A Novel Time Independent Asynchronous Communication Protocol & Its Applications

Pj Radcliffe and Xinghuo Yu (Senior Member IEEE)
RMIT University
Melbourne 3001, Australia
pjr@rmit.edu.au x.yu@rmit.edu.au

Abstract – **This paper proposes a novel communications protocol called Time Independent Asynchronous (TIA) communications. This protocol constitutes a new category which has unique properties very useful in a variety of applications including embedded controller communications. The 2-wire TIA communications system proposed is implemented using software controlled IO. Analysis of this system shows that traditional Signal Transition Graphs (STGs) may fail to predict livelock and deadlock in software based systems. A modified form of STG called STG For Threads (STG-FT) is proposed to better model the behaviour of software driven systems and is shown to correctly detect livelock and deadlock that a normal STG model may miss. The performance of the new 2-wire TIA system is reported and livelock and deadlock properties found to match the STG-FT simulation. The new 2-wire TIA communication system has particular application to communications in products and industrial systems with low end microprocessors and any microprocessor that is heavily loaded with time critical applications.**

I. INTRODUCTION

More and more processors are being used in industrial and consumer products For example, the 2005 7-series BMW and S-Class Mercedes each contain about 100 processors [1]. According to Garner research in 2002 the average American home had 200 microprocessors [2]. Many products and systems contain multiple microprocessors and inter-processor communications represents a financial cost [3]. Any method that can reduce the cost of the communications link has the potential to reduce the cost of the overall product.

An ideal low cost communications system uses no specialised peripherals, a minimum number of IO pins, and has minimal impact on any real time tasks [4]. To the best of our knowledge, there is no bidirectional communications system that satisfies these constraints with only two pins. Such a system would be very useful to any low end microprocessor and any microprocessor that is heavily loaded with real time applications.

In this paper we develop a novel, highly economic solution to this problem particulary applicable to small to medium enterprises where cost is a major concern. We propose a new category of communications called Time Independent Asynchronous (TIA) communications. This is a distinct category from asynchronous communications and has a range of useful and unique attributes. We have found that the Petri net based Signal Transition Graphs (STGs) do not correctly predict livelock and deadlock behaviour and so we have developed a modified form we call STG-FT (STG For Threads) to better model software implementations of asynchronous systems.

A 2-wire TIA communications protocol is developed that uses two microprocessor pins and has minimal impact on any existing real time code. An analysis of livelock and deadlock is given using the STG-FT model and the simulator we have developed.

This paper is organised as follows : Section II details the new proposed communications protocol and discusses the issues in relation to TIA, section III outlines the problems faced while applying a standard STG model. Section IV describes a modified form of STG that better models software driven systems and section V gives an example which shows how the STG model fails and the new model successfully predicts behaviour. Section VI reports on the modelling and performance of the proposed 2-wire TIA system.

II. TIA COMMUNICATIONS

We propose a new category of asynchronous communications called Time Independent Asynchronous (TIA) communications. Unlike many asynchronous communications methods, we adopt a simple rule that there are no restraints on timing, only the order of signals is important. Unlike Time Free communications [5], gross error conditions such as a host lockup may be corrected with time-outs. For normal signalling there are no timing requirements on an individual signal, nor any timing relationship between signals. Adopting this simple rule has several important implications-

- A host may work as fast or as slow as it likes, and change speeds arbitrarily without causing problems. This can be useful when a host can be interrupted with application or operating system tasks or has low processing power. Another use is when a system is intermittently unavailable as can be found in some power saving strategies.
- The response time to signal changes does not matter and as a result a host does not have to make any guarantees about response time to a signal change. This enables TIA communications to be run at background level which often has considerable CPU time available but on an intermittent basis. Time critical applications or operating system programs are only marginally effected by the TIA communications running in the background.
- TIA can be efficiently implemented with general IO pins which is useful when there is no dedicated communications or special purpose hardware available.
- TIA can cope with a communications medium that distorts timing though not to the stage where signals

are delivered out of order.

- Data overrun of the communications system becomes impossible as a host can simply stop receiving until it is ready to resume.

A small number of existing communications protocols are TIA in nature but most require many signals wires and so are impractical for embedded processor use. Example TIA systems include the old DEC Unibus [6] and the basics of the GPIB bus [7]. TIA systems are used with VLSI chips [8], Yakalov et al [9] propose a 4 wire system, Bainbridge and Furber [10] developed a 3 wire system. Takahashi and Hanyu propose a multi-level logic based system [11] but this does not suit simple digital logic. We have developed a new and novel 2-wire bidirectional TIA bus that only requires 2 pins on the microcontroller and particularly suits embedded controller communication.

III. STG PROBLEMS

Our 2-wire TIA protocol was conceived using a timing diagram but it was clear that complex behaviour could result from variations in host timing. A Petri net style simulation was essential to discover key properties such as deadlock, livelock and correctness. The modelling could have been performed using state machines but as Cortadella, Yakovlev et al explain [12] "(FSMs) cannot explicitly express the notions of concurrency, causality and conflict. Petri nets can naturally capture these notions." Signal Transition Graphs such as those used by Yakovlev [13] and Kishinevsky [14] are a form of Petri net (PN) that are useful for modelling asynchronous logic systems and would appear to be very suitable for modelling the 2-wire TIA system. Key STG constraints [15] include the 1-safety requirement, a free choice PN structure, transitions only have binary triggers, and ordinary tokens (no values or colours). An STG can be mapped onto one or more Finite State Machines (FSMs) for the purposes of implementation.

We found that the STG model did not properly model a system implemented with software rather than asynchronous logic. The main reason was that STGs did not model behaviour at the atomic level. We also found problems related to state allocation and automatic translation to code.

Atomic behaviour: There are differences at the atomic (indivisible) level of behaviour between the asynchronous logic view of STGs and operation of a system implemented with several software threads.

- For a thread based system a transition that tests an input is polled : it occurs at an instant in time and is then not rechecked until the software thread again executes the test instruction. There is a dead zone in which a transition may be true but not acted upon. Other transitions implemented in other FSMs that come true later, may fire before the first transition because of these polling delays.
  Modeling must be at an atomic (indivisible) level of activity to properly capture the range of behaviors of the thread based system.
- Asynchronous logic can respond very quickly to one of several possible transitions firing whereas a thread based system on one CPU can only consider one transition at a time. Multiple test transitions from a single STG place are polled : they are tested sequentially with dead zone periods between each test. The transition to fire may not be the first one to come true due to the polling sequence and delays. Again modeling must describe these tests at an atomic (indivisible) level to properly capture the range of behaviors of the system.
- STG transitions are triggered by rising or falling edges of signals [15]. With the aid of dedicated hardware a software thread may detect edge changes but in general a software thread can only test for high or low.
  A software thread may also implement very complex tests where many lines of code result in a true or false result.

State allocation: minimizing of the number of states is an important topic in the design of asynchronous logic [13][14][15] and can result in fewer gates. When using software threads state variables and other data are effectively free as memory is inexpensive. A penalty of being too free with states and data comes when simulation is executed. If the reachability tree (signal graph) is to consider all possible system states then state explosion may occur. If the reachability tree is examining all possible operational paths then states and variables may produce numerous initial states which can again cause a state explosion.

Automatic translation: Any translation requiring human judgement and expertise provides an opportunity for errors to creep in. Ideally the translation from one form to another should follow a strict set of rules that can be automated, or at least followed in an automated manner by a human being.

An STG may be mapped onto FSMs and this has been automated in tools such as Petrify [16]. Once an FSM is available it can be translated automatically into the code structure of a programming language. State tables are a convenient solution as the driver code, the state table, and the empty functions for action and test routines can be automatically generated. The user must add the code inside each action and test routine.

Standard STGs face several problems when applied to a system that must be implemented using several FSMs run by software threads-

- The splitting of an STG in to several FSMs is difficult to automate.
- The traditional labeling does not differentiate input and output signals which can cause readability problems and so encourage errors.
- The traditional poor naming conventions inhibits readability and also encourages errors.

IV. STG FOR THREADS (STG-FT) SOLUTION

We propose a new and novel form of STGs we call “STG For Threads” (STG-FT) that will accurately model a system implemented using software threads. The following process will convert an STG model into an STG-FT model.

1. Atomic modelling: The first set of constraints to apply to an STG model aim to enforce atomic level modelling so as to properly capture the behaviour of a software thread based system.

- Atomic transition : each transition may represent an atomic test or action but not both (the terms test and action have been chosen to avoid confusion with the terms input and output which define relationships between places and transitions). An atomic transition must operate at one instant in time, or at least sufficiently fast that no part of the system can see any intermediate results and the active FSM cannot see any change in the system.
- Atomic output transitions: consider the output transitions of a place. An STG-FT may have only one output transition for a place, or if the output transition is a test transition "Y/" then there may be one additional test transition "!Y/" which is the inverse of the first test.
- An atomic action transition: this is a transition that defines an output or a change to data. An action transition always fires if there is a token in its input place. It is labeled as follows-

  / output_name

  An action transition effecting output pin A then pin B labeled " / set_A_clr_B" would not be atomic. It must be split into to sequential action transitions labeled "/set_A" and "/clr_B", or the reverse order as appropriate (see Fig. 1).
- An atomic test transition: this is a transition that performs a simple boolean test that can be made at one instant in time from the point of view of the FSM containing the test. For example the test may consider multiple data structures providing they cannot change during the period spent executing the test. If the test is true and the input place contains a token then the transition fires. An atomic test is labeled as follows-

  atomic_test_name /

  A single test transition labeled "strobe_lo_reset_hi/" which looks at two external inputs would not be atomic . Two tests leaving the one place, labeled strobe_lo/ and the other reset_hi/, would not be atomic. The test must be broken into two atomic tests as per figure 2. The order of testing is important and may effect operation.

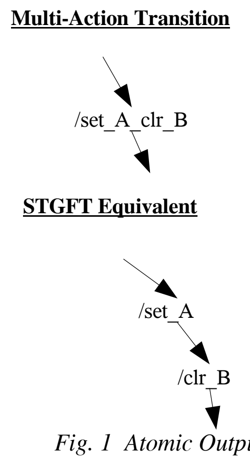

Fig. 1 Atomic Output Transitions

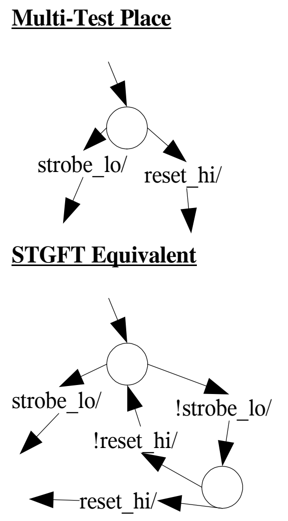

Figure 3 Multiple Test Implementation

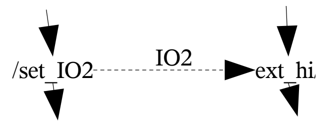

Figure 2 Inter-FSM Communication

2. FSM translation: The next set of constraints to apply to an STG model aim to make it easy to translate the STG-FT into multiple FSMs.

- FSM object: an STG-FT may model several FSMs and places and transitions are grouped into FSM objects each of which will implement one FSM. FSM objects, as with software objects, should have good encapsulation. Each FSM object should have a clear purpose and have a minimal and well defined interface to the rest of the system. A dotted ring may be used to denote an FSM object.
- Place = state: a place (a vector under STGs which may contain a circle, and may show an initial marking) represents a single state in an FSM. Only one place in an FSM object may contain a token.
- Place naming: each FSM object has its own name. Each place has its own number and may have a descriptive name as well.

  For example X2 : wait_for_busy

  X is the FSM name, 2 is the place number within the FSM object, and wait_for_busy is an optional descriptive name.

3. Inter-FSM communication: Once places and transitions are arranged into FSM objects there will be some places and transitions that link the FSM objects. In reality these communication between FSMs often represent an IO pin, or information sent via a media. Such a link is eliminated by placing an action transition in the source that effects a variable which may be as simple as a boolean IO pin, or a complete data structure. There will be one or more test transitions in the destination FSM object that can test the resulting variable. None of these variables have any state transition behaviour of their own and depend on the FSMs to change values. Variables may depend on each other and be modelled with boolean logic.

The eliminated linkage place and transition may be represented as a dotted arrow from the source action transition to the destination test transition and this represents a cause-effect relationship (see Fig. 3). The dotted arrow is an optional readability aid rather than an artefact to guide automatic implementation.

All variables must have a known value at the start of a simulation. If they can have different values then the simulation must be run for every possible data combination of all variables. The number of variables and the range of values they can take should be minimised to avoid state explosion.

FSM objects may be able to directly test the state variable of another FSM and so avoid the need for an extra variable. A dotted arrow may still be used to indicate a cause-effect relationship.

The procedure outlined should have produced an STG-FT model that will accurately model a software based implementation of a system.

V. STG-FT EXAMPLE

This example will show how an STG which is not fully in STG-FT format, will not accurately model a software implemented system. The procedure for creating STG-FT is applied and the resulting model does correctly predict behaviour.

Consider the waveforms in Figure 5. The unit to be designed must read a clock and a wait signal and generate a data bit from some internal source. If the wait signal is high when the clock goes high then the data must stay stable until after the next rising clock edge where wait is low.

The first pass STG-FT is shown at the top of figure 4. The STG-FT with Sx labelling is the data source that must be created. The diagrams labelled with Cx (clock generator) and Wx (wait generator) implement signal generators that are only used in simulation and as such they can break the atomic operation rule. They allow for the generation and testing of one cycle of wait.

We have developed an STG-FT simulator which has been used to simulate the problem. The simulator examines the full reachability tree from every initial condition through every possible valid path across all state machines. This results in full detection of livelock, deadlock and correct or incorrect operation. The translation from STG-FT to software code in the simulator is automatic in nature except that the user must create the initial conditions, the code that implements each action and test transition, and the conditions which terminate the simulation.

The simulation will show no problems and indicate correct operation with no livelock and deadlock for all possible sequences of operation. When implemented using software on CPUs the system will fail. Problem scenarios include-

- wait just before clock : the CPU polls wait and finds it low, shortly after wait goes high but the CPU does not see this due to the polling sequence. The CPU tests for clock high and finds this true and then asserts new data. Clearly a wait request has been missed and a data bit will be lost.
- wait just after clock : the CPU samples clock just before it goes high and finds it low, clock then goes high but this is not detected by the CPU due to the polling sequence. The signal wait goes high and the CPU tests wait and finds it high and so delays the sending of new data. Clearly the wait request has been applied one cycle too early.

If the STG-FT process introduced in section 4 is applied to the first pass STG-FT in Figure 4, it can be seen that it has a non-atomic test transition at S2. Note how this place has been redrawn at the bottom of Figure 4 to satisfy the atomic modelling rules. Simulation of the corrected STG-FT will detect the problem scenarios listed above. This example shows the necessity of the STG-FT approach when a system is implemented using software threads.

The data generator example as implemented with threads cannot be made to work as correct operation requires the

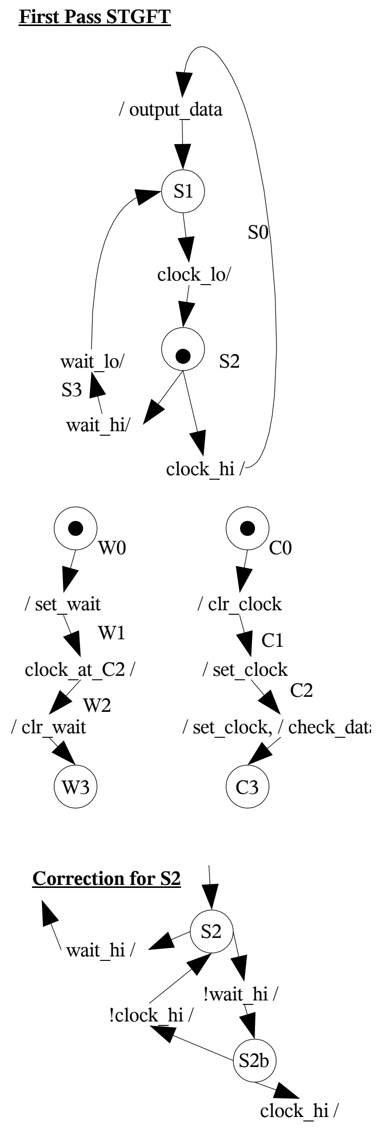

Fig. 4 Data Generator With Wait

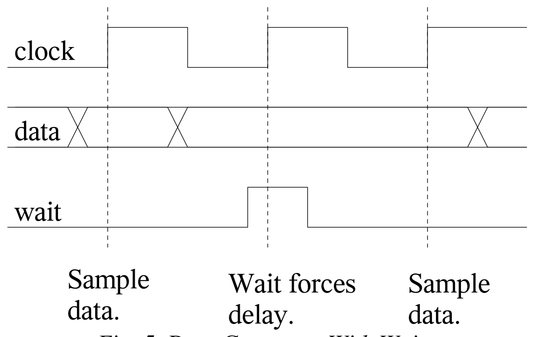

Fig. 5 Data Generator With Wait

detection of the rising edge of clock and simultaneously reading the state of wait. The sampling delays of a thread mean the exact point of the clock transition is missed, and even if it is detected then it is not possible to sample wait at the same time. Some additional hardware is required, for example wait could be sampled by hardware – perhaps a D flipflop clocked by clock with the D input connected to wait.

The original STG-FT in Figure 4, with non-atomic elements, could be made to work with asynchronous logic though meta-stability [17] is an issue as wait could be changing just as the clock is rising.

VI. 2-WIRE TIA SIMULATION & IMPLEMENTATION

In mid 2005 we faced an apparently simple problem as part of a larger research project. A small microcontroller based sensor was developed which used all of the microcontroller's peripherals and most of the available CPU time. A requirement was added that meant the controller had to communicate with an IBM-PC in a simple and cost effect manner. One option was to select a more powerful and expensive microcontroller. In some products such a cost increase would have a significant impact. We chose a different path – to develop a novel 2-wire TIA communications method that would enable the existing, low cost, microprocessor to be used.

The proposed 2-wire TIA system was simulated using an STG-FT simulator we have developed. It followed a similar path to the example -: the initial model was not atomic, following a more traditional STG approach, and on simulation showed no livelock or deadlock. The model was updated to STG-FT format and several livelocks were found and eliminated by modifying the protocol.

One remaining problem concerns a livelock loop between the communicating master and slave. Both ends have a wait loop and if they executed at exactly the same rate then a live-lock situation could develop. If the delays of the master and slave loops are not identical then the live-lock will drop out to normal operation.

Another test for any system is to calculate the reachability tree from any initial system state (all possible FSM object state combinations). While data errors are expected there should be no livelock or deadlock. At first glance it seems unreasonable to investigate all possible system states as many of them are "impossible". In practice it can be surprising the state combinations that do occur due to conditions such as forced reset, spurious signals, corrupted signals, unexpected signals and software delays. In general it is best to take a robust approach and check the behaviour from every possible initial system state.

The 2-wire TIA system has a master with 13 states, a slave with 10 states, and several media bits making 5148 initial system states. The STG-FT simulator automatically generates all initial states and investigates the reachability tree for each and checks for livelock and deadlock. The simulator had to examine some 866 million nodes which took approximately 30 minutes on a 1.6 GHz Celeron running Fedora Core 3.

The result showed the 2-wire TIA system had several deadlock conditions. The solution was to add a time-out to one state in the master in order to break the deadlock and return to normal operation. With the time-out added the 2-wire TIA system is deadlock free.

The 2-wire TIA system has been implemented using an IBM-PC as a master and an Atmel Tiny26 microprocessor as a slave where only two IO pins were free. The communications system can run in the background and is not effected by delays in either the PC or the Tiny26, and does not interfere with the Tiny26 real time code.

Preliminary performance of the system is reported in Table 1. The PC was running Fedora Core 3 using the KDE desktop. A simple PC hang-up program was used to stress the communications system and gauge the effect of system load. A hang-up program is a simple forever loop that uses up all the CPU time the operating system allows it. The Tiny26 had no application code running and just serviced the communications. It is an interesting anomaly that the 20 MHz Tiny26 was slower than the 8 MHz Tiny26.

Table 1 2-wire TIA Performance between a PC and an Atmel Tiny26

PC condition (1.6 GHz Celeron, Fedora Core 3, KDE desktop)	**Bit Transfer Rate to Tiny26 (kilobits/sec each way)**			
	1 MHz Tiny26	4 MHz Tiny 26	8 MHz Tiny 26	20 MHz Tiny 26
Master process priority -20 (highest) only KDE desktop running.	12.0	29.0	44.0	36.0
Master process priority 0 (normal) only KDE desktop running.	11.9	29.0	43.0	35.0
Master process priority +20 (lowest) only KDE desktop running.	11.8	28.3	42.0	32.0
Master process priority -20 (highest) KDE and hang-up running.	10.6	26.0	40.5	31.0
Master process priority 0 (normal) KDE and hang-up running.	6.0	14.5	22.8	17.3
Master process priority +20 (lowest) KDE and hang-up running.	0.55	1.3	2.0	1.6

Examination of the waveforms showed that the PC and 20 MHz Tiny26 timing was nearly synchronous, exhibited a short term livelock cycle as predicted by the STG-FT model, and so reduced data throughput. In all other respects the 2-wire system works as intended.

VII. CONCLUSION

The "simple" sensor application has spun off some new insights and a novel communications system. The proposed communications category TIA communications, is different from the simple asynchronous category and has important attributes useful to small embedded controllers and those heavily loaded with real time tasks. TIA communications has the potential to reduce costs in products and industrial systems.

Since the Petri net based STGs are excellent for modelling asynchronous logic, but may not properly model a system implemented using software threads, we have proposed and trialed a modified form of STGs called STGs For Threads (STG-FT) that appears to correctly model livelock and deadlock in thread based systems.

Finally a 2-wire TIA communications system has been proposed and successfully implemented between a PC and a small embedded controller. The behaviour matches the predictions of the STG-FT model and the observed performance is quite useful.

The new developments outlined in this paper raise some interesting questions that may be worth pursuing. Do the attributes of TIA communications make it useful in other domains? Is it possible to devise a 2-wire TIA system that has higher data throughput? Is a 1 wire TIA system possible using only digital logic?

The STG-FT model appears to be very successful in predicting the behaviour of software thread driven systems. It would be very interesting to apply it to a range of such systems and compare the model predictions to the actual performance.

VIII. REFERENCES

[1] J. Turley, "Motoring with Microprocessors", *Embedded Systems Programming*, Dec 2002; retrieved from www.embedded.com/showArticle.jhtml?articleID=13000166.

[2] O. Christ,, E. Fleisch, F. Mattern, "M-Lab : The Mobile and Ubiquitous Computing Lab Phase II", ETH Zurich & University of St. Gallen, 2002. Retrieved from www.m-lab.ch/about/MLabIIProjectPlan_e.pdf

[3] G. Lee, M. Lee, Hui H. Shao, X. Zhao, "Networked intelligent controller based on embedded system", *30th Annual Conference of IEEE Industrial Electronics Society*, 2004. IECON 2004. Volume 3, 2-6 Nov. 2004, pp. 2942 – 2945.

[4] M. Castro, R. Sebastian, F. Yeves, J. Peire, J. Urrutia, J. Quesada, "Well-known serial buses for distributed control of backup power plants. RS-485 versus controller area network (CAN) solutions", 28th Annual Conference of the IEEE Industrial Electronics Society (IECON02), Volume 3, 5-8 Nov. 2002 pp. 2381 - 2386.

[5] G. Le Lann, Asynchrony and real-time dependable computing, *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems*, 2003. (WORDS 2003), 15-17 Jan. 2003. pp18 – 25.

[6] *DEC Unibus specification*, retrieved from- http://www.bitsavers.org/pdf/dec/unibus/UnibusSpec1979.pdf

[7] ANSI/IEEE Std 488.1-1987, *IEEE standard digital interface for programmable instrumentation,* IEEE, 1987

[8] J. Kessels, "Register-communication between mutually asynchronous domains", *Proceedings 11th IEEE International Symposium on Asynchronous Circuits and Systems, 2005 (ASYNC 2005)*, 14-16 March 2005, pp. 66 – 75.

[9] A. Yakovlev, S. Furber, R. Krenz, A. Bystrov, "Design and analysis of a self-timed duplex communication system", IEEE Transactions on Computers, Volume 53, Issue 7, July 2004, pp. 798 – 814.[10] W.J. Bainbridge and S.B. Furber, "Asynchronous macrocell interconnect using MARBLE", *Proceedings Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 30 March-2 April 1998, pp. 122 – 132.

[11] T. Takahashi and T. Hanyu, "Multiple-valued multiple-rail encoding scheme for low-power asynchronous communication", *Proceedings of the 34th International Symposium on Multiple-Valued Logic* (IMSVL'04) , 19-22 May 2004, pp 20-25.

[12] J. Cortadella, M. Kishinevsky, L. Lavagno, A. Yakovlev, "Synthesizing Petri nets from state-based models", *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, Dec. 1995, pp. 164-171.

[13] Alex Yakovlev, "On limitations and extensions of STG model for designing asynchronous control circuits", *Proceedings International Conference on Computer Design: VLSI in Computers and Processors*, *ICCD '92.* IEEE, 11-14 Oct. 1992, pp. 396 – 400.

[14] M. Kishinevsky, J. Cortadella , Alex Kondratyev, "Asynchronous interface specification, analysis and synthesis", *Proceedings : Design Automation Conference*, 15-19 Jun 1998, pp. 2 – 7.

[15] J. Cortadella , M. Kishinevsky, Al. Kondratyev, L. Lavagno, A. Yakovlev, "Petrify: Method and Tool for Synthesis of Asynchronous Controllers and Interfaces". A tutorial from ASYNC2003 retrieved from http://www.staff.ncl.ac.uk/alex.yakovlev/home.formal/async03-tut-demo.ppt

[16] J. Cortadella, "Petrify : a tutorial for the designer of asynchronous circuits", (no date or revision number given). Retrieved from http://www.cs.technion.ac.il/~cs234305/petrify/docs/tutorial.ps

[17] H. Johnston and M. Graham, *High Speed Digital Design*, Prentice-Hall, Englewood Cliffs NJ, 1993.