# Automatic Extraction of Non-Iterated System Behavior from Verilog Specifications

Lubomir Ivanov
Department of Computer Science, Iona College, 715 North Avenue, New Rochelle, NY 10801
livanov@iona.edu

## ABSTRACT

*In this paper we present an algorithm for automatic extraction of system behavior from a structural Verilog specification. The algorithm generates a series-parallel poset expression for the behavior of the system, which is then used for verification purposes in the context of the SPPV Formal Verification Environment. The issues of correctness and time complexity of the algorithm are briefly discussed. A small example is presented to illustrate the conversion process from specification to a series-parallel poset expression.*

**Keywords:** *Series-parallel poset, formal verification, Verilog behavior extraction*

## 1. INTRODUCTION

Over the past decade, the research area of formal verification has seen a dramatic growth, and has been accepted by the industry as a standard phase in the design and development of complex hardware and software systems. Many powerful verification methods have emerged [1-7], and some have led to the development of software tools for automatic verification of design correctness, used extensively by the computer manufacturing industry. In a series of papers [8-17], we introduced a new, efficient formal verification method – Series-Parallel Poset Verification – and demonstrated its usefulness in verifying the behavior of complex real-world systems and protocols such as the PCI bus protocol, the MESI cache coherence protocol, the T414 Transputer microcode, and the integer pipeline of the DLX microprocessor. The verification of these large-scale systems was carried out with the SPPV Verification Environment – a software based on the series-parallel poset methodology. The SPPV Environment provides a convenient GUI interface (accessible as a stand-alone application or as an applet on the web), which allows a user to specify a system behavior and a set of properties to be verified, and initiate the automatic verification process. If a particular property is not satisfied, an explanation is provided to help the designer with the debugging process.

One deficiency of the SPPV Environment is that the user must become familiar with the language of series-parallel posets, and learn to accurately specify behaviors and properties as series-parallel poset expressions. While learning to use series-parallel posets is not difficult or time consuming, specifying the behavior of a large, complex real-world system as a series-parallel poset with hundreds of thousands of elements can be a daunting and error-prone task. Moreover, hardware systems are usually specified using a hardware description language (most commonly Verilog or VHDL), and to carry out a formal verification with SPPV, the user will have to convert such a HDL description to a series-parallel poset expression.

This paper presents an algorithm for automatic extraction of system behavior from a structural Verilog specification to a series-parallel poset form, suitable for use in the SPPV Verification Environment. The availability of this algorithm and its incorporation into the SPPV Verification Environment greatly simplifies the task of the designer performing verification with SPPV, and allows the use of the SPPV Environment with minimal prior knowledge of the underlying mathematical methodology.

The paper is organized as follows: We begin by outlining the most essential aspects of series-parallel poset verification and the SPPV Verification Environment. We then present the algorithm for automatic behavior extraction from structural Verilog description, and briefly discuss its efficiency and correctness. We illustrate the use of the algorithm with a small example, and, finally, outline directions for future work.

## 2. SERIES PARALLEL POSET VERIFICATION

### 2.1. Methodology

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 (•) and shuffle (⊗) - a *series-parallel poset* over an alphabet Σ is defined inductively as follows:

- The empty poset, ***1***, 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•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 $SP(\Sigma^*)$ [18]. 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 sequencing and independence of events extends naturally to sets of events.

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., i.e. ***P*** and ***Q*** are independent if the set of predecessors of ***P*** does not involve any event from ***Q*** and vice versa.

---

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

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

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

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

$\boldsymbol{B} = ((\boldsymbol{e_1} \otimes \boldsymbol{e_2}) \bullet \boldsymbol{e_3}) \otimes \boldsymbol{e_4}$

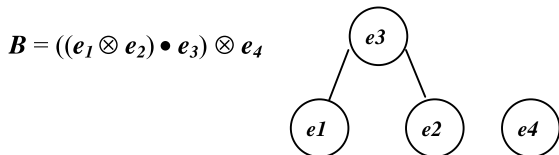


Fig.1 A Simple Series Parallel Poset Example

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 $\boldsymbol{e_3}$ does. $\boldsymbol{P_1}$ does not always precede $\boldsymbol{P_2}$ since one possible sequence of events is $\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \boldsymbol{e_4}$. But $\boldsymbol{P_1}$ may sometimes precede $\boldsymbol{P_2}$ since another possible 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 the *behavior* of a system in terms of the sequences of events occurring during its operation. In [8] we presented an approach to modeling the behavior and properties of non-iterated systems with series-parallel posets. A *non-iterated system* is one, in which events are distinct and not repeated.[2]

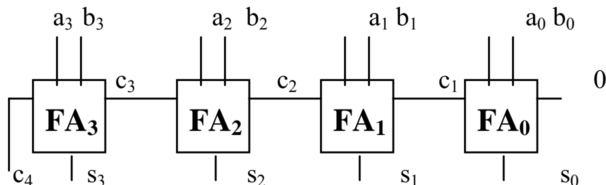


Fig.2 A Non-Iterated System: 4-bit Binary Adder

Fig.2 presents an example of a non-iterated system – a 4-bit binary adder. If we label the change of the sum and carry outputs due to a change in the inputs as events, the behavior of the above non-iterated system can be modeled as the following series-parallel poset expression:

$$\boldsymbol{B} = (e_1 \otimes (e_2(e_3 \otimes (e_4(e_5 \otimes (e_6(e_7 \otimes e_8)))))))$$

For a more detailed explanation of this and other examples, refer to [8].

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, and cache protocols, sequential circuits, feedback control systems, etc.

To describe behavior of *iterated systems*, 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, + is used to denote that either one or another event sequence occurs, i.e. it represents a 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 $\boldsymbol{S}=(\boldsymbol{S}, +, \bullet, \otimes, *, \boldsymbol{0}, \boldsymbol{1})$.

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 of iterated systems, the first two of the above "ingredients" are defined as follows:

- The *Behavior*, $\boldsymbol{B}$, of an iterated system is an element of the star-shuffle semiring $\boldsymbol{S}$. 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.

There are four normal forms of behavior and property expressions:

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

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**($\boldsymbol{B}$, $\boldsymbol{P}$) 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 the property can sometimes be traced within the behavior.
- **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 the property can always be traced within the behavior.

[2] Not all non-iterated systems can be expressed with series-parallel posets. See [8] for details.

The formal definitions of the two verification predicates **SS**(***B***, ***P***) and **AS**(***B***, ***P***) are quite complex, and examine the conditions for satisfiability of all four types of property normal forms with respect to the four types of behavior normal forms. Refer to [13] for the formal definition, correctness, and complexity arguments of the predicates and the verification algorithms that they give rise to.

### 2.2.The SPPV Formal Verification Environment

The SPPV Formal Verification Environment is an integrated software tool, based on the series-parallel poset verification methodology outline above. SPPV is Java based, and can, therefore, be easily ported to any system. It is available as a stand-alone application, or as web-based applet. The SPPV Environment can be used for verifying properties pertaining to the correct sequencing of events in the broadest possible sense in a variety of iterated and non-iterated system. We have used the software for the verification of communication and cache coherence protocols, control microcode, integer pipeline operation. The advantages of using SPPV over other commercial verification products are its extremely efficient operation ($O(n+m^3)$ worst case, $O(n+m^2)$ average case time complexity, and $O(m)$ space complexity, where $n$ is the number of events in the behavior, and $m$ is the number of events in the property being verified, which is usually very small), fully automated verification with no need for user interaction, and a simple, easy to use graphics user interface (Fig.3).

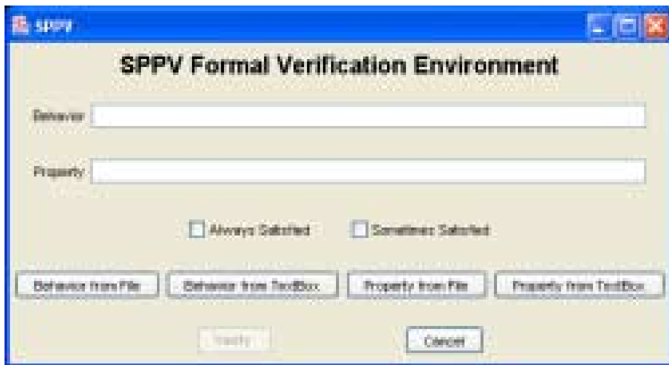


Fig.3 The SPPV Environment

The weaknesses of the SPPV Verification Environment are that it is not applicable to some types of systems (see [10, 13] and the remarks in the next section for more details), and that, until now, it required the user of the software to become familiar with the language of series-parallel posets in order to be able to specify the system behavior and verification properties. These can be entered in real time into the appropriate text boxes, or prerecorded in files, which are loaded into SPPV before the verification process begins. While specifying verification properties is relatively easy since they usually do not involve many events, specifying the behavior of a complex system that may sometimes involve millions of events with various mutual interdependencies is a daunting, if not an impossible task even for a practitioner well versed in the theory of series-parallel posets. Therefore, the development of the automatic behavior extraction algorithm not only significantly alleviates the task of the user of SPPV, but also makes possible the verification of systems for which describing the behavior by hand would have been tremendously impractical.

## 3. AUTOMATIC EXTRACTION OF BEHAVIORS FROM VERILOG SPECIFICATION

The Verilog hardware description language has established itself as the de-facto industry standard for system specification and design. It allows the system to be described either structurally or behaviorally. The structural description of a system describes its components, and their interconnections and interdependencies. The behavioral description ignores the structural issues, concentrating on the input-output behavior of the system, i.e. the reaction of the system to input stimuli over time.

Our conversion software reads a structural Verilog description of a non-iterated system from an input Verilog file, automatically extracts the system behavior, and formulates the corresponding behavior series-parallel poset expression, which can then be used for verification with the SPPV Environment. The conversion software is written in Java and is too involved and large to be presented here. Instead we present a simplified, pseudo-code version of the conversion algorithm, followed by a detailed explanation of its operation.

```
/* -- Driver function -- */
boolean extractor()
{
   Parse(Verlog_file);
   Associations();
   e = Output_node();
  // Select an unvisited output node
   string B = Bb(e);

   while (e = Output_node() != null)
   B = " @ " + Bb(e);
   write(B, SPP_Behavior_file);
}

/* -- Backward Behavior -- */
string Bb(string e)
{
   if (sizeof(pred(e)) == 0)
  return e;
  else if (sizeof(pred(e)) == 1)
  {
  if (pred_associated(e, L) == 0)
    return Bb(pred(e)) + " . " + e;
  else if (pred_associated(e, L) == 1)
    return Bb(pred(e)) + " . ( " + e + " @ " +
                                Bf(L)+ " ) ";
  else
  {
     string B = Bb(pred(e)) + " . ( " + e;
     Iterator i = L.iterator();
     while (i.hasNext())
        B += " @ " + Bf(i.next());
     B += " ) ";
```

```
      return B;
    }
}
else
{
  string B = " ( ";
  Iterator i = pred(e).iterator();
  B += Bb(i.next());
  while (i.hasNext())
    B += " @ " + Bb(i.next());
  B += " ) . ";
  if (pred_associated(e, L) == 0)
    return B + e;
  else if (pred_associated(e, L) == 1)
    return B+" ( "+e+" @ "+Bf(L)+" ) ";
  else
  {
    B += " ( " + e;
    Iterator i = L.iterator();
    while (i.hasNext())
      B += " @ " + Bf(i.next());
    B += " ) ";
    return B;
  }
}
done[e] = true;
}

/* -- Forward Behavior -- */
string Bf(string e)
{
  if (sizeof(succ(e)) == 0)
    return e;
  else if (sizeof(pred(e)) == 1)
  {
   if (succ_associated(e, L) == 0)
     return e + " . " + Bf(succ(e));
   else if (succ_associated(e, L) == 1)
     return " ( "+ e +" @ " + Bb(L) + " ) . "
                                          + Bf(succ(e));
    else
    {
       string B = " ( " + e;
       Iterator i = L.iterator();
       while (i.hasNext())
           B += " @ " + Bb(i.next());
       B += " ) . " + Bf(succ(e)) ;
return B;
    }
  }
  else
  {
    string B = "( ";
    Iterator i = pred(e).iterator();
    B += Bf(i.next());
    while (i.hasNext())
       B += " @ " + Bf(i.next());
    B += " ) ";
    if (pred_associated(e, L) == 0)
      return e + " . " + B;
    else if (pred_associated(e, L) == 1)
      return " ( " +e+" @ "+Bb(L)+" ) . "+B;
    else
    {
        string B1 += " ( " + e;
        Iterator i = L.iterator();
        while (i.hasNext())
           B1 += " @ " + Bb(i.next());
        B1 += " ) . " + B;
        return B1;
     }
  }
  done[e] = true;
}
```

The conversion involves the following steps:

- *Parsing the Verilog specification*
- *Determining event association*
- *Generating a series-parallel poset expression*

Parsing the Verilog specification involves reading through the list of components and interconnections, associating an "event" with each component, and extracting information about each event's predecessors and successors. The information is recorded in several data structures used further in the conversion to a series-parallel poset. This step of the algorithm is accomplished in linear time in the size of the circuit.

The second step of the conversion involves determining event association. Two or more events are predecessor-associated if they share the same set of predecessors. Two or more events are successor-associated if they share the same set of successors. If two events share no predecessors/successors, then they are not associated. The conversion algorithm compares the predecessor set of each event with the predecessor sets of the remaining events, and lists the events with identical predecessor sets as predecessor-associated. The procedure is repeated with the successor sets to determine successor associativity. At this point, it is important to point out an aspect of series-parallel poset theory, which has a bearing of the verifiability of a circuit or system: in a series-parallel poset, any events that share a common predecessor, must share all their predecessors. In other words, if event $e_3$ has predecessors $e_1$ and $e_2$, and event $e_4$ has only $e_1$ as a predecessor, then a series-parallel poset expression cannot be constructed to relate $e_1$, $e_2$, $e_3$, and $e_4$. Formally, this series-parallel poset property can be expressed as the implication:

$$\text{pred}(e_i) \cap \text{pred}(e_j) \neq 0 \rightarrow \text{pred}(e_i) = \text{pred}(e_j)$$

which must hold for all elements of the series-parallel poset. Thus, any system for which the above implication does not hold cannot be modeled in the context of the series-parallel poset methodology. Therefore, if the algorithm encounters two events which share some, but not all of their predecessors, the conversion is terminated and an error message is displayed indicating to the user that the system cannot be modeled in the context of the series-parallel poset methodology. Determining event associativity takes a quadratic time in the size of the circuit.

The third step in the conversion algorithm is formulating the series-parallel poset expression. The idea is to form expressions for each of the system's independent sub-systems, and then combine these expressions with the shuffle operator $(\otimes)$[3]. To compute the expressions for each independent subsystem, the conversion algorithm selects an output node which has not been previously examined, and applies to it the recursive function Bb(), which computes the backward behavior of that event.

[3] Of course, if the system does not consist of independent sub-components, a single expression is generated.

First, function Bb() has to determine the number of predecessors of the input event. If the event has no predecessors, then the expression returned is the label of the event itself (base case of the recursion).

If the event has one predecessor, function Bb() must determine if there are other events in the system, which share the same predecessor with the input event. It consults the predecessor associativity list of the event, and if the list is empty, the function returns the backward behavior of the single predecessor concatenated with the event itself, i.e. Bb(pred(e))•e, where Bb(pred(e)) initiates another recursive call to Bb(). If there is one other event which has the same predecessor as the input event, then the function returns the backward behavior of the predecessor concatenated with the shuffle of the input event and forward behavior of its predecessor-associate peer, i.e. Bb(pred(e))•(e⊗Bf(L))[4]. The forward behavior is computed by recursive function Bf(), which is dual to Bb(). Finally, if there are multiple events predecessor-associated with the input event of Bb(), the function returns the backward behavior of the predecessor concatenated with the shuffle of e and the forward behaviors of all of its predecessor-associated peers.

If the input event, e, of Bb() has multiple predecessors, then the shuffle of their backward behaviors is computed and concatenated either with the event e only, or with the shuffle of e and the forward behaviors of all its predecessor-associated peers.

After each event has been examined, it is marked "done", and removed from further consideration. Thus, in essence, each event is visited only once, which implies that the running time of this step is linearly bounded by the size of the circuit. Therefore, the overall running time of the conversion algorithm is dominated by the time needed to compute the event associations, and hence the time complexity of the conversion is $O(n^2)$. A detailed proof of the correctness of this algorithm is beyond the page limit of this paper. It suffices to say that the recursive calls have well-defined base cases, and, since each event is examined only once, that there is a finite number of recursive calls. Thus, the recursion terminates in a finite number of steps, yielding the desired series-parallel poset expression.

## 4. A SMALL EXAMPLE

To illustrate the operation of the conversion algorithm, we present a very small example. Consider the circuit presented in Fig.4.

We begin by labeling each component by $e_i$ as indicated in the logic diagram.

[4] L points to the head of the association list

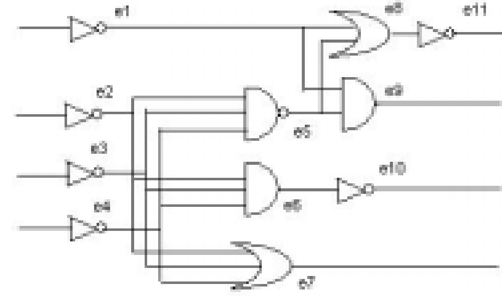


Fig.4

The first and second steps of the algorithm parse the description and generate the table below:

| ***event*** | ***pred*** | ***pred-assoc.*** | ***succ.*** | ***succ-assoc.*** | ***done*** |
|---|---|---|---|---|---|
| $e_1$ | - | - | $e_8$ , $e_9$ | $e_5$ | |
| $e_2$ | - | - | $e_5$ , $e_6$, $e_7$ | $e_3$ , $e_4$ | |
| $e_3$ | - | - | $e_5$ , $e_6$, $e_7$ | $e_2$ , $e_4$ | |
| $e_4$ | - | - | $e_5$ , $e_6$, $e_7$ | $e_2$ , $e_3$ | |
| $e_5$ | $e_2$ , $e_3$, $e_4$ | $e_6$ , $e_7$ | $e_8$ , $e_9$ | $e_1$ | |
| $e_6$ | $e_2$ , $e_3$, $e_4$ | $e_5$ , $e_7$ | $e_{10}$ | - | |
| $e_7$ | $e_2$ , $e_3$, $e_4$ | $e_5$ , $e_6$ | - | - | |
| $e_8$ | $e_1$, $e_5$ | $e_9$ | $e_{11}$ | - | |
| $e_9$ | $e_1$, $e_5$ | $e_8$ | - | - | |
| $e_{10}$ | $e_6$ | - | - | - | |
| $e_{11}$ | $e_8$ | - | - | - | |

To form the series-parallel poset expression, we find an output node (i.e. node without a successor), and compute its backward behavior, Bb(). For the sake of the example, let us choose node $e_7$. Applying function Bb() to $e_7$ yields the following expression:

$$(\mathrm{Bb}(e_2) \otimes \mathrm{Bb}(e_3) \otimes \mathrm{Bb}(e_4)) \bullet (e_7 \otimes \mathrm{Bf}(e_5) \otimes \mathrm{Bf}(e_6))$$

This reflects the fact that $e_2$, $e_3$, and $e_4$ are predecessors of $e_7$, while $e_5$ and $e_6$ are predecessor-associated peers of $e_7$. The subsequent recursive calls to function Bb() with $e_2$, $e_3$, and $e_4$ as parameters return $e_2$, $e_3$, and $e_4$ themselves as these nodes have no predecessors, and no predecessor-associated peers. The two calls to function Bf() with $e_5$ and $e_6$ as parameters expand the above expression to the following:

$$(e_2 \otimes e_3 \otimes e_4) \bullet (e_7 \otimes (e_5 \otimes \mathrm{Bb}(e_1)) \bullet (\mathrm{Bf}(e_8) \otimes \mathrm{Bf}(e_9)) \otimes e_6 \bullet \mathrm{Bf}(e_{10}))$$

One more level of recursion expands the expression to the following:

$$(e_2 \otimes e_3 \otimes e_4) \bullet (e_7 \otimes (e_5 \otimes e_1) \bullet ( e_8 \bullet \mathrm{Bf}(e_{11}) \otimes e_9) \otimes e_6 \bullet e_{10})$$

The last recursive call to $\mathrm{Bf}(e_{11})$ expands the expression to its final form:

$$(e_2 \otimes e_3 \otimes e_4) \bullet (e_7 \otimes (e_5 \otimes e_1) \bullet ( e_8 \bullet e_{11} \otimes e_9) \otimes e_6 \bullet e_{10})$$

Since there are no other independently functioning sub-systems, the above expression constitutes the overall system behavior. This expression can now be fed in as input to the SPPV Formal Verification Environment.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an algorithm for automatic extraction of system behavior from structural Verilog specification, and conversion to a series-parallel poset expression, which can be used in the context of the SPPV Formal Verification Environment for the purpose of verifying event sequencing, independence, and other timing properties. The algorithm and the software developed from it eliminate the necessity for a user to become thoroughly familiar with the formal theory and use of series-parallel posets, thus making the SPPV Environment easily accessible to any practicing engineer with a working knowledge of Verilog hardware description language.

Further work will explore the issue of automatic behavior extraction for iterated systems – in essence, systems with feedback, such as sequential circuits. As noted in section 2.1 of this paper, the series-parallel poset expressions are significantly more complex in the case of iterated systems involving sets of series-parallel posets and four operators rather than the two considered in this paper. Work on this topic is currently under way.

A further development will be the implementation of algorithms for automatic behavior extraction from a behavioral Verilog specification. Once this is accomplished, it will be fairly easy to extend the methodology to other hardware description languages such as VHDL, since the core algorithms remain the same, with only the initial parsing of the system specification requiring modification. This will allow an even larger number of engineers and hardware developers to avail themselves of the SPPV Formal Verification Environment, and begin using it as a natural step of the product development cycle.

## 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*", 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*", Proc 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*", 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] L.Ivanov "*Modeling and Verification of a Pipelined CPU*" MWSCAS'02, Tulsa, OK, 2002

[17] L.Ivanov "*SPPV: A New Formal Verification Environment*" MWSCAS'02, Tulsa, OK, 2002

[18] S. Bloom, Z. Esik, "*Free Shuffle Algebras in Language Varieties*", TCS 163 (1996) 55-98, Elsevier