FORMAL VERIFICATION OF MICROINSTRUCTION SEQUENCING

LUBOMIR IVANOV

Department of Computer Science
Iona College
715 North Avenue
New Rochelle, NY 10801
LIVANOV@IONA.EDU

The complexity of the instruction set of modern microprocessors often leads to faults in the microinstruction sequencing and timing errors in the implementation of the processor control. These errors are difficult to detect with conventional simulation methods. As an alternative, formal verification uses a mathematical model of the system to verify its correct behavior by constructing a formal proof. Recently we introduced a new partial order formal verification method based on the notion of series-parallel posets. The associated verification algorithms have a low order space- and time complexity, and have been successfully applied to the verification of properties of real-world systems such as the PCI local bus protocol and the MESI cache coherence protocol. In this paper we use series-parallel posets to model and verify the behavior of the DLX microprocessor control.

1 Introduction

The complexity of designing a modern pipelined/superscalar processor leads to a significantly increased probability of serious design faults such as improper microinstruction sequencing and timing errors, while limiting the usefulness of the classical simulation and testing methods for uncovering these design faults. 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 gained significant popularity, and have led to the development of industrial-level verification tools (SMV, FormalCheck, etc.). Unfortunately, the power and expressiveness of these methods is offset by the high computational complexity of their verification algorithms. This imposes limits on the size of the circuits to which such general techniques can be applied. In the meantime, a number of new verification methods have emerged, which, while relatively less expressive, guarantee 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 occurring during a run of the system. In addition, partial order models are usually very clear and intuitive, and the verification algorithms can be fully automated.

In [8, 9, 10, 11] we introduced a new formal verification method for proving timing properties of complex systems. The method is based on the inductively defined notion of series-parallel posets. The verification algorithms are characterized by a low-order polynomial complexity. In [12] the technique was used to verify the behavior of a Handshaking Communication Protocol, and the popular PCI local 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 multiprocessor system.

In this paper we present another important application of our series-parallel poset methodology - the modeling and verification of the DLX microprocessor control. We begin with a description DLX, and the stages of its instruction cycle. We then present the formal model of the DLX control using series-parallel posets, and demonstrate the verification of a two properties. The main presentation is followed by a brief introduction to series-parallel posets, and an outline of our verification approach for iterated systems. Finally, we briefly discuss some strengths and weaknesses of our methodology in the context of other formal verification work.

2 The DLX microprocessor

The DLX processor was introduced by Hennessy and Patterson in [14]. It incorporates many features of popular commercial microprocessors such as Intel i860, SPARCstation-1, etc. Architecturally, it has thirty-two 32-bit general purpose registers (R0 hardwired to 0), thirty-two floating-point registers, which can be used for single-precision or (in pairs) for double precision floating point computation, and a set of special purpose registers for accessing status information. Memory is accessed through loads and stores which can transfer a byte, a halfword or a word. The address is 32 bits wide. The instructions set includes 4 types of instructions:

- *Load/Stores*, (e.g. LW R3, 100(R1), or SB 45(R7), R1)
- *ALU operations*, (e.g. ADD R1, R2, R3, or SUBI R1, R5, #4)
- *Branches and Jumps*, (e.g. JR R5, or BEQZ R12, NEXT)
- *Floating Point Operations*, (e.g. ADDD F0, F1, F2)

For further architectural details, refer to [14].

The internal organization of DLX (except the FPU) is given in *figure 1* below. MAR and MDR are the memory address- and data registers, IAR is the interrupt address register, IR is the instruction register, and the A, B, C ports are used for accessing the 32 registers, R0 - R31.

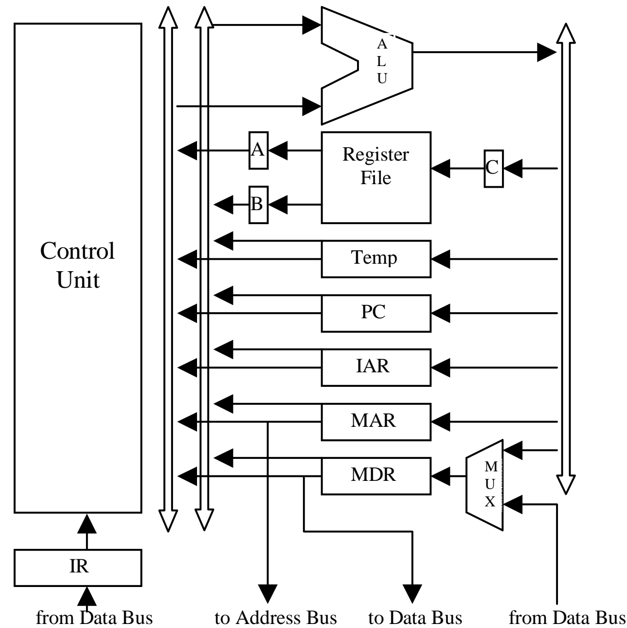

Figure 1 DLX Internal Organization

The instruction cycle for all (except FPU) DLX instructions consists of 5 stages:

- *Fetch:* MAR ← PC; IR ← Mem[MAR]
- *Decode:* A ← RegS1; B ← RegS2; PC ← PC +4
- *Execute:* - Mem reference: MAR←A+$(IR_{16})^{16}$##$IR_{16..31}$; MDR ← RegD
  - ALU instr.: ALUout ← A op (B or $(IR_{16})^{16}$##$IR_{16..31}$)
  - Branch/Jump: ALUout ← PC + $(IR_{16})^{16}$##$IR_{16..31}$
- *Mem.Access:* MDR ← Mem[MAR] or Mem[MAR] ← MDR or
  If (cond) PC←ALUout
- *Write Back:* RegD ← ALUout or MDR

Here, RegS1 and RegS2 are the two source registers, and RegD is the destination register. The notation "$(IR_{16})^{16}$##$IR_{16..31}$" means that the contents of the lower half (bits 16-31) of the IR register are sign extended by replicating bit 16 sixteen times.

3 Modeling the DLX Control

Each stage of the instruction cycle involves the execution of a sequence of microinstructions. In most cases the sequencing must be strictly enforced, e.g. the ALU should not attempt to carry out an arithmetic operation before the two register file ports, A and B, have been loaded with data from the source registers.

However, in some cases, the order is insignificant, and the microinstructions can be executed in any sequence, independently of each other. For example, the 3 microinstructions of the decode phase are independent. Our series-parallel poset methodology is aimed precisely at modeling the notions of dependence and independence, and offers a convenient way to model the DLX 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 DLX control is presented below:

$$\boldsymbol{B} = (e_0 \bullet e_1{}^* \bullet (e_2 \bullet e_3 + (e_{4a} \otimes e_{4b} \otimes e_{4c}) \bullet (\boldsymbol{B_{DT}} + \boldsymbol{B_{ALU}} + \boldsymbol{B_{Set}} + \boldsymbol{B_{Jmp}} + \boldsymbol{B_{Br}})))^*,$$

where:

$$\boldsymbol{B_{DT}} = (e_5 \bullet (e_6 \bullet e_7{}^* + e_8{}^* \bullet (e_9 + e_{10} + e_{11} + e_{12} + e_{13})) + e_{14}) \bullet e_{15} + e_{16}$$

$$\boldsymbol{B_{ALU}} = (e_{17} + e_{18}) \bullet (e_{19} + e_{20} + e_{21} + e_{22} + e_{23} + e_{24} + e_{25} + e_{26} + e_{27}) \bullet e_{15}$$

$$\boldsymbol{B_{Set}} = (e_{17} + e_{18}) \bullet (e_{28} + e_{29} + e_{30} + e_{31} + e_{32} + e_{33}) \bullet (e_{34} + e_{35}) \bullet e_{15}$$

$$\boldsymbol{B_{Jmp}} = e_{36} + e_{37} + e_{38} \bullet (e_{36} + e_{37}) \bullet e_{39} + e_2 \bullet e_{40}$$

$$\boldsymbol{B_{Br}} = (e_{41} + e_{42}) \bullet (e_{37} + 1)$$

Each event in the DLX control model is a unique microinstruction as follows:

Event	***Microinstruction***	***Event***	***Microinstruction***
e_0	MAR ← PC	e_{21}	C ← A & Temp(AND)
e_1	IR ← Mem[MAR]	e_{22}	C ← A $\mid$ Temp (OR)
e_2	IAR ← PC	e_{23}	C ← A⊕ Temp (XOR)
e_3	PC ← 0; Clear intrpt	e_{24}	C ← A<<Temp (SLL)
e_{4a}	PC ← PC + 4	e_{25}	C ← A>>Temp (SRL)
e_{4b}	A ← RegS1	e_{26}	C←Temp<<16 (LHI)
e_{4c}	B ← RegS2	e_{27}	C←$(A_0)^{Temp}$##$(A>>Temp)_{Temp..31}$ (SRA)
e_5	MAR←A+$(IR_{16})^{16}$##$IR_{16..31}$	e_{28}	A == Temp
e_6	MDR ← B	e_{29}	A != Temp
e_7	Mem[MAR] ← MDR	e_{30}	A < Temp
e_8	MDR ← Mem[MAR]	e_{31}	A <= Temp
e_9	C←$(MDR_{24})^{24}$##$MDR_{24...31}$	e_{32}	A > Temp
e_{10}	C ← 0^{24}##$MDR_{24...31}$	e_{33}	A >= Temp
e_{11}	C ← 0^{16}##$MDR_{16...31}$	e_{34}	C ← 1; true
e_{12}	C ←$(MDR_{16})^{16}$##$MDR_{16...31}$	e_{35}	C ← 0; false
e_{13}	C ← MDR	e_{36}	PC ← A
e_{14}	C ← IAR	e_{37}	PC←PC+$(IR_{16})^{16}$##$IR_{16..31}$
e_{15}	RegD ← C	e_{38}	C ← PC
e_{16}	IAR ← A	e_{39}	R31 ← C
e_{17}	Temp ← B	e_{40}	PC ← $(IR_{16})^{16}$##$IR_{16..31}$
e_{18}	Temp ← $(IR_{16})^{16}$##$IR_{16..31}$	e_{41}	A == 0
e_{19}	C ← A + Temp (ADD)	e_{42}	A != 0
e_{20}	C ← A - Temp (SUB)		

Table 1 Microinstructions of the DLX Control Unit

4 Verifying the DLX 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 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 of only two properties from the complete set of properties mentioned above:

Property $\boldsymbol{P_1}$: "Memory addresses remain fixed for the entire duration of a memory access."

For the address to remain fixed throughout an instruction fetch or a memory data access, the contents of the Memory Access Register, MAR, must not be updated until the memory read/write is complete. By examining Tbl. 1, we observe that MAR gets modified only by microinstructions $MAR \leftarrow A+(IR_{16})^{16}\#\#IR_{16..31}$ (event e_5) and MAR ← PC (event e_0). Memory is accessed by the microinstructions IR ← Mem[MAR] (event e_1), Mem[MAR] ← MDR (event e_7), and MDR ← Mem[MAR] (event e_8). Therefore, property $\boldsymbol{P_1}$ can be specified as the following series-parallel poset expression:

$$\boldsymbol{P_1} = ((e_1{*} + e_7{*} + e_8{*}) \bullet (e_0 + e_5))^*$$

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

$$\boldsymbol{B} = (e_0 \bullet e_1{*} \bullet e_5 \bullet (e_7{*} + e_8{*}))^*$$

For the processor to operate correctly, property $\boldsymbol{P_1}$ must always be satisfied. Hence, we will use the Always Satisfied verification predicate[1]:

AS($\boldsymbol{B}$, $\boldsymbol{P_1}$):
Since both $\boldsymbol{B}$ and $\boldsymbol{P_1}$ are iterated,
$AS(\boldsymbol{B}, \boldsymbol{P_1}) = AS(e_0 \bullet e_1{*} \bullet e_5 \bullet (e_7{*}+e_8{*}),\ (e_1{*}+e_7{*}+e_8{*}) \bullet (e_0+e_5))$
$AS(e_0 \bullet e_1{*} \bullet e_5 \bullet (e_7{*}+e_8{*}),\ (e_1{*}+e_7{*}+e_8{*}) \bullet (e_0+e_5))$:

- $(e_1{*}+e_7{*}+e_8{*}) \bullet (e_0+e_5) \notin SP(\Sigma^*)$ (i.e. the property has iteration)
- $AS(e_0 \bullet e_1{*} \bullet e_5 \bullet (e_7{*}+e_8{*}),\ (e_1{*}+e_7{*}+e_8{*})) =$ [2]
  $AS(e_1{*} \bullet (e_7{*}+e_8{*}),\ (e_1{*}+e_7{*}+e_8{*}))$:

---

[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

$AS(e_1{*}\bullet(e_7{*}+e_8{*}), e_1{*}) =^2 AS(e_1{*}, e_1{*}) = TRUE$
$AS(e_1{*}\bullet(e_7{*}+e_8{*}), e_7{*}) =^2 AS(e_7{*}, e_7{*}) = TRUE$
$AS(e_1{*}\bullet(e_7{*}+e_8{*}), e_8{*}) =^2 AS(e_8{*}, e_8{*}) = TRUE$
$\Rightarrow AS(e_0\bullet e_1{*}\bullet e_5\bullet(e_7{*}+e_8{*}), (e_1{*}+e_7{*}+e_8{*})) = TRUE$
- $AS(e_0\bullet e_1{*}\bullet e_5\bullet(e_7{*}+e_8{*}), e_0+e_5) =^2 AS(e_0\bullet e_5, e_0+e_5)$:
  $AS(e_0, e_0) =^2 TRUE$
  $AS(e_5, e_5) =^2 TRUE$
$\Rightarrow AS(e_0\bullet e_1{*}\bullet e_5\bullet(e_7{*}+e_8{*}), e_0+e_5) = TRUE$
- $pred_B(\{e_0\}) = \{e_0, e_1, e_5, e_7, e_8\} \cap \{e_0\} = \{e_0\}$
- $pred_B(\{e_5\}) = \{e_0, e_1, e_5, e_7, e_8\} \cap \{e_5\} = \{e_5\}$

$\Rightarrow AS(\boldsymbol{B}, \boldsymbol{P_1}) = TRUE$

Property $\boldsymbol{P_2}$: "Before any ALU operation is attempted, the register file ports A and B are already loaded with data from the source registers."

Data is written into ports A and B by microinstructions A ← RegS1 (event e_{4b}) and B ← RegS2 (event e_{4c}). The microinstructions which use ports A and/or B in an ALU operation are labeled by events e_5, e_{19}, e_{20}, e_{21}, e_{22}, e_{23}, e_{24}, e_{25}, e_{27}, e_{28}, e_{29}, e_{30}, e_{31}, e_{32}, e_{33}, e_{41}, and e_{42}. Therefore, the property can be expressed as the following series-parallel poset expression:

$$\boldsymbol{P_2} = ((e_{4b}\otimes e_{4c}) \bullet(e_5+e_{19}+e_{20}+e_{21}+e_{22}+e_{23}+e_{24}+e_{25}+e_{27}+e_{28}+e_{29}+e_{30}+e_{31}+e_{32}+e_{33}+e_{41}+e_{42}))^*$$

The reduced behavior with respect to the events in *set*($\boldsymbol{P_2}$) is:

$$\boldsymbol{B} = ((e_{4b} \otimes e_{4c}) \bullet (e_5+e_{19}+e_{20}+e_{21}+e_{22}+e_{23}+e_{24}+e_{25}+e_{27}+e_{28}+e_{29}+e_{30}+e_{31}+e_{32}+e_{33}+e_{41}+e_{42}))^*$$

It is now easy to see that $AS(\boldsymbol{B}, \boldsymbol{P_2})$ is satisfied.

The verification of the two properties presented in this paper was carried out by hand to illustrate the operation of the verification algorithms. The actual verification of the entire property set of the DLX control was carried out with the help of a software package, developed at Iona College on the basis of the theoretical constructs presented in [8, 9, 10, 11].

5 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* $\boldsymbol{P}=(P,\leq ,l)$ 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 $\boldsymbol{P}$ and $\boldsymbol{Q}$ with $P\cap Q=\emptyset$, we define two operations on labeled posets:

Concatenation (•): $\boldsymbol{P} \bullet \boldsymbol{Q} := (P \cup Q, \leq_{P \bullet Q})$

Shuffle (⊗): $\boldsymbol{P} \otimes \boldsymbol{Q} := (P \cup Q, \leq_{P \otimes Q})$, where

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

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

A *Series-Parallel Poset* (SPP) over an alphabet Σ is defined inductively:

- The empty poset, ***1***, is a SPP
- For each σ∈Σ, the singleton labeled σ is a SPP
- If ***P*** and ***Q*** are SPPs, so are ***P•Q*** and ***P⊗Q***

The set of all series parallel posets formed from ***1*** and the singletons and closed under concatenation (•) and shuffle (⊗) forms a bimonoid denoted $SP(\Sigma^*)$ [15].

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, $\boldsymbol{e_i}$. Then, the fact that event e_i precedes event e_j is represented by $\boldsymbol{e_i \bullet e_j}$. On the other hand, the independence of events e_i and e_j is represented by $\boldsymbol{e_i \otimes e_j}$. This extends naturally to sets of events.

What does it mean for two sets of events two 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 v.v. 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 v.v.

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 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:

Figure 2 A Simple Series Parallel Poset Example

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

Interpreting series-parallel posets as descriptions of the dependence or independence of sets of events allows us to model *system behavior* in terms of the sequences of events occurring during its operation. In [8] we presented a methodology for modeling the behavior and 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]

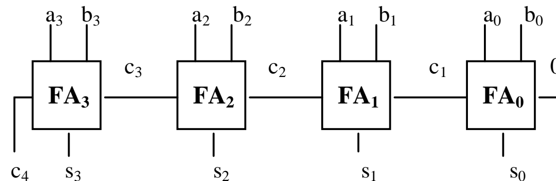

Figure 3 A Non-Iterated System: 4-bit Binary Adder

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 systems consist of non-iterated sub-systems operating in series or in parallel, such that the global output is fed back for another iteration.

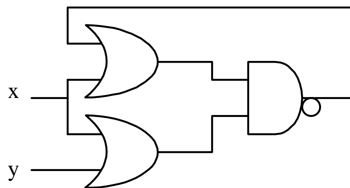

Figure 4 A Simple Globally-Iterated/Locally Non-Iterated System

The verification algorithms have a low-order polynomial time- and space complexity, which is further improved by the introduction of the behavior reduction methodology in [10].

An *iterated system* is one in which some or all events are repeated. It 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.

---

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

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

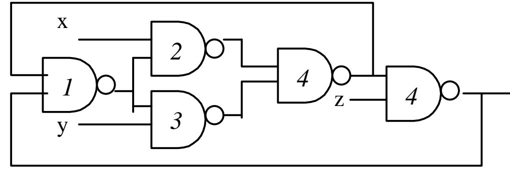

Figure 5 A Simple Iterated System (gate level)

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 $\mathbf{S} = (\mathbf{S}, +, \bullet, \otimes, *, \mathbf{0}, \mathbf{1})$ of series-parallel posets, defined as follows:

- $\mathbf{S}$ - the set of finite subsets of SP(Σ^*), closed under the semiring operations
- If $\mathbf{K}\in\mathbf{S}$ and $\mathbf{L}\in\mathbf{S}$, $\mathbf{K}+\mathbf{L} = \{P|\ P\in\mathbf{K} \vee P\in\mathbf{L}\}\in\mathbf{S}$
- If $\mathbf{K}\in\mathbf{S}$ and $\mathbf{L}\in\mathbf{S}$, $\mathbf{K}\bullet\mathbf{L} = \{P\bullet Q|\ P\in\mathbf{K} \wedge Q\in\mathbf{L}\}\in\mathbf{S}$
- If $\mathbf{K}\in\mathbf{S}$ and $\mathbf{L}\in\mathbf{S}$, $\mathbf{K}\otimes\mathbf{L}=\{P\otimes Q|\ P\in\mathbf{K} \wedge Q\in\mathbf{L}\}\in\mathbf{S}$
- If $\mathbf{K}\in\mathbf{S}$, then $\mathbf{K}^* = \mathbf{1} + \mathbf{K} + \mathbf{K}\bullet\mathbf{K} + \ldots = \Sigma_{i=0..\infty}\, \mathbf{K}^{\mathbf{i}} \in\mathbf{S}$, where $\mathbf{K}^{\mathbf{i}} = \mathbf{K}\bullet\mathbf{K} \bullet\ldots\bullet\mathbf{K}$, **i** times.
- $\mathbf{0}$ is the empty set of posets
- $\mathbf{1}$ is the empty poset

We define the *behavior*, $\mathbf{B}$, of an iterated system to be an element of the star-shuffle semiring $\mathbf{S}$, i.e. $\mathbf{B}\in\mathbf{S}$. 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 $\mathbf{e_i}$, then the behavior of the system in *Figure 5* is given by the following expression $\mathbf{B} = ((\mathbf{e_1} \bullet (\mathbf{e_2} \otimes \mathbf{e_3}) \bullet \mathbf{e_4})^* \bullet \mathbf{e_5})^*$.

We can represent the *verification properties* as sets of series-parallel posets as well, i.e. $\mathbf{P}\in\mathbf{S}$. 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 $\mathbf{P} = (\mathbf{e_2} \otimes \mathbf{e_3}) \bullet \mathbf{e_4}$.

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

- **SS**($\boldsymbol{B}$, $\boldsymbol{P}$) is a binary predicate, interpreted as “The property $\boldsymbol{P}$ is sometimes satisfied within the behavior, $\boldsymbol{B}$”. The predicate takes a behavior and a property and verifies that $\boldsymbol{P}$ can sometimes be traced within the behavior, $\boldsymbol{B}$.
- **AS**($\boldsymbol{B}$, $\boldsymbol{P}$) is a binary predicate, interpreted as “The property $\boldsymbol{P}$ is always satisfied within the behavior, $\boldsymbol{B}$”. The predicate takes a behavior and a property and verifies that $\boldsymbol{P}$ can always be traced within the behavior, $\boldsymbol{B}$, of the system.

There are four normal forms of behavior and property expressions:

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

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* $Pr(\boldsymbol{B}, set(\boldsymbol{P}))$, which takes a behavior, $\boldsymbol{B}$, and a set of events, and returns a reduced behavior, $\boldsymbol{B'}$, with respect to the events in the specified set. The effect of the projection function is to substitute $\boldsymbol{1}$ 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**($\boldsymbol{B}$, $\boldsymbol{P}$) and **AS**($\boldsymbol{B}$, $\boldsymbol{P}$) for iterated systems. In this outline, we present only **AS**($\boldsymbol{B}$, $\boldsymbol{P}$):

AS($\boldsymbol{B}$, $\boldsymbol{P}$) iff

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

In the above definitions we made use of number of functions – the labeling functions *l*(s) and *L*({$s_{1}, \ldots, s_{n}$}), 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 property events. The space complexity is O(m).

6 Contributions, Limitations, and Conclusions

In this paper, we presented the modeling and formal verification of the DLX processor control based on the recently developed series-parallel poset methodology. The technique is less expressive than some other formal verification methods, but has a low complexity. Thus, we can model complex real-world systems and protocols. Current work is on the verification of the InMOS Transputer microcode, and modeling the behavior of dataflow computers.

The issues of event sequencing and timing has been studied for a long time by many researchers - D. Dill, B. Moszkowski, Z. Manna, etc. In many respects, our approach is close to the study of language containment of behavior and property automata [2]. However, we approach the topic from a different point of view, avoiding the issue of exhaustive substring matching. Moreover, the use of the shuffle operator (⊗), 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]. However, 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 (•, ⊗, *), but models not only system behaviors but properties as well. The emphasis is on verification, and the reduced collection of operations simplifies analysis, and improves the algorithms' efficiency.

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 algorithm. Consider the simple example below:

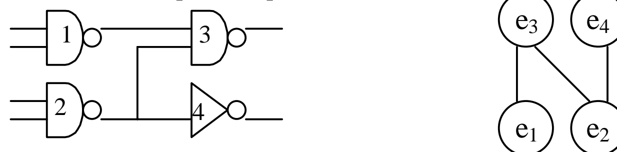

Figure 6 "N"-type event dependence in a simple system

If e_i represent the event "*gate i produces a valid output*", then the event dependency diagram has the "N"-shape described on the right. This type of dependency cannot be modeled only with operations •, ⊗, and *. Current work is aimed at extending our verification methodology to deal with "N"-type 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*", TCS, 1981

[4] V.Pratt, "*Modeling Concurrency with Partial Orders*", Int. Journal of Parallel Prog., 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 Formal Methods in Systems Design, 8 (1), 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*", Proceedings 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*", Proc. of 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] J.Hennessy, D.Patterson, "*Computer Architecture: A Quantitative Approach*", Morgan Kaufmann Publ. Inc., 1990

[15] Bloom, Z. Esik, "*Free Shuffle Algebras in Language Varieties*", Theoretical Computer Science 163 (1996) 55-98, Elsevier