Asynchronous Composition

Stavros Tripakis
University of California, Berkeley
Composition of Discrete Systems

Two major paradigms:

- **Synchronous composition**
  - All sub-systems move together: in “lock-step”
  - Main application: synchronous circuits
    - all sub-circuits having the same clock

- **Asynchronous composition**
  - Each sub-system moves at its own pace
  - Applications: concurrent software (processes, threads, ...), non-synchronized distributed systems, asynchronous circuits, ...

Common principle: the state-space of the *composite* (also called *product*) system is the product of the state-spaces of its components (subsystems).
Symbolic Asynchronous Composition

Given two KS $K_1$ and $K_2$, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

Recall: symbolic synchronous composition:

$$K_1 \times K_2 = (Init_1 \land Init_2, Trans_1 \land Trans_2)$$

is this correct?

No: need to state also that the other process doesn’t move.
Symbolic Asynchronous Composition

Given two KS $K_1$ and $K_2$, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

Recall: symbolic synchronous composition:

$$K_1 \times K_2 = (Init_1 \land Init_2, Trans_1 \land Trans_2)$$

How can the asynchronous composition $K_1 || K_2$ be represented symbolically?
Symbolic Asynchronous Composition

Given two KS $K_1$ and $K_2$, each represented symbolically as

$$K_i = (\text{Init}_i, \text{Trans}_i)$$

Recall: symbolic synchronous composition:

$$K_1 \times K_2 = (\text{Init}_1 \land \text{Init}_2, \text{Trans}_1 \land \text{Trans}_2)$$

How can the asynchronous composition $K_1|\!|K_2$ be represented symbolically?

$$K_1|\!|K_2 = (\text{Init}_1 \land \text{Init}_2, \text{Trans}_1 \lor \text{Trans}_2)$$

is this correct?
Symbolic Asynchronous Composition

Given two KS $K_1$ and $K_2$, each represented symbolically as

$$K_i = (\text{Init}_i, \text{Trans}_i)$$

Recall: symbolic synchronous composition:

$$K_1 \times K_2 = (\text{Init}_1 \land \text{Init}_2, \text{Trans}_1 \land \text{Trans}_2)$$

How can the **asynchronous** composition $K_1||K_2$ be represented symbolically?

$$K_1||K_2 = (\text{Init}_1 \land \text{Init}_2, \text{Trans}_1 \lor \text{Trans}_2)$$

is this correct?

No: need to state also that the other process doesn’t move.
Symbolic Asynchronous Composition: second attempt

Given two KS $K_1$ and $K_2$, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

where $\vec{x}_i$ are the variables of $K_i$.

Symbolic asynchronous composition (2nd attempt):

$$K_1 || K_2 = (Init_1 \land Init_2, (Trans_1 \land \vec{x}'_2 = \vec{x}_2) \lor (Trans_2 \land \vec{x}'_1 = \vec{x}_1))$$
Symbolic Asynchronous Composition: second attempt

Given two KS $K_1$ and $K_2$, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

where $\vec{x}_i$ are the variables of $K_i$.

Symbolic asynchronous composition (2nd attempt):

$$K_1 || K_2 = (Init_1 \land Init_2, (Trans_1 \land \vec{x}'_2 = \vec{x}_2) \lor (Trans_2 \land \vec{x}'_1 = \vec{x}_1))$$

Is it correct now?
Symbolic Asynchronous Composition: second attempt

Given two KS $K_1$ and $K_2$, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

where $\vec{x}_i$ are the variables of $K_i$.

Symbolic asynchronous composition (2nd attempt):

$$K_1 \parallel K_2 = (Init_1 \land Init_2, (Trans_1 \land \vec{x}'_2 = \vec{x}_2) \lor (Trans_2 \land \vec{x}'_1 = \vec{x}_1))$$

Is it correct now?

What if the two systems share variables?
Symbolic Asynchronous Composition: second attempt

Given two KS $K_1$ and $K_2$, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

where $\vec{x}_i$ are the variables of $K_i$.

Symbolic asynchronous composition (2nd attempt):

$$K_1 || K_2 = (Init_1 \land Init_2, (Trans_1 \land \vec{x}_2' = \vec{x}_2) \lor (Trans_2 \land \vec{x}_1' = \vec{x}_1))$$

Is it correct now?

What if the two systems share variables?

No problem if no shared written variables, but problems otherwise.
Symbolic Asynchronous Composition: second attempt

Consider two asynchronous processes writing to a shared variable $x$:

$\begin{align*}
\text{s}_0 & \xrightarrow{x++} \\
\text{p}_0 & \xrightarrow{x++}
\end{align*}$

Composite transition relation according to 2nd attempt:

$\begin{align*}
x' &= x + 1 \\
\text{from process 1} \\
x' &= x + 1 \\
\text{from process 2} \\
\iff & \text{false}
\end{align*}$
Symbolic Asynchronous Composition: second attempt

Consider two asynchronous processes writing to a shared variable $x$:

```
x++
```

$s_0$ || $p_0$

Composite transition relation according to 2nd attempt:

$$x' = x + 1 \land x' = x \lor x' = x + 1 \land x' = x$$

from process 1

from process 2
Symbolic Asynchronous Composition: second attempt

Consider two asynchronous processes writing to a shared variable $x$:

$$x++$$  \hspace{1cm} || \hspace{1cm} x++$$

\hspace{1cm}

$S_0$  \hspace{2cm} $P_0$

Composite transition relation according to 2nd attempt:

$$x' = x + 1 \land x' = x \lor x' = x + 1 \land x' = x \iff \text{false}$$

from process 1 \hspace{1cm} from process 2
Symbolic Asynchronous Composition: second attempt

Consider two asynchronous processes writing to a shared variable $x$:

$\quad x++$

$\quad s_0$

$\quad \parallel$

$\quad x++$

$\quad p_0$

Composite transition relation according to 2nd attempt:

$$x' = x + 1 \land x' = x \lor x' = x + 1 \land x' = x \iff false$$

Need to talk explicitly about shared variables.
Given two KS $K_1$ and $K_2$, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

their asynchronous composition $K_1||K_2$ can be represented symbolically as

$$K_1||K_2 = (Init_1 \land Init_2, (Trans_1 \land \vec{x}'_2 = \vec{x}_2) \lor (Trans_2 \land \vec{x}'_1 = \vec{x}_1))$$

where:

- $\vec{x}_i$ are the variables **owned** by $K_i$: they can be read by any process, but they can be **written only by** $K_i$.
- $Trans_i$ may refer also to **shared** variables $\vec{v}$, which are written by both $K_1$ and $K_2$. 
Symbolic Asynchronous Composition: example

Consider our previous example again:

Only one variable, \( x \), shared.

Composite transition relation:

\[
x' = x + 1 \quad \text{from process 1} \quad \text{or} \quad x' = x + 1 \quad \text{from process 2}
\]
Symbolic Asynchronous Composition: example

Consider the modified example:

\[ \begin{align*}
  x &++ \\
  y &++ \\
  z &++ \\
\end{align*} \]

\[ s_0 \parallel p_0 \]

\( x \) shared, \( y \) owned by process 1, \( z \) owned by process 2.

Composite transition relation:
Symbolic Asynchronous Composition: example

Consider the modified example:

```plaintext
x++
++

y++

z++
```

$x$ shared, $y$ owned by process 1, $z$ owned by process 2.

Composite transition relation:

\[
x' = x + 1 \land y' = y + 1 \land z' = z \lor x' = x + 1 \land z' = z + 1 \land y' = y
\]

from process 1

from process 2
Asynchronous Process Communication

Two prominent paradigms:

- **Shared memory (the one we just saw)**
  - A common pool of shared (global) variables
  - Standard in concurrent programming models of today (e.g., threads).
  - Common problems: avoid corrupt values, races, deadlocks (e.g., when semaphores are used), ...

- **Message passing**
  - Transmitter process sends messages to receiver process.
  - Usually some type of **message queue** is used to store messages.
  - Used in several modeling and programming languages and tools, e.g., UML/SysML, Erlang, Go, MPI, TCP/IP, UDP, ...
  - Many different versions, depending whether queues are finite or infinite, single-writer/reader, or multiple-writer/reader, FIFO, lossy, read is blocking, etc.
Communication via Message Passing

Examples of formalisms using message passing:

- **Kahn Process Networks** [Kahn, 1974] (studied in *Systems, Models, and Algorithms* course): infinite queues, single-writer, single-reader, blocking read. A deterministic model!


- **Rendez-vous**:
  - Can be seen (as in Spin/Promela) as message passing with queues of **size zero**.
  - Message cannot be stored in the queue (because queue size is 0) \(\Rightarrow\) transmitter and receiver must **synchronize**.
  \(\Rightarrow\) transmission and reception occurs simultaneously.
  - Common in **process algebras**, e.g., CSP [Hoare, 1985], CCS [Milner, 1980], etc.
Rendez-vous communication: example

CSP notation:

\[ a! \parallel a? = \tau \]

CCS notation:

\[ a \parallel \overline{a} = \tau \]

\( \tau \): silent (or internal) action.
FAIRNESS
Fairness: Motivation

Consider the following asynchronous composition example:

\[
\begin{align*}
  & s_0 \\
  & x++ \\
\end{align*}
\]

\[
\begin{align*}
  & p_0 \\
  & x \leq 4 \\
\end{align*}
\]

\[
\begin{align*}
  & p_1 \\
  & x > 4 \\
\end{align*}
\]

Will the rightmost process ever get to move to \( p_1 \)?
Fairness: Motivation

Consider the following asynchronous composition example:

\[ x++ \]
\[ s_0 \]
\[ x \leq 4 \]
\[ p_0 \]
\[ x > 4 \rightarrow p'=p_1 \]
\[ p_1 \]
\[ x \leq 4 \rightarrow p'=p_0 \]
\[ x'=x \]

Will the rightmost process ever get to move to \( p_1 \)?

Asynchronous composition transition relation:

\[ x' = x + 1 \land p' = p \lor (x > 4 \rightarrow p' = p_1) \land (x \leq 4 \rightarrow p' = p_0) \land x' = x \]

The transition relation allows a process to be neglected forever.
Fairness: Motivation

Consider the following asynchronous composition example:

\[
\begin{align*}
    x &++ \\
    s_0 &\quad || & x &\leq 4 \\
    p_0 &\quad x > 4 \\

    x' &= x + 1 \land p' = p \lor (x > 4 \rightarrow p' = p_1) \land (x \leq 4 \rightarrow p' = p_0) \land x' = x
\end{align*}
\]

Will the rightmost process ever get to move to \( p_1 \)?

Asynchronous composition transition relation:

The transition relation allows a process to be neglected forever. Not realistic: no matter how slow the rightmost process is, it \textbf{will} move at some point \( \Rightarrow \) need to exclude unrealistic behaviors.
**Fairness: Motivation**

Fairness is a mechanism to exclude such unrealistic (unfair) behaviors.

Indispensable for proving properties of systems, e.g.:

- A message will eventually reach its destination: need to assume that the communication channel will not keep losing the message forever. This is a fairness assumption.
- In a distributed protocol, say, leader election, a leader will eventually be elected: need to assume that nodes will not keep failing. Again, a fairness assumption.
- Every bank transaction eventually completes: need to assume that a given transaction will not constantly be overlooked due to other transactions (no starvation). Again, a fairness assumption.
- ...

Observe that the above are liveness properties.

Do we need fairness assumptions to establish safety properties?
Fairness: Motivation

Fairness is a mechanism to exclude such unrealistic (unfair) behaviors.

Indispensable for proving properties of systems, e.g.:

- A message will eventually reach its destination: need to assume that the communication channel will not keep losing the message forever. This is a fairness assumption.
- In a distributed protocol, say, leader election, a leader will eventually be elected: need to assume that nodes will not keep failing. Again, a fairness assumption.
- Every bank transaction eventually completes: need to assume that a given transaction will not constantly be overlooked due to other transactions (no starvation). Again, a fairness assumption.
- ... 

Observe that the above are liveness properties.

Do we need fairness assumptions to establish safety properties?
Defining fairness

We need to be precise: what exactly constitutes a “fair” behavior?

Two basic types [Manna and Pnueli, 1991]:

- **Weak fairness**: a process cannot be enabled forever after some point on, without getting to move.
- **Strong fairness**: a process cannot be enabled infinitely often without getting to move.

where some process $i$ is **enabled** means that the overall system (consisting of process $i$ and potentially other processes) is at a state where process $i$ **can** move.
Defining fairness

We need to be precise: what exactly constitutes a “fair” behavior?

Two basic types [Manna and Pnueli, 1991]:

- **Weak fairness**: a process cannot be enabled forever after some point on, without getting to move.
- **Strong fairness**: a process cannot be enabled infinitely often without getting to move.

where some process $i$ is **enabled** means that the overall system (consisting of process $i$ and potentially other processes) is at a state where process $i$ can move.

There are other types of fairness one may define. Depending on the application, different types of fairness assumptions are used, sometimes expressed in temporal logic. E.g., instead of verifying that $\phi$ holds, we verify that $\phi_{\text{fair}} \rightarrow \phi$ holds.
Weak Fairness

We will define fairness on transition systems directly. Such a transition system may be the result of composition of some processes.

Given a transition system $K$, a state $s$ of $K$, and a transition $a$ of $K$, we say that $a$ is enabled at $s$ iff $K$ has a transition $s \xrightarrow{a} s'$ for some $s'$.

Then we can define weak fairness:

If a transition is always enabled after some point on, it will eventually be taken.
Weak Fairness

We will define fairness on transition systems directly. Such a transition system may be the result of composition of some processes.

Given a transition system $K$, a state $s$ of $K$, and a transition $a$ of $K$, we say that $a$ is enabled at $s$ iff $K$ has a transition $s \xrightarrow{a} s'$ for some $s'$.

Then we can define weak fairness:

*If a transition is always enabled after some point on, it will eventually be taken.*

or better:

*A run $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$ is unfair w.r.t. weak fairness if there exists a transition $a$ and some integer $K$, such that $a$ is enabled at all states $s_i$ with $i \geq K$, but never taken, i.e., $\forall i \geq K : a_i \neq a$. 

Stavros Tripakis (UC Berkeley)

EE 144/244, Fall 2016
Weak Fairness: example

Consider our earlier example. Weak fairness solves this problem:

\[ x++ \]
\[ s_0 \]
\[ x \leq 4 \]
\[ p_0 \]
\[ x > 4 \]
\[ p_1 \]

The run where the transition from \( p_0 \) to \( p_1 \) never happens is unfair w.r.t. weak fairness.
Weak Fairness is Sometimes too Weak

Here, the run where the transition from $p_0$ to $p_1$ never happens is not unfair, because the transition is not constantly enabled after some point on.
Weak Fairness is Sometimes too Weak

More realistic application:

How to ensure that both processes eventually enter their critical section?
Strong Fairness

If a transition is infinitely-often enabled after some point on, it will eventually be taken.

or better:

A run $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$ is unfair w.r.t. strong fairness if there exists a transition $\alpha$ and some integer $K$, such that $\alpha$ is enabled at state $s_i$ for infinitely many $i$’s, but never taken after step $K$, i.e., $\forall i \geq K : a_i \neq \alpha$. 
Strong Fairness: example

Consider again our last example:

\[
\begin{align*}
&x++ \\
\quad s_0 \quad \| \quad p_0 \\
&\text{else} \\
\quad x > 4 \land \text{even}(x) \quad p_1
\end{align*}
\]

Here, the run where the transition from \(p_0\) to \(p_1\) never happens is unfair w.r.t. strong fairness, because the transition is infinitely-often enabled (more precisely: enabled every other step).
Model-checking in the presence of fairness

Suppose we are trying to check $M \models \phi$:

- Our attempt fails because some traces of $M$ violate $\phi$.

- Suppose all these traces are unfair.

- How to exclude them from consideration?
  
  Hint: suppose we have a way to characterize the fair traces by a temporal logic formula $\phi_{\text{fair}}$. 
Model-checking in the presence of fairness

Suppose we are trying to check $M \models \phi$:

- Our attempt fails because some traces of $M$ violate $\phi$.
- Suppose all these traces are unfair.
- How to exclude them from consideration?
  
  Hint: suppose we have a way to characterize the fair traces by a temporal logic formula $\phi_{\text{fair}}$.

Instead of checking $\phi$, check a different formula:

$$M \models \phi_{\text{fair}} \rightarrow \phi$$

Meaning: only the fair traces (those satisfying $\phi_{\text{fair}}$) must satisfy $\phi$. 
Fairness: additional remarks

Fairness is not about asynchronous composition only: in synchronous but *nondeterministic* systems, we might want to exclude behaviors where some of the nondeterministic choices are constantly ignored.

Example:

```plaintext
MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = FALSE
TRANS
  next(output) = !input | next(output) = output

This models a non-deterministic transition system.
Possible fairness requirement: if input switches, output must eventually also switch.
```
Fairness: additional remarks

Fairness is not about asynchronous composition only: in synchronous but **nondeterministic** systems, we might want to exclude behaviors where some of the nondeterministic choices are constantly ignored.

Example:

```plaintext
MODULE inverter(input)
VAR
   output : boolean;
INIT
   output = FALSE
TRANS
   next(output) = !input | next(output) = output
```

This models a non-deterministic transition system.
Possible fairness requirement: if input switches, output must eventually also switch.

Another example: a communication channel cannot keep on losing a message forever (the choice to lose or to transmit is nondeterministic).
Fairness: additional remarks

Fairness vs. probabilities: we could view fairness as an abstraction of probabilities.

- Example: consider a communication channel, which loses a message with probability $p = 10^{-6}$ and transmits it correctly with probability $1 - p$.

- In this system, a behavior where the message is always lost has zero probability. So, in principle, probabilistic systems do not need fairness, since unfair behaviors have zero probability of occurring.

- Fairness allows us to avoid specifying probabilities. Even if we don’t know what $p$ is, we can still claim that a certain behavior is unfair.

- Also, probabilistic systems are (other things being equal) harder to verify than nondeterministic systems (because in addition to state-space exploration, we have to deal with the numbers).
Composition: summary

Composition semantics:

- **Synchronous**: all processes synchronize at every move.
- **Asynchronous**: processes interleave (some may synchronize due to communication, e.g., by rendez-vous).

Communication semantics (more/less orthogonal to composition semantics):

- Shared memory
- Message passing
- Synchronization (rendez-vous)

Spin/Promela offers asynchronous composition with all three communication options [Holzmann, 2003].

NuXMV offers synchronous composition (asynchronous is deprecated).

Fairness important concern in both.
Bibliography I


