Assertion Based Verification of Multiple-Clock GALS Systems

Rostislav (Reuven) Dobkin, Tsachy Kapshitz, Shaked Flur and Ran Ginosar
VLSI Systems Research Center, Technion—Israel Institute of Technology, Haifa 32000, Israel
rostikd@tx.technion.ac.il
June 2008

Abstract

Standard EDA ABV tools fall short of verifying multiple clock domain systems on chip (MCD SoC), asynchronous systems and Globally Asynchronous Locally Synchronous (GALS) systems. This paper describes a method for verifying asynchronous and multi-clock behavior in such systems using PSL and standard ABV tools. We convert STG (signal transition graphs), a common form for specifying asynchronous behavior, into PSL statements, employ standard ABV tools, and prove complete verification. The proposed ASE (automatic sequence extraction) algorithm was applied to a MCD SoC model that employed a network-on-chip (NoC) for asynchronous inter-modular communications.

1. Introduction

Large systems on chip (SoC) may incorporate multiple modules operating at different frequencies. Moreover, in dynamic voltage and frequency scaling (DVFS) systems, frequency and voltage may dynamically change during operation [1]-[3]. The resulting multiple clock domains (MCD) SoCs are treated as Globally Asynchronous Locally Synchronous (GALS) systems [4][5]. Inter-modular communications in MCD GALS systems are best implemented by asynchronous logic, eliminating multiple synchronization latencies and complex distribution of multiple clocks. Indeed, the ITRS predicts that by 2020 40% of SoC global signaling will be performed asynchronously [6]. However, to reliably employ asynchronous signaling, suitable verification techniques are required.

In a typical design and verification flow, the specification is converted into a design and also into verification statements (e.g. in PSL [7]). The design is typically verified with an 'assertion based verification' (ABV) tool [8]. ABV may be based on either simulation [9][12] or formal verification [13][14] . In addition, advanced ABV supports temporal expression and/or data validity verification (PSL, e-language, System-Verilog, etc.) [7]-[12]. However, this scheme is often limited to clocked designs that employ a single clock, due to language limitations and tool constraints. Thus, verification by ABV is usually inapplicable to MCD systems and to any asynchronous circuits that may be included in the design.

Verification techniques for pure asynchronous logic [15]-[20] mostly employ custom tools, complicating their integration into typical design and verification flows. GALS system verification and test method was discussed in [21], where a special test extension was added to each GALS wrapper. The test extensions disconnect locally synchronous islands during test data transfer between different GALS wrappers, allowing stand-alone massive testing of the wrappers and their interconnections. This technique appears to be more test-oriented. In [22] a GALS wrapper was modeled by Petri nets and verified for reachability and deadlock using model checking [23]. Clock domain crossing (CDC) verification was discussed in [24], where structural and functional synchronizer verification was performed using PSL. These references do not provide a complete verification method for GALS systems.

One common form of specifying asynchronous behavior is based on signal transition graphs (STG) [25] which define untimed ordering of transitions. However, typical ABV tools cannot employ STG for verification of the design. In this work we combine STG specifications and temporal PSL expressions to enable CDC and asynchronous logic verification in MCD GALS systems. First, clock domain crossings and other asynchronous components of the specification are presented formally using STG. Second, an algorithm is presented that converts STG specifications into PSL statements. Third, ABV is performed, using either an artificially generated clock or transition-sensitive verification. We prove that such verification is complete. In this work we address functional ABV only; formal ABV is left for future research.

The paper is organized as follows. In Sect. 2 we describe the applicable STG and PSL properties. The algorithm that converts STG to PSL is presented and analyzed in Sect. 3, and an example of a complex SoC verification is shown in Sect. 4.

2. STG and PSL Properties

In this section we survey the applicable features of STG and property specification language (PSL), providing for the description of the algorithm in Sect. 3.

2.1. Signal Transition Graph (STG)

A module behavior can be described formally with a signal transition graph (STG). An example of STG for a simple latch controller is shown in Figure 1. The STG is a special type of a Petri Net [15]. Tokens are marked by solid circles and their position (marking) determine the circuit state; the token marking in Figure 1, denotes the initial state.

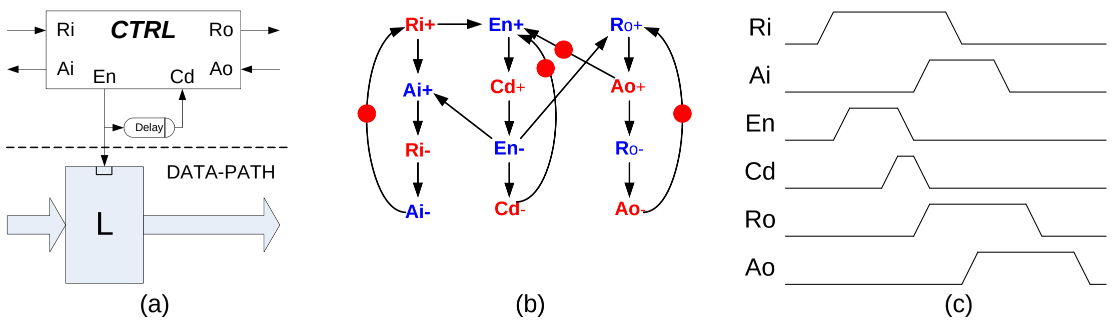


Figure 1: Latch controller example (a) Controller Interfaces, (b) STG, (c) Timing Diagram

Change of state is denoted by moving tokens along directed edges. A transition of node n is enabled when every incoming arc holds a token. When the transition takes place (node n “fires”), all incoming tokens are consumed and new tokens are produced on each outgoing arc. STG may also specify choice and merge conditions, which are not shown in this paper, but can be also treated by the ASE approach.

The STG can be used for logic synthesis, for example using Petrify tool, which also performs formal verification of the synthesized logic [15]. Unfortunately, Petrify cannot be used for large system synthesis and verification. In addition, when gate-level asynchronous design is obtained manually or by tools without formal verifier inside, the verification of internal structure is an essential condition for the design sign-off.

Large STG can be handled by the hierarchical approach employed in this work. A large STG is decomposed into multiple hierarchical levels, and each level is verified separately. Each higher abstraction level hides some of the internal details (signals) of lower levels, maintaining tractable STG sizes.

Note that, in Figure 1, STG tokens circulate in the STG for each new word in a repeating manner. While the cross-relation between the tokens may change for different cycles, the path that a single token goes through is never changed (this is partially true for STGs with choice, where current path is chosen from a certain number of predefined paths according to choice input value)

2.2. Property Specification Language (PSL)

The PSL language provides operators for defining and verifying timed sequences. For example, the following expression employs the '->' and 'eventually!' PSL operators to verify that acknowledge signal AI is asserted each time request signal RI is asserted.

```
property req_ack_in_p is always (RI->eventually! AI);
```

More complex relations can be defined by Sequential Extended Regular Expressions (SERE). A SERE makes it easier to define long sequences, allows re-use of shared sequences and can be used in conditional statements. For example, a simple four-phase handshake protocol (RI+ → AI+ → RI- → AI-) can be defined as follows:

```
sequence hs_init is {not RI; RI};
sequence hs_body is {RI; AI; not RI; not AI};
property sere_examp is always { hs_init } |-> { hs_body };
assert sere_examp;
```

The brackets define sequences. hs_init expresses the initial transition of the sequence (RI+) and hs_body contains the remaining transitions. The sere_examp property uses the "always" operator to specify that it must be valid at all times. The assert statement actually initiates verification of the property. The SERE fails when an unspecified signal sequence is encountered (e.g. RI+→AI-).

These sequence examples do not employ any clock. This is important when verifying multiple clock domains: PSL is defined only for a single clock. This verification code may be used in two ways. First, the ABV may be event-based, and asynchronous transitions are handled at arbitrary times rather than on any external clock ticks. Alternatively, a default verification clock may be defined. In any case, the verification is independent of any clock event ordering of external multiple clocks.

Verification effectiveness is measured by coverage. The next example collects coverage for the sequence hs_body:

```
cover hs_body;
```

3. Automatic Sequence Extraction (ASE)

Our goal is to generate assertion expressions for ABV from system level specification of GALS system. We use STGs for specification and then apply the Automatic Sequence Extraction (ASE) algorithm. ASE decomposes the STG into STC-1, a set of cyclic non-splitting cycles, which are then transformed into PSL assertions. In this section we present all definitions, provide a formal analysis to prove the correctness of the STG decomposition into STC-1 (Sect. 3.1), describe the ASE algorithm in Sect. 3.2 and provide an example in Sect. 3.3.

3.1. STG Decomposition and Complete Verification

Definitions

1. *Signal transition graph* (STG) is a connected directed graph G=(*V*,*E*,*T*), where *V* is a set of nodes representing signal transitions ("+" and "-"), *E* are directed edges showing precedence relations, and *T* is the initial marking ('marking' is a set of edges having tokens on them). The STG follows three sets of rules:
    a. When all edges leading into a transition have tokens, the transition may "fire", the said tokens are consumed and new tokens are placed on all edges emanating from the fired transition.
    b. In this work, STGs are free from deadlocks, 1-bounded (no more than one token per edge), and have only input free choices [15][26]. This also means that there are neither source nodes nor sink nodes in the STG, and every node may be revisited infinitely many times. More general STGs are left for future research.
    c. The STG specifies a speed-independent system, namely it has consistent state assignment (transitions strictly alternate between "+" and "-") and is persistent (enabled transitions must eventually fire) [15][26].

An STG specifies a logic circuit (and possibly its environment) by implying the following 'STG properties':

a. The sets V, E, T.
b. Sequences of transitions that are allowed in the logic circuit (transition sequences are ordered sets of transitions that are defined by paths in the directed STG).
c. Sequences that are allowed to happen concurrently.
d. The STG may also imply choice; this paper does not treat choice.

2. *Simple Cycle* (SC) in STG: A sequence of transitions that starts and ends at the same transition and does not contain any transition more than once (except for the first transition).
3. *Signal transition cycles* (STC): A STG decomposed into a set of Simple Cycles. The decomposition extracts all existing SCs for a given STG. The original STG may be reconstructed by combining the SCs of the STC back together.
   a. Note that a transition sequence that exists in the original STG may consist of segments that belong to different SCs in the STC.
   b. STC-1 is an STC where each SC contains exactly one token.
4. *Concurrent transitions* may happen in any order. Two transition sequences are concurrent sequences if any transition from one sequence is concurrent to all transitions of the other sequence. This implies:
   a. They share a starting common transition and an ending common transition.
   b. Other than the starting and ending transitions, they do not share any other common transitions, and there are no cross paths from one sequence to the other.

   Concurrent sequences are concurrently enabled when the starting common transition has fired (enabling the concurrent sequences) and the ending common transition has not yet fired [27].
   For the purpose of verification, the specification STG is converted into STC-1, and the STC-1 is converted into PSL assertions.
5. *Verification*: proving that a set of rules (PSL assertions) is fulfilled by the logic circuit ("design under verification", DUV). In other words, the DUV conforms to the PSL assertions. Verification may be performed by either a simulation-based ABV tool or formally.
6. *Complete verification* is satisfied if:
   a. The DUV is verified, namely it fulfills the PSL assertions.
   b. The PSL assertions cover all STG properties of the specification.
   c. Consequently of a+b, the DUV fulfills at least all transition sequences allowed by the STG [15].

Complete verification is demonstrated in Figure 2. Condition *a* is guaranteed by the ABV tool. Condition *b* is proven by the Claim below, and by discussing the ASE algorithm in Sect. 3.2 and 3.3 below.

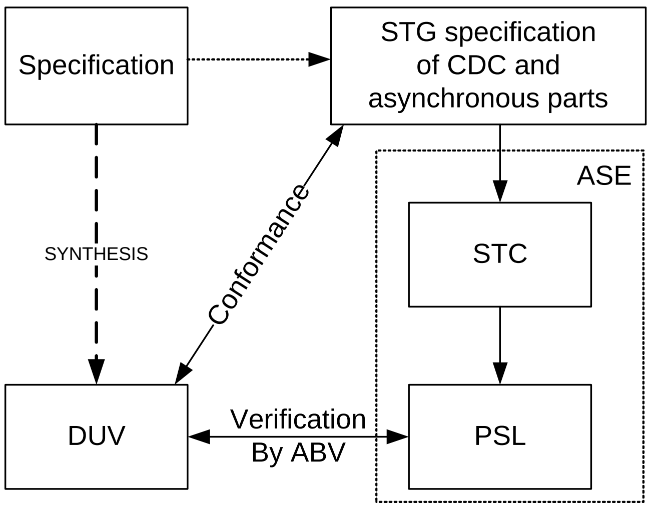


Figure 2: A specification may be designed and synthesized into a DUV. The CDC and asynchronous part of the spec is presented as STG. The STG is converted into PSL assertions (via STC) by the ASE algorithm, and the DUV is verified. This implies that the CDC and asynchronous parts of the design conform to the STG

The STG is decomposed into a STC-1, and we prove that:
Claim: The STC-1 preserves all STG properties.

The Claim is proven by means of the following lemmata. The proofs are given in the Appendix.
Lemma 1: The STG always includes a SC that contains both "+" and "-" transitions of the same signal (S+, S-).
Lemma 2: The STC-1 covers all STG transitions.
Lemma 3: The STC-1 covers all STG edges.
Lemma 4: Each SC has at least one token.
Lemma 5: A SC with more than one token is redundant and is covered by a set of SCs with single token (STC-1).
Lemma 6: The STC-1 preserves the initial marking of the STG.
Lemma 7: The STC-1 preserves the concurrent branches of the STG.

3.2. ASE Algorithm

The ASE algorithm is applied to a reduced STG (which contains no redundant edges). The search is performed following real token flow inside the STG. The cycle search starts from a token position and forks each time there is a split in STG. For each search path, each transition is visited only once, and thus the complexity of a single search path is linear in the size of STG. The total algorithm complexity is linear in the size of the STG, in number of initial tokens and in number of splits inside the STG. Instead of the mentioned approach a standard DFS algorithm can be used. Note that all algorithms can use early completion condition once CC (complex cycles) are found. The algorithm main steps are:

1. Find all SCs in the STG. Add the SCs to a STC.
2. Remove from the STC all SCs with multiple-tokens, resulting in STC-1.
3. Remove from the STC-1 all SCs that can be covered by a combination of other SCs from the STC-1.
4. Re-arrange the transitions inside each SC such that the first transition holds the token.
5. Convert each SC into a PSL assertion (as demonstrated below).

3.3. Algorithm Example

For demonstration, the algorithm is applied to the STG of Figure 1b. The STG is decomposed into four SCs shown in Figure 3. The extracted SCs are mapped into SEREs. To verify that only one transition happens at a time in each SC, the signal transitions are transformed into predicates that represent mutually exclusive events. These events are internally implemented by pulses, because it is easier to express mutual exclusion of pulses in PSL. For instance, a pulse is defined for each rising edge and another pulse for each falling edge of each signal, and the event S+ is replaced by the predicate “S+ and none of the other transitions of the same SC”, where each component is actually represented by pulses or lack thereof. Similarly a ‘transition complete’ predicate (TC) specifies no transitions, and it is inserted after each transition predicate to verify that the transition is complete before the next one starts. Note that the pulse width is chosen to be finer than the gate timing resolution, avoiding transitions during TC toggling. We have also studied other predicate types (e.g. without TC). However, providing a similar functionality, the other approaches lead to larger PSL expressions.

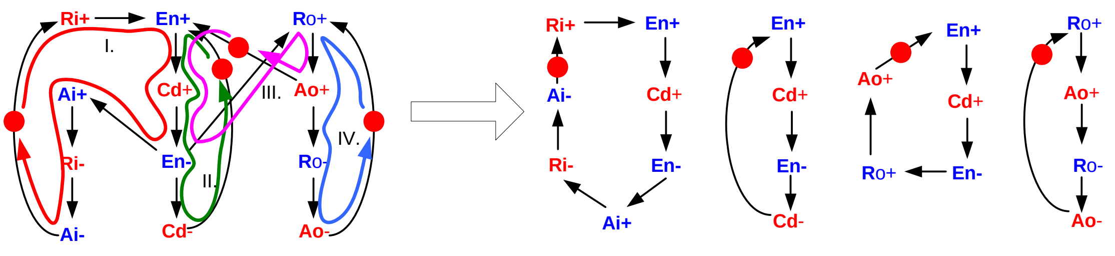


Figure 3: Extraction of Simple Cycles from STG

SEREs are constructed using only the predicates. First the algorithm generates SEREs representing the initial conditions and the SCs:

```
sequence cycle1_i  is {not Ri_r; Ri_r);
sequence cycle23_i is {not En_r; En_r);
sequence cycle4_i  is {not Ro_r; Ro_r);
sequence cycle1_c is {Ri_r; TC; En_r; TC; Cd_r; TC; En_f; TC;
                       Ai_r;  TC;  Ri_f;  TC;  Ai_f;   TC;  Ri_r};
sequence cycle2_c is {En_r; TC; Cd_r; TC; En_f; TC; Cd_f; TC; En_r};
sequence cycle3_c is {En_r; TC; Cd_r; TC; En_f; TC; Ro_r; TC; Ao_r; TC; En_r};
sequence cycle4_c is {Ro_r; TC; Ao_r; TC; Ro_f; TC; Ao_f; TC; Ro_r};
```

Next the initial condition and cycle sequences are combined into properties, asserted and monitored for coverage:

```
property cycle1_p is always {cycle1_i}  |-> {cycle1_c};
assert cycle1_p; cover cycle1_c;
property cycle2_p is always {cycle23_i} |-> {cycle2_c};
assert cycle2_p; cover cycle2_c;
property cycle3_p is always {cycle23_i} |-> {cycle3_c};
assert cycle3_p; cover cycle3_c;
property cycle4_p is always {cycle4_i}  |-> {cycle4_c};
assert cycle4_p; cover cycle4_c;
```

Note that any transition that is not specified in the original STG will be reported as failure by functional verification of the PSL expressions obtained by ASE.

4. SoC/NoC Verification Example

The ASE algorithm and a simulation-based ABV were applied to verify the SoC of Figure 4. The SoC contains locally synchronous modules each having its own local clock, interconnected by an asynchronous network-on-chip (NoC). The interfaces between the modules and the network comprise input and output ports (IP and OP) and provide synchronization and handshake, following [28]. The asynchronous NoC consists of links and asynchronous routers [29] and employs packet-based communication and wormhole routing [30].

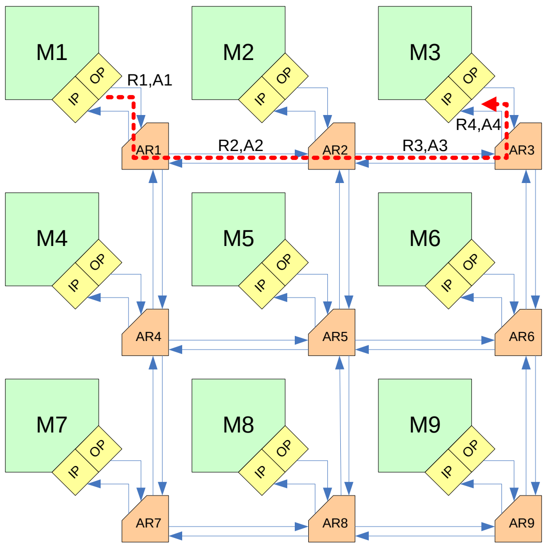


Figure 4: Design example. Multi-clock domain SoC with NoC

An example segment of the SoC (Figure 5) consisting of a sender module, three intermediate routers, and a receiver module. The STG specification of the port asynchronous controllers is shown in Figure 6a,b. The asynchronous routers employ a four-phase bundle data protocol (Figure 6c). A new data flit is sent out for each new R2+ event.

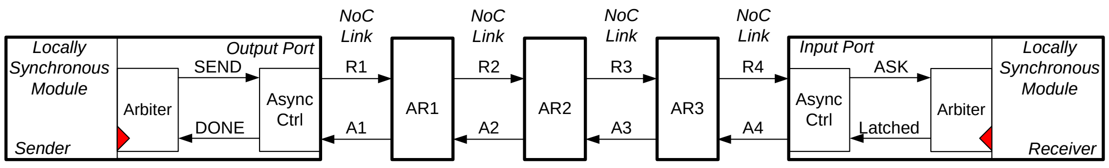


Figure 5: Verification Example: Module and NoC interfaces

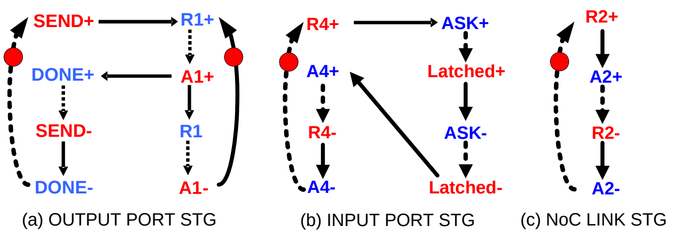


Figure 6: STG specifications

The NoC traffic pattern is usually known for each specific application, allowing optimizing the NoC in terms of buffers and links. We specify each point-to-point NoC traffic path by means of STG; Figure 7 shows an example path from M1 to M3. The example STG represents 2-deep buffers (e.g. R1+, R1+/1).

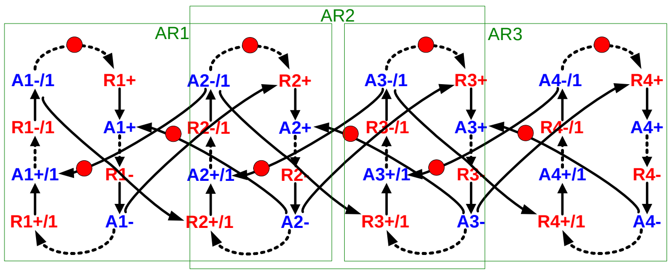
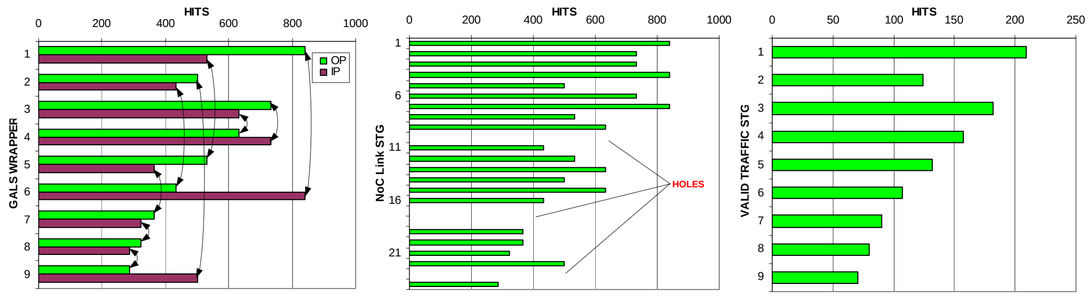


Figure 7: Traffic path specification for 2-buffer routers

The specification STGs were converted into PSL assertions by ASE, and the circuits were successfully verified using NC-Sim ABV. Coverage results for the input and output ports are shown in Figure 8a. The results are correlated through traffic addresses as shown by the arrows, thanks to the fact that in this example each output port always sends packets to the same destination.

Coverage results for NoC links are shown in Figure 8b. The coverage holes may be repaired by extending the verification cases as needed. In this example we defined nine valid traffic patterns, whose coverage is shown in Figure 8c.



(a) GALS wrapper port coverage results (b) NoC link coverage results (c) NoC traffic coverage results

Figure 8: Verification example coverage results

5. Conclusions

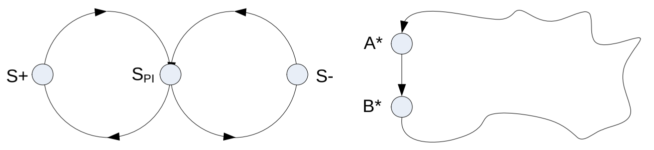
A novel verification approach for multiple-clock domain (MCD) GALS systems was presented. Previously, the asynchronous aspects of MCD GALS systems could be specified with STG, but it was impossible to verify the design against the STG using standard ABV. Further, normal ABV could verify PSL expressions, but they were practically limited to single-clock systems, and it was impossible to verify MCD systems. This work overcomes these difficulties by converting STG into PSL and employing standard ABV to achieve verification of MCD GALS systems.

In this work the asynchronous aspects of the MCD GALS system are specified by an STG. The STG is converted into PSL statements by the novel Automatic Sequence Extraction (ASE) algorithm. A correctness proof for the transformation was provided. In addition, a verification example is shown which applies the new technique to a complex MCD SoC that uses NoC (network on chip). Future research includes extending this approach to timed and choice STG as well as formal verification.

APPENDIX

Proof of Lemma 1:

Since the STG is consistent, for any signal S, the STG contains both a path from S+ to S- and a path from S- to S+. Otherwise S+ and S- are independent and can happen concurrently. The combination of these two paths is thus cyclical. We now prove that the set of such possible cyclical paths contains at least one SC. Assume on the contrary that this set does not contain any SC. Then for each cyclic path p in the set there is always at least one transition Sp that appears at least twice on p (its crossing point). This crossing point divides p into two cycles where S+ and S- belong to different cycles (Figure 9a). Therefore, S+ can happen concurrently with S-. Since the STG is consistent, this situation is impossible. Hence there exists at least one SC that contains both S+ and S-. QED.



(a) Crossing Point (b) Edge (A*, B*) implies a path (B*,A*)

Figure 9: Proof of Lemma 1 and 3

Proof of Lemma 2:

Since for each signal S∊V, there exists a SC that contains S+ and S- (Lemma 1), all transitions exist in the STC. Hence, the STC covers all transitions of the original STG. QED.

Proof of Lemma 3:

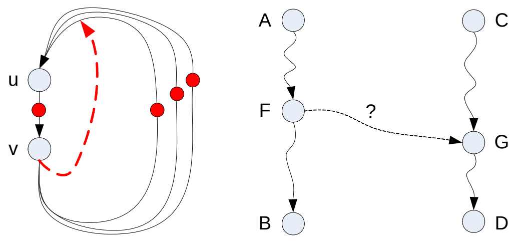
Consider two signals A and B, where the STG contains an edge (A*,B*) ('*' is either '+' or '-'). Observe Figure 9b, which demonstrates that for the given edge (A*, B*) there must also exist the return path (B*, A*), or else A* may be enabled twice before B* happens, in contrast with 1-boundedness of the STG. Since (A*,B*) is a single edge that cannot be crossed, the edge (A*,B*) and the path (B*,A*) constitute a SC. Thus, the STC covers all edges of the STG. QED.

Proof of Lemma 4:

A SC has at least one token otherwise none of its transitions ever fires. This would contradict the property that every node in the STG can be fired infinitely many times. QED.

Proof of Lemma 5:

Assume that an edge (u,v)=e∊E is covered only by SCs that have more than one token (the edge is covered by at least one SC according to Lemma 3). When u fires, a single token from each of the covering SCs is consumed, resulting in a single token on e. At this state the SCs are left with at least one more edge eT≠e that has a token (Figure 10a). The propagation of that token towards u can be blocked only by a path from v that would bring the token from edge e to the blocking transition (dashed line in Figure 10a). Such a path would result in a SC with a single token, contradicting the assumption. On the other hand, if the propagation is not blocked this could result in multiple tokens on edge e, contradicting 1-boundness and consistency. Thus, the assumption is incorrect and there exist at least one SC that does not have more than one token. Considering Lemma 4, we conclude that for each edge e there is at least one SC covering e with exactly one token. Thus, all transitions and edges can be covered by single token SCs (designated STC-1), and SC with multiple tokens are redundant and can be discarded. Since STC is covered by STC-1, Lemmata 2,3 are also applicable for STC-1. QED.



(a) Single token SC covers each edge (b) There are no cross paths between concurrent sequences

Figure 10: Proof of Lemma 5 and 7

Proof of Lemma 6:

The ASE algorithm does not add new tokens to the STC-1. It only copies existing tokens, with the edges they belong to (all edges are preserved, Lemma 3). Therefore, the markings of STC-1, containing SCs with single token each, match the initial marking T of the STG. QED.

Proof of Lemma 7:

Since the STC-1 covers all transitions and edges of the STG (Lemmata 2, 3), all transition sequences that exist in the original STG exist also in STC-1 (sequences may span multiple SCs). Assume that STC-1 does not preserve concurrency. Then STC-1 must contain at least one pair of concurrent sequences (A, B) and (C, D) which have a cross path between two intermediate transitions F, G such that $F\in(A,B)$, $G\in(C,D)$ (Figure 10b). The path (F,G) does not exist in the original STG and since STC-1 does not add any new transition or edges, such path cannot exist in the STC-1. Thus, concurrent sequences of the STG are also concurrent in the STC-1. QED.

Proof of the Claim:

The combination of lemmata 2, 3, 5 proves that the original STG is fully covered by STC-1 (STC-1 contains the sets V, E). Since STC-1 does not add any additional transitions or edges, the sequences of transitions are also preserved. In addition, STC-1 preserves the initial marking T (Lemma 6) and sequence concurrency (Lemma 7). Thus, STC-1 preserves all the STG properties listed in Sect. 3. EOP Claim.

References

[1] G. Semeraro, D.H. Albonesi, S.G. Dropsho, G. Magklis, S. Dwarkadas, M.L. Scott, "Dynamic frequency and voltage control for a multiple clock domain microarchitecture," IEEE/ACM Int. Symp. on Microarchitecture, 356-367, 2002.

[2] L. S. Nielsen, C. Niessen, J. Sparsø, C. H. van Berkel, "Low-power operation using self-timed and adaptive scaling of the supply voltage," TVLSI Systems, 2(4):391-397, 1994.

[3] W.R. Daasch, C.H. Lim, G. Cai, "Design of VLSI CMOS Circuits Under Thermal Constraint," TVLSI Systems, 49(8):589-593, 2002.

[4] D.M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems," PhD Dissertation, Stanford University, 1984.

[5] D. Bormann, P. Cheung, "Asynchronous Wrapper for Heterogeneous Systems," ICCD, 307-314, 1997.

[6] International Technology Roadmap for Semiconductors (ITRS), 2005, www.itrs.net.

[7] IEEE P1850 Standard for PSL, Property Specification Language. http://www.eda-stds.org/ieee-1850.

[8] B. Wile, J.C. Goss, W. Roesner, "Comprehensive Functional Verification: The Complete Industry Cycle," Morgan Kaufmann, 2005.

[9] E-language, http://www.cadence.com/partners/industry_initiatives/e_page/index.aspx.

[10] System-Verilog language, http://www.systemverilog.org.

[11] Synopsys, Vera. http://www.synopsys.com/products/vera/vera.html.

[12] Mentor, Modelsim. http://www.model.com.

[13] E.M. Clarke, O. Grumberg, D. Peled, "Model Checking," MIT Press, 1999.

[14] RuleBase, http://www.haifa.il.ibm.com/projects/verification/RB_Homepage.

[15] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," IEICE Trans. on Information and Systems, E80-D(3), 315–325, 1997.

[16] D.L. Dill, S.M. Nowick, R.F. Sproull, "Automatic verification of speed-independent circuits with Petri net specifications," Int. Conf. on Computer Design: VLSI in Computers and Processors, 212-216, 1998.

[17] D. Borrione, M. Boubekeur, L. Mounier, M. Renaudin, A. Sirianni, "Validation of asynchronous circuit specifications using IF/ CADP", VLSI-SOC, 85-100, 2003.

[18] J. Gong, E.M.C. Wong, "Verification of asynchronous circuits with bounded inertial gate delays," 7th Asian Test Symposium, 399-401, 1998.

[19] T.W.S. Lee, M.R. Greenstreet, C.J. Seger, "Automatic Verification of Asynchronous Circuits," Design and Test of Computers, 1995.

[20] D.L. Dill, Trace theory for automatic hierarchical verification, MIT Press, 1988.

[21] F.K. Gurkaynak, T. Villiger, S. Oetiker, N. Felber, H. Kaeslin, W. Fichtner, "A functional test methodology for globally-asynchronous locally-synchronous systems," ASYNC, 181-189, 2002.

[22] S. Dasgupta, A. Yakovlev, "Modeling and Verification of Globally Asynchronous and Locally Synchronous Ring Architectures," DATE, 568-569, 2005.

[23] V. Khomenko, "Model checking based on Petri net unfolding prefixes," PhD thesis, School of Computer Science, University of Newcastle upon Tyne, 2002.

[24] T. Kapschitz, R.Ginosar, "Formal Verification of Synchronizers," CHARME, 363-366, 2005.

[25] T.A. Chu, C.K.C. Leung, T.S. Wanuga, "A Design Methodology for Concurrent VLSI Systems", ICCD, 407-410, 1985.

[26] Sparsø and Furber, Principles of Asynchronous Circuit Design, Kluwer Academic Publishers, 2001.

[27] Y. Wolfsthal, M. Yoeli, "An Equivalence Theorem for Labeled marked Graphs," Trans. on Parallel and Distributed Systems, 5(8):886-891, 1994.

[28] R. Dobkin, R. Ginosar, C. P. Sotiriou, "High Rate Data Synchronization in GALS SoCs," TVLSI, 14(10):1063-1074, 2006.

[29] R. Dobkin, V. Vishnyakov, E. Friedman, R. Ginosar, "An Asynchronous Router for Multiple Service Levels Networks on Chip," ASYNC, 44-53, 2005.

[30] W.J. Dally, A VLSI Architecture for Concurrent Data Structures, Kluwer Academic Publishers, 1987.