Check for updates

The Composition of Concurrent Programs

K. Mani Chandy, Stephen Taylor
California Institute of Technology

31 August 1989

1 Introduction

This paper describes a notation for concurrent programs called PCN for Program Composition Notation. The notation is being implemented at Caltech on multicomputers (a network of computers that communicate by sending and receiving messages). A fragment of this notation has been implemented on a data-parallel computer — the Connection Machine — by Rajive Bagrodia at UCLA. PCN is an outgrowth of research on UNITY [1] and Strand [2]. The central ideas underlying PCN are discussed next.

1.1 Composition

The research goal of PCN is to study program composition: methods by which smaller programs can be put together to obtain larger ones. Languages such as Fortran have only one method of program composition — sequential composition: After one statement is completed, control flow passes to the next statement. Versions of Fortran have been proposed with another method of program composition: parallel composition. Functional programs employ functional composition. A variety of composition operators are discussed in [3]. Our primary goal, here, is to propose a notation that has a variety of program composition methods, and that allows programmers to specify their own composition operators in terms of the primitive operators in the notation.

PCN deals with just *one* kind of programming entity: a program. In particular, PCN does not include constructs such as processes or objects. The programs that are composed may be written in PCN itself, or in some other notation such as C, Fortran, Lisp, or ADA.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 089791-341-8/89/0011/0557 $1.50

1.2 Single Assignment Variables

Arguments have been made that logic programming, in which a variable is assigned a value at most once, is less prone to error than imperative programs, in which a variable can be assigned several values during the course of a program execution, because the meaning of a variable is clearer if it is assigned only once. However, destructive assignment allows less copying and more effective use of memory than single-assignment notations. Also, program proof techniques can be used to reduce (if not eliminate) error in programs that employ destructive assignment. PCN allows both single-assignment variables, and mutable variables, i.e., variables that can be assigned values an arbitrary number of times during an execution.

For reasons that will become clear, we call a single-assignment variable a *definition* variable. A definition variable must be assigned a value at most once during an execution. A (run-time) error occurs if a definition variable is assigned a value more than once during an execution.

A *definition* is a constant (i.e., integer, floating point number, character, string), or a definition variable, or a tuple whose elements are definitions. A tuple is a sequence of constants, variables (definition variables or mutable variables), and tuples between braces '{, and '}. For example, $\{x, \{y, \{2, 3\}\}\}$, is a tuple. More about tuples and constants later.

The initial value of a definition variable is a special symbol that indicates that it is undefined. The value of a definition variable is either undefined or it is a definition. The value of a definition variable can change in only one way: the value of a variable that is undefined can become a definition.

At a point in a computation, a definition e *reduces to* a definition e' if e' can be obtained from e by replacing occurrences of a variable in e by its definition, or (transitive closure) if there exists a definition d such that e reduces to d and d reduces to e'. For example, at a point in a computation, $\{y, 2\}$ reduces to $\{\{z, 1\}, 2\}$, if y is defined at that point as $\{z, 1\}$; furthermore, $\{y, 2\}$ reduces to $\{\{\{0\}, 1\}, 2\}$, if z is defined at that point as $\{0\}$.

At a point in a computation, an occurrence of a definition variable x can be replaced by its definition e, or by any definition e' where e reduces to e'.

By *contrast*, mutable variables have arbitrary values initially and may take on arbitrary values during the course of a program execution Furthermore, since the value of a mutable variable may change during an execution, a mutable variable cannot be replaced by its (current) value in all contexts.

Programs that are both clean, and efficient in memory usage, can be developed by judicious use of definitions and mutable values. There are cases where it appears helpful to start program development by writing programs that use only definitions, and later introduce mutable variables to obtain greater efficiency.

1.3 Concurrency

A great deal of research on concurrency deals with different methods of communication between entities — blocked and unblocked sends/receives for message-passing systems, and different methods of locking data for shared-memory systems. Programs in PCN communicate with each other by means of shared variables, but PCN has no explicit constructs for locking.

PCN has four basic composition operators, only one of which deals with concurrency explicitly: The parallel composition operator, '||'. The parallel composition operator is associative and commutative; so the order in which programs are composed in parallel is immaterial.

The central idea about shared variables in parallel composition is as follows.

A programmer must demonstrate, for a set S of programs composed in parallel, that during an execution:

1. A variable is modified by at most one program in S, and
2. a variable that is assigned a value by one program in S, and that is read by another program in S, is a definition variable.

Therefore, a variable that is accessed by two or more programs composed in parallel can change value in only one way: from undefined to defined. This monotonicity property makes locking unnecessary.

1.4 Nondeterminism

PCN programs can be nondeterministic or deterministic. Nondeterminism is helpful because a system that is to be developed may be inherently nondeterministic. Nondeterminism is also helpful because it reduces overspecificity; for instance, we may not want to insist that a program produce the correct results the *same way* each time it is called, provided that it does produce correct results. On the other hand, PCN programs can be written in a manner that guarantees that the programs are deterministic. Thus nondeterminism can be used in cases where it is helpful and avoided in cases where it is unhelpful.

2 Data Types

PCN has the usual basic data types (as in C, for instance), and the following structured data types: tuples, lists and arrays. Arrays are conventional (as in C).

A tuple is a sequence of items between braces '{' and '}'. The elements of a tuple can be arbitrary simple data types, or tuples themselves. A tuple is identical to a one-dimensional array, except that the elements of a tuple need not all be of the same type. The function, *sizeof*, applied to a tuple returns the number of elements in the tuple. The elements of a tuple x are $x[i]$ where $0 \leq i < sizeof(x)$. The number of steps required to access $x[i]$ is independent of i.

A list is a sequence of values between square brackets [, and]. A list is a special case of a tuple: An empty list, [], is {}, and a non-empty list, x, is a 2-tuple, $\{hx, tx\}$, where hx is the head of x and tx is the tail of x. For example, the list, $[0, 1, 2]$, is implemented as $\{0, \{1, \{2\}\}\}$.

A *constant* is a number or a character, or a tuple (or list) whose elements are constants. A constant only reduces to itself.

3 Programs

A program is a program heading, type declaration of arguments and local variables, and a *block*. The scope of a variable is the program in which it appears. A variable that is used in a program and which is not declared in that program is assumed to be a single-assignment variable. Parameters are passed by reference.

3.1 Blocks

The syntax of a block is described next in BNF. All nonterminal symbols are in italics, and all terminal symbols are in plain or boldface type. The notation $\prec$ *su* $\succ$, where *su* is a syntactic unit represents a list of zero or more instances of the syntactic unit where multiple instances are separated by commas.

block :: *assignment* |
equation |
procedure-call |
guard → *block* |
{ *primitive-composition-operator* $\prec$ *block* $\succ$ } |
{ *user-defined-composition-operator* $\prec$ *argument* $\succ$ }

3.2 Assignments

An assignment is of the form $x := e$, where x is a variable and e is an expression; x can be a definition variable or a mutable variable, and e can name definition variables and mutable variables. An assignment, $x := e$, is executed as follows.

1. Wait until all definition variables named in e reduce to constants, and
2. then assign to x the value of e with all definition variables replaced by the constants to which they reduce, and all mutable variables replaced by their values.

For example, the execution of $x := 1 + y + z$, where y is a definition variable and z is a mutable variable, and z has value 3, is as follows: Wait until y reduces to a constant, say 2, and then assign 6 to x because $1+y+z = 1+2+3$.

If the left-hand side of the assignment is a definition variable, the execution of the assignment defines the variable as the value of the right-hand side, and the variable retains this definition thereafter. If the left-hand side of the assignment is a mutable variable, the execution of the assignment causes the variable to get the value of the right-hand side, and this value can be changed by subsequent assignments.

3.3 Equations

An equation is of the form, $x = y$, where x is a definition variable, and y is a constant, a variable or a tuple. A definition y' is obtained from y by replacing all mutable variables in y by their values. The equation is executed as follows: A definition y' is obtained from y by replacing all mutable variables in y by their values, and the value of x becomes the definition y'. The execution of an equation always terminates, whereas the execution of an assignment may not terminate because the right-hand side of an assignment may name definition variables that never reduce to constants.

3.4 Guarded Blocks

The meaning of a block which has the syntax *guard* $\rightarrow$ *block* is given next. The syntax of a guard is:

guard :: *guard_element* | *guard_element*,*guard*
guard_element :: *true* | *false* | *expression binary_relation expression*

where *binary_relation* is $=$, $<$, $\leq$, $>$, or $\geq$, or a special symbol $\leftarrow$ for pattern-matching.

A (pattern) match is a syntactic convenience to refer to elements of tuples. The expression on the left side of the match is a tuple, and the expression on the right side is a variable. An example of a match is $\{x, y, 2\} \leftarrow z$. The syntax for the left side of a match is:
$\{ \prec pattern\_identifier \succ \}$
where pattern-identifier is an identifier that is not an argument or a local variable of the program; a pattern-identifier serves as a convenient place-holder for elements of a tuple.

A guard evaluates to one of three values: *succeeds*, *suspends*, or *fails*. For brevity, we shall say 'a guard succeeds ' instead of saying 'a guard evaluates to succeeds'; we use similar abbreviations for suspends and fails. We describe the evaluation of a guard by first giving the evaluation of a guard-element, and then defining the evaluation of a guard recursively. A guard is evaluated from left to right.

The guard-element *true* succeeds, and *false* fails. The guard-element $e * e'$, where $*$ is one of the usual comparison operators, $=$, $<$, $\leq$, $>$, or $\geq$, suspends if e or e' names a definition variable that does not evaluate to a a constant; otherwise, the expression $e * e'$ is evaluated in the usual way, by replacing each mutable variable by its value, and each definition variable by the constant to which it reduces; if $e * e'$ holds the guard-element succeeds, and fails otherwise.

For example, the guard-element $a + b = x + y$ suspends if a, b, x or y is a definition variable that does not reduce to a constant. If x and y are definition variables that reduce to 3 and 4, respectively, and a, b are mutable variables with values 2 and 5, respectively, then the guard-element succeeds because 2+5 = 3+4.

A guard-element, $\{t_0, .., t_{k-1}\} \leftarrow x$, suspends if x is a definition variable that does not reduce to a constant. If x reduces to a constant then the guard-element succeeds if x is a tuple of size k and fails otherwise. If the guard succeeds, then all instances of pattern-identifier t_i in the guard and its associated block are replaced by $x[i]$, and thus t_i serves as a place-holder for $x[i]$.

For example, the guard-element $\{m, xs\} \leftarrow x$ succeeds if x is a 2-tuple, and in the guard and its associated block, m is replaced by $x[0]$, and xs by $x[1]$.

The elements of a guard are evaluated from left to right until:

1. An element is found that fails in which case the guard fails, or
2. an element is found that suspends in which case the guard suspends, or
3. all elements succeed, in which case the guard succeeds.

The execution of a block b with syntax $g \rightarrow c$ is as follows:

1. Wait until g succeeds or to fails, and
2. if g succeeds then execute c else skip.

The meaning of an unguarded block, b is the same as that of the guarded block $true \rightarrow b$.

4 Composition Operators

4.1 Sequential Composition

The execution of a block, $\{; \prec block \succ\}$, is as follows: the blocks composed by sequential composition are executed in sequence. A sequential composition block terminates when the last block in its block-list, $\prec block \succ$, terminates.

4.2 Parallel Composition

The execution of the block, $\{\| \prec block \succ\}$, is as follows: all blocks composed by parallel composition are executed in parallel. A parallel composition block terminates when all blocks that it composes terminate.

The execution of a parallel composition block is fair: For each block b that is composed in parallel: It is always the case that b has terminated or a statement in b will be executed eventually.

4.3 Choice Composition

Restrict attention to choice blocks of the form: $\{? \prec guard \rightarrow block \succ\}$. (An unguarded block is treat as a guarded block with a *true* guard.)

The execution of a choice block is as follows: Wait until at least one guard in the choice block succeeds or all guards fail; in the former case execute any block with a guard that succeeds, and in the latter case, skip.

The execution of a choice block terminates if a guard succeeds and the block corresponding to the guard terminates, or if all guards in the block list fail.

4.4 Fair Composition — UNITY

This composition operator is the UNITY union operator [1].

As in the case of choice blocks, restrict attention to choice blocks of the form: $\{\square \prec guard \rightarrow block \succ\}$. The execution of a fair block is as follows.

While at least one guard succeeds or suspends: select a guard nondeterministically and fairly, and execute the corresponding block if the guard succeeds.

The block terminates if and only if all guards fail.

5 Quantification

A sequence of items can be defined using quantification as in UNITY.

A quantified-form has the syntax:

$\ll identifier$ in
integer-expression .. integer-expresion :: lexical-unit $\gg$.

The quantified-form, $\ll i$ in n .. $m :: exp \gg$ is a sequence in which the lexical unit exp appears once for each instance of i where $n \leq i \leq m$, and in the k-th appearance of exp, all instances of i in exp are replaced by k.

Example $\ll i$ in 0..2 :: $x[i] := y[i], u[i] := v[i] \gg$ is the sequence:
$x[0] := y[0], u[0] := v[0], x[1] := y[1], u[1] := v[1], x[2] := y[2], u[2] := v[2]$.

6 Examples

In this section a few programs are developed to illustrate the composition operators. Documentation within programs is in *slanted font.* We begin with a simple program *flip* that has two integer arguments, and *flip* interchanges the values of its arguments. The body of the program is a sequential composition block.

```
flip(u, v)
int u, v
int w
{;
   w := u,   u := v,   v := w
}
```

Declare parameters of program.
Declare local variables of program.
Begin sequential composition block.
Block-list
End sequential composition block and end program.

Next consider a program f with three arguments, an integer n, a one-dimensional array x indexed i where $0 \leq i < n$, and an index j where $0 < j < n$. The program flips $x[j-1]$ and $x[j]$ if they are not in ascending order.

```
f(n, j, x)
int n, j, x[n];
(x[j - 1] > x[j]) → flip(x[j - 1], x[j])
```

Single statement program.

Next we write programs, $s0$, $s1$, $s2$, each with two arguments: n and x where x is an array indexed $[0 .. n-1]$, and where the postcondition of each program is that x is in ascending order. For purposes of exposition assume that the elements of x are distinct.

6.1 Simple Sort

Fair Composition The simplest sorting routine is to repeatedly flip *any* pair of elements of x that are out of order until all pairs are in order.

$fc(n, x)$
int n,$x[n]$
$\{\square \ll i$ in $1 \,..\, n-1 :: (x[i-1] > x[i]) \rightarrow flip(x[i-1], x[i]) \gg$
$\}$

Choice Composition Fair selection is not necessary in the last example, and so we can use choice composition as shown next. This program flips any pair of elements of x that are out of order, and then calls itself.

$s0(n, x)$
int n,$x[n]$
$\{? \ll i$ in $1 \,..\, n-1 :: (x[i-1] > x[i]) \rightarrow \{; flip(x[i-1], x[i]), s0(n, x)\} \gg$
$\}$

Sequential Composition The bubble sort is defined in the obvious way using sequential composition.

$s1(n, x)$
int n, $x[n]$
$\{; \ll t$ in $1 \,..\, n-1$, i in $1 \,..\, n-t :: f(n, i, x) \gg \}$

Program as a Set of Equations Next, we give the specification of a sort program as a set of equations, and then we represent the equations in PCN, using definition variables.

Specification The specification employs arrays x and y, and states that y is a sort of x. The specification uses a local array z indexed t, i where $0 \leq t < n+2$. The specification of the program is the following set of equations:

$\forall i$ where $0 \leq i \leq n-1 :: z[0,i] = x[i]$
$\forall i$ where $0 \leq i \leq n-1 :: y[i] = z[n+1,i]$
$\forall i, t$ where $0 \leq i \leq n-1$, and where $0 \leq t \leq n ::$
$z[t+1,i] = \min(z[t,i], z[t,i+1])$ if $(i-t) \bmod 2 = 0$,
$\max(z[t,i], z[t,i-1])$ if $(i-t) \bmod 2 \neq 0$

Program The program is a syntactic transformation of the specification:

$s3(n, x, y)$
int n,$x[n]$,$y[n]$
definition $z[n+2][n]$
$\{\|$
$\ll i$ in $0 \,..\, n-1 :: z[0,i] := x[i]$,
$y[i] := z[n+1,i]$,
$\ll t$ in $0 \,..\, n :: z[t+1,i] :=$
$((i-t) \bmod 2 = 0 \rightarrow \min(z[t,i], z[t,i+1])$,
$(i-t) \bmod 2 \neq 0 \rightarrow \max(z[t,i], z[t,i-1]))$

$\gg\gg\}$

Parallel Composition The odd-even transposition sort is defined using sequential and parallel composition. On every odd step, for all odd i, $x[i-1]$ and $x[i]$ are flipped if they are out of order, and on even steps the same is done for even i. Thus, on step number t, program $f(n, i, x)$ is called for all i such that $(t-i) mod\ 2 = 0$.

$s2(n, x)$
int n,$x[n]$
$\{; \ll t$ in $0 \,..\, n ::$
$\{\| \ll i$ in $1 \,..\, n :: ((t-i) \bmod 2 = 0) \rightarrow f(n, i, x) \gg \}$
$\gg$
$\}$

7 Conclusion

This very brief paper gives something of an overview of PCN. A great deal has been omitted for lack of space. Details can be obtained from the authors.

Acknowledgement

This work was supported in part by JTFPMO grant coordinated by JPL and by DARPA, order numer 6202, monitored by ONR, N00014-87-K-0745.

References

[1] K.M Chandy, and J.Misra, *Parallel Program Design: A Foundation*, Addison-Wesley,1989

[2] I.Foster and S.Taylor, *Strand, New Concepts in Parallel Programming*, Prentice-Hall, 1989.

[3] C.A.R.Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1984.