CHP and CHPsim: A Language and Simulator for Fine-Grain Distributed Computation

Alain J. Martin & Christopher D. Moore
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125, USA

***Abstract*—This paper describes a complete and stable version of CHP and the simulator CHPsim. CHP is a language for fine-grain distributed computation with asynchronous VLSI as its main application. Several partial versions of the language are already widely used, but CHP has never been presented as a complete language. This presentation includes constructs like meta process, connect statement, value probe, peek, and templated type. CHPsim allows for mixed-level simulation of a system, with different components at various levels of implementation (CHP, HSE, PRS).**

INTRODUCTION

CHP (*Communicating Hardware Processes*) is a programming language for fine-grain distributed computations with asynchronous VLSI as its main application. A CHP program consists of a fixed set of concurrent processes communicating by messages.

CHP already has a long history. It took shape in the early 1980's to describe distributed computations in general. Starting from C.A.R. Hoare's CACM version of CSP[5], and Dijkstra's guarded commands[4], the early prototype of CHP introduced the notion of channels, and the probe construct, neither of which exists in CSP. The combination of processes with channels, send/receive communication, and probe operation on channel produced a very useful, albeit restricted, distributed programming model which was successfully used both for teaching distributed computing and for implementing distributed computations on experimental multiprocessors. But it was the development of a synthesis method for asynchronous VLSI that prompted the extension of the prototype notation into CHP[7].

Compared to other languages for distributed computations, CHP contains both restrictions and extensions. The restrictions are required by the limitations of hardware to finite data ranges, and by the impossibility to create physical resources during the execution of a computation in VLSI. Dynamic objects, like certain data types, inner blocks for variable scoping, general recursion, and the dynamic creation of processes are also excluded. The extensions are mainly concerned with communication, which is pervasive in VLSI and must be performed efficiently, and with the description of complex process structures. The resulting notation is vastly different from CSP and it would be a mistake to consider CHP as a syntactic variation on CSP.

CHP has gone through many modifications since it was introduced. It has been used to design several important projects in asynchronous VLSI at Caltech (among others the first asynchronous microprocessor in 1988[1] and the Caltech MiniMIPS in 1998[8]) and in other async labs around the world [13], [14]. It is also used to model other types of fine-grain distributed computation, in particular neuromorphic systems. See, e.g., [16], [17]. CHP is used in industry, not only in startups, but also in major companies like ST, FranceTelecom, and CEA/LETI.

It is the recent development of an "industrial-strength" CHP simulator, *CHPsim*, that forced us to stabilize the language definition. This paper describes what we consider a mature and stable version of CHP and the CHPsim simulator[3] attached to it. The simulator will be made available to the community through the GPL. The description of the semantics has been kept informal, but there already exist several definitions of the semantics of the main constructs of CHP in the literature,[15].

CHPsim is primarily designed for interactive debugging of a CHP system in particular with the use of *cosimulation,* i.e. the simulation of a system whose components are at different stages of the QDI synthesis: CHP, HSE, PRS. The main distinguishing feature of CHPsim is the ability to detect any violation of the QDI requirements of *stability and non-interference* via a complete randomization of the sequencing of parallel events. These violations are a fairly common type of bug, and they can be one of the most difficult to fix. However, CHPsim allows the designer to manually specify intentional timing assumptions, so it is not strictly limited to QDI design.

I. THE STRUCTURE OF A CHP PROGRAM

A CHP program usually consists of the parallel composition of several concurrent components called *processes.* Concurrent composition of processes is the main source of concurrency. The code of a process is mostly sequential—as we shall see, CHP allows a restricted (but important) form of concurrency inside a process. CHP processes do not share variables. They communicate exclusively by passing messages through communication channels. The set of processes and the communication channels connecting them form a graph with the processes as vertices and the channels as edges, called the *process graph*. A process is first declared in a declaration statement and then instantiated in an instantiation statement. In the usual modular ("structured") approach to system design, a system is constructed by composing processes in a hierarchical fashion. A process is either a *simple* or a *meta* process. A process that consists of the composition of a number of subprocesses is called a

meta process. Otherwise, it is a *simple* process. A simple process is identified by the label chp. A meta process is identified by the label $meta$.

II. Sequential CHP

CHP is a strongly typed imperative language: the state of a computation is changed by explicit assignments of values to variables. Sequential CHP contains a few unconventional constructs. (1) The control structures ("if-statement" and "do-loop") are those of Dijkstra's guarded commands, which introduce the notion of wait on a condition. Wait is equivalent to an abort in a sequential computation since nothing can modify the wait state, but it is extremely useful for concurrency. (2) Finite-integer arithmetic is not defined since it is implemented from scratch for each design; the choice made in the *CHPsim* simulator is used as default. (3) All objects are permanent once they are created, and therefore CHP has no notion of dynamic scoping, stack, pointers. No general recursion is allowed.

A. Variables, Data Types, and Assignment

All variables are local to a process. Inside a process, a variable may be local to a procedure or function. Variable declaration is superfluous for specifying the scope of the variable since its scope is always the whole process, or procedure or function, in which it appears. Declarations are used solely to specify variable types.

There are three *generic* variable types: Boolean (*bool*), integer (*int*), and symbol. The generic types integer and symbol can be restricted by the definition of *specific* types. An *integer-type* is a specific type defining a set of integer values between a lower bound and an upper bound. The two bounds define the *range* of the type. For example, the type declaration

$$\textbf{type}\ x = \{0..7\}$$

defines the *integer-type* x with values ranging from 0 to 7. A variable y can then be declared to be of type x. Variable y can be also declared directly as

$$\textbf{var}\ \ y = \{0..7\}$$

keeping the type anonymous. Similarly, the *symbol-type* $color$ can be defined as

$$\textbf{type}\ \ color = \{blue, white, red\}\ .$$

The assignment $x := expr$ assigns the value of the expression *expr* to variable x. The compiler checks that the generic type of *expr* is the same as the generic type of x. If x is defined as a specific type, the check that the value of the expression is within the range of x is done at run-time by the simulator.

For b Boolean, the command $b := true$, is also denoted $b\uparrow$; and the command $b := false$ is denoted $b\downarrow$.

III. Arrays and Records

The two mechanisms for structuring data are *array* and *record* . (Arrays are also used for processes and ports. Ports are defined in Section V.)

A. Array

CHP uses the usual array mechanism and notation with the following extension. In hardware, it is often useful to address one bit, or a range of bits (a "slice"), of the Boolean representation of an integer variable. Therefore, *an integer variable is always implicitely declared as a Boolean array as well.*

Given integer variable x, the *integer index* $x[7]$ represents the bit 7 of the binary representation of x. It is of type Boolean. Hence, the test for parity of x can be implemented as $x[0]$. The *integer slice* $x[0..3]$ is an integer variable of type $\{0..15\}$. When a slice of an integer variable is used repetitively in a program, its array range can be given a name by declaring an *integer field*. For instance, integer variable i in the CHP of the Caltech Asynchronous Microprocessor (CAM) contains the current instruction being executed. Several slices of i are recognized as standard fields of the instruction and are declared as integer fields. For an instruction of type "alu", the fields are

$$\begin{array}{lrcl}\textbf{field} & op & = & [12..\ 15];\\ \textbf{field} & x & = & [8\ ..\ 11];\\ \textbf{field} & y & = & [4\ ..\ 7];\\ \textbf{field} & z & = & [0\ ..\ 3];\end{array}$$

i.op contains the "opcode" of the instruction; $i.x$ and $i.y$ contain the indices of the registers to be used as parameters of the instruction, and $i.z$ contains the index of the register in which the result of the instruction execution is to be stored. They are integer variables. The array declaration is perhaps a little unusual: For example, a one-dimensional array A of $integer$ ranging from 0 to $n-1$ is declared as

$$A[0..n-1] : integer\ .$$

Hence, a *load* instruction is described as

$$\begin{array}{l}\textbf{type}\ \ word = \{0..2^{16}-1\};\\ \textbf{var}\ \ dmem[0..m-1] : word;\\ \textbf{var}\ \ reg[0..15] : word;\\ reg[i.z] := dmem[reg[i.x] + reg[i.y]]\ .\end{array}$$

B. Records

A variable declared as a *record* is a (finite and usually small) collection of variables (*record-fields*) each with its own type. For example, the integer variable i representing the currently executed instruction in the CAM could have been declared as a record of several types depending on the type of the instruction. For ALU instructions, the type would be:

$$\textbf{type}\ alu : \textbf{record}\{op, x, y, z : \{0..3\}\}\ .$$

C. Sequential Composition

Inside a process, arbitrary program parts can be composed sequentially by the usual semicolon operator. The semicolon is a composition operator. It is not a termination symbol.

IV. Selection and Repetition

The two control structures are the *selection* and the *repetition* of Dijkstra's guarded commands. The selection is a generalization of the usual if-statement. The repetition is a generalization of the usual do-loop. Both selection and repetition allow

for an arbitrary number of cases (*guarded commands*) like in a case-statement. They also allow for non-deterministic choice (arbitration).

It is in general difficult, and often impossible, to determine at "compile-time" which selections require arbitration. We therefore introduce two sets of control structures, a deterministic one and a non-deterministic one, and let the designer explicitly indicate where arbitration is needed.

A. Selection

The execution of the *deterministic selection* command

$$[G_1 \longrightarrow S_1 [] ... [] G_n \longrightarrow S_n] \ ,$$

where G_1 through G_n are Boolean expressions, S_1 through S_n are program parts, (G_i is called a "guard," and $G_i \rightarrow S_i$ is called a "guarded command") amounts to the execution of the arbitrary S_i for which G_i holds. At most one guard holds at any time. If none of the guards is true, the execution of the command is suspended until one guard is true. The traditional *if*-statement: $if\ B\ then\ S$ is expressed as

$$[B \longrightarrow S [] \neg B \longrightarrow skip] \ ,$$

(*skip* is the statement that does nothing but terminates). The traditional *if-then-else*-statement, $if\ B\ then\ S1\ else\ S2$, is written

$$[B \longrightarrow S1 [] \neg B \longrightarrow S2] \ .$$

The *non-deterministic selection* command

$$[G_1 \longrightarrow S_1 | \ \ ... | \ G_n \longrightarrow S_n]$$

is identical to the deterministic one, except that several guards may be found true during a guard evaluation. In such a case, an arbitrary true guard is selected, and the corresponding statement is executed. The execution is suspended if no guard is true.

B. Wait

The net-effect of the program $[B \rightarrow S]$ can be described operationally as "wait until B holds, then execute S." The concept of a wait as a computing primitive makes sense only in the context of concurrency. If B is in terms of local variables only, either B holds when it is evaluated or the wait for B to become true never terminates. The statement "wait until B holds" is used mostly when the value of B can be changed by another process, which is possible either with shared variables, which are not allowed in CHP, or with the "probe" construct which will be introduced shortly. The program statement $[B]$, where B is a Boolean expression, is a shorthand notation for $[B \rightarrow skip]$, and thus for "wait until B holds." Hence, "$[B];\ S$" and "$[B \rightarrow S]$" are equivalent.

C. Repetition

The execution of the *deterministic repetition* command

$$*[G_1 \longrightarrow S_1 [] ... [] G_n \longrightarrow S_n] \ ,$$

where G_1 through G_n are Boolean expressions, and S_1 through S_n are program parts, amounts to repeatedly selecting the arbitrary S_i for which G_i holds, and executing S_i. At any time, at most one guard is true. If none of the guards is true, the repetition terminates.

The traditional *while*-loop, $while\ B\ do\ S$, is written:

$$*[B \longrightarrow S] \ .$$

Non-deterministic repetition exists but is rarely used.

D. Infinite Repetition and Reactive Processes

Since many computations in CHP are modeled as non-terminating processes, we introduce a shorthand notation for the non-terminating repetition: $*[S]$ stands for $*[\mathbf{true} \rightarrow S]$ and, thus, for "repeat S forever." A *reactive repetition* is a construct of the form:

$$*[[G_1 \longrightarrow S_1 [] ... [] G_n \longrightarrow S_n]] \ ,$$

which can be described as "repeat forever: Wait until some G_i holds; execute the S_i for which G_i holds." This structure is used frequently in describing hardware components. Such a component is a non-terminating process waiting for some condition on its input ports to become true. When condition G_i becomes true, the component "reacts" by executing S_i, and then returns to waiting for the next external condition to be true. The reader must appreciate the difference between the reactive repetition above and the simple repetition:

$$*[G_1 \longrightarrow S_1 [] ... [] G_n \longrightarrow S_n] \ .$$

The simple repetition terminates when none of the G_i's is true. The reactive repetition never terminates.

E. The Replication Construct

VLSI algorithms are characterized by an extensive use of replication. A typical example is that some action has to be performed (sequentially or concurrently) on all the Booleans that represent an integer. Another example is that of an *n*-place buffer constructed as the concurrent composition of n identical one-place buffer. CHP therefore contains a syntactic operator, called the *replication construct,* which makes it possible to replicate any program part into a number of instances in order to simplify the coding of long lists of objects. We omit the further description of the replication construct.

F. Procedures and Functions

Procedures and functions have been kept as simple as possible and have the same restrictions as processes concerning the use of variables, so as to make the translation to processes straightforward.

Procedures and functions use only local variables: no global variables, no persistent variables, and no communication ports are allowed. The communication between a procedure or a function and its environment is entirely through the parameter mechanism.

Because of the above restrictions on the use of variables inside procedures and functions, a procedure (or a function) declaration is entirely "portable": the procedure (or function) can be declared anywhere and used anywhere. As a consequence, procedure names and function names must be unique inside the whole name space. But a procedure or a function call inside a

process is local to the process: if the same procedure is called inside two different processes, the two programs implementing the two calls can be viewed as operating on two disjoint sets of variables. The further description of procedures and functions is omitted.

V. Concurrent CHP

A. Parallel Composition of Processes

We postulate that the parallel composition of non-terminating processes is *weakly fair: If, at a certain point of the parallel execution of processes $p1$ and $p2$, x is the next action of $p1$ to be executed, then x will be executed after a finite number of actions of $p2$, if the computation is dadlock-free.*

The parallel bar never explicitely appears in the CHP code and is not part of the language syntax: processes created by an instantiation command are automatically composed in parallel.

B. Ports, Channels, and Connect Statement

Processes communicate with each other by communication commands on *ports*. A process can *send* on an output port or *receive* on an input port. A port of a process is paired with a port of another process to form a *channel*. A channel cannot be formed by two ports of the same process. A channel is shared by exactly two processes; but it is possible to generalize the definition to more than two processes sharing one channel. The ports of a simple process are all *external*: Each port is to be connected to a port of another process to form a channel. A meta process, say p, is the parallel composition of several subprocesses. The subprocesses are connected with each other by channels. For example, a channel between port X of subprocess $p1$ and port Y of subprocess $p2$ is created by the declaration

$$\textbf{connect}\ p1.X,\ p2.Y\ .$$

Ports X and Y are internal to process p. The ports that are not internal to a meta process are, of course, external. The external ports of a (simple or meta) process are declared in the heading of the process declaration. A typical declaration heading is:

$$\textbf{process}\ p()(L?:int; R!:bool; S)\ .$$

The first pair of parentheses following the name p of the process declares a list of meta parameters. (In this example, the list is empty meaning that the process does not use meta parameters.) Meta parameters are assigned a value when the process is instantiated, and remain constant thereafter. For example, a buffer process may be declared with an integer meta parameter n that will define the size of the buffer (the number of buffer stages) when it is instantiated.

The second pair of parentheses encloses the list of external ports of process p. In this example, the ports have names L, R, and S. The question mark following L identifies it as an input port (receive port). The exclamation point following R identifies it as an output port (send port). The type declarations indicate that the messages received on L are of type integer, and the messages sent on R are of type Boolean. Port S is not identified as input or output: it is a synchronization port. No data is transferred on S, and therefore no type is declared.

A group of ports of a process can be structured as a *port array or a port record*. In an example, a process *ring* is declared as having two port arrays U and D, as follows:

$$\textbf{process}\ \ ring(n)(U[1..n]?, D[1..n]! : type)$$

C. Meta-code and chp-code

There is a distinction between the "meta-code" and the "chp-code". A meta-process body may contain code used to expand the process graph. This meta-code is not subjected to the same restriction as the regular chp-code that is the body of simple processes. The code of a meta process is executed during the pre-processing phase and disappears after instantiation. In particular, it may contain recursion as we shall see in an example. In practice, the combination of connect, connect-all, port array, repetition and tail recursion is enough for the meta code to generate most regular graphs of interest.

D. Instantiation

A process declaration, say p, is always a declaration of a **process type** p. The instantiation statement **instance** creates a process of a given type. For instance, after process p has been declared, two instances of type p are created by the command:

$$\textbf{instance}\ p1, p2 : p$$

The instantiation creates processes $p1$ and $p2$ and implicitely composes them in parallel. The execution of a CHP program can be understood as the following sequence of steps:

1) expanding the process graph by pre-processing all meta code, i.e. instantiating meta processes (with actual values of meta parameters, if any) and simple processes, and establishing port connections and channels,
2) expanding all remaining replications and macros inside simple processes,
3) starting all simple processes. (The order in which the processes are started is irrelevant.)

VI. Examples

Any process graph can be generated by just enumerating all the processes by a sequence of **instance** commands, and all the channels by a sequence of **connect** commands. But, for anything but very simple process graphs, the designer may want to exploit the structure of the graph, if any, and generate the process graph by using repetition and recursion constructs, which are allowed in the meta body. Here are some standard examples. (The unspecified port type $type$ is used. A direction for the ports has been chosen arbitrarily.)

A. A Two-Process Example

Process *main* is created as the concatenation of two processes: $p1$ of type *proc1* and $p2$ of type *proc2*, (see Figure 1):

$$\begin{array}{l}
\textbf{process}\ \ proc1()(L?, R! : type)\\
chp\{\ldots\}\\
\textbf{process}\ \ proc2()(L?, R! : type)\\
chp\{\ldots\}\\
\textbf{process}\ \ main()(W?, E! : type)\\
meta\{\textbf{instance}\ \ p1 : proc1;\ \ p2 : proc2;\\
\qquad \textbf{connect}\ p1.R, p2.L;\ \ W, p1.L;\ \ E, p2.R\\
\quad \}
\end{array}$$

The connect statements are used in two different ways. The statement $\mathbf{connect}\ p1.R,\ p2.L$ creates a channel by connecting the output port R of $p1$ to the input port L of $p2$. In the two other cases, the connect statement is used to identify a port of a subprocess with an external port of the meta process. No channel is created. A connect statement used to create a channel requires that the two ports be either both synchronization ports, or one output port and one input port with the same data type. A connect statement used to create an external port requires that the two connected ports be of the same type: both synchronization ports or same port directions and same data types.

1) Notation: The ports of a subprocess of a metaprocess are identified in the connect statements of the metaprocess by prefixing their name with the name of the subprocess: in the above example, $p1.R$, $p2.R$, and $p2.L$. The ports of the metaprocess are directly identified by their local name, in the above example W and E, without the need to prefix them with the process name.

Fig. 1. *Process main as the composition of processes $p1$ and $p2$. Port R of $p1$ is connected to port L of $p2$ to form an internal channel of main. Port L of $p1$ becomes port W of main; port R of $p2$ becomes port E of main.*

B. A Chain of Processes

As a generalization of the previous example, n simple processes of previously-defined type $proc1$, are composed in a linear chain to create the meta-process $chain$.

$$
\begin{array}{l}
\mathbf{process}\ \ chain(n)(W?, E! : type) \\
meta\{\mathbf{instance}\ p[1..n] : proc1; \\
\qquad \mathbf{connect}\ \ \mathbf{all}\ i : 1..n-1 :\ \ p[\,i\,].R, p[i+1].L; \\
\qquad \mathbf{connect}\ \ W, p[\,1\,].L;\ \ \ E, p[\,n\,].R \\
\qquad \}
\end{array}
$$

The simple processes used to construct the chain are declared as an array $p[1..n]$ of processes of type $proc1$. The *connect all* statement connects a list of port pairs, here to create a list of internal channels for each value of the running index i, in the declared range, from 1 to $n-1$. The following connect-statement connects external ports W and E with ports $p\,[1]\,.L$ and $p\,[n]\,.R$ of subprocesses $p\,[1]$ and $p\,[n]$.

C. A Ring of Processes

A ring of processes is composed of n elementary processes of type $proc3$. A process of type $proc3$ has four ports L, R, U, D; the ports L and R are connected in a ring, and the ports U and D are left as external ports of the meta-process $ring$. Notice the use of both port array and process array. See Figure 2.

$$
\begin{array}{l}
\mathbf{process}\ \ proc3()(L?, R!, U?, D! : type) \\
chp\{\ldots\} \\
\mathbf{process}\ \ ring(n)(U[1..n]?, D[1..n]! : type) \\
meta\{\mathbf{instance}\ p[1..n] : proc3; \\
\qquad \mathbf{connect}\ \ \mathbf{all}\ i : 1..n : p[\,i\,].R, p[(i+1)\mathbf{mod}n].L; \\
\qquad\qquad\qquad\qquad\qquad\quad p[\,i\,].U, U[\,i\,]; \\
\qquad\qquad\qquad\qquad\qquad\quad p[\,i\,].D, D[\,i\,] \\
\qquad \}
\end{array}
$$

Fig. 2. *A ring of processes: Ports L and R of each process are used to connect the processes in a ring. Ports U and D are left unconnected: Port U of process p[i] becomes port U[i] in the port array U[1..n] of the meta process 'ring', and similarly for D.*

VII. Concurrent CHP: Communication and Slack

In CHP, a communication action on a port is identified by the name of the port. If port X is a synchronization port (no data), a communication on X is identified in the program of the process by the mere name X. The name of a port identifies both the port in the process declarations and the communication actions on that port in the program text.

Matching communication actions are mainly used to implement a form of distributed assignment statement, to "pass messages." In that case, the pair of commands is specified to consist of an input command (receive) and an output command (send) by adjoining to them the symbols "?" and "!", respectively: $X!$ denotes an output command (or send) on output port X, and $Y?$ denotes an input command (or receive) on input port Y.

Matching communication actions can be used for synchronization only, in which case the two commands are identified by the name of the ports only, without input or output symbol.

A. Slack

If output port X of $p1$ has been connected with input port Y of $p2$ to form a channel, then executions of action X in $p1$ are synchronized with executions of action Y in $p2$. For a command A, let $\mathbf{c}A$ denote the number of completed actions A at some point of the computation. The weakest form of synchronization between the send actions X and the receive actions Y is that, at any moment, the number $\mathbf{c}Y$ of completed receive actions is at most equal to the number $\mathbf{c}X$ of completed send actions:

$$\mathbf{c}Y \leq \mathbf{c}X$$

The difference $\mathbf{c}X - \mathbf{c}Y$ is the number of messages sent that have not yet been received. The maximal value of $\mathbf{c}X - \mathbf{c}Y$ is called the *slack* of the channel. The slack represents the amount of buffering available in the channel.

(The notion of slack was introduced in [6].)

Allowing message buffering in the channels requires that channels be implemented as storage devices. In view of our intention to use communication as an elementary sequencing and synchronization mechanism, we opt for as simple an implementation of channels as possible. The simplest implementation is one in which no buffering of messages is required: slack zero.

For slack zero primitives, at any time, the number of completed X-actions in $p1$ is equal to the number of completed Y-actions in $p2$; the completion of the n-th X-action "coincides" with the completion of the n-th Y-action.

If, for example, $p1$ reaches the n-th X-action before $p2$ reaches the n-th Y-action, the completion of X is suspended until $p2$ reaches Y. The X-action is then said to be *pending*. When, thereafter, $p2$ reaches Y, both X and Y are completed. The predicate "X is pending" is denoted as $\mathbf{q}X$. A pair (X, Y) of slack-zero communication commands satisfies the synchronization properties:

$$\mathbf{c}X = \mathbf{c}Y$$
$$\neg\mathbf{q}X \vee \neg\mathbf{q}Y$$

VIII. Slack Elasticity

Many properties of a CHP system will be proved true under the assumption that all communications have slack zero. However, CHP processes to be implemented in VLSI are, except in the most simple cases, subjected to decomposition transformations, like pipelining, that usually add slack to the channels. The slack may also be increased to improve the performance (throughput) of the system through the procedure called *slack matching* and also to avoid deadlock.

The designer wants to be able to perform those transformations and be assured that the system still fulfils its specification after the slack has been increased.

Slack Elasticity. A CHP system is said to be slack-elastic when the slack of any channel can be increased by an arbitrary finite amount without changing the correctness of the design.

- A deterministic system with no probed selection is slack-elastic.
- A deterministic system with probed selection is not guaranteed to be slack-elastic as adding buffer on a probed channel may change the result of the selection, and thus the behavior of the system.
- A non-deterministic system is slack-elastic if adding slack does not increase non-determinism: Any non-deterministic choice ("a decision") that is made after the slack has been increased could have been made before increasing the slack.[11]

The requirement of slack elasticity has a profound influence on the CHP design style. In particular, it limits the use of the probe and complicates the implementation of cosimulation in CHPsim.

IX. The Buffer Process

The term "buffer" (or "one-place buffer", or "L/R-buffer") is used to describe a process that, in its simplest form, alternatingly receives a message of a certain type, say $type$, on its input port L, and sends the message received on its output port R. It is described by the following CHP code:

$$\begin{array}{l}\textbf{process}\ \ buf1()(L?:type; R!:type)\\ chp\{\textbf{var}\ \ x:type;\\ \quad *[L?x;\ \ R!x]\\ \}\ .\end{array}$$

The notion of slack extends from channel to process. A one-place buffer is said to have slack one since $0 \leq \mathbf{c}L - \mathbf{c}R \leq 1$. When the process is in the state identified by the semicolon between L and R, it is storing one token of data. A one-place buffer added to a channel increases the slack of the channel by one. The above solution requires an internal variable x of the same type as the messages in order to buffer the last message received and not yet sent. Such a buffering can be avoided. The messag ereceived on L can be sent directly on R by writing the body of the buffer as

$$*[R!(L?)]\ .$$

The construct $R!(L?)$ is called the *pass*. However, there is an important difference between the two solutions. In the *pass* solution, the slack between L and R is zero. Reducing the slack by the introduction of a *pass* can lead to deadlock.

X. Internal Concurrency: The Comma

So far, concurrency has only been possible between parallel programs (processes) that do not share variables, hence eschewing all conflicts related to concurrent reads and writes of variables. However, we cannot avoid introducing a restricted form of internal concurrency between program parts inside a process, hence opening the Pandora's box of variable sharing. There are two important reasons: efficiency and deadlock avoidance. We will clarify the two motivations with an example. Internal concurrency is denoted by the comma. Inside a process,

$$S1, S2$$

denotes the concurrent execution of $S1$ and $S2$. *Internal concurrency is defined only between non-interfering programs.*

Non-interference between two programs $S1$ and $S2$ can be enforced in two ways. (1) *Local non-interference*: any variable x modified (written) in $S1$ is not used in $S2$ (neither written nor read). (2) *Global non-interference*: variable x is written in $S1$ and used (written or read) in $S2$, but the accesses to x in $S1$ and $S2$ are in sequence. The sequencing of the accesses to x is enforced through their ordering with communication actions and rely on some proper ordering of those communications in the environment.

The concurrent execution of two communications on the same port of a process is excluded.

Local non-interference is more restrictive to the designer, but also easier to verify. It suffices for the vast majority of the designs.

As an example of locally non-interfering parallelism, consider a process that repeatedly receives two integer values x and y on ports X and Y, and sends the smallest of the two values $min(x, y)$ on port m, and the largest of the two values $max(x, y)$ on port M. In absence of the comma, the two receive commands $X?x$ and $Y?y$ on the one hand, and the two

send commands $m!min(x, y)$ and $M!max(x, y)$ on the other hand, have to be sequenced in some order. With the comma, the program becomes

$$\begin{array}{l}\textbf{process}\ \ minmax()(X?, Y? : type; m!, M! : type)\\ chp\{\textbf{var}\ \ x, y : type;\\ \quad *[X?x, Y?y;\ \ m!min(x, y), M!max(x, y)]\\ \}\ .\end{array}$$

(The comma binds more tightly than the semicolon.) The advantages are twofold. First, efficiency: ordering all four communications would be quite expensive. Second, deadlock avoidance: without the comma, one arbitrary sequencing has to be chosen for the two receives and one for the two sends. Suppose we choose X before Y. It may happen that the environment of $minmax$ imposes an order on the two matching send actions, say X' after Y', and this order is not known to the designer of $minmax$. Then, our choice of ordering for X and Y leads to deadlock. The comma makes it possible to adapt the ordering of communications to the unkwnon choice made by the environment.

An example of global non-interference will be described after we introduce the needed constructs of CHP.

In practice, determining whether the parallelism in an arbitrary program is non-interfering is intractable. As such, the default behavior of CHPsim is to unconditionally flag locally interfering parallelism as an error. Variables that are intentionally involved in local interference must be flagged so that CHPsim can fall back to checking for interference via random sequencing.

XI. The Probe: Port Selection

In order to satisfy the semantics of communication, a process, say $p1$, reaching a communication action X must be able to "decide" whether it may proceed with the execution of the communication or suspend itself until the matching communication action Y in process $p2$ has been reached. In other words, process $p1$ must be able to detect whether action Y is pending in process $p2$. Hence, a process being suspended at a communication action must be an observable state: a process reaching a communication action carries out a state transition. In CHP, this state is exposed to, and used by, the designer: it is possible to test in $p1$ whether communication action Y is pending in $p2$ by means of a Boolean command on ports, called the *probe*[9]. (It turns out that such a test is easy to implement in hardware.)

The definition of the probe states that the probe command $\overline{X}$ in process $p1$ has the same value as $\mathbf{q}Y$, and, symmetrically, the probe command $\overline{Y}$ in process $p2$ has the same value as $\mathbf{q}X$. In other words, the probe $\overline{X}$ evaluating to true in $p1$ means that a communication on port Y is pending in $p2$.

Such a mechanism is particularly useful when a process engaged in message exchanges with its environment may need to detect on which port a communication with the environment is to take place next. As we shall see, for a large class of deterministic computations, this need can be answered by the use of control messages on control channels. But this solution does not apply to non-deterministic interactions (like external interrupts for a microcontroller).

Any communication command can be probed (input or output). And both sides of a channel can be probed. However probing both sides of a channel in the same communication leads to deadlock. In practice, many designers restrict the use of the probe to the input ports.

A. Example: Single-variable Register

The register-process controls a single variable x, for instance a Boolean, that can be written (assigned a new value) through input port X, or read through output port Y. The environment has the initiative of reading or writing x. The register is a reactive process. It detects whether the next action of the environment is a read or a write by probing the ports X and Y:

$$\begin{array}{l}\textbf{process}\ \ reg()(X?, Y! : bool)\\ chp\{\textbf{var}\ \ x : bool;\\ \quad *[[\overline{X} \longrightarrow X?x[]\overline{Y} \longrightarrow Y!x]]\\ \}.\end{array}$$

This example shows that both input ports and output ports can be probed. A deterministic selection has been used, under the assumption that the environment never attempts to perform a read and a write at the same time. If this assumption cannot be made, then the non-deterministic selection has to be used instead: in the above program, the "thick bar" [] has to be replaced with the "thin bar" |.

B. Arbitration

The probe is particularly useful in conjunction with arbitration, i.e., when a non-deterministic choice has to be made between several conditions. In a system like a microprocessor, non-deterministic choice occurs in two different contexts. First, true non-determinism is caused by external signals like interrupts and exceptions. Second, non-determinism may be introduced for efficiency reasons. In the MiniMIPS, the data- and instruction caches share one single external interface (pins), and therefore the acccess to the interface is arbitrated.

C. Probe and Stable Wait

The probe construct introduces a limited form of shared variable: the process reaching a pending communication sets the probe to true, while the process at the other end of the channel reads the value of the probe. Does that mean that the designer now has to worry about the issues related to shared variables that communicating processes were meant to avoid? Fortunately no, thanks to the notion of *stable wait*.

Negated probes are used only to enforce fairness, which is seldom necessary. The most common use of the probe is in guards where the probe is not negated. Since all other program variables appearing in a guard have well-defined constant values at the point where the guard is evaluated, the guard can be reduced to a Boolean expression in terms of probes only. Such an expression can, for instance, be put in a canonical disjunctive normal form in which no probe is negated. Let $B(x_1, \ldots, x_n)$ be such an expression where $x_1, \ldots, x_n$ are the probes used in B. Let X be the vector $x_1, \ldots, x_n$. We introduce the ordering:

- $false \prec true$,

- For two vector values, U and V, of X

$$U \preceq V \Leftrightarrow (\forall i : 1..n : U.xi \preceq V.xi) \ .$$

Then it is easy to prove the *Guard Monotonicity Property*:

$$(U \preceq V) \Rightarrow (B(U) \preceq B(V)) \ .$$

Now assume that process p evaluates a guard B with probes while the environment changes the values of the probes. The environment can only change the values of the probes from *false* to *true* while p evaluates B, since a probe is reset to *false* only after the firing of the communication, i.e. after evaluation of the guard.

Therefore, thanks to the monotonicity property, once B evaluates to *true*, it remains *true* even while some of the probes are still changing: We say that the *wait* $[B]$ *is stable.* Stability of the wait is an essential property for reasoning about the probe in CHP, and for implementing concurrent computation in hardware without appealing to timing assumptions.

XII. Value Probe and Peek

A. Value Probe

The *value probe* makes it possible to evaluate a predicate on an input message while the communication is pending. Given the input port L,

$$\overline{L : B(L)} \stackrel{\text{def}}{=} \ \overline{L} \wedge B(L.val)$$

where $L.val$ is the data value "pending on L." More precisely, if $\overline{L}$ holds for the n-th L-action and $R!x$ is the matching n-th R-action, then $L.val = x$.

Observe that $\overline{L} : \neg\overline{L}$ means $\overline{L} \wedge (L.val = false)$, *i.e.*, "communication L is pending and the value of L is the Boolean $false$," which is different from the negated probe $\neg\overline{L}$ meaning "communication L is not pending." The value probe cannot be used with negated probe, since the value probe is defined only when the probe is true.

For example, given the buffer $*[L?x; R!(parity(x))]$, where $parity$ returns the value $true$ if x is even and $false$ if x is odd, suppose we want to recode it without having to store the values of x internally. With the value probe, we write:

$$\begin{array}{l} *[[\overline{L : even(L)} \longrightarrow R!true, L? \\ \ \ [] \overline{L : odd(L)} \longrightarrow R!false, L? \\ \ ]] \ . \end{array}$$

The selection in the previous example is between two values on the same ports. Therefore increasing the slack on that channel will not change the behavior of the computation: the channel is slack-elastic.

B. The Nyström Peek

The *peek* is an addition to CHP proposed by Mika Nyström[12]. Since the value-probe construct makes it possible to read the current value of a message on a pending input communication, say X, we can also assign this current value to an internal variable. Peeking the value of input port X does not complete the communication between X and Y: Y is still pending after the peek. In terms of Hoare triples, the peek of port X of channel (X, Y), denoted $X¿$ can is defined as:

$$\{\overline{Y}\}X¿x\{\overline{Y} \wedge (x = X.val)\}$$

i.e., if Y is pending before the peek, then, after the peek, Y is still pending and x has been assigned the value pending on port X. The peek can be defined in terms of the value probe as follows. Let $\{x1, x2, \ldots, xn\}$ be the range of values of x. The peek $X¿x$ is equivalent to the value-probe program:

$$\begin{array}{l} [\overline{X : X = x1} \longrightarrow x := x1 \\ []\overline{X : X = x2} \longrightarrow x := x2 \\ \qquad \vdots \\ []\overline{X : X = xn} \longrightarrow x := xn \\ ] \ . \end{array}$$

The peek simplifies the implementation of inner loops as shown in the next example. A process receives a message on input port L and repeatedly sends it on output port R as long as the Boolean c received on control port C is true. The first solution is as follows:

$$*[[L?x; c\uparrow; *[c \longrightarrow R!x; C?c]]$$

But inner loops are notoriously annoying to implement in hardware. A solution without inner loop is possible with the peek:

$$\begin{array}{lll} *[L¿; R!x; & [\overline{C : C = true} \longrightarrow C? & \\ & []\overline{C : C = false} \longrightarrow L?, C? & \\ ] &] & \end{array}$$

The commands $L¿; R!x$ send on R the value pending on L, without completing L. As long as c is true, the same value of L is sent on R. When a false value of c is received, L is completed, and thus the next value pending on L is sent on R.

The two solutions are not strictly equivalent. In the first solution, the communication on L completes immediately since the value received is kept in x. In the second solution, the communication on L cannot complete immediately since it is the value kept on L that is sent repeatedly on R until c false is received. Only then does L terminate.

C. An Example of Global Non-Interference

As an example of global non-interference between two internally parallel sections of a process, consider the following CHP of a two-port register file:

$$\begin{array}{l} \textbf{process} \ \ Reg()(Rd!, Wr? : word; Ri?, Wi? : \{1..N\}) \\ chp\{\textbf{var} ri, wi : \{1..N\}; \textbf{var} reg[1..N] : word; \\ \ \ *[Wi¿wi; Wr?reg[wi]; Wi], \\ \ \ *[Ri¿ri; Rd!reg[ri]; Ri] \\ \} \end{array}$$

This process itself does not force any ordering between the two statements $Wr?reg[w_i]$ and $Rd!reg[r_i]$ that access the elements of array reg. So it is up to the environment to ensure global non-interference. This can be done by forcing strict sequencing on Rd and Wr, or by ensuring that concurrent use of these ports take place only for different array indices (different elements of an array are considered different variables). If actions $Wr?reg[w_i]$ and $Rd!reg[r_i]$ operate on the same

array elements ($ri = wi$), then the environment must enforce sequencing between Wi and Ri.

XIII. Templated Type

Components like buffer, split, merge, stack are used over and over again in various environments. The designer could save considerable labor if the same code for a given type of component could be reused in as many different contexts as possible. One mechanism is already available that is helpful in that respect: the meta parameter.

For example, buffers or stacks of any size can be described by one piece of code as long as they differ only by their size, here referring to the maximal number of data items they can hold. Using the buffer description of 5.4.1, an instance of size 16 can be declared as

$$\textbf{instance}\ \ b16 : buf(16)\ .$$

However, all instances of a buffer must contain data of the same type as the one specified in the buffer declaration. Being able to change the type of the data handled by a component while keeping its code unchanged significantly enlarges the class of objects that can be specified by the same code.

The *templated type* is introduced for that purpose. During a proces declaration, the unknown data type is declared in the meta parameter section as a *type*. For instance, in the case of the buffer we will write:

$$\textbf{process}\ \ buf1(T : type)(L?, R! : T)$$
$$chp\{\textbf{var}\ \ x : T; *[L?x; R!x]\}\ .$$

The desired type is specified for each instantiation of a process of type $buf1$; for instance, a buffer passing integers will be instantiated as:

$$\textbf{instance}\ \ b : buf1(< int >)\ .$$

Several different templated types can be use in the same declaration for different variables or ports of the process:

$$\textbf{process}\ \ proc(T1 : type, T2 : type)(L? : T1; R! : T2)\ .$$

XIV. CHP in Examples

A. Merge, Split, and Toggle

Buffer, merge and split processes were by far the most common components in the design of the MiniMIPS.

1) Merge: The merge is a process with two input ports X and Y, and an output port Z. The process outputs on port Z a stream of messages which is an arbitrary merge of the stream of messages received on X and the stream of messages received on Y. The streams received on X and Y can each be empty, finite, or infinite. Because of the possibility that no message might be received on an input port in a current state of the system, an input port has to be probed before each input communication on the port in order to avoid deadlock. In the absence of any information on the environment, we have to assume that both probes may be true at the same time if there are pending communications on both input ports at the same time. We therefore have to use the arbitrated version of the selection statement. The solution is the *arbitrated merge*

$$\textbf{process}\ \ amerge()(X?, Y?, Z! : type)$$
$$chp\{\textbf{var}\ \ u : type;$$
$$\quad *[[\overline{X} \longrightarrow X?u; Z!u\,|\,\overline{Y} \longrightarrow Y?u; Z!u]]$$
$$\}\ .$$

The process has the typical "reactive process" structure defined earlier. Even though such a process is non-deterministic, it is slack-elastic: adding slack to one of its input channels may change the way in which the input streams are merged but since the merge is non-deterministic, it does not matter.

In many cases, it is possible to guarantee that the environment never sends a message on both inputs of the merge at once. The merge receives one message on one input port (never from both at the same time) and sends the received message on its output port. This *probed merge* can be described as follows:

$$\textbf{process}\ \ pmerge()(X?, Y?, Z! : type)$$
$$chp\{\textbf{var}\ \ u : type;$$
$$\quad *[[\overline{X} \longrightarrow X?u; Z!u[]\overline{Y} \longrightarrow Y?u; Z!u]]$$
$$\}\ .$$

This merge is no longer slack-elastic. Adding slack to an input channel may change the order in which the input ports are selected, or may violate the mutual exclusion between the input ports. A standard way of making the merge slack-elastic is by introducing an additional control channel that specifies from which input channel (X or Z) to expect the next item. This solution, which eliminates the probed selection, is called *controlled merge*:

$$\textbf{process}\ \ cmerge()(X?, Y?, Z! : type; C? : bool)$$
$$chp\{\textbf{var}\ \ u : type;\ \ c : bool;$$
$$\quad *[C?c; [\ c \longrightarrow X?u; Z!u[]\neg c \longrightarrow Y?u; Z!u]]\}\ .$$

2) Split: The split process has one input port X and two output ports Y and Z. The messages received on X are sent on Y or Z. Which output is chosen is decided by the environment. The first technique for the choice of output is the probe. We get the *probed split*

$$\textbf{process}\ \ psplit()(X?, Y!, Z! : type)$$
$$chp\{\textbf{var}\ \ u : type;$$
$$\quad *[[\overline{Y} \longrightarrow X?u; Y!u[]\overline{Z} \longrightarrow X?u; Z!u]]$$
$$\}\ .$$

The receive action $X?u$ can also be placed before the probed selection. Similarly to the merge, the split can also be controlled by a control port C. This *controlled split* is as follows:

$$\textbf{process}\ \ csplit()(X?, Y!, Z! : type; C? : bool)$$
$$chp\{\textbf{var}\ \ u : type;\ \ c : bool;$$
$$\quad *[C?c, X?u; [\ c \longrightarrow Y!u[]\neg c \longrightarrow Z!u]]$$
$$\}\ .$$

In all above solutions for the merge and split, the receive-send sequences, like $X?u; Y!u$, can be replaced by the pass construct, in this case $Y!(X?)$, with the associated loss of slack. Merge and split are the main building blocks for buses.

3) Toggle: An input toggle $toggle\_in$ is a restricted form of a merge. It has two input ports X and Y, and an output port Z. The process outputs on port Z a stream of messages which is a merge of the stream of messages received on X and the stream of messages received on Y. But now the messages are input

alternatingly from X and Y starting with X. The process can be coded as

process *toggle_in*()($X?$, $Y?$, $Z!$: *type*)
chp{**var** u : *type*;
$*[X?u;\ Z!u;\ Y?u;\ Z!u]$
} .

An output toggle *toggle_out* is a restricted form of a split. It has one input port X, and two output ports Y and Z. The messages received on X are sent alternatingly on Y or Z. The process can be coded as

process *toggle_out*()($X?$, $Y!$, $Z!$: *type*)
chp{**var** u : *type*;
$*[X?u;\ Y!u;\ X?u;\ Z!u]$
} .

Both toggle processes can be fit into a buffer template by introducing a Boolean variable c, initially $true$. We describe the output toggle only. We get

$*[X?x; [\ c \longrightarrow Y!u, c\downarrow [] \neg c \longrightarrow Z!u, c\uparrow]]$.

Alternatively, we can introduce the control process $*[U;\ V]$ to implement the two phases of the toggle:

$*[X?u; [\overline{U} \longrightarrow Y!u, U [] \overline{V} \longrightarrow Z!u, V]]$.

(U and V are synchronization channels.)

B. Buffers

1) Linear Buffer of Size n : An n-place buffer can store up to n messages. Such a buffer can be constructed as the linear composition of n one-place buffers. For $n > 1$, it can be described as the recursive composition of a one-place buffer and a buffer of size $n-1$.

process *buf*(n)($L?$, $R!$: *type*)
meta{**instance** p : *buf*1;
$[n = 1 \longrightarrow$ **connect** $\{p.R,\ buf(n).R;$
$p.L,\ buf(n).L\}$
$[] n > 1 \longrightarrow$ **instance** q : *buf*$(n-1)$;
connect$\{p.R, q.L;\ p.L, L;\ q.R, R\}$
$]$
}

2) Circular Buffer: Alternatively, a buffer of size n can be constructed as a single process using an array $b[0..n-1]$ of data items of the same type as the messages. The array is used in a circular way. Pointer i points to the next free position in the array, and pointer j points to the next (the oldest) occupied position in the array. Both pointers point to the same array position when the array is full or empty.

process *cirbuf*(n)($L?$, $R!$: *type*)
chp
{**var** $i, j, k : \{0..n-1\};\ b[0..n-1]$: *type*;
$i := 0, j := 0, k := 0;$
$*[[\overline{L} \wedge k < n \longrightarrow L?b[i]; k := k+1; i := (i+1) \mathbf{mod} n$
$[] \overline{R} \wedge k > 0 \longrightarrow R!b[j]; k := k-1; j := (j+1) \mathbf{mod} n$
$]]$}

If the buffer is initially empty, i and j should be initialized to the same (arbitrary) value in the range 0 to $n-1$, and k is initially 0. For simplicity, we have initialized all three to 0.

3) Tree Buffer: The latency of a data item traveling through a linear n-place buffer may become prohibitive when n is large. The circular buffer does not suffer from this problem, but the cost of incrementing the two pointers and of the addressing mechanism may offset the latency advantage. A better solution for n large is the "tree buffer." Assume n is a power of two. The buffer is partitioned into two buffers of size $n/2$, say $B0$ and $B1$. Data items are added alternatingly to $B0$ and $B1$, for instance $B0, B1, B0, B1, ...$, and they are removed alternatingly from $B0$ and $B1$ in the same alternation: $B0, B1, B0, B1, ...$ Hence, the FIFO property of the buffer is maintained, assuming that $B0$ and $B1$ are themselves FIFO buffers.

The alternation on the input side of $B0$ and $B1$ is realized by an output toggle whose first output port is connected to the input port of $B0$ and whose second output port is connected to the input port of $B1$. The alternation on the output side of $B0$ and $B1$ is realized by an input toggle whose first input port is connected to the output port of $B0$ and whose second input port is connected to the output port of $B1$.

process *treebuf*(n, k)($L?$, $R!$: *type*)
meta{**instance** $t1$: *toggle_out*; $t2$: *toggle_in*;
$[k = 2 \longrightarrow$ **instance** $b0, b1$: *buf*$(n/2)$
$[] k > 2 \longrightarrow$ **instance** $b0, b1$: *treebuf*$(n/2, k/2)$
$]$;
connect$\{t1.Y, b0.L;\ \ t1.Z, b1.L;$
$t2.X, b0.R;\ \ t2.Y, b1.R;$
$t1.X, L;\ \ t2.Z, R\}$
}

C. Bit Router in Mesh Structures

The goal is to construct the node of a routing network which communicates data between the different components of a distributed system, e.g., a network on chip. The data items are partitioned into a string of Boolean messages that together constitutes a *packet*. A packet is composed of a header containing the destination address of the packet, a number of messages containing the data, and a trailing token indicating the end of the message string. A router can receive a packet from one of two network inputs and can send a packet to one of two network outputs. A router can also receive a packet from its compute port in, and send a packet to its compute port out.

The router is built out of two components: a *switch* and a *selector*. A two-dimensional mesh router needs two switches and two selectors. The processor injecting a message into the network prefixes the message with a string of bits (the header) specifying the path the message will travel through the network; and it appends a trailing token to mark the end of the packet. The selector has two input ports and one output port and transmits an entire packet from the selected input port to the output port. Packets from different input ports are not interleaved: Once the selector has chosen an input port, it keeps sending the message from this port until it reaches the end-of-packet token. The switch has one input port and two output ports. It consumes the first bit of the packet received on its input port, and based on whether the bit is zero or one, passes the remainder of the packet out through either of its two output ports until the token is transmitted.

$$
\begin{array}{l}
\textbf{process}\ \ selector()(A?, B?, C! : \{0,1,F\}) \\
chp\{\textbf{var}\ \ x : \{0,1,F\}; \\
\quad *[[\overline{A} \longrightarrow A?x; C!x; *[x \neq\ \ F \longrightarrow A?x; C!x] \\
\quad\ \ [\overline{B} \longrightarrow B?x; C!x; *[x \neq\ \ F \longrightarrow B?x; C!x] \\
\quad\ ]] \\
\}\ .
\end{array}
$$

$$
\begin{array}{l}
\textbf{process}\ \ switch()(X?, S!, T! : \{0,1,F\}) \\
chp\{\textbf{var}\ \ x : \{0,1,F\}; \\
\quad *[X?x; \\
\quad\ \ [x = 1 \longrightarrow *[x \neq\ \ F \longrightarrow X?x; S!x] \\
\quad\ \ []x = 0 \longrightarrow *[x \neq\ \ F \longrightarrow X?x; T!x] \\
\quad\ ]] \\
\}\ .
\end{array}
$$

XV. Cosimulation

As Dijkstra said *"Testing can show the presence of bugs but not their absence"*. Some day, verification and correctness-by-design may be enough to guarantee the functionality of a large system, but not yet. Meanwhile, simulation can be used to both estimate the performance of a system (by augmenting the code with timing information) and debug it. CHPsim serves both purposes. CHPsim also provides some of the syntactic and runtime checks of a normal compiler: data type and range checking, type mismatch between connected ports, dangling ports, etc. CHPsim is also used to detect and locate deadlock. A more interesting and original function of CHPsim is *cosimulation*, which describes the simulation of a concurrent CHP system in which different parts of the system are at different stages of the QDI synthesis.

In the Caltech synthesis method for QDI, a system, for instance a microprocessor, is first described as a sequential CHP program; it is then decomposed into a fine-grain collection of CHP processes by a series of CHP-to-CHP transformations, called *process decomposition*. The next two steps realize the two main transformations necessary to implement a CHP program as a digital VLSI system. All target VLSI implementations have two main restrictions with respect to CHP: they allow only Boolean variables, and sequencing is not a primitive operation (while parallelism is). The next step transforms a CHP process into what we call a *handshaking expansion* or HSE, by reducing all operations (communications, and logical and arithmetic operations) to operations on Booleans. The following transformation eliminates all sequencing (all semicolons) from a HSE code by introducing a entirely concurrent language based on a single primitive operation called *production rule*. A program is a *production rule set* (PRS).

Cosimulation is the simulation of a system in which some components are in CHP, while others are in HSE or PRS representations. (SPICE will be added later.)

The translation from CHP to HSE and from HSE to PRS further relaxes the strict segregation of variables enforced by message passing—and which has already been breached by internal concurrency. The implementation of communication with handshake protocols introduces a restricted form of shared variables among two processes. A variable x that is part of a handshake protocol is set ($x\uparrow$ or $x\downarrow$) by one process and read ($[x]$ or $[\neg x]$) by the other process. This special type of shared Boolean variable is called a *wire* in CHPsim. The class of handshake protocols for QDI design are such that the waits on HSE wires are always stable. Typically, a four-phase handshake as described, for instance, in [10] is used, so a process like:

$$
\begin{array}{l}
\textbf{process}\ \ Source()(O! : 0..1) \\
chp\{*[O!0]\}
\end{array}
$$

can be written as:

$$
\begin{array}{l}
\textbf{process}\ \ Source()(O! : (e\downarrow; d[0..1]\downarrow)) \\
hse\{*[[O.e]; O.d[0]\uparrow; [\neg O.e]; O.d[0]\downarrow]\}
\end{array}
$$

process body, but the external interface of the process (the external ports) is now defined in terms of wires. In this example, the Boolean port O is implemented with three wire variables: the *enable* wire e initialized to zero, and the data wires $d[0]$ and $d[1]$ also initialized to zero.

A production rule consists of a guard and a target. The guard is an expression just like in a guarded command, and the target is an assignment to a single wire in shorthand ($\uparrow$/$\downarrow$) notation. The same *Source* process is implemented as two inverters in sequence (A description of production rules can be found in [10]).

$$
\begin{array}{l}
\textbf{process}\ \ Source()(O! : (e\downarrow; d[0..1]\downarrow)) \\
prs\{\textbf{var}\ \ x\uparrow; \\
\qquad O.e \longrightarrow x\downarrow \qquad \neg x \longrightarrow O.d[0]\uparrow \\
\qquad \neg O.e \longrightarrow x\uparrow \qquad\ \ x \longrightarrow O.d[0]\downarrow \\
\quad\}
\end{array}
$$

At the PRS level, a variable could be assigned concurrently by two different production rules since all production rules are potentially concurrent. This form of interference must be excluded and the stability of guards must be guaranteed. Stability and non-interference of a PRS are enforced by the synthesis procedure, but are easily violated during manual synthesis.

Running a cosimulation requires that CHPsim provide an interface between a CHP process on one side and an HSE or PRS process on the other side. A special interface process is inserted on the channel which can execute a CHP communication on one side and a handshake protocol on the other. A difficulty is that the slack of the original channel may be increased in a subtle way, which may hide some deadlock situations.

With these constructs and requirements on stability and non-interference, CHPsim has the ability to decompose a fully sequential CHP description of a large-scale system into a complete, explicit hardware-implementable description of the same system, and the ability to do so entirely via simple, local transformations of the code. Each transformation can be tested and debugged independently of other transformations, and a thorough testbench for the original CHP description is generally sufficient for testing all decomposition levels down to the final version.

XVI. Conclusion

CHP has already proved its usefulness through multiple successful designs of large systems. In particular, it is an almost ideal formalism for the decomposition of a large sequential code into a collection of fine-grain processes. In that context, even the austere syntax is an advantage for the quasi-algebraic manipulations of CHP code.

We also believe that it is not practical or wise to try and express the whole synthesis of a VLSI system from a high-level specification entirely within one single notation: the hierarchy of the three notations—CHP, HSE, PRS—is a better solution.

The design of a programming language is a difficult compromise between simplicity and soundness on the one hand, and expressive power on the other hand. We are confident that the version of CHP that we now present to the community is a good choice in that respect and that few further adjustments will be required.

ACKNOWLEDGMENTS

Many students have contributed to the design and refinement of CHP. It is impossible to name them all. Marcel van der Goot was the designer of the first version of CHPsim, and as such had an important role in the definition of the language. Acknowledgment is due to Sean Keller for his comments on the paper. The research described in this paper was supported by a grant from the National Science Foundation.

REFERENCES

[1] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, P.J. Hazewindus. The Design of an Asynchronous Microprocessor. *Decennial Caltech Conference on VLSI*, ed. C.L.Seitz, MIT Press, 351-273, 1989.

[2] Steven M. Burns and Alain J. Martin. Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. *Proc. Fifth MIT Conference on Advanced Research in VLSI*, ed. J. Allen and F. Leighton, MIT Press, 35-40, 1988.

[3] Marcel van de Goot, revised by Chris Moore. *CHPsim* A Simulator and Debugger for the CHP Language–User Manual. California Institute of Technology, 2002-2008.

[4] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs NJ, 1976.

[5] C.A.R. Hoare. Communicating Sequential Processes. *Comm. ACM* 21,8, pp 666-677, 1978.

[6] Alain J. Martin. An Axiomatic Definition of Synchronization Primitives. *Acta Informatica* 16,219–235, 1981.

[7] Alain J. Martin. The Design of a Self-timed Circuit for Distributed Mutual Exclusion. *1985 Chapel Hill Conference on VLSI*, ed. Henry Fuchs, Computer Science Press, 247-260, 1985.

[8] A. J. Martin, A.Lines, R.Manohar, M.Nyström, P.Penzes, R.Southworth, U.Cummings and T. K. Lee. The Design of an Asynchronous MIPS R3000 Microprocessor. *Proc.17th Conf. on Advanced Research in VLSI*, IEEE Computer Society Press, 164-181, 1997.

[9] Alain J. Martin. The Probe: An Addition to Communication Primitives. *Information Processing letters* 20, pp 125-130, 1985.

[10] Alain J. Martin and Mika Nyström. Asynchronous techniques for system-on-chip design *Proceedings of the IEEE* Vol.94,6:1089-1120, June 2006.

[11] Rajit Manohar and Alain J. Martin. Slack Elasticity in Concurrent Computing. *Proc. 4th Intern. Conf. on Mathematics of Program Construction*, LNCS 1422, J. Jeuring ed., Springer-Verlag, 1998.

[12] Mika Nyström and Alain J. Martin. Asynchronous Pulse Logic. *Kluwer Academic Publishers* Boston, 2001.

[13] M. Renaudin, P. Vivet, F. Robin. ASPRO: An asynchronous 16-bit RISC microprocessor. *Proc. ESSRCIRC99*, September 1999.

[14] Rajit Manohar and C. Kelly IV. Network on a chip: modeling wireless networks with asynchronous VLSI. *IEEE Communication Magazine*, November 2001.

[15] Hubert Garavel, Gwen Salan, Wendelin Serwe. On the semantics of communicating hardware processes and their translation into LOTOS for the verification of asynchronous circuits with CADP *Science of Computer Programming,* Vol.74,3,100-127, 2009.

[16] K.A.Boahen. A Burst-mode word-serial address-event link *IEEE Trans. on Circuits and Systems*, July 2004.

[17] G.N. Patel, M.S. Reid, D.E. Schimmel, S.P. DeWeerth An Asynchronous Architecture for Modeling Intersegmental Neural Communication *IEEE Trans. on VLSI Systems*, V.14, No.2, Feb. 2006