Example of Parallel Composition
Where a Software Program Is Synthesized

Haibo Zeng
(zenghb@eecs.berkeley.edu)

Abstract
Parallel (or asynchronous) composition models how software programs communicate. Using this form of composition, we can derive a solution $X$ which when composed in parallel with the fixed part, satisfies the specification. One way to do parallel composition is to convert the problem into one of synchronous composition, solve it and convert the answer back. In this paper, we propose some methods of doing these conversions, which would take advantage of software synthesis.

1 Introduction
The unknown component problem is to synthesize an unknown component given its context (environment) plus the specification which the component composed with its environment should satisfy. Stated in equations,

$$X \cdot C \subseteq S$$

meaning that $X$ “composed” with $C$ “conforms” to $S$. Here $C$ is the context, $S$ the specification, and $X$ the unknown. This statement of the problem covers a wide set of applications, for example sequential synthesis, game solving, testing of distributed components, and protocol conversion. To solve this problem, we really need to understand the way how systems can be described and composed, and how different compositions of languages, for example, parallel composition and synchronous composition relate to each other.

It has been noticed by Kurshan [2] and Arnold [1] that asynchronous systems can also be modeled with the synchronous interpretation by using non-determinism and self loops with null transitions at each state to keep a transition system in the same state for an arbitrary period of time. As Kurshan stated [2]: “While synchronous product often is thought to be a simple - even uninteresting - type of coordination, it can be shown that, through the use of non-determinism, this conceptually simple coordination serves to model the most general ‘asynchronous’ coordination, i.e. where processes progress at arbitrary rates relative to one another. In fact the ‘interleaving’ model, the most
common model for asynchrony in the software community, can be viewed as a special case of this synchronous product.”

This paper is organized as following: In Section 2, we introduce basic definitions about language, its operators and compositions. In Section 3, we present the recipes of doing parallel composition under various conditions, by converting the problem into one of synchronous composition, solving it and converting the answer back. In Section 4, we conclude our work.

2 Backgrounds

2.1 Languages and Operators

Definition 2.1. An alphabet $\Sigma$ is a finite set of symbols. A language is a set of finite length strings on the symbol set $\Sigma$. If we denote $\Sigma^*$ as the set of all finite strings over a fixed alphabet $\Sigma$, which includes the empty string $\epsilon$, then a language is a subset of $\Sigma^*$.

Definition 2.2. Given languages $L_1$ and $L_2$, respectively over alphabets $\Sigma_1$ and $\Sigma_2$, the union $L$ of language $L_1$ and $L_2$ is defined as

$$L = L_1 \cup L_2 = \{\alpha | \alpha \in L_1 \text{ or } \alpha \in L_2\}$$

which is over alphabet $\Sigma_1 \cup \Sigma_2$.

Definition 2.3. Given languages $L_1$ and $L_2$, respectively over alphabets $\Sigma_1$ and $\Sigma_2$, the intersection $L$ of language $L_1$ and $L_2$ is defined as

$$L = L_1 \cap L_2 = \{\alpha | \alpha \in L_1 \text{ and } \alpha \in L_2\}$$

which is over alphabet $\Sigma_1 \cap \Sigma_2$. If $\Sigma_1 \cap \Sigma_2 = \emptyset$, then $L = L_1 \cap L_2 = \emptyset$.

Definition 2.4. Given a language $L$ over alphabet $X \times Y$, consider the homomorphism $p : X \times Y \to X$ defined as

$$p(x, y) = x$$

then the language

$$L_{|X} = \{p(\alpha) | \alpha \in L\}$$

over alphabet $X$ is the projection of language $L$ to alphabet $X$.

The distributive law for projection holds [3]: Let $L_1, L_2$ be languages over alphabet $I \times U$, then projection commutes with $\cup$ and $\cap$:

$$(L_1 \cup L_2)_{|I} = L_{1|I} \cup L_{2|I}$$

$$(L_1 \cap L_2)_{|I} = L_{1|I} \cap L_{2|I}$$
Definition 2.5. Given a language $L$ over alphabet $X \times Y$, consider the substitution $l : X \to X \times Y$ defined as

$$l(x) = \{(x, y) | y \in Y\}$$

then the language

$$L_{1,Y} = \{ l(\alpha) | \alpha \in L \}$$

over alphabet $X \times Y$ is the lifting of language $L$ to alphabet $Y$.

The distributive law for lifting also holds [3]:

Let $L_1, L_2$ be languages over alphabet $U$, then lifting commutes with $\cup$ and $\cap$:

$$(L_1 \cup L_2)_{1,I} = L_{1,I} \cup L_{2,I}$$

$$(L_1 \cap L_2)_{1,I} = L_{1,I} \cap L_{2,I}$$

Definition 2.6. Given a language $L$ over alphabet $X \cup Y$, consider the homomorphism $r : X \cup Y \to X$ defined as

$$r(x) = \begin{cases} x & \text{if } x \in X \\ \epsilon & \text{if } x \in X \setminus Y \end{cases}$$

then the language

$$L_{\downarrow X} = \{ r(\alpha) | \alpha \in L \}$$

over alphabet $X$ is the restriction of language $L$ to alphabet $X$.

The distributive law for restriction also holds [3]:

Let $L_1, L_2$ be languages over alphabet $I \cup U$, then restriction commutes with $\cup$ and $\cap$:

$$(L_1 \cup L_2)_{\downarrow I} = L_{1,I} \cup L_{2,I}$$

$$(L_1 \cap L_2)_{\downarrow I} = L_{1,I} \cap L_{2,I}$$

Definition 2.7. Given a language $L$ over alphabet $X$ and an alphabet $Y$ which disjoints from $X$, consider the mapping $e : X \to X \cup Y$ defined as

$$e(x) = \{(\alpha x \beta) | \alpha, \beta \in Y^*\}$$

then the language

$$L_{\uparrow Y} = \{ e(\alpha) | \alpha \in L \}$$

over alphabet $X \cup Y$ is the expansion of language $L$ to alphabet $Y$.

The distributive law for expansion also holds [3]:

Let $L_1, L_2$ be languages over alphabet $U$, then expansion commutes with $\cup$ and $\cap$:

$$(L_1 \cup L_2)_{\uparrow I} = L_{1,I} \uparrow L_{2,I}$$

$$(L_1 \cap L_2)_{\uparrow I} = L_{1,I} \cap L_{2,I}$$
2.2 Composition of Languages

Consider two systems $A$ and $B$ with associated languages $L(A)$ and $L(B)$. The systems communicate with each other by a channel $U$ and with the environment by channel $I$ and $O$. The two composition operators that describe the external behavior of the composition of $L(A)$ and $L(B)$ are:[3]:

Definition 2.8. Given the disjoint alphabets $I, U, O$, language $L_1$ over $I \times U$ and language $L_2$ over $U \times O$, the synchronous composition of languages $L_1$ and $L_2$ is the language $\left[ (L_1)\cap (L_2) \right]_{I \times O}$, defined over $I \times O$. We denote synchronous composition as $L_1 \bullet L_2$.

Definition 2.9. Given the disjoint alphabets $I, U, O$, language $L_1$ over $I \times U$ and language $L_2$ over $U \times O$, the parallel composition of languages $L_1$ and $L_2$ is the language $\left[ (L_1)\cap (L_2) \right]_{I \cup O}$, defined over $I \cup O$. We denote parallel composition as $L_1 \circ L_2$.

3 Parallel Composition by Synchrony

3.1 Relation between Composition Operators

[3] also introduced some transformation that theoretically describes how different composition operators relate to each other. This transformation allows a sort of conversion from one type of composition to another, with the outcome that one needs to support a solution procedure only for the latter. The idea is to introduce a new alphabet symbol called $\epsilon$ that is different from any existing alphabet symbol, and add it to every alphabet under consideration. More specifically, given the languages $L_A$ over alphabet $I \cup U$ and language $L_B$ over alphabet $U \cup O$, consider the following transformation that yields the language $\tilde{L}_A$ and $\tilde{L}_B$ over alphabet $I \times U \times O$. For each string in $L_A$ or in $L_B$:

1. replace each symbol $i$ by the triple $i\epsilon\epsilon$, each symbol $u$ by $\epsilon u \epsilon$ and each symbol $o$ by $\epsilon \epsilon o$.

2. insert the regular expression $(\epsilon \epsilon \epsilon)^*$ after each word prefix.

The semantics here are that when an input comes into a module, it takes an unspecified amount of time for the module to produce an output. This will be modeled with a non-deterministic self-loop labeled with $\epsilon$.

By deleting the occurrences of the silent symbol $\epsilon$ one can go back from $\tilde{L}_A$ and $\tilde{L}_B$ to $L_A$ and $L_B$.

With these conversions, we can do synchronous composition and get the equivalent expanded result of parallel composition $S\parallel F$. Thus we need to implement only one type of compositional method - synchronous, and simply have a mapping of each machine into its extended machine to compose in parallel. Finally, we can take the synchronous solution and map it back into a finite state machine.
In Section 3.2 and 3.3, we will engineer how to correctly and efficiently convert from parallel description to synchronous one, and convert the solution back.

### 3.2 Parallel Composition with Asynchronous Automata

The transitions of asynchronous automata are in the following form:

\[(s \rightarrow^{i/o} s')\]

which has the semantic that the system takes an input, produces an output. To model that when an input comes into a module, it takes an unspecified amount of time for the module to produce an output, a new intermediate state \(s''\) is added between the input and output, thus a transition \((s \rightarrow^{i/o} s')\) becomes two transitions \((s \rightarrow^i s'') (s'' \rightarrow^o s)\). We introduce one \(\epsilon\) variable for each channel, it is defined as \(\epsilon = 1\) when the channel is active, and 0 otherwise. So all the \(\epsilon\) variables are 0 means that there is no significant event in the original alphabet happening, we can use this case to model an arbitrary amount of time is passing.

It seems that for all the accepting states, we need to add a self-loop with the condition of all \(\epsilon\) variables equal 0. But remember that we care about the significant events in the system instead of how many \(\epsilon\) are there in between these events. So we add self-loops to the accepting states only if there might be some significant events happening.

Algorithm 1 summarizes the above discussion:

**Algorithm 1 Parallel-to-Synchronous Conversion with Asynchronous Automata**

1. The accepting states are left without changes;
2. There are as many new accepting states as there are transitions (the symbolic name of a state in derived by concatenating the state names followed by the zero-based transition number);
3. There is one additional epsilon variable \(\epsilon_i\) for each channel \(i\);
4. The epsilon variable is 1 when the channel is active, and 0 otherwise;
5. The self-loop in the accepting state is going under condition that all epsilon variables are 0 (meaning all the channels are not active);
6. The self-loop is added only if there might be some significant events happening.
7. The dots for variables in the original specification (unused variables) become dashes in the resulting automaton (meaning don’t-cares).

Because we add self-loops to the accepting states only if there might be some significant events happening, this gives the solution a nice property that all the traces are acceptable, meaning that it takes an input, produce an output; there are no two consecutive inputs or two consecutive outputs! This built-in property in the solution largely simplifies the conversion back to asynchronous, since there is no need to trim the synchronous solution. Algorithm 2 describes
how to collapse the intermediate states to convert the synchronous solution back to parallel.

**Algorithm 2** Synchronous-to-Parallel Conversion

Find all the input-output trace \((s \rightarrow s'')(s'' \rightarrow o s')\), collapse the intermediate state \(s''\). The transitions \((s \rightarrow s'')(s'' \rightarrow o s')\) become one transition \((s \rightarrow^{i/o} s')\).

### 3.3 Parallel Composition with Blif-mv Description

Although the systems are asynchronous finite state machines in our mind, sometimes they are originally described in blif-mv i.e. as synchronous machines since that is what blif-mv does. Since in Section 3.2 we present a way in good shape of doing parallel composition, the most intuitive method here is to read in the blif-mv files and interpret them as asynchronous machines with the languages, and then follow the recipe in Section 3.2.

But it turns out that it is impossible to convert a synchronous machine to an asynchronous automata. Because we need to produce the “output \(\epsilon\)”, one tick after we get the “input \(\epsilon\)”, in the blif-mv circuit formulation the output channel should not be active in the same tick as the input channel, which means that we cannot use combinational logic to produce the output \(\epsilon\) as a combinational function of the input \(\epsilon\). So we need to think another way to do this conversion. Notice that here the only thing to make sure is that the outputs come one tick later than the inputs. The idea is to add a latch to each of the channel to “store” the input values in the previous tick thus the output can be written as a combinational function of these “delayed” inputs and the state bits. Also, to make sure the channels are enabled in a correct sequence, it is necessary to add a machine called \(\epsilon\)-machine to generate the required sequence of \(\epsilon\) values. This is like to add a “shell” to the original description, as shown in Figure 1.

**Algorithm 3** Parallel-to-Synchronous Conversion with Blif-mv Description

1. The original description are left without changes;
2. There is one additional multi-value variable \(\epsilon\);
3. each value of \(\epsilon\) corresponds to only one channel is active and all the others are inactive;
4. Add an \(\epsilon\)-machine to generate the appropriate sequence of \(\epsilon\) values.
5. If the original finite state machine is not Moore machine, add a latch for each channel.

Algorithm 3 describes how we convert the blif-mv description into a synchronous automaton. After doing synchronous composition, we can take the synchronous solution and map it back into a finite state machine by following Algorithm 2.
4 Conclusion

We propose some methods of converting the parallel composition problem into one of synchronous composition, solve it and convert the answer back, which takes the advantage of software synthesis. This comes with the outcome that one needs to support a solution procedure only for synchronous composition.

References

