Proceedings of the International MultiConference of Engineers and Computer Scientists 2011 Vol II,
IMECS 2011, March 16 - 18, 2011, Hong Kong

Towards a Language Based Synthesis of NCL Circuits

Hemangee K. Kapoor, Abhinav Asthana, Tomas Krilavičius, Wenjie Zeng, Jieming Ma and Ka Lok Man

***Abstract*—This paper is an attempt to provide a language front-end to synthesise asynchronous control circuits using NCL technology. The target implementation being delay insensitive (DI), the specification language should be DI as well. Delay Insensitive Sequential Processes (DISP) is a process algebra where the behaviour of asynchronous control logic blocks is expressed by the processes.**

We show that one can confine the orphan paths in an NCL implementation by decomposing the language expressions.

A few basic DISP constructs have been successfully mapped to NCL and small cases studies performed. This is a step towards an alternative synthesis path for NCL circuits.

***Index Terms*—Specification languages, logic design, integrated circuits**

I. Introduction

WITH recent advances in the field of chip manufacturing technology, although packing density of logic on silicon wafer has increased tremendously, at the same time devices are approaching their physical limits in terms of dimensions as well as delays. Effect of delays is becoming more prominent over the timing issues of a design. With decreasing dimensions of devices there exists a possibility of increased power density. Circuits that are insensitive to delays, dissipate lesser power, faster in speed and are clock-less, are required. Clock-less or asynchronous circuits [1], [2] are obtained by replacing the clock by the additional control circuits. However, automation tools and specification languages are missing for design of such circuits.

Digital circuits are designed and implemented to satisfy the given specifications. Correct working and required performance depend on certain factors related to the physical parameters of the implementation technology, such as width of transistors, delays in the interconnects, capacitances, metal layers etc. Delay-Insensitivity is the property of delivering correct functioning, independent of the physical delays in wires and gates. Such circuits can be described using a process algebraic language called Delay Insensitive Sequential Processes (DISP) [3].

Null Conventional Logic [4]–[6] is a delay-insensitive circuit implementation methodology that claims to synthesise DI circuits. The term NCL is derived from the notion of absence of data using a special *null* (not data) value. We attempt to provide a language front-end to synthesise NCL using DISP. The similarities between the nature of the specifications from the latter, and the basic building blocks from the former, provide an opportunity to identify an alternate synthesis path.

The paper is organized as follows. The next section discusses related work. In Section 3 DISP and its syntax are presented. Section 4 discusses NCL and Section 5 gives the translation method. Case studies performed using the given technique are shown in Section 6. Finally we draw some conclusions in Section 7.

H.K. Kapoor, Indian Institute of Technology Guwahati, India, e-mail: hemangee@iitg.ernet.in

A. Asthana, Member IEEE, India, India, e-mail: abhinavasthana@ieee.org

T. Krilavičius, Baltic Advanced Technologies Institute, Vilnius, Lithuania, e-mail: t.krilavicius@gmail.com

W. Zeng, J. Ma and K.L. Man, Dept. of Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University (XJTLU), e-mails: wenjie.zeng07@student.xjtlu.edu.cn, jieming84@gmail.com and ka.man@xjtlu.edu.cn

II. Related Work

Currently, the language based synthesis of asynchronous circuits uses CSP [7] based formalisms. The related work includes languages like CHP [8], Tangram [9], Balsa [10], DI-Algebra [11] and DISP [3].

Graphical modelling techniques include Petri nets that use the tool Petrify [12] for synthesis.

There have also been attempts to use industry-standard HDLs like Verilog and VHDL to synthesise asynchronous circuits. However, these languages lack the basic constructs required for asynchronous behaviour, therefore special packages should to be added to the language. VHDL is currently used to synthesis NCL-based circuits. Verilog along with the tool pipefitter has also been applied to synthesis of large circuits including an asynchronous DLX pipeline processors [13].

III. DISP

DISP, a variant of DI-Algebra [11] and CSP [7], is a structured parallel programming language specifically designed for behavioural specification of a delay insensitive circuits. Using a DISP, a developer can specify the behaviour of a DI circuit in terms of a processes. It allows a designer to specify behaviour of asynchronous logic blocks and investigate diverse handshake protocols.

Gate net-list can be obtained from DISP specifications using a tools `di2pn` [14] and `petrify` [12]. Former is used to translate the DISP expressions into Petri net, and the later synthesises the circuit from the Petri net specification.

A. Language Syntax

Behaviour of a circuit is described as a process in DISP. Each process is associated with an input and output alphabet and is capable of absorbing inputs and emitting outputs. The language syntax is defined as follows:

$$\begin{aligned} \mathit{proc} ::= & \mathit{var} \mid \texttt{stop} \mid \texttt{skip} \mid \texttt{error} \mid \mathit{burst} \mid \\ & \mathit{proc} \ ; \ \mathit{proc} \mid \mathit{proc} \ \texttt{or} \ \mathit{proc} \mid \mathit{proc} \ \texttt{par} \ \mathit{proc} \mid \\ & \texttt{forever do} \ \mathit{proc} \ \texttt{end} \mid \texttt{select} \ \mathit{alt\text{-}set} \ \texttt{end} \\ \mathit{alt\text{-}set} ::= & [\ \mathit{burst} \ [\ \texttt{then} \ \mathit{proc}\ ] \ \{\texttt{alt} \ \mathit{burst} \ [\ \texttt{then} \ \mathit{proc}]\ \}\ ] \end{aligned}$$

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

$burst ::= siglist1 \,/\, siglist_2$ is an innput and output burst, where the set of input events ($siglist_1$) must occur before the output events ($siglist_2$) are generated. $siglist$ is a set of signals (possibly empty), specified as $siglist ::= - \mid sig\{, sig\}$ where $-$ denotes an empty list and signal names are given in comma-separated list.

A process can be identified by using a process variable. Behaviours like `stop` and `error` are used to specify processes that will eventually diverge (do anything whatsoever). A `skip` does not perform any action and terminates immediately. It can also be written as $\texttt{skip} = -/-$. A *guarded choice* behaviour can be specified using an $\texttt{alt} - \texttt{set}$ expression. The choices are guarded by the `burst` expression. The processes whose guard is satisfied are chosen for execution. *Continuous execution* is provided by the `forever` construct. The `or` operator specifies a non-deterministic choice between two processes. Processes can be composed sequentially $P \;;\; Q$ and in parallel P `par` Q. For two processes P and Q with input alphabets I_P and I_Q and output alphabets O_P and O_Q when composed in parallel, must satisfy alphabet restrictions $I_P \cap I_Q = \emptyset$ and $O_P \cap O_Q = \emptyset$. In other words, they should not generate the same outputs and cannot share inputs. However, output from one process can be input to the other process in the composition. Such signals $((I_P \cap O_Q) \cup (I_Q \cap O_P))$ are called internal signals and are not observable from the environment.

The behaviour is delay-insensitive, hence every transition on a wire must be acknowledged before another transition is sent on that same wire, because two consecutive transitions on the same wire may superimpose on each other leading to the transmission interference. I.e., pulses cannot be safely transmitted.

$$a/- \;;\; a/- \;;\; P = a/- \;;\; a/- \;;\; \texttt{error}$$
$$-/c \;;\; -/x \;;\; P = \texttt{error}$$

IV. Null Conventional Logic

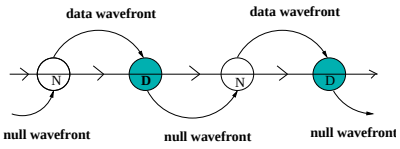


Fig. 1. Wavefronts in a data flow

Null conventional logic (NCL) [4], [15], [16] is a deviation from the conventional boolean systems where the value of the signal itself is used to show its arrival/presence/validity. Thus each variable in the expression has two values: (a) DATA (indicating the value as well as validity) and (b) NULL (indicating absence of data). DATA can be an abstract set of values. Use of a NULL value for a variable gives this system the name *Null Conventional Logic* (*NCL*). As the variables represent their own presence, the validity of the outputs is easily determined, therefore is not required to compute the stabilised output generation time. It reduces the burden of estimating the timing requirements and strengthens the logical structure of the system [6].

There are two conceptual flows for signals in an NCL implementation: the data-wavefront and the flow of NULL items (to clear all states) called the null-wavefront, fig. 1.

One set of input values leads to one set of output values. The final generation of output can be easily detected by using a completion detection circuit. To read a new set of input values, previously generated outputs are flushed by making all the inputs NULL.

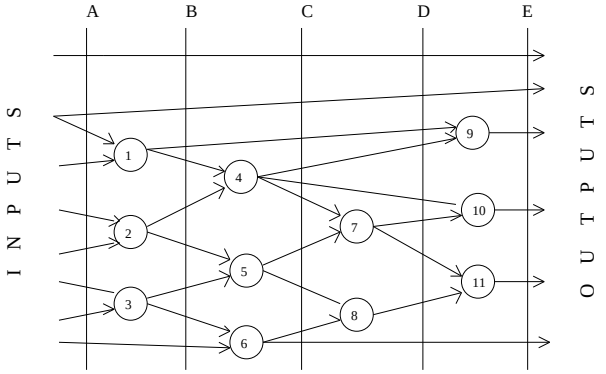


Fig. 2. Presentation (validation) boundaries for input variables

Fig. 2 shows a connection of NCL gates (1-11) forming a multi-level logic implementation. A, B, C, D and E are the logical boundaries for the representation of a signal. The final set of generated outputs (at the boundary E) is valid iff all the outputs of boundary D are valid, and so on. Following the chain, it is easy to see that all the output elements can be generated only when all the inputs have arrived [17].

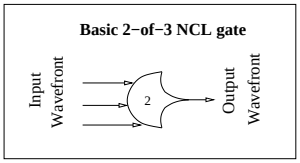


Fig. 3. Diagram for basic NCL gate

A basic NCL gate is shown in fig. 3. It has three inputs and a threshold of two, hence a 2-of-3 NCL gate. The number inside the gate denotes it's threshold. As follows from the description, to get logical 1 in output at least two out of three inputs should have value 1, e.g. let a, b and c be t he inputs and z be an output, then the gate represents a logical equation $\texttt{Output} = a.b + b.c + c.a$. To reset the output to 0, all the inputs must go to 0 irrespective of the threshold. Until all the inputs are reset the gate holds the previous state.

Note that an N-of-N NCL gate is equivalent to an N input C-element [18].

V. Mapping DISP Expressions to NCL

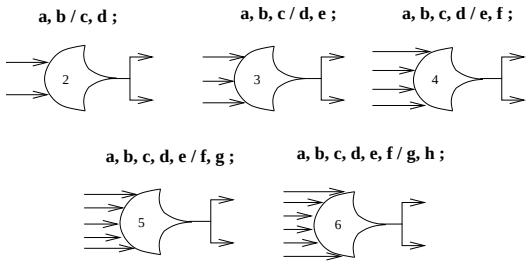


Fig. 4. Burst with N ($2 \leq N \leq 6$) inputs and two outputs

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

a) Burst expression: as discussed earlier, a DISP burst can be directly mapped to an NCL gate. Further optimisations can be performed on the implementation to take care of threshold limitations and to reduce the overall gate count. An input-output burst in DISP means that transitions on the inputs must be followed by transitions on the output. Each DISP burst therefore consists of a pair of data and null wavefront in the NCL implementation. E.g., the burst a/b in DISP is implemented in NCL to have a data wavefront on signals a and b; generation of data wavefront on b starts the null wavefront on a; which in turn starts the null wavefront on b.

To translate a DISP burst having N inputs and M outputs we use an N-of-N NCL gate with output forked in M different directions. E.g., burst $B = a, b, c \,/\, d, e$ waits for all inputs (a, b, c) to become valid before it can generate the output signals (d, e). The second gate in the fig. 4 is similar to the above burst expression. The other structures in the figure show example bursts and their NCL implementations with input signals ranging from two to six.

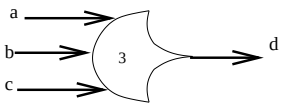


Fig. 5. Non-fragmented burst.

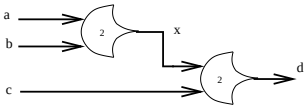


Fig. 6. Fragmented burst.

In NCL the number of inputs to a gate cannot be more than six, hence bursts with more than six inputs are decomposed. For example, the burst $B = a, b, c \,/\, d$ is implemented without decomposition as shown in fig. 5, and with decomposition as in fig. 6, using two 2-of-2 gate.

b) Guarded choice: `select` ***alt-set*** `end`. The selection can be done with the help of a decision-wait element (described in Section VI).

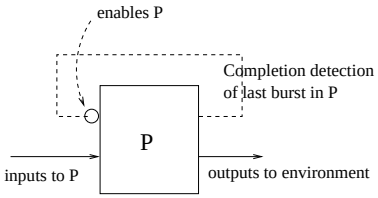


Fig. 7. A forever process

c) Infinite repetition: `forever do` *proc* `end` is required to express continuously running hardware. The process is implemented using other basic translation rules. The completion of the last burst in it will enable the first burst.

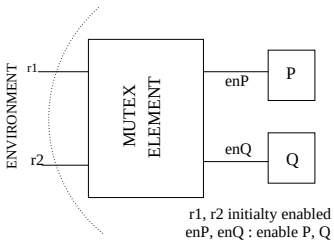


Fig. 8. Non-deterministic choice among processes

d) Non-deterministic choice: P `or` Q. To perform it we need mutual-exclusion, which can be implemented using a Mutex element, fig. 8. The two requests to the Mutex are assumed to be present when the composition is invoked by the environment. The Mutex then arbitrarily decides whether to execute P or Q.

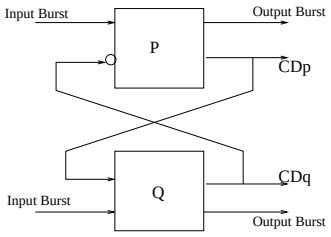


Fig. 9. Sequential Composition of Bursts

e) Sequential composition: P ; Q can be implemented using a *sequencer* element. The sequencer keeps track of the order of execution. The completion of P is an input to the sequencer, which in turn enables the execution of Q. The data and null-wavefronts for P should be completed before those of Q. When Q has signals distinct from P, the null-wavefront of P can be delayed to happen concurrently with data-wavefront of Q. A generic sequencer is shown in fig. 9, the implementation is given only for small size burst compositions (cf. Section VI).

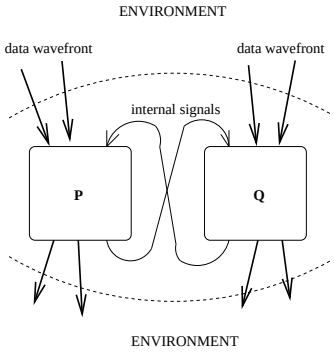
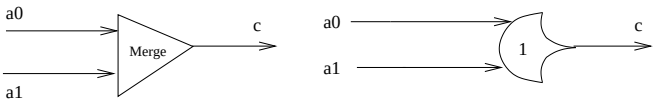


Fig. 10. Parallel Composition of Burst Expressions

f) Parallel composition: P `par` Q. Both processes are implemented using their respective process constructs and they can run concurrently. The internal signals of each are cross-connected, fig. 10.

VI. Case Studies



(a) Normal MERGE element (b) MERGE using NCL gate

Fig. 11. Model of a Merge element using NCL gates

A Merge [19] has two input terminals, and one output terminal. It 'merges' signals on the input terminals to the output terminal. Input and output signals alternate. The environment has to guarantee mutual exclusion on the inputs.

$$\textbf{merge} = \texttt{forever do select } a0 \,/\, c \texttt{ alt } a1 \,/\, c \texttt{ end end}$$

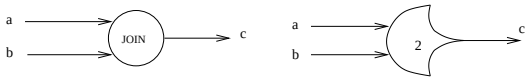
ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

The merge elements waits for an activity on the inputs and propagates the output. As activity over a single input element is enough to generate the output and the environment guarantees mutual exclusion of input elements, the merge element can be formed using a 1-of-2 NCL gate, fig. 11.

A generalised K-merge

K-merge $=$ `forever do`
`select` $a0 \,/\, c$ `alt` $a1 \,/\, c$ `alt` $\ldots$ `alt` $ak \,/\, c$ `end`
`end`

can be implemented using 1-of-K NCL gate.



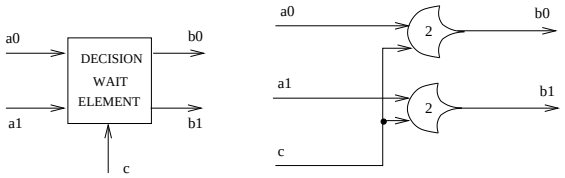
(a) Normal JOIN element (b) JOIN element using NCL gate

Fig. 12. Model of a Join element using NCL gates

Join [19] has two input terminals and one output terminal. It waits until input signals on both input terminals arrive, after which it produces a signal on its output terminal.

$$\textbf{join} = \texttt{forever do}\ a, b \,/\, c\ \texttt{end}$$

As a join element waits for both inputs to arrive, we can use a 2-of-2 NCL gate for its implementation, fig. 12. Similarly, a K-join element can be synthesised using a K-of-K NCL gate.



(a) Block diagram of Decision Wait element (b) Decision Wait element using NCL gates

Fig. 13. Model of a 2×1 Decision wait element

2×1 Decision Wait has three inputs (a0, a1,and c), and two outputs (b0, b1). It waits for a signal on one of the ai inputs and a signal on c, before it outputs on bi. The environment has to guarantee mutual exclusion on the a-inputs (shown by erroneous behaviour after both a0 and a1 arrive). Each guard is implemented using an appropriate threshold NCL gate, see fig. 13.

decision-wait $=$ `forever do`
`sel` $a0, c \,/\, b0$ `alt` $a1, c \,/\, b1$ `alt` $a0, a1 \,/\, -$ `then error end`
`end`

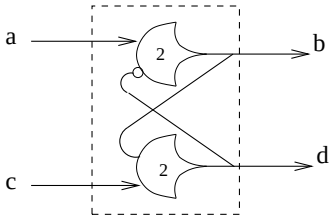
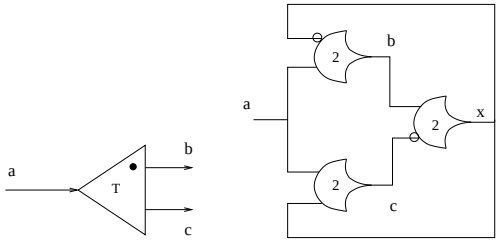


Fig. 14. Block and Internal diagrams for Sequencer element

This example shows a **sequential composition** of two burst expressions. Completion of the first burst a/b triggers the beginning of the second burst c/d. We need the concept of a state to sequence the bursts. The implementation is shown in fig. 14. The first gate is enabled initially and when a arrives, the output b is generated. It enables the second gate and disables the first gate enabling the reception of c and generation of d.

$$\textbf{sequencer} = \texttt{forever do}\ a \,/\, b\ ;\ c \,/\, d\ \texttt{end}$$

Note that in the given implementation, the data wavefront on a/b happens before the data wavefront on c/d and the null wavefront on a/b happens concurrently with the data wavefront of c/d. It becomes a limitation when we compose burst using common signals. For such cases we need to complete the null-wavefront of the first burst before the data wavefront of the second can begin. It can be implemented with using state variables as in the toggle element



(a) A Toggle Element (b) Toggle Element using NCL−gates

Fig. 15. Model for Toggle Element based on NCL gates

Toggle [19] has one input (a) and two outputs (b and c). Each input signal produces one output signal. Input and output signals alternate. Signals on the output also alternate, first on b then on c, etc.

$$\textbf{toggle} = \texttt{forever do}\ a \,/\, b\ ;\ a \,/\, c\ \texttt{end}$$

As mentioned, we need a state variable to implement such sequential composition, see fig. 15. The state variable is required because, after the data-null wavefronts on a/b are over, the circuit is in the same state as it started, and it has no way to distinguish between the first a and the second a. It is called a state coding conflict in digital logic [20], [21].

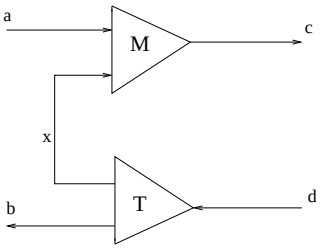


Fig. 16. Model for 2 to 4 phase converter using NCL gates

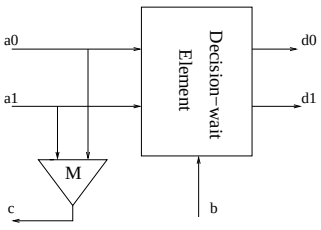


Fig. 17. Model of a call element

2 to 4 Phase Converter [19] has two inputs (a and d) and two outputs (b and c). Input a and output b alternate (together forming a passive handshake channel), as do output c and input d (together forming an active handshake channel). Every 2 phases ab enclose 4 phases cdcd.

```
Conv = forever do
  select a/c then
```

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
    select d/c then
      select d/b
        alt a/- then error end
      alt a/- then error end
  alt d/- then error end end
```

Effectively the behaviour is following

2to4 phase converter = `forever do` $a\,/\,c\,;\,d\,/\,c\,;\,d\,/\,b$ `end`

with the constraint that inputs a and b are mutually exclusive.

Observing P, we get two sequential compositions, viz., $a/c; d/c$ and $d/c; d/b$. The first one, given the constraint that a and d are mutually exclusive can be implemented using a merge-element. The second composition is similar to a toggle element. As the two compositions are connected, output of one goes as input to the other. The implementation using merge (M) and toggle (T) is shown in Figure 16, and it is same as that given in [19].

A **non-arbitrating, blocking call-element** [19] has three inputs (a0, a1, b) and three outputs (c, d0, d1). A signal appearing on either of the a_i's will produce a signal at c. The combination of a signal at a_i and b will produce a signal atd_i. It does not matter which of the two input signals arrives first. The environment of the Call must guarantee mutual exclusion of the signals on $a0$ and $a1$. The a_i and c signals alternate, the a_i and d_i signals alternate, and the b and d_i signals alternate.

```
call = forever do
  select a0/c then select b/d0
                   alt a0/- then error end
    alt a1/c then select b/d1
                   alt a1/- then error end
    alt b/- then error end end
```

The description puts constraints that inputs $a0$ and $a1$ are mutually exclusive (indicated by divergence after both $a0$ and $a1$ arrive).

The output c is generated by either $a0$ or $a1$. Hence, we can use a merge-element with inputs $a0$ and $a1$ to generate c. The overall behaviour of Call is a guarded choice implemented by a 2×1 decision-wait (DW) element. The inputs to DW are $a0, a1, b$ and outputs - $d0, d1$. See this implementation, matching one in [19], in fig. 17.

VII. Orphan Paths

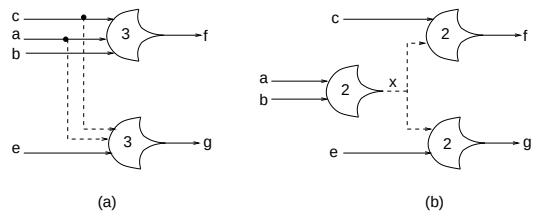


Fig. 18. (a) Orphan path, (b) Confined orphan path

An orphan path is a branched off connection carrying data to an NCL gate and is not used to produce any output [6]. When the output of an NCL implementation transitions to a complete DATA state, it implies that the input data set is complete and that the transitions to DATA have propagated over the effective path. There will also be *ineffective* paths branching off from this effective path that do not contribute to the output and therefore are not logically determined by the output. These ineffective paths are called *orphans* because they have lost all of their logical relations. In the fig. 18, orphan paths are depicted by dotted lines.

A. Identifying Orphan paths in DISP

Orphan paths can be identified at a point, when a particular data wavefront follows two different paths producing different outputs. In terms of DISP,

```
P = forever do
      select a,b,c/f alt a,b,e/g end
    end
```

Here DATA wavefronts on `a` and `b` follow two different paths to get mixed with other elements and produce various elements. Depending on the availability of other input elements, only one of the split DATA waves will result in an output burst. Other wavefronts on `a` and `b` will remain ineffective.

E.g., if we get DATA wavefront on `a, b, c`, the circuit will send a DATA wavefront on the output `f`, and the wavefront depicted by the dotted line (going to the second gate) becomes orphan.

B. Confinement of Orphan paths

Though one cannot avoid orphan paths in an NCL implementation, we suggest to confine and convert them. Orphan paths stem from the forked paths. By extracting the common burst expression, it can be implemented using an extra NCL gate. The output of this extra intermediate gate is neither an input nor an output and can be used as an internal signal. Such an internal element denotes the presence of a common input burst as well. E.g., in the above expression we separate the common input burst a,b/- and rewrite it:

```
P = forever do
      a,b/- ; select c/f alt e/g end
    end
```

Then we expand it using an internal variable (x):

```
P = forever do
      a,b/x ; select x,c/f alt x,e/g end
    end
```

Internal variable x represents a steady presence of a and b. One may notice that x gets forked and sent to the separate bursts; it results into an orphan path as well. The main difference between an ordinary orphan path and x is the following: the latter is logically determined. The idea to generate x also reduces the number of orphan paths significantly. Element x may still behave as a slow orphan but its impact is reduced, because it cannot get mixed with a DATA or NULL wavefronts. Those wavefronts are governed by the presence/absence of a and b. Instead of appearing at the beginning of the expression, the fork is *confined* inside the expression and converted into an internal variable. The confinement is illustrated in fig. 18-(b).

C. Advantage at the language level

A slow orphan is not a problem for the correctness, as the output DATA wavefront is generated using other variables. However, this trailing wavefront on the orphan may interfere with the succeeding DATA wavefront in a non-deterministic manner and cause unexpected glitches.

As discussed in [6], there may exist an implementation, where all the inputs of an NCL gate are fed by the orphan paths in the circuit. Such an NCL gate gives out an orphan

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

path as its output. Though the output of the gate may also be used by another gate, hence there must be some security measure restricting the orphan inputs to change their state. By renaming the generation of orphan output as one of the completion detection criteria of the circuit, solves the issue. The DATA wavefront is not allowed to change the state, until conflict among the orphan paths is not over.

Using DISP

Our solutions differs from the one in the earlier works, because the expressions are in the form of I/O bursts and not as combinational logic. The completion detection of a burst is determined by the generation of all the output elements. Confining the orphan path within the expression makes superfluous other completion detection criteria. The internal element generated by the extracted burst expression seems to be similar to the orphan output of [6]. However, due to the confinement of the orphan path, this internal element needs not be exposed to the environment, thereby reducing the complexity in the completion detection circuitry. Even after this fragmented implementation, the input and output elements are directly related to each other as specified by the DISP expression, unaware of the fact that the bursts were fragmented before their actual implementation.

VIII. Conclusion

The paper presented an attempt to find an alternative synthesis path for NCL based implementations. The target implementation being delay-insensitive the language chosen was DISP. The basic construct of the language, an input-output burst, was found to be directly implementable using N-of-M NCL gates.

The main idea of synthesis is to construct the basic building blocks of DI [22] using NCL gates. Then identify these in the DISP language and map them appropriately.

While translating a given behaviour, what remains is to identify such basic blocks in the language and map them to NCL implementations. However, the concept of a data followed by null wavefront forms a crucial factor. Care needs to be taken to make sure that each data wavefront is complete in order to generate the correct outputs and also the computational block goes through a null wavefront before another computation can begin. This leads to the study of orphan paths [6], paths not used by the circuit but still carrying data wavefronts. The implication of 'hysteresis' and 'variable threshold' on the implementation also need to be addressed.

Issues related to state variable insertion to solve state coding conflicts is another problem. This is due to the fact that although the logic blocks can undergo a data-null wavefront, the state variables need to be prevented from doing so immediately (in order to hold state). This controlled nullification of state variables remains to be solved. The paper demonstrated this with a small example of a toggle element which had only one state coding conflict. However, a generic method of state variable insertion requires a deeper study of the wavefronts and correct identification of places to insert state variables.

Although many issues still remain to be addressed, this initial study has given a confidence in designing such an alternative synthesis path. This is a positive step as currently there is no pure asynchronous language front-end to NCL synthesis. Automation of the complete synthesis also needs to be dealt with.

References

[1] S. Hauck, "Asynchronous design methodologies : An overview," *Proceedings of the IEEE*, vol. 83, pp. 69–93, January 1995.

[2] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design A System Prespective*. Kluwer Academic Publishers, 2001.

[3] M. B. Josephs and D. P. Furey, "A programming approach to the design of asynchronous logic blocks," in *Concurrency and Hardware Design, Advances in Petri Nets*, ser. Lecture Notes in Computer Science, vol. 2549. Springer, 2002, pp. 34–60.

[4] K. M. Fant and S. A. Brandt, "Null convention logic : A complete and consistent logic for asynchronous digital circuit synthesis," in *ASAP*. IEEE Computer Society, 1996, pp. 261–273. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ASAP.1996.542821

[5] K. Fant and S. Brandt, "Null convention logic," 1994. [Online]. Available: citeseer.ist.psu.edu/fant94null.html

[6] F. K. M., *Logically Determined Design – Clockless System Design with Null Conventional Logic*. New Jersey: Wiley Interscience, 2005.

[7] C. Hoare, "Communicating sequential processes," *Comm. ACM*, vol. 21, no. 8, pp. 666–677, aug 1978.

[8] A. J. Martin, "Programming in VLSI: From Communicating Processes to Self-timed VLSI Circuits," in *Proceedings of UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, March 1987.

[9] K. v. Berkel, *Handshake Circuits - An Asynchronous architecture for VLSI programming*. Cambridge University Press, 1993.

[10] A. Bardsley and D. Edwards, "Compiling the Language Balsa to Delay-insensitive Hardware," *Hardware Description Languages and their Applications*, pp. 89–91, April 1997.

[11] M. B. Josephs and J. T. Udding, "An overview of DI algebra." in *26th Hawaii Int. Conference on System Science (HICSS 1993)*, JAN 1993, pp. 329–338.

[12] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers," *IEICE Transactions on Information and Systems*, vol. 3, no. E80-D, pp. 315–325, 1997.

[13] M. Amde, I. Blunno, and C. P. Sotiriou, "Automating the Design of an Asynchronous DLX Microprocessor," in *Proceedings of the 40th Design Automation Conference (DAC), ACM*, 2003, pp. 502–507.

[14] D. Furey and M. B. Josephs, "Asynchronous circuit design via automated petri net generation," 2003. [Online]. Available: http://citeseer.ist.psu.edu/607069.html;http://www.sbu.ac.uk/~fureyd/petrinet_combinators.pdf

[15] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial HDL synthesis tools," in *ASYNC*. IEEE Computer Society, 2000, p. 114.

[16] S. K. Bandapati and S. C. Smith, "Design and characterization of NULL convention arithmetic logic units," in *Proceedings of the International Conference on VLSI, VLSI '03, June 23 - 26, 2003, Las Vegas, Nevada, USA*, H. R. Arabnia and L. T. Yang, Eds. CSREA Press, 2003, pp. 178–184.

[17] C. L. Seitz, "System Timing," in *Introduction to VLSI Systems*, Mead and Conway, Eds. Addison-Wesley, 1980, ch. 7.

[18] I. E. Sutherland, "Micropipelines." *Commun. ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[19] T. Verhoeff, "Encyclopedia of delay-insensitive systems (EDIS)," `http://www.win.tue.nl/~edis/edis.html`, Dept. of Math. and C.S., Eindhoven Univ. of Technology.

[20] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "A Region-based Theory for State Assignment in Speed-Independent Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, pp. 793–812, August 1997.

[21] ——, "Complete State Encoding Based on the Theory of Regions," *Proceedings of Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 36–47, March 1996.

[22] P. Patra and D. Fussell, "Building-blocks for designing DI circuits," Department of Computer Science, University of Texas at Austin, Tech. Rep., November 1993.

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)