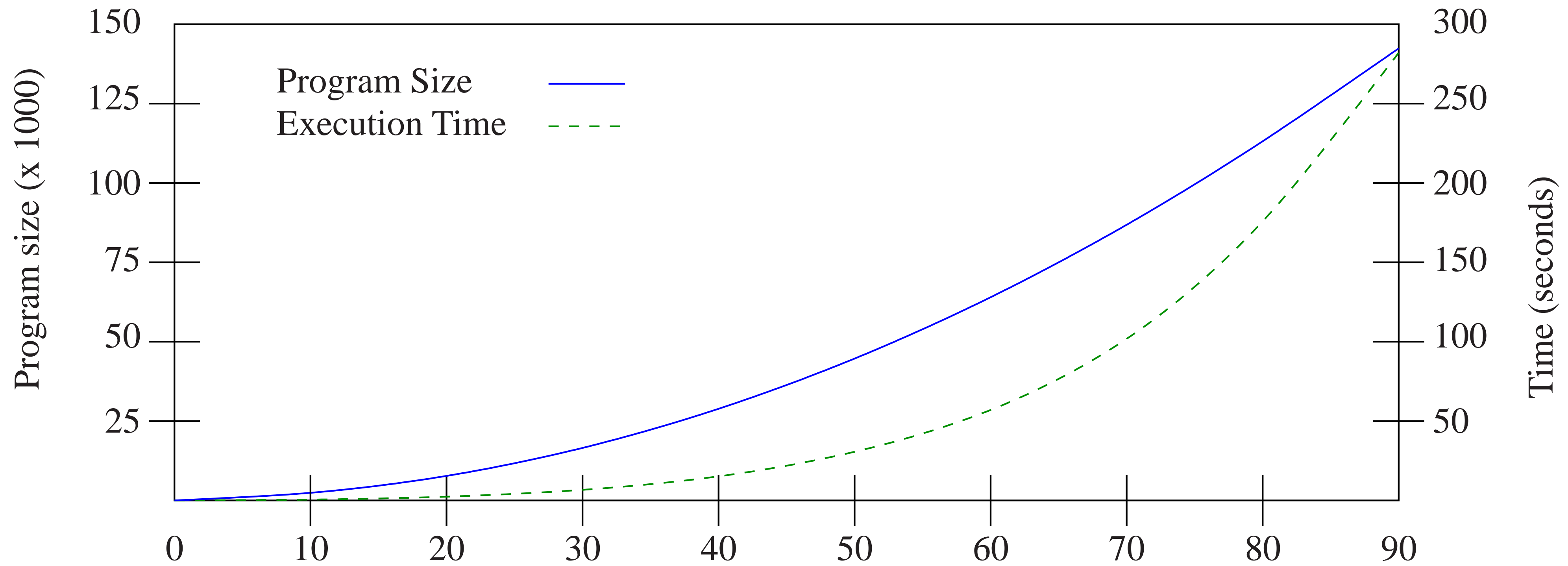
Results of Synthesis

The synthesis algorithm is based on the representation of the situation as a two-person game in which the environment is the first player, while the system is the second player, attempting to maintain the specification. The design realizing the specification can be extracted as the winning strategy for Player 2. Applying this to the Arbiter specification, we obtain the following design:



There exists a symbolic algorithm for extracting the implementing design.

Execution Times and Programs Size for Arbiter(N)



From Circuits to Programs

The previous methods provided a very satisfactory partial solution to the problem of circuit synthesis. What about the synthesis of programs?

What is the difference? In circuit synthesis we consider a synchronous system in which the inputs and outputs are synchronized by a common clock. The consequences are:

- Every change in the inputs is observable by the system.
- Every output that can be computed in a single step is immediately visible to the environment.
- A single step may modify several (possibly all) input variables while modifying at the same step several (possibly all) output variables.

In contrast, shared-variables programs are asynchronous. As a result, the above three premises are no longer valid.

Illustrating the Difference

Consider a system

with the specification

$$\square\ (y = x)$$

This specification is synchronously realizable by a circuit that implements the transition relation $y' = x'$.

It is asynchronously unrealizable. A shared-variable program cannot observe all the possible changes in x and modify y accordingly. In particular, y cannot change its value in the same step that x is modified.

The Rosner Reduction

The work in [PR-ICALP89] proposed the following reduction from asynchronous synthesis to synchronous synthesis:

Claim 1. *The specification $\varphi(x, y)$ with Boolean input x and Boolean output y is asynchronously realizable iff the formula $\mathcal{X}(x, r; y)$ with Boolean inputs x, r and Boolean output y is synchronously realizable.*

The *kernel formula* $\mathcal{X}$ is defined by

$$\mathcal{X}(x, r; y) : \quad \left(\begin{array}{ll} \overline{r} & \wedge \\ \Box\Diamond r & \wedge \\ \Box\Diamond \overline{r} & \end{array}\right) \rightarrow \left(\begin{array}{ll} \varphi(x, y) & \wedge \\ (r \vee \ominus \overline{r}) \Rightarrow (y = \ominus y) & \wedge \\ (\forall \widetilde{x}).(r \wedge \ominus \overline{r} \Rightarrow (x = \widetilde{x})) \rightarrow \varphi(\widetilde{x}, y) & \end{array}\right)$$

The auxiliary variable r is a scheduling variable which determines the points in the computation in which the system may read the value of the input variable x, or modify the value of the output variable y. Initially, $r = 0$. Variable x may be read whenever r rises from 0 to 1. Variable y may be modified whenever r drops from 1 to 0.

The proof of the claim also provides a translation of the synthesized synchronous circuit into an asynchronous program.

Problems with the Reduction

In principle, Claim 1 provides a complete solution to the problem of asynchronous synthesis. However, in comparison with the synchronous case, the complexity of the solution is unacceptable.

Even in the case that the specification φ is restricted to the GR(1) class, the clause

$$\beta_3 : \quad (\forall^{\approx}\widetilde{x}) \,.(r \wedge \ominus\overline{r} \Rightarrow (x = \widetilde{x})) \rightarrow \varphi(\widetilde{x}, y)$$

may still be of exponential size. We conjecture that this exponential blowup is inherent, by comparison to other cases of synthesis under partial observability.

Thus, while [PPS06] proposes a practical solution to the synchronous synthesis problem, we are missing a similar solution for the asynchronous case.

The Proposed Remedy

We propose to develop two formulas $\mathcal{X}_{\forall\exists}(x, \widetilde{x}, r; y)$ and $\mathcal{X}_{\psi}(x, r; y)$ that can be viewed as two-sided bounds on $\mathcal{X}$. This can be schematically represented by the "inequalities"

$$\mathcal{X}_{\psi} \quad \sqsubseteq \quad \mathcal{X} \quad \sqsubseteq \quad \mathcal{X}_{\forall\exists}$$

The properties of these two approximations can be summarized as follows:

- The over-approximation $\mathcal{X}_{\forall\exists}(x, \widetilde{x}, r; y)$ guarantees that if $\mathcal{X}$ is synchronously realizable then so is $\mathcal{X}_{\forall\exists}$. It follows that if $\mathcal{X}_{\forall\exists}$ is not synchronously realizable, then φ is not asynchronously realizable. Thus, $\mathcal{X}_{\forall\exists}$ provides us with a test for unrealizability of φ.
- The under-approximation $\mathcal{X}_{\psi}(x, r; y)$ guarantees that if $\mathcal{X}_{\psi}$ is synchronously realizable then so is $\mathcal{X}$. It follows that if $\mathcal{X}_{\psi}$ is synchronously realizable, then φ is asynchronously realizable, and we can extract from the realization of $\mathcal{X}_{\psi}$ an asynchronous implementation (program) implementing φ. Thus, $\mathcal{X}_{\psi}$ provides us with a test for the realizability of φ.
- Both $\mathcal{X}_{\forall\exists}$ and $\mathcal{X}_{\psi}$ are complexity preserving in the sense that if φ belongs to the GR(1) class, then so do $\mathcal{X}_{\forall\exists}$ and $\mathcal{X}_{\psi}$. It follows that the two described tests are practical, i.e. can be performed in time N^3.

Establishing Unrealizability

Define

$$\mathcal{X}_{\forall\exists}(x,\widetilde{x},r;y): \quad \begin{pmatrix} \overline{r} & \wedge \\ \Box\Diamond r & \wedge \\ \Box\Diamond \overline{r} & \end{pmatrix} \rightarrow \begin{pmatrix} \varphi(x,y) & \wedge \\ (r \vee \circleddash \overline{r}) \Rightarrow (y = \circleddash y) & \wedge \\ (r \wedge \circleddash \overline{r} \Rightarrow (x = \widetilde{x})) \rightarrow \varphi(\widetilde{x},y) & \end{pmatrix}$$

Recalling that

$$\mathcal{X}(x,r;y): \quad \begin{pmatrix} \overline{r} & \wedge \\ \Box\Diamond r & \wedge \\ \Box\Diamond \overline{r} & \end{pmatrix} \rightarrow \begin{pmatrix} \varphi(x,y) & \wedge \\ (r \vee \circleddash \overline{r}) \Rightarrow (y = \circleddash y) & \wedge \\ (\forall^{\approx}\widetilde{x}).(r \wedge \circleddash \overline{r} \Rightarrow (x = \widetilde{x})) \rightarrow \varphi(\widetilde{x},y) & \end{pmatrix}$$

we claim that the following implication is valid:

$$\forall x, r\, \exists y\, \mathbf{A}\, \mathcal{X}(x,r;y) \quad \rightarrow \quad \forall x, \widetilde{x}, r\, \exists y\, \mathbf{A}\, \mathcal{X}_{\forall\exists}(x,\widetilde{x},r;y)$$

This is because we can transform the left-hand side into the right-hand side by shifting the quantification $(\forall^{\approx}\widetilde{x})$ across the operator **A**, the quantification $\exists y$, and then transforming it to $(\forall\widetilde{x})$, all of which are weakening transformations.

It follows that

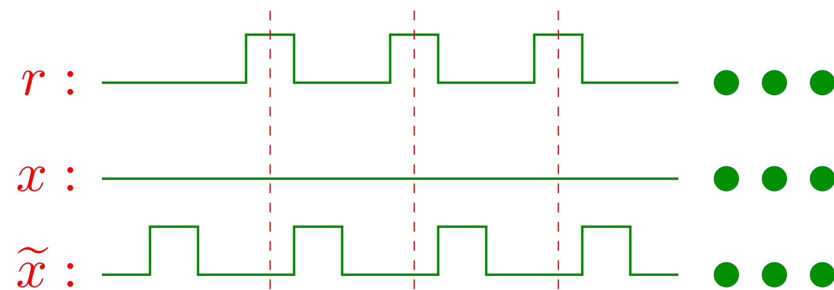
If $\mathcal{X}$ is synchronously realizable, then so is $\mathcal{X}_{\forall\exists}$.

Establishing that $\square\ (y = x)$ is Not Asynchronously Realizable

We consider the specification $\varphi_1 : \square\ (y = x)$ and show that it is asynchronously unrealizable.

To do so, we apply the unrealizability test by showing that the formula $\mathcal{X}_{\forall\exists}(x, \widetilde{x}, r; y)$ is not synchronously realizable. It is sufficient to display a sequence of states $\sigma : s_0, s_1, \ldots$ on which r oscillates infinitely many times, x and $\widetilde{x}$ agree on all reading points, and then show that σ cannot satisfy both $\square\ (y = x)$ and $\square\ (y = \widetilde{x})$.

Consider the following state sequence:



Obviously $x = \widetilde{x}$ at all reading points. However, if both $\square\ (y = x)$ and $\square\ (y = \widetilde{x})$ would have held, then we must have had $x = \widetilde{x}$ at all points. However, this is obviously not the case.

We conclude that $\square\ (y = x)$ is not asynchronously realizable.

Next Attempt

Since $y = x$ is a specification of a copying module, let us try to capture the essence of copying in a specification that can do it in an asynchronous setting.

Some relevant requirements can include:

- Whenever x rises to 1, then sometimes later y should rise to 1.
- Whenever x drops to 0, then sometimes later y should drop to 0.
- Variable y should not rise to 1, unless sometimes before x was 1.
- Variable y should not drop to 0, unless sometimes before x was 0.

A temporal formula that captures these four requirements may be given by the following specification:

$$\varphi_2 : (x \Rightarrow \diamondsuit\, y) \;\wedge\; (\overline{x} \Rightarrow \diamondsuit\, \overline{y}) \;\wedge\; (y \Rightarrow y\, \mathcal{S}\, \overline{y}\, \mathcal{S}\, x) \;\wedge\; \bigcirc(\overline{y} \Rightarrow \overline{y}\, \mathcal{B}\, y\, \mathcal{S}\, \overline{x})$$

Applying the unrealizability test to φ_2, we find that also φ_2 is not asynchronously realizable.

Restraining the Variability of x

The weakness of the specification φ_2 is that it allows the environment to modify x too quickly without waiting for an evidence that the system has noticed the most recent change. We can correct this drawback by allowing the environment to modify x only at points in which $x = y$. For example, we can suggest the following specification:

$$\varphi_3 : ((x \neq y) \Rightarrow (x = \bigcirc x)) \rightarrow \begin{pmatrix} (x \Rightarrow \Diamond y) \wedge (\overline{x} \Rightarrow \Diamond \overline{y}) \wedge \\ (y \Rightarrow y \, \mathcal{S} \, \overline{y} \, \mathcal{S} \, x) \wedge \bigcirc(\overline{y} \Rightarrow \overline{y} \, \mathcal{B} \, y \, \mathcal{S} \, \overline{x}) \end{pmatrix}$$

According to this specification, the environment may modify x only in states at which $x = y$. Applying the unrealizability test to φ_3, we find that its corresponding $\mathcal{X}_{\forall\exists}$ is synchronously realizable. Unfortunately, no conclusions can be inferred from a failure of the unrealizability test.

Establishing Realizability

Assume that we can find a formula $\psi(x, r; y)$ that satisfies the following two implications:

$$Imp : \begin{cases} \alpha \wedge ((r \vee \ominus\overline{r}) \Rightarrow (y = \ominus y)) \wedge \psi & \rightarrow \quad \varphi(x, y) \\ \alpha \wedge ((r \vee \ominus\overline{r}) \Rightarrow (y = \ominus y)) \wedge \psi \wedge (r \wedge \ominus\overline{r} \Rightarrow (x = \widetilde{x})) & \rightarrow \quad \varphi(\widetilde{x}, y) \end{cases}$$

where $\alpha : \overline{r} \wedge \square\Diamond r \wedge \square\Diamond\overline{r}$. Define the formula

$$\mathcal{X}_\psi(x, r; y) : \quad (\overline{r} \wedge \square\Diamond r \wedge \square\Diamond\overline{r}) \quad \rightarrow \quad \psi(x, r; y) \ \wedge \ ((r \vee \ominus\overline{r}) \Rightarrow (y = \ominus y))$$

Recalling the definition

$$\mathcal{X}(x, r; y) : \quad \begin{pmatrix} \overline{r} & \wedge \\ \square\Diamond r & \wedge \\ \square\Diamond\overline{r} & \end{pmatrix} \rightarrow \begin{pmatrix} \varphi(x, y) & \wedge \\ (r \vee \ominus\overline{r}) \Rightarrow (y = \ominus y) & \wedge \\ (\forall^{\approx}\widetilde{x}) . (r \wedge \ominus\overline{r} \Rightarrow (x = \widetilde{x})) \rightarrow \varphi(\widetilde{x}, y) & \end{pmatrix}$$

we can use Imp to infer the implication $\mathcal{X}_\psi(x, r; y) \ \rightarrow \ \mathcal{X}(x, r; y)$ which leads to the following

Realizability Test:

> If $\mathcal{X}_\psi$ is synchronously realizable, then φ is asynchronously realizable.

Heuristics for Finding ψ

The main question is how to find a ψ that will satisfy the implications Imp together with φ.

At present, we offer the following heuristics:

Heuristic A:

> Replace in φ one or more occurrences of $x = v$ by $(x = v) \land \ominus \overline{r} \land r$, which means that x equals v at a reading point.

For example, we may transform

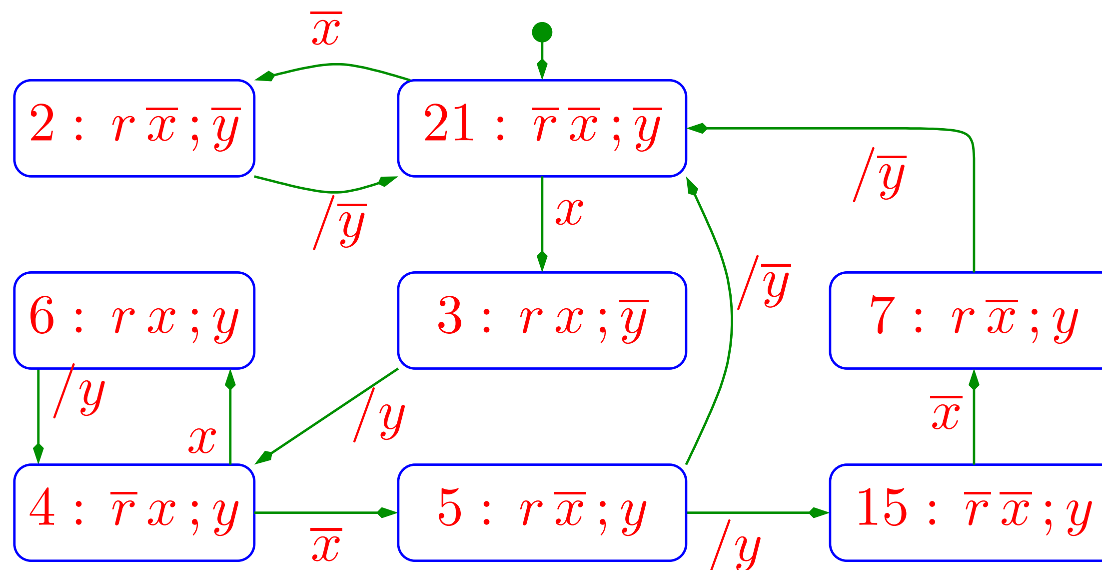
$$\varphi_3 : ((x \neq y) \Rightarrow (x = \bigcirc x)) \rightarrow \begin{pmatrix} (x \Rightarrow \diamondsuit\, y) \;\land\; (\overline{x} \Rightarrow \diamondsuit\, \overline{y}) \;\land \\ (y \Rightarrow y\, \mathcal{S}\, \overline{y}\, \mathcal{S}\, x) \;\land\; \bigcirc(\overline{y} \Rightarrow \overline{y}\, \mathcal{B}\, y\, \mathcal{S}\, \overline{x}) \end{pmatrix}$$

into

$$\psi_3 : ((x \neq y) \Rightarrow (x = \bigcirc x)) \rightarrow \begin{bmatrix} (x \Rightarrow \diamondsuit\, y) \;\land\; (\overline{x} \Rightarrow \diamondsuit\, \overline{y}) \quad \land \\ (y \Rightarrow y\, \mathcal{S}\, \overline{y}\, \mathcal{S}\, (\ominus \overline{r} \land r \land x)) \quad \land \\ \bigcirc(\overline{y} \Rightarrow \overline{y}\, \mathcal{B}\, y\, \mathcal{S}\, (\ominus \overline{r} \land r \land \overline{x})) \end{bmatrix}$$

Continuing with the Synthesis of φ_3

The validity of the implications Imp can be checked by a tool such as TLV. They proved valid for φ_3 and ψ_3. We continued to apply synchronous synthesis to $\mathcal{X}_{\psi_3}$. The algorithm declared this specification to be synchronously realizable, and yielded a synchronous automaton. From this automaton we managed to extract an asynchronous program as follows:



Conclusions

- Synthesis of programs (asynchronous systems) can also be done in a practical mode, provided we agree to restrict our specification language to a simpler fragment.
- Unlike the synchronous case, we have to rely on heuristics for finding under-approximations. Many more should be developed.
- In future research we will consider the multi-variable case.