What Is "Deterministic CHP", and Is "Slack Elasticity" That Useful?

Karl Papadantonakis

Computer Science Department
California Institute of Technology

Master's Thesis, revised 6/18/2002

What Is "Deterministic CHP", and Is "Slack Elasticity" That Useful?

Karl Papadantonakis
Master's Thesis, revised 6/18/2002

[Deterministic CHP is a trivial example of a slack elastic system].
-Rajit Manohar

Abstract

This paper addresses the issue of slack elasticity in distributed computation, as defined by the Caltech Asynchronous VLSI group. We show with a counterexample that slack elasticity is not sufficient for process decomposition. We give criteria which imply slack elasticity and which are sufficient for several forms of process decomposition, and present a hierarchy of determinism.

Motivation

In this paper we show that by using the language we call D-CHP, which avoids the probe of CHP and restricts the comma, we obtain collections of processes which are semantically equivalent to a very specific form, which we call the D-CHP-canonical form. This is interesting for three reasons:

1. The programs in this form comprise precisely the PL1 language [MN], and thus our method of putting programs into this form can be used as a last resort method of compiling an arbitrary D-CHP program into PL1, which can be compiled into production rules efficiently by the PL1 compiler [MN].
2. By restricting our attention to programs in D-CHP-canonical form, we are able to show that all D-CHP programs, and hence the Caltech MiniMIPS- and 80C51- CPUs ("-" means we remove the interrupt implementation, and make a minor modification to the MiniMIPS exception mechanism [KP]) written in a subset of the `m3-3` language that corresponds to D-CHP, have two determinism properties:
   i. The *transition graph* (to be defined) is identical after every execution. For example, this gives a simple way to prove that a program does or does not deadlock for particular inputs, just by running the program in `m3-3` (a message passing library developed by the author, in Modula-3).
   ii. If slack is added, the transition graph may expand, but contains the original as a subgraph.
   iii. Decomposition transformations of programs (which we will characterize) also only expand sequences (which we will define), even for some practical transformations not allowed in slack elastic systems.

At first, it may seem puzzling that simple decomposition transformations are not allowed in slack elastic systems, since Jack Dennis performed these transformations back in 1984. However, one should realize that in this paper we consider the more flexible dataflow design style developed over the years by the Caltech Asynchronous VLSI group; we permit such features as conditional communication, flow control, and short-circuit evaluation, which were not allowed in the original dataflow style. Hence we must justify our transformations with more care.

Table of Contents

1. L-Deterministic Communicating Hardware Processes (D-CHP)

We use the prefix “L-” (for “language”) with the word “deterministic” in order to be able distinguish it from second and third related but different widely used definitions of the word “deterministic” later on. To understand some of the notation used in this section, please refer to Appendix I.

A *D-CHP alphabet* consists of

I. A finite set of variables and function symbols, as in ISP.
II. A finite set of *channel symbols*.
III. All delimiters of the ISP alphabet (“[”,“]”,“*”,“▯”,“→”,“(”, “)”, “;”), plus “!”, “?”, and “,”.

As with ISP, a program consists of an alphabet, variable domains, function interpretations, and a program text. For the program text, however, we now add two *communication statements* (1b and 1c) to the program construction rules:

1a. $P \equiv x := f(x_1, \ldots, x_k)$
1b. $P \equiv (x_1, \ldots, x_k) := f(F_1?, \ldots, F_m?)$
1c. $P \equiv (F_1!, \ldots, F_m!) := f(x_1, \ldots, x_k)$
2. $P \equiv (P_1; P_2)$
3. $P \equiv *[f(x_1, \ldots, x_k) \rightarrow P_1]$
4. $P \equiv [f_1(x_1, \ldots, x_k) \rightarrow P_1 \;▯ \ldots ▯\; f_m(x_1, \ldots, x_k) \rightarrow P_m]$

where $x_1, \ldots, x_k$ and x are arbitrary variables, $f_1, \ldots, f_m$ and f are arbitrary function symbols, $P_1, \ldots, P_m$ are arbitrary programs of the same alphabet, and $F_1, \ldots, F_m$ are arbitrary channel symbols. All symbols are interpreted as in ISP, except the functions symbols in forms 1b and 1c, which are now k-valued.

Note: this allows parallel communications allowed in CHP such as A, B, but it does not allow complex semicolon/comma expressions, such as $(A; B), C$. Here are some examples of CHP constructs expressed in D-CHP:

CHP	D-CHP	notes
$X!x, Y!y$ $X?x, Y?y$	$(X!, Y!) := id(x, y)$ $(x, y) := id(X?, Y?)$	id is the identity function of two variables. In CHP, x and y are disjoint variables.
$Z!(X?)$	$z := id(X?); (Z!) := id(z)$	In D-CHP, there is no atomic receive-send.
$[c \rightarrow M!x ▯ \neg c \rightarrow N!x], X!x$	$[\; c \rightarrow (M!, X!) := copy(x)$ $▯ \neg c \rightarrow (N!, X!) := copy(x)]$	$\overline{copy(x)} \overset{\text{def}}{=} (\overline{x}, \overline{x})$

And clearly any CHP program that uses no commas (or probes, of course) can be directly translated into D-CHP. There are two advantages to restricting the use of the comma:

1. We do not need to worry about illegal statements such as $(A?x, B?x)$ or $(x{:=}1, x{:=}2)$. See Appendix II.
2. The canonical form is much simpler than would otherwise be possible.

Definition: Semantics. *The* **semantics** $\mathcal{S}(P)$ *of a D-CHP program P are the set of ordered predicate pairs* $\big(A[x_1, \ldots, x_k, F_1, \ldots, F_m], B[x_1, \ldots, x_k, F_1, \ldots, F_m]\big)$, *where A and B are first-order formulas,* $x_1, \ldots, x_k$ *are variables of program P, and* $F_1, \ldots, F_m$ *are channel symbols of program P, such that the Hoare triple* $\{A[x_1, \ldots, x_k, F_1, \ldots, F_m]\}\ P^*\ \{B[x_1, \ldots, x_k, F_1, \ldots, F_m]\}$ *holds for any partial execution of P.*

The first-order language containing A and B should be fixed, and should contain the variables and channel symbols of program P. To allow for communication axioms, it should also contain the relation symbol R and the function symbol V, defined as follows:

$$\begin{aligned} \overline{R(F,i)} &\overset{\text{def}}{=} \text{were } i \text{ messages sent on } F? \\ \overline{V(F,i)} &\overset{\text{def}}{=} \text{value of message number } i \end{aligned} \tag{1}$$

NB: The innovation in this definition is that the predicates are attached to the possible sequences of communicated values, but not to the state of the program at each semicolon. Thus, even if semicolons are removed from the program, the original predicates (those intended for the unmodified program) can be applied to the new program without rephrasing.

Definition: Semantic Conjugacy. *The definition for semantic conjugacy of two D-CHP programs is the same as for two ISP programs, except that we also require the two languages to have exactly the same set of channel symbols.*

Definition: D-CHP-canonical form. *A D-CHP program is in* **D-CHP-canonical form** *if its program text has the following form:*

```
x := x_1();
*[C(x) ⟶
    [< [] : i : 0..m : s(x) = i() ⟶ (S_{i,1}!, ..., S_{i,|S_i|}!) := g_i(x) >];
    [< [] : i : 0..k : r(x) = i() ⟶ x := f_i(x, R_{i,1}?, ..., R_{i,|R_i|}?) >]
 ]                                                                              (2)
```

where x is a variable, x_1 and i represent constant functions, (in fact $\overline{i()} \overset{\text{def}}{=} i$) and f_i, g_i, r, and s are function symbols. Furthermore, R_i and S_i are finite, indexed sets of channels to receive and send on, respectively. By convention, we require that f_0 is unary, and g_0 has no value, i.e. we require that $R_0 = S_0 = \emptyset$. We let $E \overset{\text{def}}{=} f_0$. Finally, $x := f_i(x, R?)$ is shorthand for $r := (R?); x := f_i(x, r)$.

Theorem: D-CHP-canonical form. *Every D-CHP program with complete variable initialization is semantically conjugate to a program in D-CHP-canonical form.*

Proof. As with ISP programs, we have a principle of induction of programs, and we can assume a program has just one variable, without increasing its syntax depth. Also, as before, we consider "program parts" i.e. programs where the variables need not be initialized; such programs are equivalent to (2) except that the first statement is $x := x_1(x)$. Proceeding with case analysis:

```
1a. P ≡ x := f(x). Let $\overline{x_1} = \overline{f}$, and $\overline{C} = 0$.
1b. P ≡ x := f(F_1?, ..., F_m?)
      ≡ y↑;  *[y ⟶ [0=0 ⟶ y↓]; [1=1 ⟶ x := f(F_1?, ..., F_m?)]]

1c. P ≡ (F_1!, ..., F_m!) := f(x)
      ≡ y↑;  *[y ⟶ [1=1 ⟶ (F_1!, ..., F_m!) := f(x)]; [0=0 ⟶ y↓]]

 2. P ≡ (P_1; P_2)
      ≡ x := x_1(x);
        *[C_1(x) ⟶< [] : i : 1..m_1 :  s_1(x) = i  ⟶  (S_{i,1}!, ..., S_{i,|S_i|}!) := g_i(x) >;
                    < [] : i : 1..k_1 :  r_1(x) = i  ⟶  x := f_i(x, R_{i,1}?, ..., R_{i,|R_i|}?) > ];
        x := x_2(x);
        *[C_2(x) ⟶< [] : i : m_1+1..m :  s_2(x) = i  ⟶  (S_{i,1}!, ..., S_{i,|S_i|}!) := g_i(x) >;
                    < [] : i : k_1+1..k :  r_2(x) = i  ⟶  x := f_i(x, R_{i,1}?, ..., R_{i,|R_i|}?) > ]
      ≡ x := x_1(x); y := 0;
        *[y≠ 3 ⟶ [< [] : i : 1..m :  ((y=0∧¬C_1)*r_1(x) + (y=2∧¬C_2)*r_2(x)) = i  ⟶
                            (S_{i,1}!, ..., S_{i,|S_i|}!) := g_i(x) >
                  []ELSE(... = 0) ⟶ skip];
                  [< [] : i : 1..k :  ((y=0∧¬C_1)*s_1(x) + (y=1)*0 + (y=2∧¬C_2)*s_2(x)) = i  ⟶
                            x := f_i(x, R_{i,1}?, ..., R_{i,|R_i|}?) >
                  []ELSE(... = 0) ⟶ x := x*(y≠1) + x_2(x)*(y=1);  y := y + 1]]
```

where *skip* is shorthand for the empty send command ():=().

```
3. $P \equiv *[f(x) \longrightarrow P_1]$
       $\equiv *[f(x) \longrightarrow x := x_1(x);$
                 $*[C(x) \longrightarrow < \square : i : 1..m : \;\; s_1(x) = i \;\; \longrightarrow \;\; (S_{i,1}!, \ldots, S_{i,|S_i|}!) := g_i(x) >;$
                          $< \square : i : 1..k : \;\; r_1(x) = i \;\; \longrightarrow \;\; x := f_i(x, R_{i,1}?, \ldots, R_{i,|R_i|}?) > \;]]$
       $\equiv y\!\downarrow; *[f(x) \vee (C(x) \wedge y) \longrightarrow$
                    $[< \square : i : 1..m : \;\; (y{*}r(x)) = i \;\; \longrightarrow (S_{i,1}!, \ldots, S_{i,|S_i|}!) := g_i(x) >$
                    $\square \neg y \longrightarrow skip];$
                    $[< \square : i : 1..k : \;\; (y{*}s(x)) = i \;\; \longrightarrow x := f_i(x, R_{i,1}?, \ldots, R_{i,|R_i|}?); \;\; y := C(x) >$
                    $\square \neg y \longrightarrow x := x_1(x); \;\; y := C(x)]]$

4. $P \equiv [f_1(x) \longrightarrow P_1 \;\square\; \ldots \;\square\; f_n(x) \longrightarrow P_n]$
       $\equiv [f_1(x) \longrightarrow$
               $x := x_1(x);$
               $*[C_1(x) \longrightarrow < \square : i : 1..m_1 : \;\; s_1(x) = i \;\; \longrightarrow \;\; (S_{i,1}!, \ldots, S_{i,|S_i|}!) := g_i(x) >;$
                        $< \square : i : 1..k_1 : \;\; r_1(x) = i \;\; \longrightarrow \;\; x := f_i(x, R_{i,1}?, \ldots, R_{i,|R_i|}?) > \;]$
          $\square$
          $\vdots$
          $\square f_n(x) \longrightarrow$
               $x := x_2(x);$
               $*[C_n(x) \longrightarrow < \square : i : m_{n-1}{+}1..m : \;\; s_n(x) = i \;\; \longrightarrow \;\; (S_{i,1}!, \ldots, S_{i,|S_i|}!) := g_i(x) >;$
                        $< \square : i : k_{n-1}{+}1..k : \;\; r_2(x) = i \;\; \longrightarrow \;\; x := f_i(x, R_{i,1}?, \ldots, R_{i,|R_i|}?) > \;]$
          $]$
       $\equiv y := 0; *[(y{=}0) \vee (y{=}1 \wedge C_1(x)) \vee \cdots \vee (y{=}m \wedge C_m(x)) \longrightarrow$
                    $[< \square : i : 1..m : \;\; (r_y(x){*}(y{\neq}0)) = i \;\; \longrightarrow (S_{i,1}!, \ldots, S_{i,|S_i|}!) := g_i(x) >$
                    $\square y{=}0 \longrightarrow skip];$
                    $[< \square : i : 1..k : \;\; (s_y(x){*}(y{\neq}0)) = i \;\; \longrightarrow x := f_i(x, R_{i,1}?, \ldots, R_{i,|R_i|}?) >$
                    $\square y{=}0 \longrightarrow y := argf(x); \;\; x := x_y(x)]]$
```

$\square$

Performance Guarantee for Translation to D-CHP-canonical Form

One may ask if the method of transforming a D-CHP program to canonical form has adverse effects on performance. We can guarantee the following:

(i) Communication statements which occur in parallel in the original D-CHP program occur in parallel in the canonical D-CHP program.

(ii) The number of semicolons executed between any two communication actions in a canonical D-CHP program is at most twice the number of semicolons executed between the same two communication actions in the original D-CHP program.

Proof:

(i) is a consequence of the canonical form theorem: the semantics of the program are not changed.

For (ii), consider any two communication actions A and B which do not occur in parallel in the original program. We show the result by induction on the number of communications that occur between A and B. For the inductive step, suppose a communication C occurs between A and B. We simply add the number of semicolons between A and C, and between C and B. This preserves the "less than twice" property.

Thus, we can focus on the base case: there are no communications between A and B. The original program executed $k \geq 1$ semicolons between A and B. Through syntactic expansion, the canonical program executed at most mk semicolons between A and B. We will show how to reduce m by 2 (in fact by almost halving m) when $m > 2$.

Suppose $m > 3$. This means that at some point, the program executed two $x := E(x)$ statements with no communication action in between. To reduce the number of times this happens, we simply replace E by E', defined thus:

$$E'(x) \overset{\text{def}}{=} \begin{cases} E(E(x)), & \text{if } C(E(x)) \wedge (s(E(x)) = 0) \wedge (r(E(x)) = 0) \\ E(x), & \text{otherwise.} \end{cases} \tag{3}$$

□

Although we have two semicolons in the canonical form, the PL1 compiler [MN] can compile such processes into circuits such that every node cycles at most once per loop iteration. So at the circuit level, each loop iteration corresponds to just one cycle. Since handshake protocols require each communication to occur in its own cycle, there is typically nothing more to improve at the CHP level of description.

Still, one may object that the canonical D-CHP with conditional communications requires more cycling than what would be required if the extended features of the PL1 language were used in order to collapse a chain of conditional communications into once cycle. This problem can only be solved by recognizing such a chain and rewriting the program.

2. Communicating State Machines: SS- and IO-Determinism

Given any set of D-CHP processes, first express them as canonical D-CHP programs. Second, notice that processes in such a system can stop in two ways: deadlock or termination. Deadlock is more powerful, and is also usually the method used in practice. Thus we eliminate the case of termination, by adding a "stop" channel to every process. This allows us to eliminate the $C(x)$ test: we simply make $C(x)$ activate the "stop" channel instead. Now all processes have the form:

$$\begin{aligned} x := x_1(); \quad & *\texttt{[[}< \texttt{[]} : i : 0..m : s(x) = i() \longrightarrow (S_{i,1}!, \ldots, S_{i,|S_i|}!) := g_i(x) >\texttt{]}; \\ & \texttt{[}< \texttt{[]} : i : 0..k : r(x) = i() \longrightarrow x := f_i(x, R_{i,1}?, \ldots, R_{i,|R_i|}?) >\texttt{]]}, \end{aligned} \tag{4}$$

which we could abbreviate

$$x := x_1; *\texttt{[}S_x!g(x);\ x{:=}f(x, R_x?)\texttt{]}, \tag{5}$$

but beware of the informal notation: R_x and S_x are actually the sets of channels $R_{r(x)}$ and $S_{s(x)}$, and the dimensions of f and g vary accordingly. We will refer to the functions f_i as the family f. We call f a *state transition function.*

The Execution Model

We define a set of *semicolon states*, where each state s has the form (x, ϕ), for $\phi \in \{0, 1\}$.

We define a set of *state transitions* of the form:

$$(\overline{x}, 0) \xrightarrow{S_{\overline{x}}!\overline{g(x)}} (\overline{x}, 1) \tag{6}$$

$$(\overline{x}, 1) \xrightarrow{R_{\overline{x}}?y} (\overline{f(x, y)}, 0) \tag{7}$$

Now we are ready to give meaning to the above state transistions by defining a sequence of guarded commands for each of the above commands, in each process.

Gather/Step (Guarded Command) Model

Definition. *A* **L-deterministic system** *is a set of D-CHP processes without shared channels. I.e., each channel F used by a process is used only for receiving or only for sending, but not both, and F is used by at most one other process for the opposite type of communication.*

To distinguish the x variables in the different processes, we write the variables as x_α, where α is a process index. (notation detail: we assume without loss of generality that the values of state variables in different processes are disjoint, so that the notation $\overline{x_\alpha}$ clearly refers to a value of the variable x_α in process α. The reason for referring to the value and not the symbol is that we will have possibly different rules for each possible *value* of each state).

Next we define a set of *expanded states*, each of the form $(\overline{x_\alpha}, \phi, T)$, where T is a set of completed channels. For each semicolon state $(\overline{x_\alpha}, \phi)$, we create an expanded state of the form $(\overline{x_\alpha}, \phi, \emptyset)$. We also define a set of *expanded state transitions*, (guarded commands) by expanding each state transition into as follows:

original state transition	exp'n	expanded state transitions	expand over
$(\overline{x_\alpha}, 0) \xrightarrow{S_{\overline{x_\alpha}}!\overline{g(x_\alpha)}} (\overline{x_\alpha}, 1)$	gather	$(\overline{x_\alpha}, 0, T) \xrightarrow{F!\overline{g_F(x_\alpha)}} (\overline{x_\alpha}, 0, T \cup \{F\})$	all $T \subseteq S_{\overline{x_\alpha}}$,
(send)	step	$(\overline{x_\alpha}, 0, S_{\overline{x_\alpha}}) \xrightarrow{\sigma} (\overline{x_\alpha}, 1, \emptyset)$	and all $F \in S_{\overline{x_\alpha}} - T$.
$(\overline{x_\alpha}, 1) \xrightarrow{R_{\overline{x_\alpha}}?y} (\overline{f(x_\alpha, y)}, 0)$	gather	$(\overline{x_\alpha}, 1, T) \xrightarrow{F?y} (\overline{x_\alpha}, 1, T \cup \{(F, y)\})$	all $T{\subseteq}R_{\overline{x_\alpha}}$, all $y{\in}\mathcal{D}(F)$,
(receive)	step	$(\overline{x_\alpha}, 1, R_{\overline{x_\alpha}} \times V) \xrightarrow{\rho} (\overline{f}(\overline{x_\alpha}, V), 0, \emptyset)$	and all $F \in R_{\overline{x_\alpha}} - T$.

where $R_{\overline{x_\alpha}} \times V$ denotes a set of ordered pairs associating each channel of $R_{\overline{x_\alpha}}$ with a corresponding value in V.

These guarded commands are executed by forming a sequence of "executed" guarded command pairs, according to the following rules:

i. Before each executed command, each process α has an associated state $(\overline{x_\alpha}, \phi, T)$, i.e. a state number $\overline{x_\alpha}$, a phase number ϕ, and a completed channel set T.

ii. A gather command $(\overline{x_\alpha}, \phi, T)\xrightarrow{F*v}(\overline{x_\alpha}', \phi', T')$ is *enabled* whenever process α is in state $(\overline{x_\alpha}, \phi, T)$, *and* the unique other process connected to channel F is simultaneously in a state $(\overline{x_\beta}, \psi, U)$ and there is a *partner* command $(\overline{x_\beta}, \psi, U)\xrightarrow{F*v}(\overline{x_\beta}', \psi', U')$.

iii. A step command $(\overline{x_\alpha}, \phi, T)\xrightarrow{\sigma}(\overline{x_\alpha}', \phi', T')$ or $(\overline{x_\alpha}, \phi, T)\xrightarrow{\rho}(\overline{x_\alpha}', \phi', T')$ is its own partner, and is enabled when α is in state $(\overline{x_\alpha}, \phi, T)$.

iv. A guarded command must be enabled immediately before it is executed. Furthermore, once a guarded command becomes enabled, it is guaranteed to be executed after a finite number of execution steps.

v. When a guarded command $(\overline{x_\alpha}, \phi, T)\xrightarrow{F*v}(\overline{x_\alpha}', \phi', T')$ is executed, the partner command $(\overline{x_\beta}, \psi, U)\xrightarrow{F*v}(\overline{x_\beta}', \psi', U')$ must be executed. After executing this pair of commands, α is in state $(\overline{x_\alpha}', \phi', T')$, and β is in state $(\overline{x_\beta}', \psi', U')$. Similarly, when α executes a step command $\ldots\rightarrow(\overline{x_\alpha}', \phi', T')$, α changes to state $(\overline{x_\alpha}', \phi', T')$.

The execution sequence for a given set of processes and initial conditions is emphatically *not* unique. This is in fact the beauty of asynchronous dataflow design: a circuit may automatically pipeline itself when the latency exceeds the cycle time, without the designer investing extra effort. However, to be sure that all possible executions are correct, one must hope for some property to hold for all executions. For an L-deterministic system, we shall find that the *transition graph* (defined below) is such a property.

Transition graph

Definition. *The* **transition graph** *of a particular gather/step execution sequence (of an L-deterministic system) consists of*

i. The nodes $(x_{j\alpha}, \phi, \emptyset)$, for each process α, and for all $j \in \mathbf{N}^+$ such that process α made at least j visits to *superstates*, i.e. states of the form $(\overline{x_\alpha}, \phi, \emptyset)$. I.e., we count state visits, ignoring states with $T \neq \emptyset$.

ii. For each node $(x_{j\alpha}, \phi, \emptyset)$, we define the *value* $\overline{(x_{j\alpha}, \phi, \emptyset)}$ of the node as the jth state of the form $(\overline{x_\alpha}, \phi, \emptyset)$ reached by α.

iii. Between each pair of nodes $\big((x_{j\alpha}, 0, \emptyset), (x_{j\alpha}, 1, \emptyset)\big)$, we create a "step" edge labelled σ, and between each pair of nodes $\big((x_{j\alpha}, 1, \emptyset), (x_{(j+1)\alpha}, 0, \emptyset)\big)$, we create a "step" edge labelled ρ.

iv. Each executed gather-send command $(\overline{x_\alpha}, 0, T)\xrightarrow{F!v}(\overline{x_\alpha}, 0, T')$ is identified by the channel F, the sent value v, and by the state $x_{j\alpha}$ that α was in immediately before the command. Furthermore, this command is by construction only executed when a partner command $(\overline{x_\beta}, 0, U)\xrightarrow{F?v}(\overline{x_\beta}, 0, U')$ is executed, immediately after β was in state $x_{k\alpha}$. If β ever reaches a state $(x_{(k+1)\beta}, 0, \emptyset)$, we define an edge labelled (F, v) from $(x_{j\alpha}, 0, \emptyset)$ to $(x_{(k+1)\beta}, 0, \emptyset)$.

For example, suppose $P_1 \equiv F!v$, $P_2 \equiv F?v, G?w$, and $P_3 \equiv G!w$. One (in fact the only, as we shall see) possible transition graph is the following:

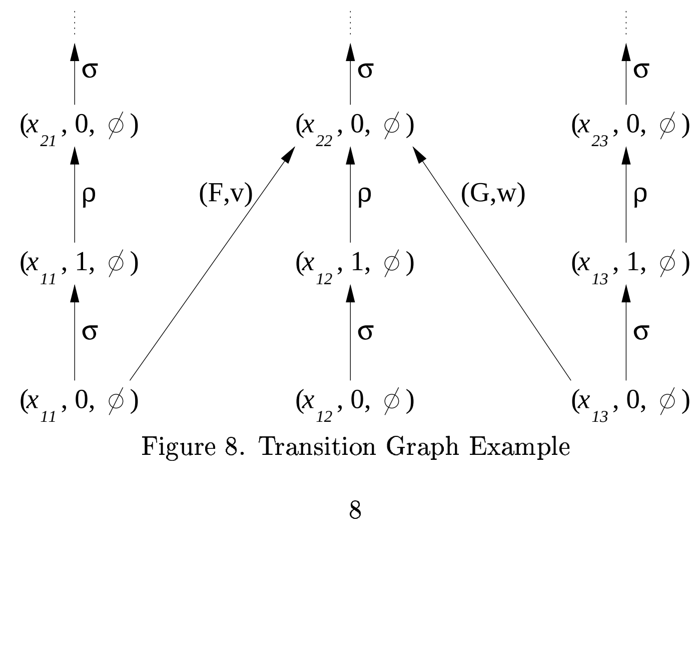

Figure 8. Transition Graph Example

Notice that the value of x_{22} actually depends on what was sent on v and w, and hence the values of x_{11} and x_{13}, in addition to the value of x_{12}.

SS-Determinism

So far, our definitions of "Determinism" have been restricted to D-CHP systems. We now consider a definition (SS stands for "state sequence") which applies to arbitrary CHP systems:

Definition: SS-deterministic. *A CHP system is SS-deterministic if for all processes X,*

i. Some subsets of the semicolons of X are identified as **supersemicolons***, and on any execution, the corresponding states are called* **superstates***.*

ii. All communication entered by X can be determined from the superstates. I.e. whether or not X reaches a communication action (with a particular value to send, if it is a send) between superstates s and t is a function solely of s, and every communication follows some s (notice this implies that there must be at least one superstate if there can be any communications).

iii. On any execution, the sequence of superstates visited by X does not change.

First Determinism Theorem. *Every L-deterministic system is SS-deterministic.*

Proof. We have already shown that any L-deterministic system can be viewed as a gather/step system. In such a system we have identified superstates that determine which communications will be entered before the next superstate. The sequence of superstates visited by the original system is entirely determined by the transition graph of the gather/step system. Hence it suffices to show that two executions E and E' of the gather/step system produce the same transition graph.

E is a sequence of commands $C_0, C_1, \ldots$. Let $P(i)$ be the statement that the portion of the transition graph G of E represented by $C_0 \ldots C_{i-1}$ is contained as a subgraph in the transition graph G' of E'. The base case $P(0)$ states that the empty graph is a subgraph of G'.

Now suppose that $P(i)$ holds. G' contains the subgraph of G represented by $C_0 \ldots C_{i-1}$. We must (assuming at least $i+1$ commands are executed in E) also show that it contains the new edges represented by C_i. The transition graph only expands when a superstate is reached, so we can assume that C_i formed a new superstate s for process X in G.

Next we show that s is formed with the same incoming edges in G'. All incoming edges of s have their other vertex in G', by assumption. These prerequsite edges are labeled with X as their target process. Hence they could only be formed by pointing to one of X's states. Since X has already reached the superstate before s, we know that the edges do not point into X before s. Thus the prerequisite edges all either point into s, or are enabled–but then we know that they must be executed eventually. So in fact s forms.

The incoming edges to s are the same in G and in G'. By induction, these edges left nodes whose values were the same in G and G'. Hence, during the gather phase, for a receive, the same value assignment was collected (though perhaps not in the same order) and hence s has the same value in G and in G'. For all other types of commands, s has the same value just because X was in the same state just before s. Thus we have $P(i+1)$.

By induction, G' includes G as a subgraph. Reversing the argument, G includes G', hence the two graphs are isomorphic. □

Clearly, the converse is false. Here is a CHP program which uses probes, yet is SS-deterministic:

$$x\!\downarrow; *[W!x; X?x; Y!x; Z!x] \quad \| \quad *[[\overline{W} \rightarrow W?x; X!x \,[]\, \overline{Y} \rightarrow Y?x; Z!x]] \tag{9}$$

This program is SS-deterministic because the execution sequence is always $*[W; X; Y; Z]$, because at each semicolon there is exactly one process which is not blocked (and only one action allowed for that process). The program is also *slack elastic*; in the next section (on that topic) we will also see an SS-deterministic program which is not slack elastic.

Clearly this means that one can devise criteria for SS-determinism that are weaker than L-determinism. However, this does not seem to be absolutely necessary, as we have designed MiniMIPS- and an 80C51-CPUs which ran user-level MIPS and 80C51 programs respectievely (including exceptions in the MIPS), and which are both entirely L-deterministic systems; i.e. the descriptions use only the four language elements described above.

Applications of the First Determinism Theorem

Suppose we are designing an L-deterministic system. The First Determinism Theorem tells us that the sequence of superstates characterizes all executions, including their output values. Thus, debugging an L-deterministic system is at least as easy as debugging a synchronous system: run the system (in a simulator such as `m3-3`), and if it produces the correct values for a given input, then the trace constitutes a proof that this input case always leads to correct output values. If the system deadlocks, then on any execution it deadlocks for the same reason.

In fact, some synchronous systems are harder to debug than L-deterministic asynchronous systems, because some of the most popular design styles "define away" deadlock, allowing many careless errors to "sneak by", even if these errors would otherwise cause deadlock.

IO-Determinism

Definition: IO-deterministic. *A CHP system is IO-deterministic if*

i. There is some subset of the channels, identified as *IO channels*. Actually these are output channels, because we require inputs to be coded in the initial conditions of the processes, for simplicity. Often we assume that all channels are IO channels, because that is usually as interesting as any other case.
ii. The sequence of values sent on each IO channel is the same on any execution.

Note: i. is "given", so when we say we are going to show that a system is IO-deterministic, we mean that for any choice of i., we will show ii.

Second Determinism Theorem. *Every SS-deterministic system is IO-deterministic.*

Proof: Consider an IO channel C. Because we have no shared channels, only one process P can send on C. The sequence of superstates of P is the same on every execution. Hence the subsequence of superstates before each send on C is the same on every execution. Hence the sequence of values sent on C is the same on every execution. □

Clearly, the converse is false. Here is a CHP program which is IO-deterministic (and slack elastic), but not SS-deterministic:

$$*\texttt{[[}1 \rightarrow X; X \,|\, 1 \rightarrow X\texttt{]]} \;\; \| \;\; *\texttt{[}X\texttt{]} \tag{10}$$

Combining the first and second determinism theorems, we have:

Corollary. *Every L-deterministic system is IO-deterministic.*

3. Slack Elasticity

Definition: Slack Elasticity. *A system is* **slack elastic** *if*

i. for each channel C, there is a *correctness function* f_C from sequences of values in $\mathcal{D}(C)$ to $\{\textbf{true}, \textbf{false}\}$. Such a function must have the property that extending a non-infinite sequence which satisfied the function must produce a new sequence still satisfying the function. We can obtain Rajit's definition (which requires the original system to be deadlock-free) by only considering correctness functions whose support is restricted to infinite sequences, though the author does not see this as strictly necessary.
ii. any execution satisfies the correctness functions.
iii. if a channel C has all receiver occurences replaced by R and all sender occurences replaced by L, and the buffer process $*[L; R]$ is added to the system, the original correctness functions will still be satisfied.

Note: i. and ii. are "given", so when we say we are going to show that a system is slack elastic, we mean that for any choice of i. satisfying ii., we will show iii.

A random number sequence generator is an example of a non-IO-deterministic slack elastic system (if this comment seems unmotivated, see the Venn diagram in the Conclusion section).

Slack Elasticity Theorem. *Every L-Deterministic system is slack elastic.*

Proof: Consider an execution E of the original system S; it satisfies the correctness functions. Consider a finite prefix I of E. If we replace each C action in I by the two actions $L; R$, we obtain a valid execution of the modified system S': L actions are enabled at least when C was enabled, and R enables whatever C enabled. It does not matter that L actions might be enabled when they were not previously enabled, because we are only considering a finite execution prefix. Hence E' executes I. (we will see a more rigorous proof by induction of this same fact when we discuss value sequence systems).

By the determinism theorem, *any* execution of S' includes I. For an infinite execution, consider all its finite prefixes. Since each channel's communication sequence in S is a prefix of that channel's communication sequence in S', the correctness function for that channel remains satisfied. □

SS-Determinism is too weak for slack elasticity.

What if we replace "L-deterministic" in the slack elasticity theorem with "SS-deterministic"? Then the theorem is emphatically false; here is an SS-deterministic system which is not slack elastic:

$$*[X?x; [\neg x \to C_0 [\!] x \to C_1]] \;\; \| \;\; *[[\overline{C_0} \to Y!0; C_0 [\!] \overline{C_1} \to Y!1; C_1]] \tag{11}$$

The above program is SS-deterministic: if n values are sent on X, then the system always does

$$<; i : n : X?x_i; x_i \to; C_i \to; Y!x_i; C_{x_i} > \tag{12}$$

since at each step there is exactly one process which is not blocked. Hence the state sequences of both processes are determined.

However, if slack is added to C_0, then the lefthand process can do C_0 and C_1 before the righthand process has done anything, hence the latter may then proceed to send either 0 or 1 as the first message on Y.

4. Beyond Slack Elasticity

Often, slack elasticity is not enough to guarantee that a desired program transformation is allowed. For example, suppose we start with the following program to compute a function of three inputs:

$$P_1 \overset{\text{def}}{=} *[A?a, B?b, C?c;\ Y!f(a, g(b,c))] \tag{13}$$

In hopes of exploiting the structure of the function, one begins by writing the following equivalent programs:

$$P_1' \overset{\text{def}}{=} *[A?a, B?b, C?c; x := g(b,c);\ Y!f(a,x)] \tag{14}$$

$$P_1'' \overset{\text{def}}{=} *[A?a, (B?b, C?c; x := g(b,c));\ Y!f(a,x)] \tag{15}$$

Next, to simplify implementation, and to allow concurrent operation of f and g one would like to "pipeline" P_1''. This can be done by putting $(B?b, C?c; x := g(b,c))$ in its own process, since this part is almost independent of the rest of P_1'': the only communication is sending the result x. This leads to:

$$P_2 \overset{\text{def}}{=} *[A?a, X?x;\ Y!f(a,x)] \parallel *[B?b, C?c; X!g(b,c)] \tag{16}$$

If P_1 was one process in a larger system, how do we justify replacing it by P_2? At first, one may think that slack elasticity of the original system is enough, because P_2 was obtained by "adding slack". But adding slack to P_1 can neither yield a process equivalent to P_2, nor a process equivalent to P_2 with some amount of slack added. In fact, the following environment for P_1 will not work with P_2, even if we add slack to any input and/or output channels:

```
$b := 1;$
$*[i := 0;\ \ *[i < b \longrightarrow B!0; i := i{+}1];$
   $C!0;$
   $[\overline{B} \longrightarrow B!0; b := b{+}1 \,|\, \neg\overline{B} \longrightarrow \textit{skip}];$
   $\{\text{now flush all channels}\}$
   $A!0;\ Y?y;\ \ i := 0;\ \ \ *[i < b{-}1 \longrightarrow A!0;\ C!0;\ Y?y; i{:=}i{+}1]$
 $]$                                                                          (17)
```

The trouble with this environment is that it assumes that just because b items could be sent on B before anything was sent on A, it would succeed in sending b items another time, even if it didn't use C the second time. But P_2 can deadlock this environment: the environment can count up to b even if only $b-2$ buffer places were added to the B channel, because the communication on C allows the second part of P_2 to receive an additional item on B. On another attempt (the next time through the main loop), the environment attempts to send b items on B *before* sending anything on C, and deadlocks.

The environment worked provided only slack was added to P_1, hence the system consisting of the environment together with P_1 *is* slack elastic. However, it does not allow the transformation that yielded P_2 from P_1. (see Appendix III for a more rigorous treatment of this counterexample). Since transformations of this form are very important in processor design, systems must have a stronger property than slack elasticity.

Using the determinism theorems to prove correctness of program transformations

We want to justify the replacement of P_1 by P_2. Even if the environment is slack elastic, this is not in general allowed, as we have seen. However, for particular environments, we can use the fact that the entire system is deterministic to justify the transformation (just as we justified adding slack, when we showed that L-deterministic systems are IO-deterministic).

Suppose the input channels A, B, and C are each connected to a process which sends n items; A sends $<; i : n : a_i >$, etc. We can verify the following possible execution sequence:

$$<; i : n : A?a_i; B?b_i; C?c_i; X!g(b_i,c_i);\ Y!f(a_i, g(b_i,c_i)) > \tag{18}$$

by writing the state of every process at every semicolon. Restricting our attention to the values sent on channel Y, we see that a particular execution sequence produces the output sequence $<; i : n : f(a_i, g(b_i,c_i)) >$. Rather than proceeding to check all possible execution sequences in the same manner, we simply use the fact that the entire system can be written in D-CHP, and hence is IO-deterministic, hence every execution sequence produces the the output sequence $<; i : n : f(a_i, g(b_i,c_i)) >$.

Value Sequence Systems

We now attempt to define a class of systems which allows function decomposition transformations. We will present a model, the *value sequence system*; a CHP system (examples below) is considered a value sequence system if its behavior can be described according to this model (and in due course we will show how to construct value sequence systems generically).

We begin with a set of channels; each channel is written F_α, and has an associated domain $\mathcal{D}(F_\alpha)$. Each channel has an associated infinite sequence of values; value i of channel F_α is written $\overline{F_{\alpha i}}$, is an element of $\mathcal{D}(F_\alpha)$, and has an associated *value variable* $\text{val}(F_{\alpha i})$, sometimes also called an *element variable* and written $F_{\alpha i}$ for short.

For each element variable $F_{\alpha i}$, we also have an associated boolean-valued *definedness variable* $\text{def}(F_{\alpha i})$; this variable represents whether or not $F_{\alpha i}$ is defined. The value of an undefined element variable will never be used.

For each element variable $F_{\alpha i}$, we have a *definedness formula* $\text{edef}(F_{\alpha i})$, which is a possibly infinite propositional formula whose literals are the definedness variables and terms of the form $f(F_{\alpha i})$, for arbitrary $f : \mathcal{D}(F_\alpha) \rightarrow \{\textbf{true}, \textbf{false}\}$.

For each element variable $F_{\alpha i}$, we also have an *evaluation formula* $\text{eval}(F_{\alpha i})$, which is actually a set of possibly infinite propositional formulas $\text{eval}(F_{\alpha i}) = k$ for all $k \in \mathcal{D}(F_\alpha)$, with the property that exactly one evaluates to true, regardless of the definedness variables. Hence $\text{eval}(F_{\alpha i})$ is effectively a $\mathcal{D}(F_\alpha)$-valued formula; we write its value $\overline{\text{eval}(F_{\alpha i})}$.

Any function from all definedness variables to $\{\textbf{true}, \textbf{false}\}$ is called a *definedness assignment*. Similarly, we can define a *defined-value assignment*, which gives values to all definedness variables and value variables.

Note: We realize that a formally represented function which depends on infinitely many variables is not usually referred to as a "formula". We will need dependence on infinitely many variables when we show that every L-deterministic system is a value sequence system. However, we will give three examples of value sequence systems and their decompositions; in these examples our formulas will be standard (finite) formulas. Furthermore, more specifically than being functions, all definedness formulas we will consider have the property that if they can be satisfied, then they can be satisfied for some assignment which defines only finitely many variables. We will never actually need this fact in order to prove that decomposition transformations are allowed, but this is simply an interesting property (meriting these functions a more distinguishing title than "functions") which is derived from the fact that a physical system cannot wait until infintely many values have been received on a channel before responding.

Definition: Extended Definedness Formula. *An extended definedness formula for a (possibly already extended) definedness formula* $\text{edef}(F)$ *is any formula resulting from substituting* $\text{edef}(G) \wedge \text{def}(G)$ *for the corresponding definedness variable* $\text{def}(G)$ *appearing in* $\text{edef}(F)$.

Our execution model will only use defined-value assignments with the property that extending a definedness formula $\text{edef}(F)$ does not change $\overline{\text{edef}(F)}$.

Definition: Stability. *An evaluation formula* $\text{eval}(F_{\alpha i})$ *and corresponding definedness formula* $\text{edef}(F_{\alpha i})$ *are* **stable** *with a value sequence system if they have the following property. For any defined-value assignment satisfying* $\text{edef}(F_{\alpha i})$, $\text{edef}(F_{\alpha i})$ *remains satisfied, and* $\overline{\text{eval}(F_{\alpha i})}$ *does not change, if the assignment is "independently" altered, i.e. altered in one of the following ways:*

i. a **false** definedness variable is reassigned **true**, or
ii. the value variable corresponding to a **false** definedness variable is assigned a different value.

This ensures that if a value $F_{\alpha i}$ can be defined by only a subset of the elements, (called a *domain* of $F_{\alpha i}$) that elements outside this domain of $F_{\alpha i}$ should not affect this value.

Specifying Definedness through Dependencies

Often, a definedness formula edef(G) has the form $< \wedge : j : k : F_j >$. For each definedness formula of this form, we can instead write $< \wedge : j : k : F_j \rightarrow G >$. This has the notational advantage of allowing us to specify an element's domain one element at a time. For example, the definedness formulas for an $*[L; R]$ buffer process can be specified through the dependencies:

$$\begin{aligned} &L_i \rightarrow R_i \\ &R_i \rightarrow L_{i+1}, \end{aligned} \tag{19}$$

for all i. We sometimes write this more compactly as $L_i \rightarrow R_i \rightarrow L_{i+1}$. What this actually means is

$$\begin{aligned} \text{edef}(R_i) &= \text{def}(L_i) \wedge \text{xdef}(R_i) \\ \text{edef}(L_{i+1}) &= \text{def}(R_i) \wedge \text{xdef}(L_{i+1}), \end{aligned} \tag{20}$$

where the "xdef" are formulas determined by the rest of the system. After we define the execution model for value sequence systems, we will see that this describes the behavior of the $*[L; R]$ buffer.

Domain Weakening

Definition: Domain Weakening. *A definedness formula* edef'$(F_{\alpha i})$ *is a* **weakening** *of* edef$(F_{\alpha i})$ *if some extension of* edef$(F_{\alpha i})$ *implies some extension of* edef'$(F_{\alpha i})$*, and if* edef'$(F_{\alpha i})$ *is stable with the new value sequence system obtained by replacing* edef$(F_{\alpha i})$ *by* edef'$(F_{\alpha i})$*. We sometimes say the* **domain** *of* $F_{\alpha i}$ *is* **weakened**.

Domain Weakening: Example 1

Suppose P_1'', as discussed earlier:

$$P_1'' \overset{\text{def}}{=} *[A?a, (B?b, C?c; x := g(b, c)); Y!f(a, x)] \tag{21}$$

is to be replaced by P_2,

$$P_2 \overset{\text{def}}{=} *[A?a, X?x; Y!f(a, x)] \parallel *[B?b, C?c; X!g(b, c)] \tag{22}$$

in a value sequence system. We specify their definedness formulas through dependencies:

$$\begin{tikzcd}
A_i \arrow[dr] & & A_{i+1} \\
B_i \arrow[r] & Y_i \arrow[ur] \arrow[r] \arrow[dr] & B_{i+1} \\
C_i \arrow[ur] & & C_{i+1}
\end{tikzcd}$$

Figure 23. Definedness for P_1''.

$$\begin{tikzcd}
& A_i \arrow[dr] & & A_{i+1} & \\
B_i \arrow[r] & X_i \arrow[r] \arrow[dr] \arrow[ddr] & Y_i \arrow[ur] \arrow[r] & X_{i+1} \arrow[r] \arrow[dr] & B_{i+2} \\
C_i \arrow[ur] & & B_{i+1} \arrow[ur] & & C_{i+2} \\
& & C_{i+1} \arrow[uur] & &
\end{tikzcd}$$

Figure 24. Definedness for P_2.

P_2 has channel X, not present in P_1''. Fortunately, we can still compare the two systems by considering a value sequence system without X, but equivalent to P_2:

$$\begin{tikzcd}
A_i \arrow[r] & Y_i \arrow[r] \arrow[dr] \arrow[ddr] & A_{i+1} \\
B_i \arrow[ur] \arrow[r] \arrow[dr] & B_{i+1} \arrow[r] \arrow[dr] & B_{i+2} \\
C_i \arrow[uur] \arrow[ur] \arrow[r] & C_{i+1} \arrow[ur] \arrow[r] & C_{i+2}
\end{tikzcd}$$

Figure 25. Definedness for projection of P_2 onto (A,B,C,Y).

because the elements of X are not necessary in order to specify which values depend on which other values (see value sequence execution model).

The domains of B_{i+2} and C_{i+2} have been weakened, while all other domains are the same. E.g. the internal contribution to the definedness of C_{i+2} changed from def(Y_{i+1}) (which can be extended to def$(Y_{i+1}) \wedge$ def$(B_{i+1}) \wedge$def(C_{i+1})) to the weaker def$(Y_i) \wedge$def$(B_{i+1}) \wedge$def(C_{i+1}), while the external contribution remained the same, by assumption.

The new definedness formulas are stable with the new system: by definition of stability, we check:

i. they are still monotonic in other definedness variables, so there is no way for a **false**-to-**true** change of another definedness variable to make the formula false.
ii. they do not depend on other value variables.

similarly, one must check stability of the value formulas:

i-ii. eval(Y_i) depends only on A_i, B_i and C_i, all of which are defined in order for edef(Y_i) to evaluate to **true**; hence if edef(Y_i) was **true** then A_i, B_i and C_i remain defined with the same values, and in particular eval(Y_i) does not change.

Domain Weakening: Example 2

In the last example, the domains of the operands were weakened, while the domain of the result remained the same. In the next example, we will see the opposite: the domain of the result will be weakened, while the domains of the operands will remain the same. Furthermore, we will see why definedness formulas might need to depend on the values of other elements (and not just their definedness).

This example is motivated by the PL1 compiler [MN]. If we give the PL1 compiler the following "AND" process:

$$*[A?a, B?b;\ Y!(a \wedge b)] \tag{26}$$

then it gives the following implementation:

$$\begin{array}{l} *[[\overline{A=0} \longrightarrow A?; B?,\ Y!0 \\ \quad |\overline{B=0} \longrightarrow B?; A?,\ Y!0 \\ \quad |\overline{A=1} \vee \overline{B=1} \longrightarrow A?, B?;\ Y!1 \\ \ ]] \end{array} \tag{27}$$

The definedness formula for Y_i changed from $\text{def}(A_i) \wedge \text{def}(B_i)$ to

$$\big(\text{def}(A_i) \wedge (\text{val}(A_i) = 0)\big) \ \vee\ \big(\text{def}(B_i) \wedge (\text{val}(B_i) = 0)\big) \ \vee\ \big(\text{def}(A_i) \wedge \text{def}(B_i)\big) \tag{28}$$

As in example 1, the new formulas are stable with the new system because the definedness formulas are monotonic in the definedness variables, and the value formulas cannot change by making an undefined variable defined. The interesting case is when the value was 0 after only one operand arrived. This 0 remains 0 after the second operand arrives. Thus we have a domain weakening.

The decomposed AND process is an example of a value sequence system which is not SS-deterministic.

Domain Weakening: Example 3

This example deals with adding slack to a value sequence system; once we show that domain weakening is a valid transformation (in the next section), this example will show that value sequence systems are slack elastic.

Consider a Value Sequence System with a channel L. Every channel has the dependence $L_i \to L_{i+1}$. Also the definedness of L_i in general has a factor (i.e. formula ANDed in) X_i due to the sending end, and a factor Y_i due to the receiving end (this modularity will be formalized in the section "Constructing Value Sequences"). I.e. just knowing that L is a channel in a value sequence system, we can write:

$$\text{edef}(L_i) = \text{def}(L_{i-1}) \wedge X_i \wedge Y_i. \tag{29}$$

We use the following picture to express Equation 29:

$$\begin{tikzcd}
X_i \arrow[d] & Y_i \arrow[dl] & & X_i \arrow[d] & Y_{i+1} \arrow[d] \\
L_i \arrow[r] & L_{i+1} & \text{i.e.} & L_i \arrow[r] & L_{i+1}
\end{tikzcd}$$

Figure 30. Value Sequence System with channel L.

If we now replace channel L by an $*[L; R]$ buffer, we move Y_i out of the definedness expression for L_i, and into the definedness expression for R_i:

$$\begin{tikzcd}
X_i \arrow[d] & Y_i \arrow[d] & & & X_i \arrow[d] & Y_i \arrow[d] \\
L_i \arrow[r] & R_i \arrow[r] & L_{i+1} & \arrow[r, Rightarrow, "\text{Project onto (L)}"] & L_i \arrow[r] & L_{i+1}
\end{tikzcd}$$

Figure 31. Value Sequence System with channel L replaced by $*[L; R]$ buffer.

comparing Figures 30 and 31, we see that edef(L_i) was weakened from def$(L_{i-1}) \wedge X_i \wedge Y_i$ (which extends to def$(L_{i-1}) \wedge X_i \wedge Y_i \wedge Y_{i-1}$) to def$(L_{i-1}) \wedge X_i \wedge Y_{i-1}$.

Motivated by the above examples, we will from now on refer to "domain weakening" transformations, instead of the more vague "decomposition transformations".

The Determinism Theorem for Value Sequence Systems

In analogy to the "first determinism theorem" for D-CHP systems, we would like to show that any execution of a value sequence system preserves underlying sequences. To begin, we must give the execution model for value sequence systems.

Execution Model

An *execution* of a value sequence system is a sequence of states s_i, each of which is a defined-value assignment. The initial state s_0 must assign all definedness variables to **false**. State s_{i+1} is obtained from state s_i by executing an enabled definedness formula. We say a definedness formula edef(G) is *enabled* if

$$(\overline{\mathrm{edef}(G)} = \mathbf{true}) \wedge (\overline{\mathrm{def}(G)} = \mathbf{false}). \tag{32}$$

And edef(G) is *executed* by assigning **true** to def(G), and $\overline{\mathrm{eval}(G)}$ to val(G). Finally, we require that an enabled formula eventually be executed.

The Determinism Theorem

Theorem. *Every Value Sequence System is IO-Deterministic.*

proof: We can suppose that every channel is an IO channel, i.e. we will show that every channel always defines the same elements, and to the same values.

Suppose we have two executions, E and E'. Let $P(i)$ be the statement that all definedness variables assigned **true** at step s_i of E are eventually defined in E', with the same values. The base case $P(0)$ follows from execution initialization (nothing is defined, so there is nothing to show). Now suppose we have the inductive assumption $P(i)$. To get to s_{i+1}, some definedness formula ϕ in E was enabled and executed. Consider all definedness variables which are true in s_i. Those were assigned true for smaller i, so by inductive assumption, they were assigned true in E', and the corresponding values are the same eventually in E'. Once a variable becomes defined, it never becomes undefined or changes its value. Hence ϕ is enabled eventually in E'. Hence it must execute eventually, after some state s'_j in E'. Since ϕ is stable with E', it suffices that the definedness variables which are true in s_i are true in s'_j; we are guaranteed that the value newly defined in s'_{j+1} equals the value newly defined in s_{i+1} (even if unrelated values are defined). Thus we have $P(i+1)$. By induction, all variables assigned in E get the same assignments in E'.

Swapping E and E', we find that all variables assigned in E' get the same assignments in E. Thus each channel which communicates a sequence in E communicates the same sequence in E'. □

Domain Weakening only Extends Value Sequences

We have seen three examples of decomposition transformations widely used in practice, that can be viewed as domain weakening. This motivates us to ask how execution can be affected by domain weakening.

Suppose we have two value sequence systems; system S' is obtained by weakening some definedness formulas in S. Consider an execution E of S. Any finite prefix of this execution is also a valid execution prefix of S'; by induction every formula enabled in S is certainly enabled in S' since the condition is weaker. Since we are only considering a finite prefix, we can ignore the requirement that an enabled formula be executed eventually.

Now we claim that for any finite sequence of variables that was defined in S, the same sequence will be defined to the same values in S'. This sequence of variables was assigned after some finite execution prefix of E. By the determinism theorem and the observation about execution prefixes, those variables will be defined in *any* execution of S'.

Finally we make the same claim, but for infinite sequences. Simply consider all prefixes of an infinite sequence. □

Projection Preserves Value Sequences

Sometimes (as in two domain weakening examples) it is necessary (for purposes of comparing different systems) to remove elements from a system S, replacing their variables by the corresponding definedness formulas, to obtain a "projected" system S'. Without loss of generality, we can assume that just one variable, G, was removed.

Consider an exection E of S. We now claim that the execution sequence E' consisting of E where the assignment to G (which we assume exists because otherwise there is nothing to show) is removed. Consider a formula ϕ executed after s_i in E. Every definedness and value variable of G in ϕ has some value in s_i, which must either be the same as the value of the corresponding formula, or def(G) = **false** if edef(G) has not been executed yet. Even in the latter case, ϕ is still enabled if def(G) is replaced by edef(G), by stability. Thus ϕ, the next executed formula in E', is enabled.

Any formula enabled in E' is enabled in E, hence executed eventually in E, hence executed eventually in E'.

By the determinism theorem, S' will never define any values that were not defined in S. □

Caution: "projection" as defined for value sequences is a much simpler operation than the "projection" technique used in CHP system design; the latter can remove backward dependencies, and hence can certainly change communication sequences. By contrast, a projection in a value sequence system preserves dependency between two element variables even when intermediate channels are removed, by expanding the dependency formula. As a consequence, a value sequence system is not always directly implementable in CHP.

Constructing Value Sequence Systems

Generic VSS Theorem. *Every L-deterministic system is a value sequence system.*

We will in fact prove a stronger (and more useful) statement: that every D-CHP process is a *value sequence process*. Thus, if we have a system which is mostly D-CHP processes, but contains a few processes with probes (perhaps because they were decomposed, as in our second domain weakening example), we can handle the D-CHP processes generically, while the remaining value sequence processes must be constructed manually (using the techniques presented in the domain weakening examples).

In order to make sense of the idea of a process in a value sequence system, we must separate constraints imposed by the receiver from those imposed by the sender. Thus for each element variable F, we create the following formulas: *sender definedness* sdef(F), *receiver definedness* rdef(F), and *sender value* sval(F). We turn these formulas into a value sequence system as follows:

$$\begin{aligned} \mathrm{edef}(F) &\overset{\mathrm{def}}{=} \mathrm{rdef}(F) \wedge \mathrm{sdef}(F) \\ \mathrm{eval}(F) &\overset{\mathrm{def}}{=} \mathrm{sval}(F) \end{aligned} \tag{33}$$

Definition. *A* **Value Sequence Process** *P is a set of channels identified as input and output channels, and a set of named formulas, such that*

i. Each input channel C is connected to exactly one other process, is an output channel in the other process, and for each i the P provides the formula rdef(C_i).

ii. Each output channel C is connected to exactly one other process, is an input channel in the other process, and for each i the P provides the formulas sdef(C_i) *and* sval(C_i).

iii. The formulas provided by P use only variables associated to the channels of P.

In light of Equation 33, the behavior of a channel is in general affected by both the sender and receiver, as in CHP. However, the stability of the system can be handled in a modular way: the stability of a formula is affected by the variables used in that formula, but not by formulas of other elements, and if sdef(F) and rdef(F) are stable, then edef(F) is stable (stability distributes over conjunction). Hence we can show that all formulas in a system are stable, just by showing that all formulas in all processes are stable. Thus we say a *process* is *stable* if all its formulas are stable, and we proceed to show that all processes are stable.

Continuing with the Theorem proof, we now show how to model a D-CHP process as a value sequence process. We begin with a D-CHP process, canonically written (see Program 4):

$$\begin{aligned} x := x_1(); \quad & *[[< [] : i : 0..m : s(x) = i() \longrightarrow (S_{i,1}!, \ldots, S_{i,|S_i|}!) := g_i(x) >]; \\ & [< [] : i : 0..k : r(x) = i() \longrightarrow x := f_i(x, R_{i,1}?, \ldots, R_{i,|R_i|}?) >]], \end{aligned} \tag{34}$$

The subtetly in rewriting this as a value sequence system is that the sequence of x_j have indices which are related in a complicated way with the element variable indices of each channel (in fact the inverse function is not recursive). Therefore we proceed slowly; first we associate definedness and value *state formulas* to each x_j (and the intermediate semicolon after each x_j). Then we include these formulas in the channel element formulas we are ultimately interested in.

We begin by defining the initial state; we define two *state formulas*:

$$\begin{aligned} \mathrm{edef}(x_1) &\overset{\mathrm{def}}{=} \mathbf{true} \\ \mathrm{eval}(x_1) &\overset{\mathrm{def}}{=} x_1() \end{aligned} \tag{35}$$

For each channel S_l, we define a formula for its starting index:

$$t(S_l, 1) \overset{\mathrm{def}}{=} 1 \tag{36}$$

Next, we express the conditions for passing the send and receive statements, assuming $\mathrm{edef}(x_j)$, i.e. assuming the loop body is reached for the jth time.

$$\begin{aligned} \mathrm{sdef}(x_j) &\overset{\mathrm{def}}{=} \mathrm{edef}(x_j) \\ &\quad \wedge \left\langle \vee i : 0..m : (s(\mathrm{eval}(x_j)){=}i) \wedge \mathrm{def}(S_{i,1}, t(S_{i,1}, j)) \wedge \cdots \wedge \mathrm{def}(S_{i,|S_i|}, t(S_{i,|S_i|}, j)) \right\rangle \\ \mathrm{rdef}(x_j) &\overset{\mathrm{def}}{=} \mathrm{edef}(x_j) \wedge \mathrm{sdef}(x_j) \\ &\quad \wedge \left\langle \vee i : 0..k : (r(\mathrm{eval}(x_j)){=}i) \wedge \mathrm{def}(R_{i,1}, t(R_{i,1}, j)) \wedge \cdots \wedge \mathrm{def}(R_{i,|R_i|}, t(R_{i,|R_i|}, j)) \right\rangle \end{aligned} \tag{37}$$

where the definedness variables appear in the following expressions:

$$\begin{aligned} \mathrm{def}(S_l, t) &\overset{\mathrm{def}}{=} \bigvee_{t'} \mathrm{def}(S_{l,t'}) \wedge (t = t') \\ \mathrm{def}(R_l, t) &\overset{\mathrm{def}}{=} \bigvee_{t'} \mathrm{def}(R_{l,t'}) \wedge (t = t') \end{aligned} \tag{38}$$

Similarly we use the value variables in the following state transition expression:

$$\mathrm{sval}(x_j) \overset{\mathrm{def}}{=} \left\langle +i : 0..k : (r(\mathrm{eval}(x_j)){=}i) * f_i\big(\mathrm{eval}(x_j), \mathrm{val}(R_{i,1}, t(R_{i,1}, j)), \ldots, \mathrm{val}(R_{i,|R_i|}, t(R_{i,|R_i|}, j))\big) \right\rangle \tag{39}$$

Now we can describe when the next state is reached:

$$\mathrm{edef}(x_{j+1}) \overset{\mathrm{def}}{=} \mathrm{rdef}(x_j) \tag{40}$$

and how the channel element indices are updated for each S_l:

$$t(S_l, j+1) = t(S_l, j) + 1 * \left(S_l {\in} S_{s(\mathrm{eval}(x_j))} \right) \tag{41}$$

Notice how we avoid circular definition: starting from $j = 1$, we apply equations 37-41 in that order, and repeat to obtain formulas for increasing j. Next we can define element formulas, assuming state formulas are defined for all j.

A value is sent on S_l after state x_j if $S_l \in S_{s(\mathrm{eval}(x_j))}$. Hence value number t' is sent on S_l if some such x_j for which $t(S_l, j) = t'$ has become defined. Formally,

$$\mathrm{sdef}(S_{l,t'}) \stackrel{\mathrm{def}}{=} \Big\langle \vee\, j : \infty : \mathrm{edef}(x_j) \wedge \Big(t(S_l, j){=}t'\Big) \wedge \Big(S_l \in S_{s(\mathrm{eval}(x_j))}\Big)\Big\rangle. \tag{42}$$

Because this is an unbounded search, it is *not* in general a recursive function (see below). That is the price (worth paying, for system modularity) of having local channel indices that are not synchronized with some global index of the system.

We also describe which value is sent:

$$\mathrm{sval}(S_{l,t'}) \stackrel{\mathrm{def}}{=} \left\langle +j : \infty : \left(\mathrm{edef}(x_j) \wedge \Big(t(S_l, j){=}t'\Big) \wedge \Big(S_l \in S_{s(\mathrm{eval}(x_j))}\Big)\right) * g_{il}\left(\mathrm{eval}(x_j)\right)\right\rangle, \tag{43}$$

where $g_{il}(x)$ is the component of $g_i(x)$ to be sent on S_l.

And we describe when value number t' is received on R_l:

$$\mathrm{rdef}(R_{l,t'}) \stackrel{\mathrm{def}}{=} \Big\langle \vee\, j : \infty : \mathrm{sdef}(x_j) \wedge \Big(t(R_l, j){=}t'\Big) \wedge \Big(R_l \in R_{r(\mathrm{eval}(x_j))}\Big)\Big\rangle. \tag{44}$$

Notice that the searches (Equations 42-44) can have at most one j-term **true** at a time, since only one communication can have a given index t'.

Finally, we must show that the resulting $\mathrm{sdef}(S_{l,t'})$, $\mathrm{rdef}(R_{l,t'})$, and $\mathrm{sval}(S_{l,t'})$ formulas are stable. Consider a defined-value assignment. Suppose $\mathrm{sdef}(S_{l,t'})$ holds; the term for some j is **true**. By construction, $\mathrm{edef}(x_{j'})$ for all $j' \leq j$ must also hold. By $\mathrm{sdef}(x_j)$ (Equation 37), all terms of the form $\mathrm{def}(S_{i,p}, t(S_{i,p}, j'))$ are **true**. Similarly, all terms of the form $\mathrm{def}(R_{i,p}, t(R_{i,p}, j'))$ are **true**. Hence we can assume that all these definedness variables remain true in a new assignment (part i. of stability), and that the corresponding value variables $\mathrm{val}(R_{i,p}, t(R_{i,p}, j'))$ do not change (part ii. of stability). With these values known, we can reconstruct $\mathrm{sdef}(x_{j'})$, $\mathrm{rdef}(x_{j'})$, $t(S_l, j')$, and $\mathrm{eval}(x_{j'})$ with the same values, by induction on j', up to j. Thus the searches (Equations 42-44) find the same j-term to be **true**, and with the same $\mathrm{eval}(x_j)$, hence all element formulas evaluate to the same numbers. □

When are the definedness and value formulas computable?

Equations 42-44 define a value sequence system. I.e. they tell us that at any point in an execution, the status (definedness and value) of the next element on a channel is a function (actually an infinite formula) of the previously defined elements.

In fact the functions are recursively enumerable, in the sense, for example, that if $\mathrm{sdef}(S_{l,t'})$ is **true**, the unbounded searches will terminate. Now we ask when they are recursive.

A process is said to *loop internally* if there is some state x such that $r(x) = s(x) = 0$, and all states in the sequence of iterates $f_0^{\circ n}(x)$ also have this property. I.e. an internally looping process can go into an infinite loop which is independent of external communication. We have:

Value Sequence Computability Theorem. *Every non-looping L-deterministic process is a value sequence process in which the defining formulas Equations 42-44 are computable.*

Proof: Suppose such a value sequence process does not loop internally (e.g. because it was constructed that way by the system designer). Now consider any defined-value assignment which might occur in a value sequence execution. First, notice that by induction, any such sequence has at most finitely many $\mathrm{def}(x_j)$ assigned **true**: as we define the x_j (by building successively larger formulas from Equations 37-41), we can only visit finitely many states with $r(x) = s(x) = 0$ (since there is no internal looping) before depending on a definedness variable (of which there are only finitely many) being **true**. Second, notice that the x_j are defined consecutively, starting with x_1. Thus $\mathrm{sdef}(S_{l,t'})$ and the other formulas are recursive: we can terminate the search immediately as soon as we find an undefined x_j, and we have argued that such an x_j exists. □

Conclusion

By designing L-deterministic systems, we obtain the properties indicated in this containment diagram:

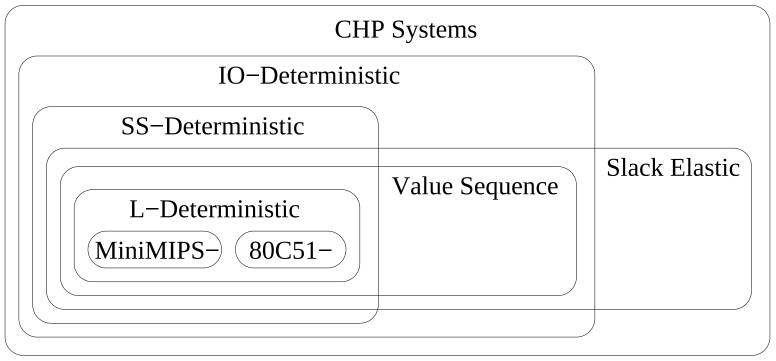

Figure 45. Containment of the different kinds of deterministic systems, and relation to transformation properties (slack elasticity and value sequence systemness).

All containments are strict, as we have seen

i. a slack elastic system which is not IO-deterministic.
ii. a slack elastic system which is IO-deterministic, but not SS-deterministic.
iii. a slack elastic system which is SS-deterministic, but not L-deterministic.
iv. an SS-deterministic system which is neither L-deterministic nor slack elastic.
v. an IO-deterministic, slack elastic system which is not a value sequence system.
vi. a value sequence system which is not SS-deterministic.
vii. a slack elastic, SS-Deterministic system which is not a value sequence system (because it does not allow function decomposition)

and we can easily obtain

viii. a system which is neither slack elastic nor IO-deterministic, by combining a non-slack-elastic system with a non-IO-deterministic system.
ix. a system which IO-deterministic but neither SS-deterministic nor slack elastic, by combining a non-SS-deterministic IO-deterministic system with a non-slack-elastic IO-deterministic system.
x. a non-IO-deterministic, slack elastic system which is not a value sequence system, by combining an IO-deterministic, non-value-sequence, slack elastic system with a non-IO-deterministic, slack elastic system.
xi. a large L-Deterministic system other than a MiniMIPS- or an 80C51-, by combining both in one design (not that anyone would actually do that).
xii. a slack elastic, non-SS-Deterministic non-value-sequence-system, by combining a slack elastic, non-value-sequence-system with a slack elastic, non-SS-Deterministic system.

We have seen how systems that remain IO-deterministic after a transformation is applied can be proven correct just by exhibiting a single trace which behaves like the original system. Furthermore, we have introduced the notion of *value sequence* systems, which can withstand more decomposition transformations than slack elastic systems can. Finally, we have proven that L-deterministic systems are IO-deterministic in two ways: by showing that they are SS-deterministic, and by showing that they are value sequence systems.

Future Work

One could define “nondeterministic value sequence systems”, which allow the value functions to be nondeterministic. Such systems are still slack elastic, and allow domain weakening transformations, but are just not IO-determinstic.

As we have seen that slack elasticity is not a sufficient condition for the correctness of the decomposition of functions of more than two inputs, one must beware of problems that will arise in generalizing arguments for the decomposition of two input functions in slack elastic systems, such as those in [MLM].

Many of the ideas used in this paper could be generalized to production rules. For example, one might define an analog of SS-deterministic for production rules, and analyse its relation to PRS stability and non-interference.

Appendix I: Isolated Sequential Processes (ISP)

In the first section of the paper, we showed that there is a canonical form for D-CHP programs. If that argument was hard to follow, here is a simpler example: we show the Turing thesis for isolated CHP programs (called Isolated Sequential Processes).

Definition: ISP alphabet. *An ISP alphabet is a union of the following:*

I. A finite set of *variables.*
II. A finite set of *function symbols.*
III. Delimiters "[","]","*","▯","→","(", ")", and ";".

Definition: ISP program. *An ISP program P consists of the following ordered quadruple:*

i. An ISP alphabet.
ii. Domain sets for each of the variables. I.e. for each variable x, there is some domain $\mathcal{D}(x)$, and one may speak of x having an interpretation $\overline{x} \in \mathcal{D}(x)$. The reader may suppose that the domain sets are finite, though none of our results will depend on that.
iii. Interpretations of each of the function symbols, i.e. for each f, there is some *canonical stable argument list* $\text{val}(y), y$, and some function
$$\overline{f} : \mathcal{D}(y_1), \ldots, \mathcal{D}(y_k) \mapsto \mathcal{D}(y) \tag{46}$$
iv. A *program text* (simply denoted P when no confusion can arise) consisting of a string of symbols from the alphabet, constructed through one of the following rules:
   1. $P \equiv x := f(\text{val}(x))$
   2. $P \equiv (P_1; P_2)$
   3. $P \equiv *[f(\text{val}(x)) \rightarrow P_1]$
   4. $P \equiv [f_1(\text{val}(x)) \rightarrow P_1 \ ▯ \ldots ▯ \ f_m(\text{val}(x)) \rightarrow P_m]$

where $\text{val}(x)$ and x are arbitrary variables, $f_1, \ldots, f_m$ are arbitrary function symbols, and $P_1, \ldots, P_m$ are arbitrary programs of the same alphabet as P. (note: we write optional commas in our argument lists, but we do not consider the "," as part of our language, to avoid confusion with the concurrency comma used in more powerful languages).

Furthermore, whenever $f(\text{val}(x))$ appears in a program, we require the argument list $(\text{val}(x))$ to be *stable* with f, i.e. if f has canonical stable argument list $\text{val}(y), y$, then $\mathcal{D}(x_i) = \mathcal{D}(y_i)$ for all i. Also, for programs in form 1, we require that $\mathcal{D}(x) = \mathcal{D}(y)$, and for programs in forms 3 and 4, we require $\mathcal{D}(x) = \{\textbf{true}, \textbf{false}\}$.

Note: For simplicity, we do not provide syntax for complex expressions. The solution is to replace each complex expression by a new function symbol with an appropriate interpretation.

Definition: Semantics. *The* **semantics** $\mathcal{S}(P)$ *of an ISP program P are the set of ordered predicate pairs* $\big(A[\text{val}(x)], B[\text{val}(x)]\big)$, *where A and B are first-order formulas, and* $\text{val}(x)$ *are variables of program P, such that the Hoare triple* $\{A[\text{val}(x)]\}\ P\ \{B[\text{val}(x)]\}$ *holds.*

Note: the first-order language containing A and B should be fixed, and should contain the variables of program P, but it may be larger. For example, it may contain arithmetic symbols, and function symbols used in P.

Each ISP program is a CHP program, so for each ISP statement, we define the set of Hoare triples that hold to be the set of Hoare triples which hold for the corresponding CHP program. One exception is that we will not allow a program to block at a selection statement (it would never unblock in that situation anyway).

Definition: Semantic Equivalence. *Two programs of the same language are* **semantically equivalent** *if they have identical semantics.*

The first fact that we will borrow from CHP is that ";" is associative; i.e. a transformation which changes the parenthesization of a program constructed through repeated application of rule 2 preserves semantics. Thus for the purposes of finding semantically equivalent programs, we omit these parentheses.

The following two definitions will also allow us to rename the variables in a program, without changing the semantics (in the sense defined below).

Definition: Semantic Semiconjugacy. *An ISP program P_1 of a language with variables* $\mathrm{val}(x)$ *is* **semantically semiconjugate** *to an ISP program P_2 of a language with variables $y_1, \ldots, y_m$ if there exists a* **conjugation function**

$$\overline{\sigma} : (\mathrm{val}(\overline{x})) \mapsto (\overline{y}_1, \ldots, \overline{y}_m) \tag{47}$$

such that

$$\big(A[\mathrm{val}(x)], B[\mathrm{val}(x)]\big) \in \mathcal{S}(P_1) \iff \big(A[\sigma(\mathrm{val}(x))], B[\sigma(\mathrm{val}(x))]\big) \in \mathcal{S}(P_2). \tag{48}$$

Definition: Semantic Conjugacy. *Two ISP programs are* **semantically conjugate** *if they are semiconjugate via a bijective conjugation function.*

Definition: ISP-canonical form. *An ISP program is in* **ISP-canonical form** *if its program text is the following:*

$$x := x_1(); *[C(x) \to x := E(x)]. \tag{49}$$

Of course, the different possible interpretations of the variable x, and the functions C and E and the 0-ary (constant) function x_1 lead to vastly different possible semantics for programs in ISP-canonical form, in fact so vastly different that the following theorem holds:

Theorem: ISP-canonical form. *Every ISP program whose behavior is independent of the values of its variables before execution is semantically conjugate to a program in ISP-canonical form.*

Proof. We will prove the stronger statement that every ISP program corresponds to a program of the following form:

$$x := x_1(x); *[C(x) \to x := E(x)]. \tag{50}$$

As a lemma, we show that any program corresponds (is semantically conjugate to) a program of the same syntactic depth in which there is only one variable. Proof: suppose the program has variables $x_1 \ldots x_n$. We have encoding functions α and $\alpha_1 \ldots \alpha_n$ such that $\overline{\alpha_i(\alpha(x_1, \ldots, x_n))} = \overline{x_i}$ for all $\overline{x_i}$. We replace each use of x_i with $\alpha_i(x)$, and each assignment of $x_i := E$ with $x := \alpha(\alpha_1(x), \ldots, E, \ldots \alpha_n(x))$ (where the E is in position i).

Our program construction rules give us a syntactic principle of induction on programs; we can show a property about a program assuming that it holds for the program parts. In this case the propery is that the program is syntactically conjugate to Program 50. By the lemma, we can suppose that the program has only one variable.

1. $P \equiv x := f(x)$. In this case, let $\overline{x_1} = \overline{f}$, and $\overline{C} = 0$.
2. $P \equiv (P_1; P_2)$. By assumption, P_1 and P_2 have the form (50). Applying the lemma, we find that P_1 and P_2 remain in form (50) and have the same variable. Thus

$$\begin{aligned} P \equiv\ & x := x_1(x);\ *[C_1(x) \longrightarrow x := E_1(x)];\ x := x_2(x); *[C_2(x) \longrightarrow x := E_2(x)] \\ \equiv\ & x := x_1(x); y := 0; \\ & *[y \neq 3 \longrightarrow x := (y{=}0)*E_1(x) + (y{=}1)*x_2(x) + (y{=}2)*E_2(x); \\ & \qquad y := (y{=}0 \wedge \neg C_1(x))*1 + (y{=}1)*2 + (y{=}2 \wedge \neg C_2(x))*3] \end{aligned} \tag{51}$$

which has the proper form because each $x := f(x); y := g(x, y)$ becomes $x := F(x); x := G(x)$ by the lemma, and we change that to $x := G(F(x))$.

3. $$\begin{aligned} P \equiv\ & *[f(x) \longrightarrow P_1] \\ \equiv\ & *[f(x) \longrightarrow x := x_1(x);\ *[C(x) \longrightarrow x := E(x)]] \\ \equiv\ & y{\downarrow}; *[f(x) \vee (C(x) \wedge y) \longrightarrow x := \neg y * x_1(x) + y*E(x);\ y := C(x)] \end{aligned}$$

4. $$\begin{aligned} P \equiv\ & [f_1(x) \longrightarrow P_1\ \square\ \ldots\ \square\ f_m(x) \longrightarrow P_m] \\ \equiv\ & [f_1(x) \longrightarrow x := x_1(x); *[C_1(x) \longrightarrow x := E_1(x)]\ \square\ \ldots \\ & \square f_m(x) \longrightarrow x := x_m(x); *[C_m(x) \longrightarrow x := E_m(x)]] \\ \equiv\ & y := 0; *[(y{=}0) \vee (y{=}1 \wedge C_1(x)) \vee \cdots \vee (y{=}m \wedge C_m(x)) \longrightarrow \\ & \qquad y' {:=} \mathit{argf}(x);\ x {:=} x_{y'}(x)*(y{=}0) + E_y(x)*(y{\neq}0);\ y {:=} y'*(y{=}0) + y*(y{\neq}0)] \end{aligned}$$

where $\mathit{argf}(x)$ is the unique i such that $f_i(x)$ evaluates to **true**.

Appendix II: Concurrency Within Processes

It has been suggested that the CHP comma operator (parallel statement composition) be considered in this paper. There are two objections:

i. The comma operator is not powerful enough to describe all possible intraprocess concurrency. For example, dependencies in the form of a cross diamond cannot be expressed:

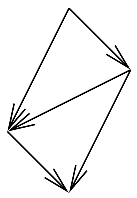

Figure 52. Cross Diamond.

ii. The comma operator complicates the process. Even worse, the execution of the process becomes much more complicated; it no longer consists of a sequence of simple states.

Fortunately, we can solve both problems in one step. Given a CHP program, we replace each statement of the form S_1, S_2 with the *fork* statement $L_1!x, L_2!x$, followed by the *join* statement $x := f(R_1?, R_2?)$. We add two processes $*[L_1?x; S_1; R_1!x]$ and $*[L_2?x; S_2; R_2!x]$. We have to assume that no shared channels result from doing this. But such an assumption is not unreasonable; after all there are many perfectly acceptable CHP programs, such as

$$*[(A?x; B?x), B?x] \quad \| \quad x\uparrow; *[B!x; A!x; B!x]$$

in which it is not syntactically obvious that they do not concurrently access channels and variables, even if one can show (as in the above example) that they do not, with a little work.

Instead of allowing arbitrary CHP programs, we provide the fork and join operations. In fact, given any dependency graph (e.g. Figure 52), we can implement each node as a process, with a join over all incoming edges and a fork over all outgoing edges.

Appendix III: The Counterexample Annotated

We now show more rigorously that the environment Program (17) forms a slack elastic system together with P_1, but not with P_2.

The undecomposed system is slack elastic.

Suppose arbitrary slack is added to channels A, B, C, and Y. Of course we do not encode the amount of slack into the processes under consideration (because if this is allowed then slack elasticity would not make any sense) but we can refer to the amount of slack in arguments about the system's execution. We use $\#B$ to denote the number of communications on B, and $s(B)$ to denote the amount of slack (number of $*[L; R]$ buffer processes) added to channel B. Consider the environment program:

```
$b := 1;$
$*[$
    (i)                              $\{\ \#A = \#B = \#C = \#Y$ and $b \le s(B) + 1\ \}$
    $i := 0; *[i < b \to B!0; i := i{+}1];$

    (ii)                             $\{\ \#A = \#B - b = \#C = \#Y\ \}$
    $C!0;$

    (iii)                            $\{\ \#A = \#B - b = \#C - 1 = \#Y\ \}$
    $[\overline{B} \to B!0; b := b{+}1 \mid \neg\overline{B} \to \mathit{skip}];$

    {now flush all channels}
    (iv)                             $\{\ \#A = \#B - b = \#C - 1 = \#Y$ and $b \le s(B) + 1\ \}$
    $A!0;\ Y?y;$

    (v)                              $\{\ \#A = \#B - (b-1) = \#C = \#Y\ \}$
    $i := 0; *[i < b{-}1 \to A!0;\ C!0;\ Y?y;\ i{:=}i{+}1]$

$]$
```

together with the undecomposed function process

$$P_1 \overset{\text{def}}{=} *[A?a, B?b, C?c; Y!f(a, g(b, c))].$$

Invariant (i) states that the system is "reset": all processes (including the buffer processes) are at the beginning of their main loops. Also we know that b does not exceed one plus the amount of slack on B. Knowing this, we can say that the loop statement (i) always terminates, and we reach (ii), having sent b items on B. Since no A or C action has occured since reset, we know that P_1 has not passed its first explicit semicolon at this point, hence (iii) is reached. (iii) obviously can never block, so (iv) is reached. Suppose that b was incremented. Then $B!0$ occured, so we maintain $\#B - b = \#Y$. Also, no A action has occured since reset, so we know that $b \le s(B) + 1$, because (iv) was reached, hence the only way b items could have been sent without deadlock is if P_1 (which still has not received anything on A, hence remains before its first explicit semicolon) received one item, and there were $b - 1$ buffers on B to receive the other items.

From (iv), the goal is only to reset the system (but to remember b, which is an important feature of the counterexample). Since B and C have been sent, but P_1 has not reached its first explicit semicolon, we do $A!0; Y?y$. Then P_1 is at its first semicolon. Since there are $b - 1$ outstanding B messages, we will for the same reason do $A!0; C!0; Y?y$, $b - 1$ times. This restores the loop invariant.

If the original correctness function was that infinitely many values are sent on Y, then the *undecomposed* system with arbitrary slack added *is* a valid implementation.

Decomposing P_1 introduces deadlock not removable by adding slack.

Now consider the same environment process together with the decomposed function process

$$P_2 \overset{\text{def}}{=} *[A?a, X?x; Y!f(a,x)] \parallel *[B?b, C?c; X!g(b,c)].$$

Also, suppose that arbitrary slack has been added to channels A, B, C, X, and Y. The reader can suppose this added slack is 0, though we will show that any amount of added slack cannot prevent deadlock.

Since we are only interested in constructing a deadlock situation, we can (for simplicity of the argument) suppose for a contradiction that the decomposed system does not deadlock. All annotations except for $b \leq s(B)+1$ still hold, because the number of communication counts is always represented in the variables of the environment process alone (also because the decomposed system is capable of all executions the original system was capable of).

Since we only need to construct a single deadlock example, we will build a particular execution trace, by specifying what happens at step (iii). Suppose that $b \leq s(B)+1$. At (iii), we have sent at least one item on B, and one on C, so we can arrange that the process $*[B?b, C?c; X!g(b,c)]$ has sent on X, which is allowed because $*[A?a, X?x; Y!f(a,x)]$ has not started since the last reset. Therefore we can also have a *second* B item consumed by $*[B?b, C?c; X!g(b,c)]$, if $b \geq 2$. Then, assuming that $b \leq s(B)+1$, we can use the $b-1$ buffers to accept $b-2$ more B items, while leaving the last buffer empty at (iii). Thus we can always increment b, only assuming that $b \leq s(B)+1$.

Now consider what happens when we get to (i) with $b = s(B)+2$. We must send $s(B)+2$ items to the B input of $*[B?b, C?c; X!g(b,c)]$ *before* sending anything on C. This causes deadlock.

If the original correctness function was that infinitely many values are sent on Y, then the *decomposed* system with arbitrary slack added is *not* a valid implementation.

The undecomposed system is SS-deterministic.

Unfortunately the environment uses a probe, so it is not syntactically obvious that the system including this environment is SS-deterministic. In fact, if slack is added then it is *not* ss-deterministic. However, if no slack is added, then $b = 1$ invariantly. Thus if we treat every semicolon in the environment (and in the other processes as well) as a superstate, we find that the sequence of superstates is the same on any execution, and each semicolon is followed by a unique communication. I.e. the system is SS-deterministic.

Bibliography

[JD] Jack B. Dennis. "Data Flow Computation". *Proceedings of the NASA Advanced Study Institute on Control Flow and Data Flow.* NATO ASI Series F, 1984, pp. 345-398.

[AL] Andrew Lines. *Pipelined Asynchronous Circuits*, Masters thesis, Caltech, revised June 1998.

[AM] Alain J. Martin. *Synthesis of Asynchronous VLSI Circuits*, Caltech Computer Science Tech Report, 1990.

[RM] Rajit Manohar. *The Impact of Asynchrony on Computer Architecture*, PhD thesis, Caltech, 1998.

[MM] Rajit Manohar and Alain J. Martin. Slack Elasticity in Concurrent Computing. *Proceedings of the Fourth International Conference on the Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pp. 272-285, Springer-Verlag 1998

[MLM] Rajit Manohar, Tak-Kwan Lee, and Alain Martin. "Projection: A Synthesis Technique for Concurrent Systems". *Proceedings, Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society, April 18-22, 1999, Barcelona, pp. 125-134.

[MN] Mika Nyström. *Asynchronous Pulsed Logic Circuits*, PhD thesis, Caltech, 2001.

[KP] Karl Papadantonakis, *MiniMIPS Decomposition.*