

SPPV: A NEW FORMAL VERIFICATION ENVIRONMENT

Lubomir Ivanov

Michael Shute

Department of Computer Science, Iona College, 715 North Avenue, New Rochelle, NY 10801

livanov@iona.edu

mshute@iona.edu

ABSTRACT

Formal Verification has become an integral part of the product development cycle leading to a demand for powerful, yet easy to use tools, which conceal the complexity of the underlying mathematical arguments through the use of convenient interfaces and automatic verification. In this paper we present a new formal verification environment – SPPV – based on series-parallel poset verification. SPPV allows fast, automated verification of event sequencing in complex systems. The system model and properties can be expressed as series-parallel poset expressions or in Verilog.

1. INTRODUCTION

Over the past decade, the research area of formal verification has developed rapidly, and finds its place in the product development cycle of many established computer companies as an alternative to exhaustive product testing and simulation. The goal of formal verification is to prove the correctness of a particular design by creating a mathematical model of the system against which various properties are checked. Many different methods of formal verification have emerged. Some are extremely powerful, but cannot be fully automated (e.g. Theorem Proving). Others such as Model Checking [1] and ω -Automata Verification [2] are almost as powerful and can be automated, but suffer from the state explosion problem, i.e. the size of the system model is exponential in the number of system variables, which makes the verification task very time-consuming. Nevertheless, these methods have led to the development of practical industrial verification tools, which, used on small to medium sized circuits, perform extremely well. Examples of such tools include Verity, SMV, FormalCheck, etc. To improve the speed of the verification process, some methods sacrifice some generality and expressiveness, focusing instead on concrete verification tasks such as timing verification. To achieve such speedup, many of these methods use the mathematical notion of partial orders [3-7]. The goal is to avoid the study of all possible interleavings of system events. In [8-15], we introduced a new partial order verification method, referred to as *series-parallel poset verification*. The method has been used to verify the behavior of various complex systems and protocols, including the PCI bus protocol, MESI cache protocol, and T414 Transputer microcode.

In this paper we present the SPPV Formal Verification Environment based on series-parallel poset verification. The software is useful for proving the correct sequencing of events in complex systems. A convenient graphics interface allows the user to specify the system behavior, properties to be checked, and other parameters of the verification process. The behavior and properties can be specified either as series-parallel poset expressions, or with the Verilog hardware description language. If a property fails to be satisfied within the system model, an explanation for the failure is provided. The remainder of the paper outlines the series-parallel poset verification method, and introduces the SPPV verification environment along with a few simple examples of its use. A large scale example of the use of SPPV – a pipelined CPU verification – will appear as a separate paper in the proceedings of MWSCAS'02.

2. OVERVIEW OF SERIES-PARALLEL POSET

Series-parallel poset verification is based on the notion of a *partially ordered set (poset)*, which is a set with a reflexive, antisymmetric, and transitive relation defined on the set elements. With the help of two operations – concatenation (\bullet) and shuffle (\otimes) – a *series-parallel poset* over an alphabet Σ is defined inductively:

- The empty poset, I , is a series-parallel poset
- For each $\sigma \in \Sigma$, the singleton poset labeled σ is a series-parallel poset
- If P and Q are series-parallel posets, 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 and shuffle forms a bimonoid denoted $SP(\Sigma^*)$ [16]. For purpose of formal verification, the alphabet, Σ , consists of all distinct events occurring in the system whose behavior is to be verified. The concatenation operation is used to indicate event sequencing, whereas shuffle denotes event independence. If e_1 and e_2 are two events, then their concatenation $e_1 \bullet e_2$ represents the fact that event e_1 occurs before event e_2 , while the shuffle $e_1 \otimes e_2$ indicates that both events occur but in an unspecified order.¹ The definitions of shuffle and concatenation are naturally extended to sets of event, and used as the basis for the verification algorithms for non-iterated systems – systems in which events are distinct and not repeated. To describe behavior of iterated systems (in which some or all events may be repeated) a generalization of the idea of series-parallel posets is needed:

- The shuffle and concatenation operations are redefined to operate over sets of posets

- Two new operations are introduced – plus (+) and star (*)

In the context of verification, plus is used to denote that either one or another event sequence occurs, i.e. choice of event sequences. The star operation (modeled after the Kleene star) is used to represent repetition of event sequences, i.e. the possibility that a sequence of events occurs 0 or more times. Thus, the set of finite subsets of $SP(\Sigma^*)$ closed under the above four operations defines a new structure – the star shuffle semiring $\mathcal{S} = (\mathcal{S}, +, \bullet, \otimes, *, \theta, I)$.

Formal verification requires that three components be specified:

- *System Behavior*: A mathematical model of the system
- *Verification Properties*: A set of expressions – one for each of the properties to be tested.
- *Proof Method*: A formal technique to allow each property to be verified within the system behavior

In the context of series-parallel poset verification, the first two of the above “ingredients” are defined as follows:

- The *Behavior*, \mathcal{B} , of an iterated system is an element of the star-shuffle semiring \mathcal{S} . Thus, the behavior of an iterated system is a set of series-parallel posets.
- A *Verification Property* is also a set of series-parallel posets but over a subset of the alphabet Σ .

¹ For a formal definition of shuffle, concatenations, and other concepts introduced below, refer to [8, 10, 13].

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: $B = B_1^*$ & $P = P_1^*$

Before the verification process begins, the behavior expression is reduced to include only the events of interest specified in the property. This simplifies the reasoning about sets of events and the complexity of the verification algorithms [10, 13].

The third “ingredient” – the proof method – is implicit in the *verification questions* that can be asked about a property. The verification questions are specified as *predicates*:

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

The formal definitions of the two verification predicates **SS(B, P)** and **AS(B, P)** are given below:

SS(B, P) iff

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

AS(B, P) iff

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

The definitions use a number of functions and auxiliary predicates:

- Labeling functions:
 - $l: S \rightarrow \Sigma, l(s) = \sigma, s \in S$ and $\sigma \in \Sigma$
 - $L: \wp(S) \rightarrow \wp(\Sigma), L(\{s_i \mid 0 \leq i < n\}) = \{l(s_i) \mid 0 \leq i < n\}$, where *S* is the set of singleton posets, $\wp(S)$ the powerset of *S*, and Σ the alphabet of system events
- Function *set(B)*, which takes a series-parallel poset expression, *B*, and returns the set of its elements.
- Function predecessor, *pred_B(P)*, which takes the set of vertices of a series-parallel poset expression *P* and returns the set of predecessors of those vertices in a series-parallel poset expression, *B*, or \emptyset if the predecessor set is empty.

Function *NI(B)*, which allows the verification of a non-iterated property within a single iteration of an iterated system with behavior *B* by reducing an iterated behavior to a non-iterated one.

- Function *lisc(B, e)*, which returns the least iterated sub-component of an iterated behavior *B*, which contains the event *e*, e.g. $\text{lisc}((e_1^* \otimes e_2)^* \bullet e_3^*, e_2) = (e_1^* \otimes e_2)^*$
- Predicate **Independent_B(P, Q)**, which verifies that two sets of series-parallel posets of events, *P* and *Q*, are independent within the context of a behavior, *B*.
- Predicates **AS_NI(B, P)** and **SS_NI(B, P)**, which verify that a non-iterated property, *P*, is always/sometimes satisfied within the context of a non-iterated behavior, *B*.

The definitions of the verification predicates **AS(B, P)** and **SS(B, P)** serve as the basis of our verification algorithm. The algorithm works by recursively decomposing the specified property into subproperties, which are verified and their interdependence tested. For further details refer to [13].

3. THE SPPV ENVIRONMENT

The SPPV (Series-Parallel Poset Verification) Environment is a software tool based on the theory of series-parallel poset verification outlined above. The software consists of three components:

- The Verifier
- The Graphics User Interface (GUI)
- Verilog-to-SPP Expression Converter

3.1 The Verifier

The Verifier is the core piece of the SPPV Environment, being responsible for proving the correctness of properties within the specified system behavior. The inputs to the verifier are a behavior and a property expression in series-parallel poset format, and a verification choice – test for sometimes satisfied, always satisfied, or both. If the property is satisfied within the behavior, the Verifier confirms this fact and prints an appropriate message. If the property is not satisfied, a reason for the failure is printed to help the user debug his/her design. The Verifier implements the **AS(B, P)** and **SS(B, P)** predicates defined in the previous section along with all auxiliary functions and predicates. The implementation is in C++. To guarantee maximum efficiency of the verification process, the implementations of some of the functions depart from

the recursive definition given in the theory. The semantics however remain unchanged.

3.2 The Graphics User Interface



The GUI component of the SPPV Environment allows a convenient way to specify the behavior, properties, and verification parameters. There are two ways to enter a behavior and a property expression – interactively, by typing it directly into the appropriate window, or from a plain text file, previously typed in a text editor. If the user opts to enter a behavior in Verilog format, the interactive input option is automatically disabled, requiring the Verilog description to be prepared in a file. The user can specify the chosen method of input by selecting one of the two radio-buttons labeled Verilog and Series Parallel Poset in the main window. In addition, the user must select one of the three verification options – “Sometimes Satisfied”, “Always Satisfied”, or “Test Both”. After the behavior and the property have been entered, and the verification choice selected, the user clicks on the “Verify” button to begin the automatic verification process. When completed, the appropriate results are displayed in a separate message window.



3.3 Verilog-to-SPP Expression Converter

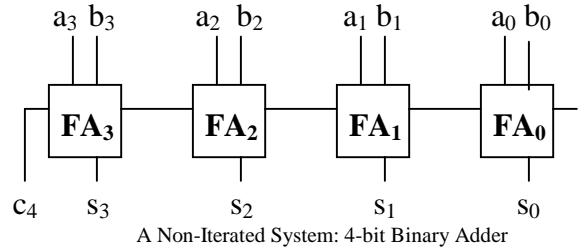
Allowing the user to specify a system description in Verilog format necessitates conversion to series-parallel poset expression format required by the Verifier. The conversion is carried out automatically by a program, which reads the name of the Verilog file specified by the user, parses the descriptions, carries out the

appropriate conversion, and updates the internal behavior variable passed on to the Verifier. The converted behavior in series-parallel poset form is displayed in a window for the user to examine. So far, only a very restricted subset of Verilog is supported, limited to basic modules implemented with gates and buffers. We are actively working on extending the capabilities of our software to support a much wider set of Verilog specifications.

4. USING THE SPPV ENVIRONMENT

We will illustrate the use of the SPPV Environment with two simple examples:

Example 1:

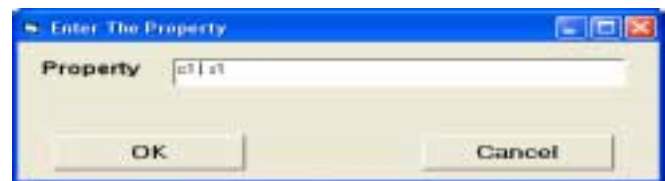


Let us denote by s_0 the event “full adder FA₀ produces a valid sum bit s_0 ”, by c_1 the event “FA₀ produces carry out bit c_1 ”, etc. The behavior of this simple circuit then is given by the following series-parallel poset expression:

$$B = (s_0 \otimes (c_1 \bullet (s_1 \otimes (c_2 \bullet (s_2 \otimes (c_3 \bullet (s_3 \otimes c_4)))))))$$

The first property we wish to test is “Carry bit c_1 always precedes sum bit s_1 ”, expressed as a series-parallel poset as $c_1 \bullet s_1$.

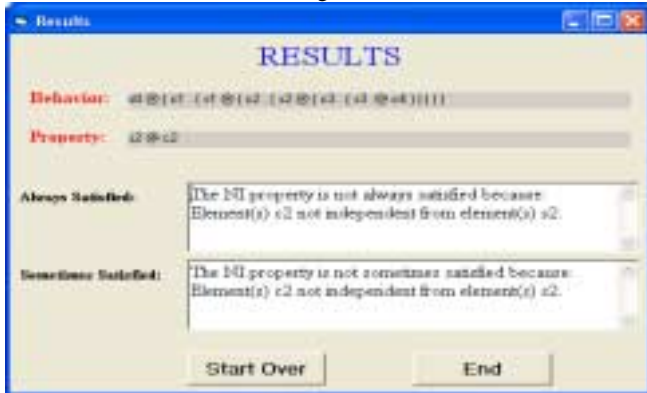
The behavior and property are entered interactively:



Having selected the “Always Satisfied” option on the main SPPV window, and clicked “Verify”, we obtain the following result:



The second property we wish to test is “The sum bit s_2 independent of the carry bit c_2 ”, expressed as a series-parallel poset as $s_2 \otimes c_2$. This time we obtain the following:



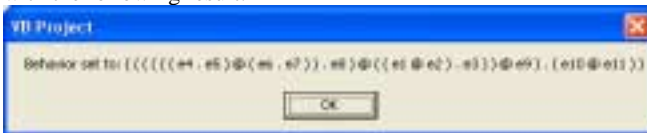
Example 2: This time we use the following Verilog circuit description from a text file prepared in advance:

```

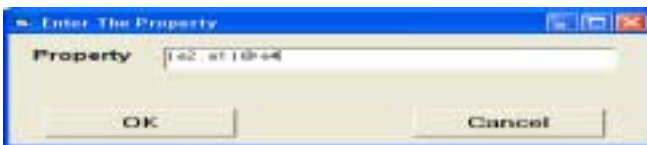
module Circuit(In1, In2, In3, In4, In5, Out1, Out2, Out3);
    output Out1, Out2, Out3;
    input In1, In2, In3, In4, In5;
    buf (e1, In1), (e2, In2),
        (e4, In3), (e6, In4),
        (e9, In5);
    and (e3, e1, e2);
    nand (e10, e3, e8, e9);
    or (e11, e3, e8, e9);
    nor (e8, e5, e7);
    not (e5, e4), (e7, e6), (Out1, e10),
        (Out2, e11), (Out3, e9);
endmodule

```

The conversion to series-parallel posets is carried out automatically with the following result:



A sample property is specified interactively as follows:



The result of the verification with the “Sometimes Satisfied” option enabled is given below:



5. CONCLUSIONS

In this paper we presented a new formal verification tool – the SPPV Environment - for proving the proper sequencing and/or independence of events in hardware systems. The tool allows the inputs to be specified as series-parallel poset expressions or as Verilog descriptions. Relative to other verification software, the scope of the tasks that can be handled by our product is more limited, but the low-order complexity guarantees that even large-scale commercial designs can be verified in relatively short time. We have used SPPV in the verification of a pipelined CPU and other complex systems.

Current work involves extending the series-parallel poset theory and its implementation to handle an even larger class of systems, and redesigning the Verilog-to-Series-Parallel Poset conversion software to handle more complex Verilog descriptions.

6. REFERENCES

- [1] K. McMillan, “Symbolic Model Checking”, Kluwer, 1993
- [2] R.Kurshan, “Computer Aided Verification of Coordinating Processes: The Automata-Theoretic Approach”, Princeton Series in Computer Science, 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”, IJPP, 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”, Proc of IFIP, 1996
- [7] D. Peled, “Combining Partial Order Reductions with On-the-Fly Model Checking”, JFMSD, 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”, Proceedings of ASIC/SOC’99, 1999
- [11] L. Ivanov, R. Nunna, “Modeling and Verification of an Interconnect Bus Protocol”, Proc. . . .of MWSCAS’00, 2000
- [12] L.Ivanov, R.Nunna “Modeling and Verification of Cache Coherence Protocols”, ISCAS’01, Sydney, Australia, 2001
- [13] L.Ivanov, R.Nunna “Modeling and Verification of Iterated Systems and Protocols”, MWSCAS’01, Dayton, OH, 2001
- [14] L.Ivanov “Formal Verification of a Microprocessor Control”, MWSCAS’01, Dayton, OH, 2001
- [15] L.Ivanov “Formal Verification of Microinstruction Sequencing”, Proceedings of ICCIT’01, Montclair, NJ, 2001
- [16] S. Bloom, Z. Esik, “Free Shuffle Algebras in Language Varieties”, TCS 163 (1996) 55-98, Elsevier