Specification and Analysis of NCL Circuits

J. Ma, H.K. Kapoor, T. Krilavičius, K.L. Man, N. Zhang, E.G. Lim, T.T. Jeong, S.U. Guan and J.K. Seon

***Abstract*— Due to a number of existing limiting factors in synchronous circuit design, the semiconductor industry gives renewed interest to the application of asynchronous technology. NCL (NULL Conventional Logic) is a Delay-Insensitive (DI) clockless paradigm convenient for implementing asynchronous circuits. Efficient analysis methods and tools are proposed to specify and verify such DI systems. Based on DISP (Delay Insensitive sequential Process) specification, this paper exemplifies application of formal methods by applying Process Analysis Toolkit (PAT) to model and verify behavior of NCL circuits. A few useful constructs, such as Boolean logic gates, binary half adder and pipeline ring, are successfully modeled and verified by using PAT. The flexibility and simplicity of modeling, simulation and verification show the usefulness and applicability of PAT for NCL circuit design and verification.**

***Index Terms*—NCL circuits, CSP#, specification, integrated circuits.**

I. Introduction

SYNCHRONOUS and clocked architectures have dominated digital design for many years. With the development of manufacturing technology, tens of billions of transistors can be integrated on to a single chip. At the same time, however, concerns have been raised due to many limiting factors of the synchronous design, including increasing clock frequency, decreasing chip size, and increasing power consumption. It seems that the clock is getting harder and harder to manage and the increasing difficulties of synchronous design have renewed the interest in asynchronous digital design, which is thought to be a potential solution for many inherent defects of clocked system.

NULL Conventional Logic (NCL) integrates the expression of data transformation and the expression of control into a single symbolically determined expression [1]. It is one of the promising methods that can design and implement of asynchronous circuits. Unlike Boolean logic, NCL circuits perform complete function independent of the wire delays. The signal values in NCL are directly used for representing the data arrival, and they monotonically transit between 'complete data' and 'all NULL (no data)'. NCL paradigm is attractive since it has the merit of asynchronous circuits with considerably less design cost and risk.

Many language based approaches have already been proposed for asynchronous circuit synthesis. In [2], several basic Delay Insensitive Sequential Process (DISP) [3] constructs have been successfully mapped to NCL and that shows a step towards an alternative synthesis path for NCL circuits. However, these DISP constructs lack of formal verification support.

Communicating Sequential Programs (CSP#) (presented in the toolkit PAT [5]) is a programming language that can be used for both modeling and verifying the behavior of variety of concurrent systems. This paper seeks the way of mapping NCL circuits to CSP# constructs, which allows the use of the Process Analysis Toolkit (PAT) [5] to model and verify the behavior of NCL circuits through CSP# constructs. The operator of CSP# are based on the classic CSP process algebra [4]

This paper has been organized in the following way. The next section gives an overview of NCL circuits. Section III lays out the DISP language syntax, and presents how to convert DISP into CSP#. The methodology of verification in PAT is then described in the last part of Section III. Through several case studies, Section IV presents the synthesis, characterization and verification of several NCL circuit models using PAT. Finally, we draw conclusions in Section V.

II. Overview of NCL

Since Boolean functions determine the output values based on only the input value, and since the speed of different signal paths is varied, a series of intermediate result transitions may be delivered ahead of valid stable transitions. It is hard to express the boundaries of instantiation and resolution by traditional time-dependent and symbolic-value-dependent Boolean logic.

TABLE I
Boolean Truth Tables with NULL Value

	T	F	N
T	T	F	N
F	F	F	N
N	N	N	N

AND

	T	F	N
T	T	T	N
F	T	F	N
N	N	N	N

OR

	F
T	F
F	T
N	N

NOT

In order to make Boolean logic symbolically complete, additional logical expression NULL are added to the traditional Boolean truth table as shown in TABLE I. True (T) and False (F) are data values while NULL is not. A data can be asserted data value only if there is 'complete data' present

Manuscript received June XX, 20XX; revised July XX, 20XX.

J. Ma is with the department of Computer Science, University of Liverpool, UK (e-mail:jieming@liverpool.ac.uk).

T. Krilavičius is with the Baltic Institute of Advanced Technology, Vilnius, Lithuania and Informatics faculty, Vytautas Magnus University, Kaunas, Lithuania (e-mail:t.krilavicius@gmail.com).

H. K. Kapoor is with the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, Assam, India (e-mail:hemangee@iitg.ernet.in).

K.L. Man, N. Zhang, E.G. Lim, and S.U. Guan are with Xi'an Jiaotong-Liverpool University, Suzhou, P.R.China. (e-mail: {ka.man;nan.zhang;enggee.lim;steven.guan}@xjtlu.edu.cn).

T.T. Jeong is with Myongji University, Korea (e-mail:ttjeong@mju.ac.kr).

J.K. Seon is with LS Industrial Systems, Korea (e-mail: jkseon@lsis.biz).

at the input. Similarly, NULL value will be asserted to the output only when there is 'all NULL' at the input. The completeness of input criteria is a significant feature of DI circuits. Consider the combinational expression in Fig. 1., circles denote the logic operators and A, B, C and D are the logical boundaries for the presentation of a signal. The symbols crossing D cannot be valid until all of the symbols crossing C are complete data, and so on.

A logic that includes a NULL value and recognizes completeness relationships in the primitive logic operators will be referred to as a NCL [1]. NCL circuits have no time relationships and are insensitive to the propagation time of symbols among their components.

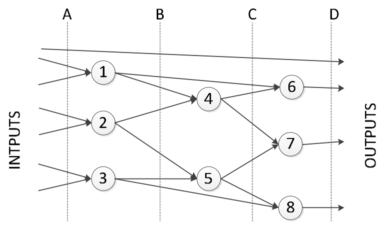


Fig. 1. Presentation boundaries for a combinational expression.

TABLE II
DUAL-RAIL ENCODING

Logic value	Encoding	
	D_0	D_1
Data1	0	1
Data0	1	0
Null	0	0
Invalid	1	1

2NCL[1] is a kind of NCL logic that obtains only one data value, which indicates that the signal path can transit between "DATA" and "NULL" without intermediate values. Multiple mutually exclusive values are normally expressed by multiple signal paths. In a binary system, a dual-rail signal D, which is transmitted by two mutually exclusive wires (D_0 and D_1), is used to express True and False. As seen in TABLE II, Boolean logic 0 and 1 are equivalent to the Data0 (D_0=0, D_1=1) and Data1 (D_0=1, D_0=0) respectively. NULL state comes only when both the inputs receive logic 0 and the state D_0=1, D_1=1 is not permitted.

The basic logic elements of 2NCL are threshold gates. THmn gate, shown in Fig. 2, is a primary type of threshold gates. It has n input terminals with threshold m, where $1 \leqslant m \leqslant n$, i.e. it becomes activated if at least m of n inputs are active.

Since 2NCL circuits follow the input-completeness criterion [1], the output will be asserted only if all the inputs are asserted DATA and the output will not transit from DATA to NULL until all inputs have transited from DATA to NULL. Weighted threshold gate is another widely used type. When wR ($1 \leqslant wR < m$) denotes the weight of inputR ($1 \leqslant R < n$), a threshold gate can be represented as THmnWw1w2⋯wR, where w1, w2, ⋯ wR are integer weights of input1, input2, ⋯ inputR respectively and they should be larger than 1. Fig.3 shows a TH33w2 threshold gate that has 3 inputs with 3 thresholds. The weight of input A is 2, which implies the output cannot be asserted until A is asserted along with input B, C or both. Then the gate can be represented by a logical equation Z=AB+AC.

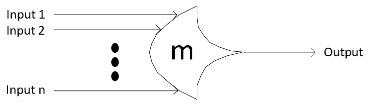


Fig. 2. A THmn NCL threshold gate.

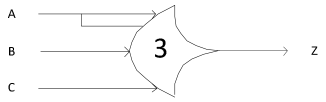


Fig. 3. A TH33w2 threshold gate: Z=AB+AC.

An asynchronous cycle path normally consists of the completeness detection path, the acknowledge path and the data path which may include asynchronous combinational circuits. Fig. 4 shows a structure for the basic NCL pipeline. Asynchronous combinational circuits are involved between two NCL registers, which control the request and acknowledge signals. The input request signal (Ki) is from the completion detection path of the next cycle. The output acknowledge signal (Ko), however, will not be generated by the current acknowledge path until the current computation cycle is complete. If the previous cycle does not receive the acknowledge signal, the data will be blocked and the subsequent process cannot continue.

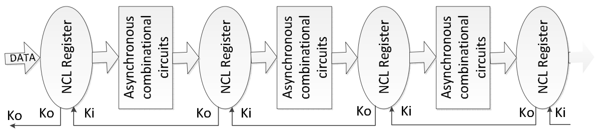

DATA
NCL Register
Asynchronous combinational circuits
NCL Register
Asynchronous combinational circuits
NCL Register
Asynchronous combinational circuits
NCL Register
Ko
Ko
Ki
Ko
Ki
Ko
Ki
Ko
Ki

Fig. 4. Basic NCL pipeline.

III. LANGUAGE BASED SYNTHESIS

A. *DISP based synthesis*

The term Delay Insensitive (DI) is generally understood to mean performing correct circuit operation regardless of the wire delays. DI-Algebra is an algebra used for describing the processes that communicate in terms of DI, and it has been applied in the design, as well as decomposition and verification of DI circuits [10, 11, 12, 16, 17].

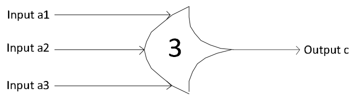


Fig.5. A 3-of-3 C-element.

DISP, however, provides a much more simple way to specify the asynchronous circuits and is supported by Computer Aided Design (CAD) tools like di2pn [13] and Petrify [5] for specification and verification. DISP is similar to the handshaking behavior but obtains uniform treatment to signals. Behaviors of 2NCL circuits are DI and thus they can be expressed by processes in DISP. The concrete syntax of DISP is defined as follows:

proc :: = stop | skip | error | *burst* | select *choice* end | forever do proc end | proc ; proc | proc par proc

choice ::= burst [then proc] [alt choice]
burst ::= *siglist*/*siglist*

The entity *siglist* is a list of signal names. The simplest process is a burst (input/output burst), where all signals in the output burst will not happen until all the signals in the input burst are absorbed. The *burst* can be straightforwardly translated to the burst behavior of a THmn threshold gate. For instance, C-element [6] is a kind of THmn gates whose threshold number is equivalent to the number of input terminals. By using burst construct and an infinite repetition, which is usually represented by forever-do-end construct, a C-element3 (as seen in Fig. 5) can be mapped easily as follows:

C-element3 = forever do a1, a2, a3/c end

(a1, a2, a3) are inputs signals while c is output signal. Output c will be generated only after a1, a2 and a3 become valid. The select-end process delimits a process from a choice, which is restricted to a number of guarded processes. Together with infinite repetition, a single select-end process is applied to describe the behavior of a TH1k (k⩾1) 2NCL gate as seen in Fig. 6. Its DISP specification is expressed as follows:

TH1k = forever do select a1/c alt a2/c alt … alt ak/c end end

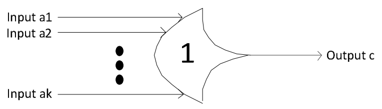


Fig. 6. A TH1k threshold gate.

Processes can also be composed either sequentially or concurrently. Fig. 7 shows a block and internal diagrams for Sequencer element. The second burst c/d cannot be processed until the first burst a/b is completed. In the following DISP expression, sequential behavior is stated by a semicolon.

Sequencer = forever do a/b; c/d end

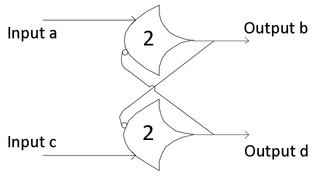


Fig. 7. A TH1k threshold gate.

The process proc par proc is used to express the parallel composition of two processes. The behavior of a C-element3 can be decomposed into two parallel processes as given in the following expression:

C-element3=forever do a1, a2, a3/c end
=forever do a1,a2/d end par forever do d,a3/c

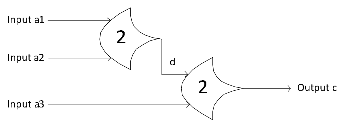


Fig. 8. Fragmented C-element3.

It is worth pointing out that a DISP program always uses signals in a consistent way, either for input from the environment or for output to the environment, or for local communication in a parallel composition [3]. A case in point is the decomposed expression of C-element3, which is described by a/b par b/c rather than a/b;b/c. Furthermore, sharing input or output signals is not permitted in the parallel composition.

B. Mapping between DISP and CSP#

Though DISP can be applied to specify the 2NCL asynchronous circuits in a simple manner, as far as we are aware, there is a lack of tools for DISP verification.

CSP# is a modeling language, which integrates high-level modeling operators with low-level procedural codes, for the purpose of efficient mechanical system verification [10]. Since most of the CSP# syntax can match with the syntax of DISP, it is easy to convert DISP to CSP#. For a detail of mapping between DISP and CSP# at the semantical level will be subjected to future work. The target CSP# codes can be conveniently modeled and verified by PAT.

The following is BNF [11] description of the CSP# expressions.

```
P ::= Stop | Skip | e{prog} -> P
      | P; Q |P [] Q |
      | [b]P | P || Q | P ||| Q |
```

P and Q are processes. e is an event name and the sequential program *prog* is optional. b states a Boolean expression here.

Stop is a deadlock process that does absolutely nothing. Compared to Stop, Skip processes a special terminating event first, and then behaves exactly the same as Stop. Event prefixing e->P performs event e first and then behaves as process P.

The expression of sequential composition in DISP is the same as that in CSP#: the process (P; Q) starts P first and Q starts only when P has been terminated. A general choice can be stated by []. For instance, P [] Q describes either P or Q may be processed. The symbol || is used to denote parallel composition, which synchronizes common events in the alphabets of P and Q. Interleaving, however, runs all processes independently. In a guarded process [b]P, P will not be executed until condition b is satisfied. Recursion in CSP# can be expressed by process referencing. The following process gives a simple example of mutual recursion.

```
P() = a -> Q();
Q() = b -> P();
System() = P() || Q();
```

As seen in TABLE III, basic 2NCL behavior described in the previous section can be easily translated at the syntactical level to the equivalent CSP# expression.

TABLE III
DISP AND CSP# EXPRESSION FOR BASIC PROCESS

Process	DISP	CSP#

C-element3	forever do
	a1, a2, a3/c
		P1() = a1 ->c ->Skip;
		P2() = a2 ->c ->Skip;
		P3() = a3 ->c ->Skip;
		P4() = P1()||P2()||P3();
		F() = P4();F();

TH1k threshold gate	forever do
	select
	a1/c alt
	a2/c alt
	…
	alt ak/c
	end
	end
		P1() = a1 -> c -> Skip;
		P2() = a2 -> c -> Skip;
		P3() = a3 -> c -> Skip;
		…
		Pk() = ak ->c -> Skip;
		P4()=P1()[]P2()[]…[]Pk();
		F() = P4();F();

Sequencer	forever do
	a/b; c/d
	end
		P1() = a -> b -> Skip;
		P2() = c -> d -> Skip;
		P3() = P1();P2();
		F() = P3();F();

Fragmented C-element3	forever do
	a1,a2/d
	end
	par
	forever do
	d,a3/c
		P1()=a2->a1->d->Skip;
		P2()=a1->a2->d->Skip;
		P3()=P1()[]P2();
		P4()=d->a3->f->Skip;
		P5()=a3->d->f->Skip;
		P6()=P4()[]P5();
		F1()=P3();(F2()[]F1());
		F2()=P6();(F1()[]F2());
		system()=F1()||F2();

C. Modeling and verification tool

PAT is a generic and extensible framework for supporting composing, simulating and reasoning of concurrent, real-time systems and other possible domains [14]. It implements a number of different model checking techniques catering for different properties such as deadlock freeness, divergence-freeness, reachability, and complete Linear Temporal Logic (LTL) [15] properties. Furthermore, PAT supports customized semantics and stage reduction techniques; and has a friendly graphic user interface. The main modeling language supported in PAT is CSP# process algebra which has high level modeling operators, parallel composition, interleaving, channels, etc.

Assertion-based verification is a methodology that has been dormant for many years and is now widely applied in hardware verification. Besides plenty of modeling features, a number of useful assertions are supported in PAT. Assertions assist to capture the design intent. They monitor behaviors during simulation, detect and report errors. By means of assertions, verification can start in earlier design stage, bugs can be detected and resolved easily, and design engineers can incorporate their intent into programs to minimize integration issues.

Given P() as a process, the basic assertions used are described as follows:

#assert P() deadlockfree: performs Depth-First-Search or Breath-First-Search algorithm to detect the states with no further transitions except successfully terminated states.

#assert P() divergencefree: checks if there is a process performing transitions forever without useful events.

#assert P() deterministic: asks if there is no two out-going transitions with the same events leading to different states.

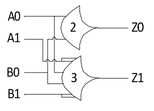
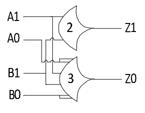
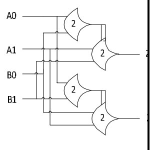
#assert P() nonterminating: Depth-First-Search or Breath-First-Search algorithm is applied to detect the state with no further move, including successfully terminated states.

IV. Case Studies

A. 2NCL expression mappings for basic Boolean logic gates

Though Boolean function cannot avoid the expressional shortcomings, it is a convenient expression to specify the output results of an electronic system by Boolean functions first and then map these functions to 2NCL expressions. Thus it is necessary to seek a way that can effectively transfer Boolean functions to its corresponding 2NCL combinational expressions.

TABLE IV
EXPRESSION MAPPINGS FOR BASIC BOOLEAN LOGIC GATES

	OR	AND	XOR
Boolean Functions	$Z = A + B$	$Z = A \cdot B$	$Z = A \cdot B + C \cdot D$
2NCL Expression	$Z_1=A_1A_0+B_1B_0+A_1B_0+A_0B_1+A_1B_1$ $Z_0=A_0B_0$	$Z_1=A_1B_1$ $Z_0=A_1A_0+B_1B_0+A_1B_0+B_1A_0+A_0B_0$	$Z_1=A_0B_1+A_1B_0$ $Z_0=A_0B_0+A_1B_1$
2NCL Schematic Views			
DISP	forever do select A0,B0/Z0 alt A1,A0/Z1 alt B1,B0/Z1 alt A1,B0/Z1 alt B1,A0/Z1 alt A1,B1/Z1 end end	forever do select A1,B1/Z1 alt A1,A0/Z0 alt B1,B0/Z0 alt A1,B0/Z0 alt B1,A0/Z0 alt A0,B0/Z0 end end	forever do select A0,B1/Z1 alt A1,B0/Z1 alt A0,B0/Z0 alt A1,B1/Z0 end end

CSP#	P1()=A0->Z1->Skip;
	P2()=B0->Z1->Skip;
	P3()=A1->Z1->Skip;
	P4()=B1->Z1->Skip;
	P5()=A0->Z0->Skip;
	P6()=B0->Z0->Skip;
	
	z1()=(P3()||P1())[](P4()||P2())[](P3()||P2())[](P4()||P1())[](P3()||P4());
	z0()=P5()||P6();
	F() = (z1()[]z0());F();
		P1()=A1->Z1->Skip;
		P2()=B1->Z1->Skip;
		P3()=A1->Z0->Skip;
		P4()=B1->Z0->Skip;
		P5()=A0->Z0->Skip;
		P6()=B0->Z0->Skip;
		
		z0()=(P3()||P5())[](P4()||P6())[](P3()||P6())[](P4()||P5())[](P5()||P6());
		z1()=P1()||P2();
		F() = (z1()[]z0());F();
			P1()=A0->Z1->Skip;
			P2()=B0->Z1->Skip;
			P3()=A1->Z1->Skip;
			P4()=B1->Z1->Skip;
			P5()=A1->Z0->Skip;
			P6()=B1->Z0->Skip;
			P7()=A0->Z0->Skip;
			P8()=B0->Z0->Skip;
			
			z0()=(P7()||P6())[](P8()||P5());
			z1()=(P1()||P2())[](P3()||P4());
			F() = (z1()[]z0());F();

As explained in Section II, there must be two signal paths to exclusively express the meaning of True and False in a 2NCL binary system and therefore NULL function should be expressed in addition to desired data function. Take an OR gate as an example. As shown in TABLE IV, the Boolean function of OR gate is Z=A+B, where Z, A and B can be either logical 1 or logical 0. The conventional OR gate implemented using Boolean logic will deliver logical 1 at the output Z if one or both inputs of the gate are asserted logical 1. 2NCL combinational expression, however, presents logical 1 and 0 by a dual-rail signal Z and has to be defined by two individual equations.

Since the generic equation of Z1 is AB+AC+AD+BC+BD, its 2NCL expression can be mapped to a TH34w22 gate. Similarly, the Z1 equation, whose generic expression is AB, can be mapped to a TH22 gate conveniently. In terms of the language based synthesis technology discussed in section III, the behaviors of a 2NCL OR gate can be expressed by a series of selected constructs in DISP. CSP# expressions are easier to use. Input and output behavior can be defined as process by a number of simple events. Then these processes can be organized either sequentially or in parallel manner.

PAT was applied to analyze the phases for such an OR gate model. In Fig. 9(a), a simulation run for the OR gate expressed as a transition graph is shown. Basic circuit properties were also verified. Verification results show that OR gate is deadlock-free, divergence-free, nonterminating and deterministic.

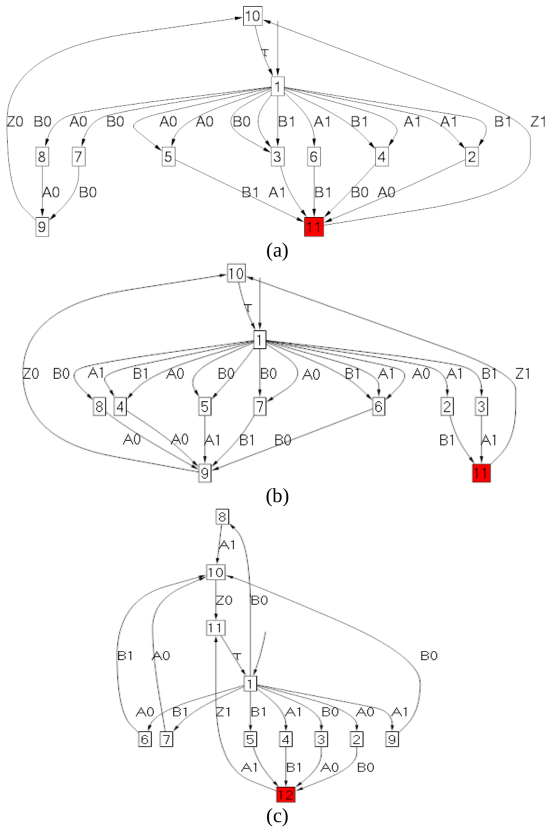


Fig. 9. (a) A simulation run of a 2NCL OR gate as transition graph (simulated by PAT).
(b) A simulation run of a 2NCL AND gate as transition graph (simulated by PAT).
(c) A simulation run of a 2NCL XOR gate as transition graph (simulated by PAT).

2NCL AND and XOR gates can be implemented in a similar way. The Boolean function and 2NCL expressions can be seen in TABLE IV. Fig. 9 (b) and (c) show the simulation results.

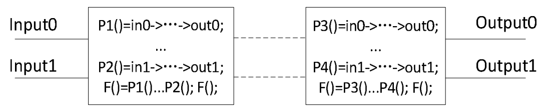


Fig. 10. Connection between modules.

Since PAT well supports stage reduction, it is not hard to connect modules by redefining inputs and outputs even if duplicate event stages exist in expressions. Fig. 10 shows two modules, where basic events and their relationship are defined in the box. The two modules running in parallel can be connected by means of sequential composition. Furthermore, the outputs of the first module should be defined as the inputs of the subsequent module. The example CSP# code is shown as follows.

```
P1()=input0->…->out0;
P2()=input1->…->out1;
F1()=P1()…P2();
P3()=out0->…->output0;
P4()=out1->…->output1;
F2()=P3()…P3();
system()=(F1()||F2());system();
```

B. 2NCL binary half adder design

A half adder is a combinational logical circuit that can perform an addition operation between two binary digits. The result is either 0, 1 or 2 and therefore two bit output terminals (SUM and CARRY) are required to represent the value. Their Boolean function expressions and Karnaugh map are given in TABLE VI and TABLE V respectively.

TABLE V
KARNAUGH -MAP FOR A BINARY HALF-ADDER

SUM	Y_0	Y_1
X_0	Z_0	Z_1
X_1	Z_1	Z_0

CARRY	Y_0	Y_1
X_0	C_0	C_0
X_1	C_0	C_1

TABLE VI
A BINARY 2NCL HALF-ADDER

Boolean Functions	Z = X xor Y
	C = X and Y

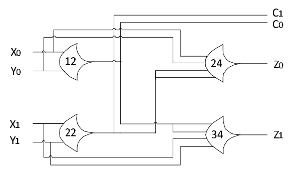
2NCL Expression	$Z0 = X_0Y_0+X_1Y_1$ $Z1 = X_1Y_0+X_0Y_1=X_1Y_0+X_0Y_1+X_0X_1+Y_0Y_1$ $= (X_0+Y_0)X_1+(X_0+Y_0)Y_1$ $C0 = X_0Y_0+X_0Y_1+X_1Y_0=X_0Y_0+X_0Y_1+X_0Y_0+X_1Y_0$ $= X_0+Y_0$ $C1 = X_1Y_1$
2NCL Schematic Views	 C1 C0 X0 Y0 12 24 Z0 X1 Y1 22 34 Z1

TABLE VII
DISP AND CSP# EXPRESSIONS FOR A BINARY HALF-ADDER

DSIP	CSP#
forever do	P1()=X1->Y1->C1->Z->Skip;
select	P2()=Y1->X1->C1->Z->Skip;
select	P3()=P1()[]P2();//C1
X1,Y1/Z0	P4()=X0->Y0->C0->Z->Skip;
alt X0,Y0/Z0	P5()=Y0->X0->C0->Z->Skip;
end	P6()=X1->Y0->C0->Z->Skip;
alt	P7()=Y0->X1->C0->Z->Skip;
select	P8()=X0->Y1->C0->Z->Skip;
C0,X1/Z1	P9()=Y1->X0->C0->Z->Skip;
alt	P10()=P4()[]P5()[]P6()[]P7()[]P8()[]P9();
C0,Y1/Z1	//C0
end	P11()=Y0->X1->Z1->Z->Skip;
end	P12()=X1->Y0->Z1->Z->Skip;
end	P13()=X0->Y1->Z1->Z->Skip;
par	P14()=Y1->X0->Z1->Z->Skip;
forever do	P15()=P11()[]P12()[]P13()[]P14();
select	//Z1
X1,Y1/C1	P16()=X1->Y1->Z0->Z->Skip;
alt	P17()=Y1->X1->Z0->Z->Skip;
select	P18()=X0->Y0->Z0->Z->Skip;
X0/C0	P19()=Y0->X0->Z0->Z->Skip;
alt	P20()=P16()[]P18()[]P19();
Y0/C0	//Z0
end	F1()=(P3()[]P10());(F1()[]F2());
end	F2()=(P15()[]P20());(F2()[]F1());
end	system()=F1()||F2();

Since 2NCL applies dual-rail signals for representing circuit logic, there are two equations for the sum and two equations for the carry. With the aim of simplifying 2NCL circuits, mutually exclusive signals can be inserted to the 2NCL expressions. For example, the original logic expression of Z1 is X1Y0+X0Y1, where a TH22 and a TH23w2 gates are required to deliver the correct result. In case that X0X1 and Y0Y1 are inserted, the logic expression can be mapped to a single TH34w2 gate. According to the dual-rail encoding, the situation where X_0 and X_1 will not be asserted at the same time and thus the operator will not respond to any combinations of X without Y. The schematic view of the optimized 2NCL logic can be seen in TABLE VI. TABLE VII provides the corresponding DISP and CSP# expressions for the binary half adder. A dummy state Z indicates the completeness of the process because the transition graph is complicated and hard to read. Since the logic with mutually exclusive expressions is redundant, only the relevant behavior for the half adder is specified.

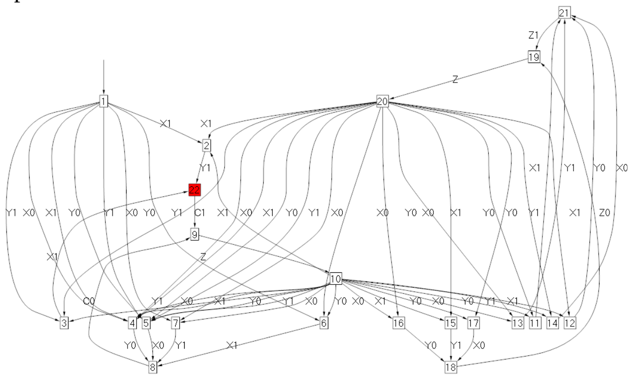


Fig. 11. A simulation run of a 2NCL binary half adder as transition graph (simulated by PAT).

The simulation (see Fig. 11) and verification results (refer to deadlock-free, divergence-free, nonterminating and deterministic) show the correctness of the 2NCL half adder construct.

C. 2NCL asynchronous register with completion detection circuits

Pipeline is widely applied in a variety of digital systems. Asynchronous registers, as well as completion detection circuits, are basic components of 2NCL pipeline. Fig. 12 shows a 2-bit dual-rail encoded 2NCL register with completion detection. Ki and Ko are handshaking signals. When the subsequent cycle finishes computation, the register will be informed by a Ki signal to store the data from a0, a1, b0 and b1. As soon as the output D_a0, D_a1, D_b0, D_b1 receives the data, the completion detection circuits will generate a Ko signal to acknowledge the completeness of this cycle. As seen in Fig. 12, the register is implemented by TH22 gates, which has the same function as Boolean AND gates. The 2-bit output results are:

$$D\_a0 = A_0 \cdot K_i$$
$$D\_a1 = A_1 \cdot K_i$$
$$D\_b0 = B_0 \cdot K_i$$
$$D\_b1 = B_1 \cdot K_i$$

The completion detection circuits are implemented by one TH22 gate and two TH12 gates, and their function can be expressed by the following equations:

$$Ko = \neg (D\_a0 + D\_a1) \cdot \neg (D\_b0 + D\_b1)$$

where ¬ denotes the signal inversion.

In order to make processes convenient for expressing, the logic equation Ko can be re-expressed as ¬(D_a0 +D_a1+D_b0+D_b1) according to the DeMorgan's Law.

TABLE VIII lists the DISP and CSP# expression for this register construct while Fig. 13 shows a simulation run of the register as transition graph (simulated by PAT).

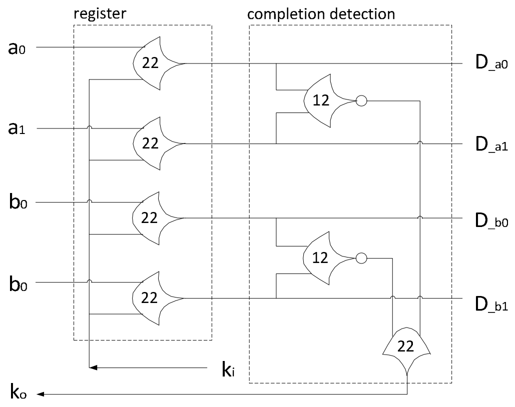


Fig. 12. Schematic view of a 2-bit 2NCL register and completion detection circuits.

TABLE VIII
K-MAP FOR A 2-BIT 2NCL REGISTER AND COMPLETION DETECTION CIRCUITS.

DISP	CSP#

forever do
select
a0,Ki/Da0
alt a1,Ki/Da1
alt b0,Ki/Db0
alt b1,Ki/Db1
end
end
par
forever do
select
Da0/inv
alt Da1/inv
alt Db0/inv
alt Db1/inv
end
end
par
forever do
inv/Ko
end
	P1()=a0->Da0->Skip;
	P2()=a1->Da1->Skip;
	P3()=b0->Db0->Skip;
	P4()=b1->Db1->Skip;
	P5()=Ki->Da0->Skip;
	P6()=Ki->Da1->Skip;
	P7()=Ki->Db0->Skip;
	P8()=Ki->Db1->Skip;
	P9()=(P1()||P5())[](P2()||P6())[](P3()||P7())[](P4()||P8());
	
	P10()=Da0->inv->Ko->Skip;
	P11()=Da1->inv->Ko->Skip;
	P12()=Db0->inv->Ko->Skip;
	P13()=Db1->inv->Ko->Skip;
	P14()=(P10()[]P11()[]P12()[]P13());
	
	F()=(P9()||P14());F();

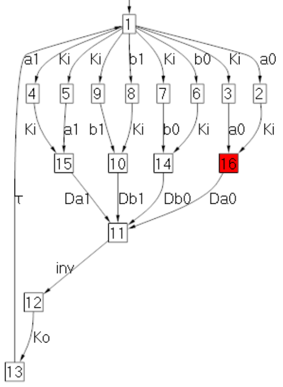


Fig. 13. A simulation runs of a 2-bit 2NCL register and completion detection circuits as transition graph (simulated by PAT).

D. 2NCL pipeline ring

A complex structure normally contains several basic pipeline structures and a pipeline structure is composed of a few of cycles. The term pipeline ring comes to be used to refer to the pipeline whose outputs are connected to from a continuous ring of cycles. Fig. 14 shows a simple ring with three processes. Process I begins by sending a signal R to B. As soon as Process B receives signal R, Process B will begin and will then inform Process A to start after it is completed. In a similar way, signal F will be sent by Process A to initiate Process I again. The three processes I, B and A can be expressed as follows:

Process I: F↑, R↑, F↓, R↓
Process B: S↑, F↑, S↓, F↓
Process A: R↑, S↑, R↓, S↓

Where up arrow (↑) indicates active and down arrow (↓) indicates passive. By defining a series of active and passive events, the behavior of the simple ring can be expressed in CSP# easily as follows.

```
P1()=Fa->Ra->Ff->Rf->Skip;
F1()=P1();(F1()[]F2());
P2()=Sa->Fa->Sf->Ff->Skip;
F2()=P2();(F2()[]F3());
P3()=Ra->Sa->Rf->Sf->Skip;
F3()=P3();(F2()[]F1());
system()=(F1()||F2()||F3()); system();
```

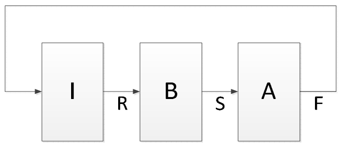


Fig. 14. A simple ring with three processes.

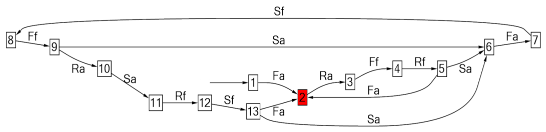


Fig. 15. A simulation run of a simple ring as transition graph (simulated by PAT).

Fig. 15 shows the dependency graph for the three process ring. The event Fa denotes that the signal F is active while Ff denotes that the signal F is passive, and so forth. The verification has proven that the construct is deadlock free.

A 2NCL pipeline ring usually consists of 2NCL registers, completion detect circuits and combinational circuits. A 4-cycle 2NCL pipeline ring can be seen in Fig. 16, where R refers to register, F refers to combinational function block for a single stage and D refers to completion detection component. The half adder described in Section IV.B can be used as a combinational function block. The pipeline consists of 4 functional blocks (e.g. R1, D1 and F1 form Block 1 in Fig. 16). The behavior of each block is described as below:

Block 1: F1↑, R2↑, D2↓, R1↓, F1↓, R2↓, D2↑, R1↑, F1↑
Block 2: F2↑, R3↑, D3↓, R2↓, F2↓, R3↓, D3↑, R4↑, F4↑
Block 3: F3↑, R4↑, D4↓, R3↓, F3↓, R4↓, D4↑, R3↑, F3↑
Block 4: F4↑, R1↑, D1↓, R4↓, F4↓, R1↓, D1↑, R4↑, F4↑

The corresponding CSP# expression is as follows and its dependency graph given by PAT is shown in Fig.17.

```
P1()=D2f->R1f->F1f->R2f->D2r->R1r->F1r->R2r->Skip;
F1()=P1();(F1()[]F2());
P2()=F2r->R3r->D3f->R2f->F2f->R3f->D3r->R2r->Skip;
F2()=P2();(F2()[]F3());
P3()=F3r->R4r->D4f->R3f->F3f->R4f->D4r->R3r->Skip;
F3()=P3();(F3()[]F4());
P4()=F4r->R1r->D1f->R4f->F4f->R1f->D1r->R4r->Skip;
F4()=P4();(F4()[]F1());
system()=(F1()||F2()||F3()||F4());system();
```

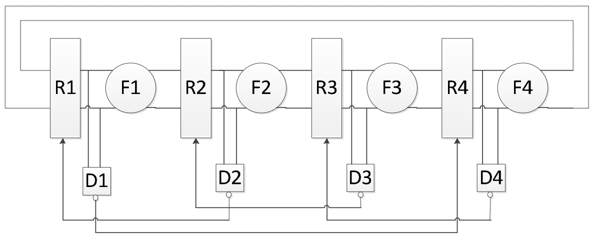


Fig. 16. A 2NCL pipeline ring.

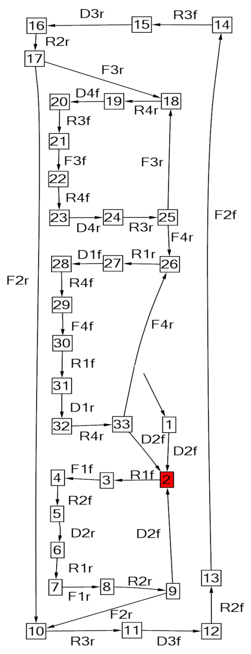


Fig. 17. Dependency graph of the 2NCL pipeline ring (given by PAT).

V. Conclusion

We have shown that our formal models developed for fundamental NCL/2NCL circuits, including logic gates, combinational circuits and pipeline structures, can be generally simulated and verified using PAT. All these inspire us to keep working on this direction to verify more complex formal NCL electronic systems.

However, the formal models of NCL/2NCL circuits presented in this paper are simple and the mappings between

DISP and CSP# are currently limited at the syntactical level only. So, we aim at developing more formal models of NCL circuits including complex systems (e.g. MIPS-based systems-on-a-chip) as well as defining formal translation between DISP and CSP#.

REFERENCES

[1] K. M. Fant, S. A. Brandt, "Null convention logic. A complete and consistent logic for asynchronous digital circuit synthesis," in *Proc. of the International Conference on Application Specific Systems, Architectures*, Chicago, 1996, pp. 261–273.

[2] H. K. Kapoor, A. Asthana, T. Krilavicius, W. Zeng, J. Ma and K. L. Man, "Towards a Language Based Synthesis of NCL Circuits," in *Proc. of the 2011 International MultiConference of Engineers and Computer Scientists*, IMECS 2011, 16-18 March, 2011, Hong Kong pp.1033-1038.

[3] M. B. Josephs, D. P. Furey, "A programming approach to the design of asynchronous logic blocks," in Proc. of *Concurrency and Hardware Design, Advances in Petri Nets*, vol. 2549, London, 2002, pp.34-60.

[4] C. Hoare, "Communicating sequential processes," in *Comm. ACM* 1978, pp. 21- 8.

[5] PAT. (2009). "PAT: Process Analysis Toolkit." Available at: http://www.comp.nus.edu.sg/~pat/.

[6] K. M. Fant, "Logically Determined Design – Clockless System Design with Null Conventional Logic," New Jerey: John Wiley & Sons, Inc, 2005.

[7] J. Sun, Y. Liu, J. S. Dong, C. Chen, "Integrating Specification and Programs for System Modeling and Verification," in: *Proc. of the Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, Wanshington, DC, 2009, pp. 127-135.

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

[9] G. E. Sobelman, K. M. Fant, "CMOS Circuit Design of Threshold Gates with Hysteresis," in *Proc. of the 1998 IEEE International Symposium on Circuits and Systems*, Monterey, CA, 1998, pp. 61-65.

[10] W. C. Mallon, "Theories and Tools for the Design of Delay-Insensitive Communicating Processes," PhD thesis, Dept. of C.S., Univ. of Groningen, The Netherlands, January 2000

[11] M. B. Josephs, J. T. Udding, "Delay-Insensitive Circuits: An Algebraic, Approach to their Design," in *Proc. of CONCUR'90*, 1990, pp. 342–366.

[12] M. B. Josephs, P. G. Lucassen, J. T. Udding, T. Verhoeff, "Formal Design of an Asynchronous DSP Counterflow Pipeline: A Case Study in Handshake Algebra," in *Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'94)*, Salt Lake City, UT, 1994, pp. 206–215.

[13] M. B. Josephs, D. P. Furey, "A Programming Approach to the Design of Asynchronous Logic Blocks," in *Proc. Concurrency and Hardware Design*, 2002, pp.34-60.

[14] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, "Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers," *IEICE Transactions on Information and Systems*, vol. 3(E80-D), pp. 315–325, 1997.

[15] G. J. Holzmann, "The SPIN Model Checker: Primer and Reference Manual", Boston: Addison Wesley Professional, 2003.

[16] M. B. Josephs and H. K. Kapoor, "Controllable Delay-Insensitive Processes," in *Fundamenta Informaticae*, vol.78, No.1, pp. 101-130, 2007.

[17] H. K. Kapoor, M. B. Josephs, D. P. Furey, "Verification and Implementation of Delay-Insensitive Processes in Restrictive Environments," in *Fundamenta Informaticae*, vol.70, No.1-2, pp.21-48, 2006.

Jieming Ma (M'11) received Bachelor of Engineering degree from Nantong University, P.R. China, in 2007. He then received Master of Science from the University of Bristol, UK, in 2010. Currently, he is a PhD student in the department of Computer Science at the University of Liverpool, UK. His research interests include logic design, VLSI, and Low power design methodologies for integrated circuits and System-on-a-Chip (SoC).

Hemangee K. Kapoor received Bachelor of Engineering degree (computer engineering) from College of Engineering, Pune, India, in 1998; Masters of Technology (computer science and engineering) from Indian Institute of Technology Bombay, India, in 2000; and Ph.D. degree (computer science) from London South Bank University, London, UK, in 2004.
Presently she is Assistant Professor at the Department of Computer Science and Engineering at the Indian Institute of Technology Guwahati, Assam, India. Her current research interests include formal methods, process calculi, asynchronous systems, system-on-chip interconnects, network-on-chip design and computer architecture. Dr. Kapoor is a member of the IEEE.