

Formal Verification of a Microprocessor Control

Lubomir Ivanov

Department of Computer Science

Iona College

715 North Avenue

New Rochelle, NY 10801

livanov@iona.edu

Abstract

The complexity of the instruction set of modern processors often leads to faults in the microinstruction sequencing, and timing errors, which are difficult to detect with conventional simulation methods. Formal verification offers a powerful alternative for dealing with these problems. In this paper we present a mathematical model of the microcode of a Transputer-like microprocessor, and demonstrate how to test for the satisfaction of desired properties and the absence of improper microinstruction sequencing. The verification is based on a recently introduced technique using the inductively defined notion of series parallel posets, which offers low time- and space complexity.

Introduction

The complexity of designing a modern processor leads to a significantly increased probability of serious design faults. Pipeline hazards and superscalar dependencies, for example, arise, in part, from the inability of the available hardware to support overlapped or parallel execution of instructions, as well as from instruction interdependencies. Thus, proper microinstruction sequencing must be ensured to avoid potential conflicts. At the same time, the ever-increasing processor clock speed can lead to potential timing errors, which are difficult to detect using classical simulation and testing methods. The recent examples of "bugs" in the microcode of the Pentium® processors illustrate the severity of the problem. A promising alternative is offered by the field of formal verification, which, based on a mathematical model of the system under consideration, attempts to prove or disprove facts about the system model, guaranteeing that all desired properties are satisfied, and unwanted properties and design faults are absent. An excellent overview of the field of formal verification can be found in [1]. Some powerful formal verification methods such as *Symbolic Model Checking* [1] and *ω -Automata Verification* [2] have become the basis of industrial-level verification tools (SMV, FormalCheck, etc.). Unfortunately, the high computational complexity of these verification methods imposes limits on the size of the circuits to which such powerful techniques can be applied. As an alternative to the powerful but computationally expensive verification methods, a number of new verification approaches have emerged, which sacrifice part of their expressive power, but ensure a significantly improved efficiency. Among these, several methods have been based on using partial orders to describe the dependence or independence of sets of events occurring in a hardware system [3, 4, 5, 6, 7]. The main appeal of using partial orders in modeling and verifying system behavior is in avoiding the study of all possible interleavings of events in the system. In [8, 9, 10, 11] we introduced a new formal verification method for proving timing properties of complex systems, based on the inductively defined notion of series-parallel posets. The verification algorithms have a

low-order polynomial time- and space complexity. In [12] the technique was used to verify the behavior of a handshaking communication protocol, and the PCI bus interconnect protocol. In [13] the method was applied to the modeling and formal verification of the MESI cache coherence protocol for a system of n write-back cache memories in a Shared Memory MIMD system.

In this paper we demonstrate how our series-parallel poset methodology can be applied to the modeling and verification of the control of a small microprocessor similar to the InMOS Transputer T414. We begin with a description of the processor, its datapath, instruction set, and the stages of its instruction cycle. We then present the formal model of the processor control, and demonstrate the verification of properties. The main presentation is followed by a brief introduction to series-parallel posets, and an outline of the series-parallel poset verification approach for iterated systems. Finally, we briefly discuss some strengths and weaknesses of our methodology in the context of related formal verification work.

Processor Organization and Architecture

The processor we model is a slightly simplified version of the InMOS Transputer T414 - a small 2-bus RISC architecture with three general-purpose registers in a stack organization, a small 4Kb on-chip memory, and four communication links. We chose to model the T414 to avoid the complexity of modeling virtual channel processor and the FPU of the later models (T805, T9000, etc.). These issues do not reveal any intricate aspects of our conceptual framework, and will only distract the reader from the issues of verification. However, in an upcoming paper, we present the modeling and verification of a pipelined processor, in which the above issues are considered in greater depth.

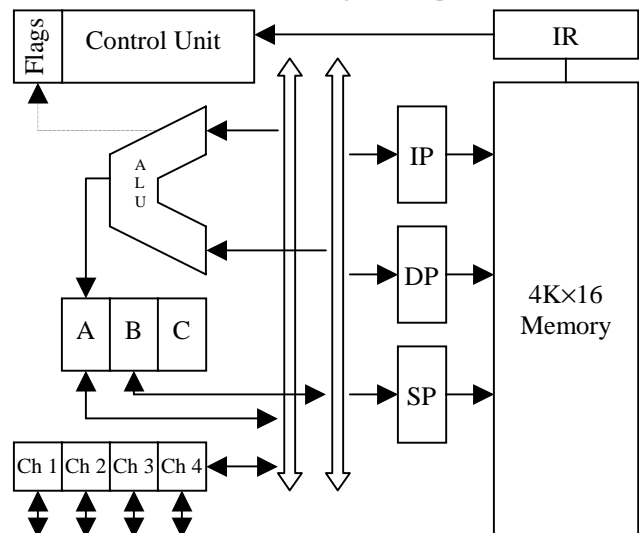


Fig. 1 - Processor Organization

The processor organization is presented in Fig.1. There are three general-purpose registers organized as a hardware stack with register A at the top. The 4K×16 memory is pointed to by an Instruction Pointer (IP), Data Pointer (DP), and Stack Pointer (SP) special purpose registers. The fetched instructions are moved to the Instruction Register (IR), and decoded by the control unit. There are four bi-directional communication channels, used for communicating with peripherals and/or other processors.

The instruction set of the processor has only sixteen instructions:

| Instruction | RTL Description | Instruction | RTL Description |
|-------------|---------------------------|-------------|--|
| ADD | $A \leftarrow A + B$ | LD addr | $A \leftarrow \text{Mem}[\text{addr}]$ |
| SUB | $A \leftarrow A - B$ | ST addr | $\text{Mem}[\text{addr}] \leftarrow A$ |
| SLL | $A \leftarrow A \ll B$ | SEND Ch | $\text{Ch} \leftarrow A$ |
| SRL | $A \leftarrow A \gg B$ | RECV Ch | $A \leftarrow \text{Ch}$ |
| AND | $A \leftarrow A \& B$ | CALL addr | Procedure call |
| OR | $A \leftarrow A B$ | RET | Procedure return |
| XOR | $A \leftarrow A \oplus B$ | J addr | Uncond. Jump |
| NOT | $A \leftarrow !A$ | BEQZ addr | Branch Equal 0 |

The instruction cycle can be expressed with the following set of state transition diagrams:

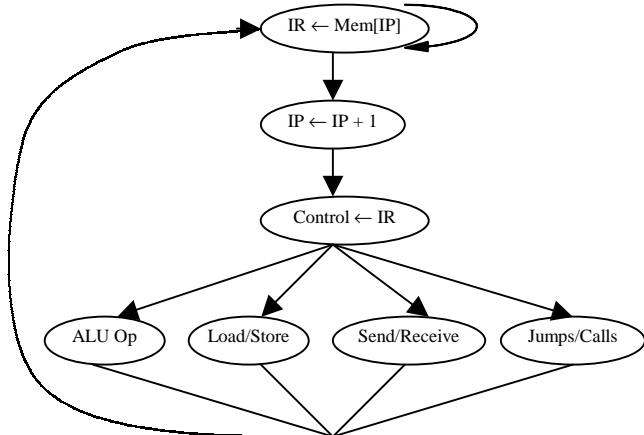


Fig. 2 - Fetch-Decode-Execute Instruction Cycle

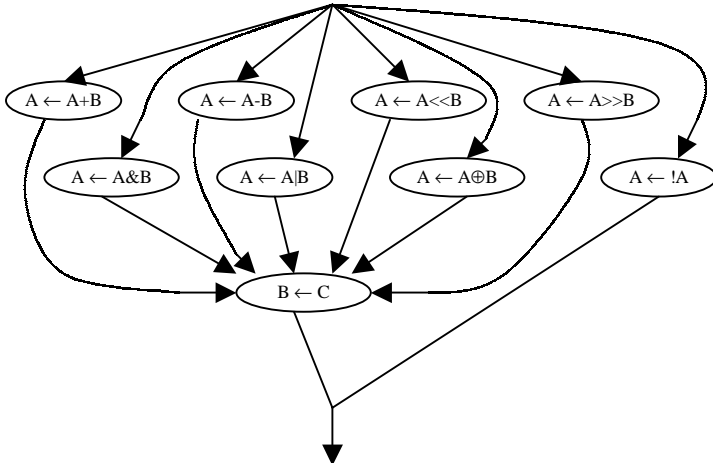


Fig. 3 - ALU Operation State Diagram

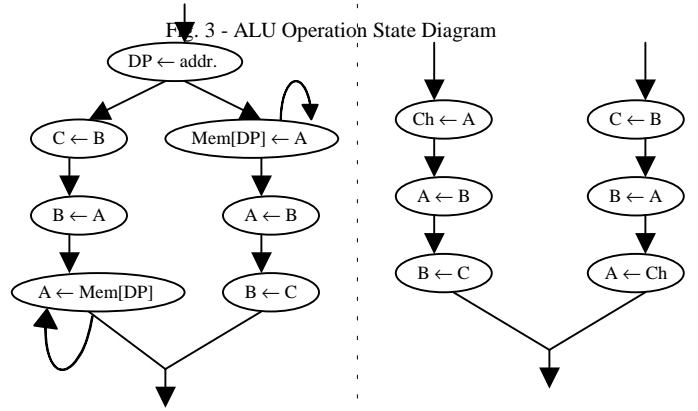


Fig. 4 - Load/Store and Send/Receive State Diagrams

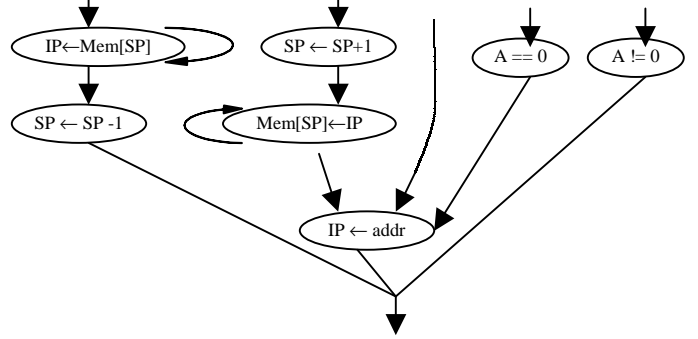


Fig. 5 - Jumps/Calls State Diagram

The state diagram in Fig.2 describes the general structure of the fetch-decode-execute instruction cycle, branching into the state diagrams in Fig. 3, 4, 5 depending on the type of instruction. Each stage of the instruction cycle involves the execution of several microinstructions, whose sequencing must be strictly enforced, e.g. IP must not be incremented before the instruction fetch is complete. However, certain microinstructions are independent, and can be executed in any order. For example, incrementing IP and decoding the fetched instruction can be done in parallel, independently of each other. Our series-parallel poset methodology is aimed at modeling the notions of dependence and independence, and offers a convenient way to model the processor's control.

Modeling the Processor Control

Our model defines the execution of each microinstruction to be an event, and specifies the correct ordering or the independence of microinstruction execution with the help of operations concatenation, •, (for sequencing), shuffle, ⊗, (for independence), Kleene star, *, (for iteration/repetition), and union, +, (for choice). The formal model of the processor control is given below:

$$B = (e_0^* \bullet (e_1 \otimes e_2) \bullet (B_{ALUop} + B_{L/S} + B_{S/R} + B_{J/C})^*)^*$$

where:

$$B_{ALUop} = (e_3 + e_4 + e_5 + e_6 + e_7 + e_8 + e_9) \bullet e_{10} + e_{11}$$

$$B_{L/S} = e_{12} \bullet (e_{13} \bullet e_{14} \bullet e_{15}^* + e_{16}^* \bullet e_{17} \bullet e_{10})$$

$$B_{S/R} = e_{18} \bullet e_{17} \bullet e_{10} + e_{13} \bullet e_{14} \bullet e_{19}$$

$$B_{J/C} = (I + e_{20} \bullet e_{21}^* + e_{25}) \bullet e_{24} + e_{22}^* \bullet e_{23} + e_{26}$$

Each event in the above processor control model represents a unique microinstruction as follows:

| Event | Microinstruction | Event | Microinstruction |
|-----------------|------------------|-----------------|------------------|
| e ₀ | IR ← Mem[IP] | e ₁₄ | B ← A |
| e ₁ | IP ← IP + 1 | e ₁₅ | A ← Mem[DP] |
| e ₂ | Control ← IR | e ₁₆ | Mem[DP] ← A |
| e ₃ | A ← A+B | e ₁₇ | A ← B |
| e ₄ | A ← A-B | e ₁₈ | Ch ← A |
| e ₅ | A ← A<<B | e ₁₉ | A ← Ch |
| e ₆ | A ← A>>B | e ₂₀ | SP ← SP+1 |
| e ₇ | A ← A&B | e ₂₁ | Mem[SP]←IP |
| e ₈ | A ← A B | e ₂₂ | IP←Mem[SP] |
| e ₉ | A ← A⊕B | e ₂₃ | SP ← SP -1 |
| e ₁₀ | B ← C | e ₂₄ | IP ← addr |
| e ₁₁ | A ← !A | e ₂₅ | A == 0 |
| e ₁₂ | DP ← addr. | e ₂₆ | A != 0 |
| e ₁₃ | C ← B | | |

Tbl.1 - Microinstructions of the Control Unit

Verifying the Processor Control

To verify the proper sequencing of microinstructions, and check for event dependence/independence, we need to specify a set of properties, which tests all critical aspects of the system operation. Once the properties are identified and converted to series-parallel poset expression format, they can be verified using the predicates defined in [11] (and briefly outlined in the "Overview" section). For lack of space, we shall limit ourselves to demonstrating the verification process of only one very simple property from the complete set of properties mentioned above.

Property P_1 :

"Memory addresses remain fixed during an instruction fetch."

For the address to remain fixed throughout an instruction fetch, the contents of the IP register must not be updated until the memory read/write is complete. By examining the microinstructions in Tbl. 1, we observe that IP gets modified by the microinstructions IP←IP+1 (event e₁), IP←Mem[SP] (event e₂₂), and IP←addr (event e₂₄). To carry out an instruction fetch, memory is accessed by the microinstructions IR←Mem[IP] (event e₀). Therefore, property P_1 can be expressed with the following series-parallel poset expression:

$$P_1 = (e_0^* \bullet (e_1 + e_{22} + e_{24}))^*$$

Once the property has been specified, the next step of our verification algorithm is to reduce the size behavior expression, B , by eliminating all events not among the set of events specified in the property. The reduced behavior with respect to the events in $set(P_1) = \{e_0, e_1, e_{22}, e_{24}\}$ is:

$$B = (e_0^* \bullet e_1 \bullet (e_{22} + e_{24}))^*$$

For the processor to operate correctly, property P_1 must always be

true. Therefore, we need to verify that property P_1 is always satisfied. Hence, we will use the AS verification predicate¹:

$AS(B, P_1)$:

Since both B and P_1 are iterated,

$$AS(B, P_1) = AS(e_0^* \bullet e_1 \bullet (e_{22} + e_{24}), e_0^* \bullet (e_1 + e_{22} + e_{24})) :$$

- $e_0^* \bullet (e_1 + e_{22} + e_{24}) \notin SP(\Sigma^*)$ (i.e. the property has iteration),

- $AS(e_0^* \bullet e_1 \bullet (e_{22} + e_{24}), e_0^*) = {}^2 AS(e_0^*, e_0^*)$

Since $e_0^* \notin SP(\Sigma^*)$ (i.e. the property has iteration),

$$AS(e_0^*, e_0^*) = AS(e_0, e_0) = TRUE$$

$$\Rightarrow AS(e_0^* \bullet e_1 \bullet (e_{22} + e_{24}), e_0^*) = TRUE$$

- $AS(e_0^* \bullet e_1 \bullet (e_{22} + e_{24}), (e_1 + e_{22} + e_{24})) = {}^2$

$$AS(e_1 \bullet (e_{22} + e_{24}), (e_1 + e_{22} + e_{24})):$$

$$AS(e_1 \bullet (e_{22} + e_{24}), e_1) = {}^2 AS(e_1, e_1) = TRUE$$

$$AS(e_1 \bullet (e_{22} + e_{24}), e_{22}) = {}^2 AS(e_{22}, e_{22}) = TRUE$$

$$AS(e_1 \bullet (e_{22} + e_{24}), e_{24}) = {}^2 AS(e_{24}, e_{24}) = TRUE$$

$$\Rightarrow AS(e_0^* \bullet e_1 \bullet (e_{22} + e_{24}), (e_1 + e_{22} + e_{24})) = TRUE$$

- $pred_B(\{e_1\}) = \{e_0\} \cap \{e_0\} = \{e_0\}$

- $pred_B(\{e_{22}\}) = \{e_0, e_1\} \cap \{e_0\} = \{e_0\}$

- $pred_B(\{e_{24}\}) = \{e_0, e_1\} \cap \{e_0\} = \{e_0\}$

$$\Rightarrow AS(B, P_1) = TRUE$$

The verification of the above property was carried out by hand to illustrate the operation of the verification algorithms. The actual verification of the entire property set was done using a software package, developed at Iona College on the basis of the theoretical constructs presented in [8, 9, 10, 11], and outlined below.

Overview of Series-Parallel Posets

A *partially ordered set (poset)* is a set with a reflexive, antisymmetric, and transitive relation defined on the set elements. A Σ^* -labeled poset $P=(P, \leq, l)$ consists of a poset (P, \leq) , and an assignment of a nonempty word (a label) $l(v) \in \Sigma^*$ to each vertex v in P . Given posets P and Q with $P \cap Q = \emptyset$, we define two operations on labeled posets:

$$\text{Concatenation } (\bullet): \quad P \bullet Q := (P \cup Q, \leq_{P \bullet Q})$$

$$\text{Shuffle } (\otimes): \quad P \otimes Q := (P \cup Q, \leq_{P \otimes Q}),$$

$$\text{where:} \quad v \leq_{P \bullet Q} v' \Leftrightarrow v \leq_P v' \vee v \leq_Q v' \vee (v \in P \wedge v' \in Q)$$

$$v \leq_{P \otimes Q} v' \Leftrightarrow v \leq_P v' \vee v \leq_Q v'$$

The notion of a *Series-Parallel Poset (SPP)* over an alphabet Σ is defined inductively as follows:

- The empty poset, I , is a SPP
- For each $\sigma \in \Sigma$, the singleton labeled σ is a SPP
- If P and Q are SPPs, so are $P \bullet Q$ and $P \otimes Q$

The set of all series parallel posets formed from I and the singletons and closed under concatenation (\bullet) and shuffle (\otimes) forms a bimonoid denoted $SP(\Sigma^*)$. For further details see [14].

For our purposes, the alphabet, Σ , will consist of all distinct events occurring during a run of the system under consideration. Let each event, e_i , occurring in a system be represented by a singleton poset, e_i . Then, the fact that event e_i precedes event e_j is represented by $e_i \bullet e_j$. On the other hand, the independence of events e_i and e_j is represented by the series-parallel poset $e_i \otimes e_j$. This extends naturally to sets of events.

¹ To follow the outlined verification process, use the algorithm in the Overview of Series-Parallel Posets section or consult [11].

² Reducing the behavior to include only events from the property

What does it mean for two sets of events to be dependent or independent?

Two sets of events, P and Q , are *independent* if no event in P triggers a chain of events leading to the occurrence of an event in Q and vice versa. In other words P and Q are independent if the set of events, which are predecessors of P does not involve any event from Q and vice versa.

A set of events P *always precedes* a set of events Q if all events in P occur before any event in Q does, i.e. when each event in Q has all events in P as predecessors.

A set of events P *partially precedes* a set of events Q if P sometimes occurs before Q . This is so when each event in Q has at least one predecessor from P , or when P and Q are independent.

Let us illustrate the definitions with an example. Consider the following series-parallel poset:

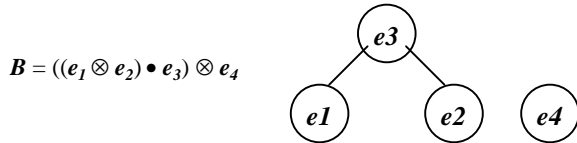


Fig.6 - A Simple Series Parallel Poset Example

Consider now the sets of events $P_1 = \{e_1, e_4\}$, and $P_2 = \{e_3\}$. Clearly, P_1 and P_2 are not independent since e_1 must occur before e_3 . P_1 does not always precede P_2 since a possible event sequence is e_1, e_2, e_3 , and then e_4 . But P_1 may sometimes precede P_2 since another possible event sequence is, for example, e_1, e_2, e_4, e_3 .

Interpreting series-parallel posets as descriptions of the dependence or independence of sets of events allows us to model the *behavior* of a system in terms of the sequences of events occurring during its operation. In [8] we presented a methodology for modeling the behavior and the properties of non-iterated systems with series-parallel posets. A *non-iterated system* is one, in which the events are distinct and not repeated.³

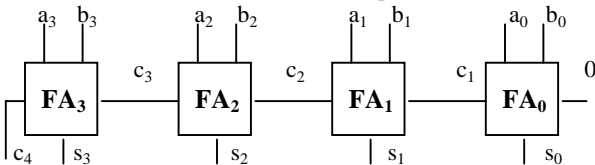
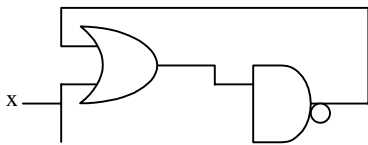


Fig.7 - A Non-Iterated System: 4-bit Binary Adder

We presented an algorithm, which can be used to verify that a particular property is always satisfied or sometimes satisfied within a given behavior. In [9], the methodology was further expanded to deal with *globally iterated/locally non-iterated systems*. These are systems, which consist of non-iterated sub-systems operating either in series or in parallel, but such that the global system output is fed back for another iteration.



³ Not all non-iterated systems can be expressed with series-parallel posets. See the section on Contributions and Limitations.

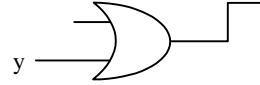


Fig.8 - A Simple Globally-Iterated/Locally Non-Iterated System
The verification algorithms have a low-order polynomial time- and space complexity, which is further improved with the introduction of the behavior reduction methodology presented in [10].

An *iterated system* is one in which some or all events are repeated. Thus, an iterated system consists of a number of components, which function in series or independently so that each component is either an iterated- or a non-iterated system. A wide variety of systems can be considered iterated:

- Communication-, Interconnect, or Cache Protocols
- Asynchronous Sequential Circuits
- Feedback Control Systems, etc.

Concrete examples of iterated systems to which we have applied our methodology are the *Peripheral Component Interconnect* (PCI) bus protocol, which allows 32-bit or 64-bit high-speed data transfers between devices, and the *Modified/Exclusive/Shared/Invalid* (MESI) cache coherence protocol used to synchronize the operation of cache controllers in shared-memory MIMD systems, as well as to maintain the consistency between the level-1 and level-2 caches of the Intel Pentium® microprocessor [12, 13]. Here is a simple example of an iterated system at the logic gate level:

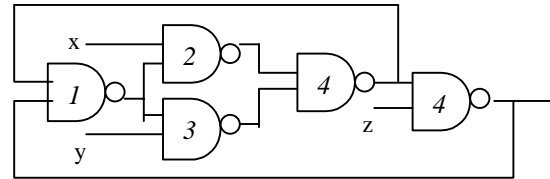


Fig.9 - A Simple Iterated System (gate level)

The notion of a series-parallel poset is not sufficient to describe the behavior of a system with iteration. We, therefore, need to introduce a new structure - the star shuffle semiring $S = (S, +, \bullet, \otimes, *, \emptyset, I)$ of series-parallel posets, defined as follows:

- S - the set of finite subsets of $SP(\Sigma^*)$, closed under the semiring operations
- If $K \in S$ and $L \in S$, $K+L = \{P \mid P \in K \vee P \in L\} \in S$
- If $K \in S$ and $L \in S$, $K \bullet L = \{P \bullet Q \mid P \in K \wedge Q \in L\} \in S$
- If $K \in S$ and $L \in S$, $K \otimes L = \{P \otimes Q \mid P \in K \wedge Q \in L\} \in S$
- If $K \in S$, then $K^* = I + K + K \bullet K + \dots = \sum_{i=0, \dots, \infty} K^i \in S$, where $K^i = K \bullet K \bullet \dots \bullet K$, i times.
- \emptyset is the empty set of posets
- I is the empty poset

We define the *behavior*, B , of an iterated system to be an element of the star-shuffle semiring S , i.e. $B \in S$. Thus, the behavior of an iterated system is a set of series-parallel posets. For example, if we denote the event “gate i produces a valid output” by e_i , then the behavior of the system in Fig.5 is given by the following expression $B = ((e_1 \bullet (e_2 \otimes e_3) \bullet e_4)^* \bullet e_5)^*$.

We can represent the *verification properties* as sets of series-parallel posets as well, i.e. $P \in S$. Unlike behaviors, however, properties are usually be defined over a subset of the alphabet Σ ,

since we are most often interested in the mutual dependence or independence of a relatively small subset of system events. For example the property "Gates 2 and 3 produce valid outputs independent of each other, but gate 4 depends on both gates 2 and 3" is given by the expression $P = (e_2 \otimes e_3) \bullet e_4$.

The verification questions are specified as predicates over sets of series-parallel posets. These predicates are:

- **SS(B, P)** is a binary predicate, which we shall read as "The property P is sometimes satisfied within the behavior, B ". The predicate takes a behavior and a property and verifies that P can sometimes be traced within the behavior, B , of the system.
- **AS(B, P)** is a binary predicate, which we shall read as "The property P is always satisfied within the behavior, B ". The predicate takes a behavior and a property and verifies that P can always be traced within the behavior, B , of the system.

There are four normal forms of behavior and property expressions:

- Concatenation: $B = B_1 \bullet B_2 \bullet \dots \bullet B_n$ & $P = P_1 \bullet P_2 \bullet \dots \bullet P_m$
- Shuffle: $B = B_1 \otimes B_2 \otimes \dots \otimes B_n$ & $P = P_1 \otimes P_2 \otimes \dots \otimes P_m$
- Plus: $B = B_1 + B_2 + \dots + B_n$ & $P = P_1 + P_2 + \dots + P_m$
- Star: $B = B_1^*$ & $P = P_1^*$

To simplify the reasoning about sets of events and the complexity of the verification algorithms we introduce the notion of a reduction of the system behavior. It is prompted by the fact that, while the system behavior may involve hundreds of thousands of events, in most cases the verification property involves only a few events. The reduction is carried out by a recursively defined projection function $Pr(B, set(P))$, which takes a behavior, B , and a set of events, and returns a reduced behavior, B' , with respect to the events in the specified set. The effect of the projection function is to substitute I in place of all events not in $set(P)$ without modifying the ordering of events in the behavior.

Based on a number of theorems, corollaries, and lemmas, which examine the satisfaction of all forms of properties with respect to all forms of behaviors, we derive the formal definition of the two verification predicates **SS(B, P)** and **AS(B, P)** for iterated systems. In this outline, we present only **AS(B, P)**:

AS(B, P) iff

- $P = e \wedge B = e$
- $P = P_1^* \wedge B = B_1 \otimes B_2 \otimes \dots \otimes B_n \wedge AS(B, P_1) \wedge \forall i \in [n] B_i = B_{i1}^*$
- $P = P_1^* \wedge B = B_1^* \wedge AS(B_1, P_1)$
- $P = P_1 \otimes P_2 \otimes \dots \otimes P_m \wedge B = B_1^* \wedge AS(B_1, P)$
- $P = P_1 \otimes P_2 \otimes \dots \otimes P_m \wedge B = B_1 \otimes B_2 \otimes \dots \otimes B_n \wedge P \notin SP(\Sigma^*) \wedge \forall i \in [m] AS(B, P_i) \wedge \forall i \in [m-1] \text{Independent}_B(P_i, P_{i+1}) \wedge \forall i \in [m] (P_i = (P_{i1})^* \rightarrow \forall e \in P_i L(set(lisc(B, e))) \subseteq L(set(P_i)))$
- $P = P_1 \bullet P_2 \bullet \dots \bullet P_m \wedge (B = B_1 \bullet B_2 \bullet \dots \bullet B_n \vee B = B_1^*) \wedge P \notin SP(\Sigma^*) \wedge \forall i \in [m] AS(B, P_i) \wedge \forall i \in [m] (P_i = (P_{i1})^* \rightarrow \forall e \in P_i L(set(lisc(B, e))) \subseteq L(set(P_i))) \wedge \forall i \in [m-1] (\forall e \in P_{i+1} L(pred_B(\{e\})) \cap L(set(P_i)) = L(set(P_i)))$
- $B = B_1 + B_2 + \dots + B_n \wedge \forall i \in [n] AS(B_i, P)$
- $P = P_1 + P_2 + \dots + P_n \wedge \exists i \in [n] AS(B, P_i)$

In the above definitions we made use of number of functions – the labeling functions $l(s)$ and $L(\{s_1, \dots, s_n\})$, the predecessor function, $pred(P)$, and the functions $set(P)$, "Non-Iterated", $NI(B)$, and "Least Iterated Sub-Component", $lisc(B, e)$. The exact definition of these functions is presented in [11] and omitted here for lack of space. We also used the auxiliary predicate **Independent_B(P, Q)**.

The predicates serve as a basis of a verification algorithm. The analysis of its requirements shows that the worst-case time complexity is $O(n+m^3)$, and the average case time complexity is $O(n+m^2)$, where n is the number of events in the behavior (before the reduction), and m is the number of events in the property. The space complexity is $O(m)$.

Conclusions

In this paper, we presented the modeling and formal verification of processor control for a simplified version of the InMOS Transputer microprocessor. The approach is based on the developed series-parallel poset methodology. The technique is less expressive than other formal verification methods (e.g. symbolic model checking), but guarantees a low algorithmic complexity, which makes it applicable to complex real-world systems and protocols. Current work involves the verification of pipelined microprocessor microcode, and modeling the behavior of dataflow computers.

In many respects, our approach is close to the studying of the language containment of behavior and property automata [2]. However, we approach the topic from a purely syntactic point of view, avoiding the issue of exhaustive substring matching. Moreover, the use of the shuffle operator (\otimes), significantly simplifies and speeds up the verification task by avoiding the study of all possible independent event interleavings. Closest to our work is that of V.Pratt [4]. There are however important differences. The main stress in [4] is on modeling system behavior with the help of an extensive collection of operations. Our technique uses a far smaller collection of operations ($\bullet, \otimes, *$), but models not only system behaviors but properties as well. The emphasis of our work is on verification, and, the reduced collection of operations simplifies the analysis, and helps the efficiency of the algorithms.

One important shortcoming of our technique is the inability to model "N"-type dependencies among the events occurring in a system. These are encountered quite often in real systems and significantly limit the general applicability of our algorithms. We are currently working on extending our verification methodology to deal with this type of event dependencies as well.

References

- [1] K. McMillan, "Symbolic Model Checking", Kluwer Academic Publishing, 1993
- [2] R.Kurshan, "Computer Aided Verification of Coordinating Processes: The Automata-Theoretic Approach", Princeton Series in CS, Princeton, 1994
- [3] M.Nielsen, G.Plotkin, and G.Winskel, "Petri nets, event structures, and domains", part I, TCS, 13:85-108, 1981
- [4] V.Pratt, "Modeling Concurrency with Partial Orders", International Journal of Parallel Programming, 1986
- [5] P. Godefroid, "Partial Order Methods for the Verification of Concurrent Systems: an Approach to the State Explosion Problem", Doctoral Dissertation, University of Liege, 1995
- [6] R. Nalumasu, G. Gopalakrishnan, "A New Partial Order Reduction Algorithm for Concurrent System Verification", Proceedings of IFIP, 1996
- [7] D. Peled, "Combining Partial Order Reductions with On-the-Fly Model Checking", Journal of FM in System Design, 1996
- [8] L.Ivanov, R.Nunna, S.Bloom, "Modeling and Analysis of Non-Iterated Systems: An Approach Based on Series-Parallel Posets", Proceedings of ISCAS'99, 1999
- [9] L.Ivanov, R.Nunna, "Formal Verification with Series-Parallel Posets of Globally-Iterated Locally-Non-Iterated Systems", Proc. of MWSCAS'99, 1999
- [10] L.Ivanov, R.Nunna, "Formal Verification: A New Partial Order Approach", Proc. of ASIC/SOC'99, 1999

- [11] L. Ivanov, R. Nunna, "*Modeling and Verification of Iterated Systems and Protocols*", submitted to MWSCAS'01, 2001
- [12] L. Ivanov, R. Nunna, "*Modeling and Verification of an Interconnect Bus Protocol*", Proc. of MWSCAS'00, 2000
- [13] L. Ivanov, R. Nunna, "*Modeling and Verification of Cache Coherence Protocols*", Proc. of ISCAS'01, Sydney, 2001
- [14] Bloom, Z. Esik, "*Free Shuffle Algebras in Language Varieties*", Theoretical Computer Science 163 (1996) 55-98, Elsevier