A VHDL Based Environment for System Level Design and Analysis

Gnanasekaran Swaminathan, Ramesh Rao, James H. Aylor, and Barry W. Johnson
University of Virginia, Charlottesville

Abstract
The UVA uninterpreted modeling methodology uses a set of predefined primitive elements to model computer systems that can be used to explore different design alternatives. In this paper, we present an overview of the VHDL perspective of our UVA design methodology. We present Colored Petri Net models for the primitive elements that formalizes the UVA methodology. We then present a translation algorithm which translates the Petri Net (PN) model to VHDL so that the PN model can be simulated. In order to speed up the simulation, we also present a set of reduction rules that reduces the complexity of the PN model.

I Introduction
As the set of possible designs satisfying a given specification at the system level is very large, it is difficult to pick a good design that has the maximum performance for a given cost. The UVA uninterpreted modeling methodology allows a designer to identify performance bottlenecks and to quickly find the effect of any design decisions on the performance of the system [1]. Thus, by eliminating the designs that give poor performance, the UVA methodology reduces the design space. Also, the UVA methodology uses simple primitive modules to model computer systems thereby eliminating the need for the designer to understand complex queuing theory or Petri Net (PN) theory to analyze the performance of the system. That is, the UVA methodology enhances the productivity of the designer by reducing the time and the cost needed to arrive at a good design.

A problem facing fault-tolerant designers is the high level of modeling expertise required by the current reliability tools to produce meaningful results. This high level of expertise has often had the effect of postponing reliability calculations until near the end of the design process, and has often required the services of a reliability expert, rather than the system designer, to perform the dependability analysis [2]. The UVA methodology, by providing facilities for automated reliability analysis of a model constructed using the primitive modules, eases the design of fault-tolerant systems to a great extent.

Besides integrating performance and dependability evaluation tools into a Computer Aided Design environment, UVA methodology enables a design engineer to capture the following information about a system in a single unified model:
1. Data-flow and control-flow through the system
2. Input-Output relationships of components and subsystems
3. Data processing rates and component delay information
4. Dependability characteristics of the components such as failure rates and coverage

By using a single model, our methodology eliminates the inconsistency between different models used to perform system level analysis and trade-offs. The primitive modules may be interconnected to model hardware, software, and the interaction between the two. The designer can independently refine elements of the design and simulate the system models in which different components are described at different levels of abstraction and interpretation [1].

The behavior of each primitive module is defined by a Colored Petri Net (CPN) [3] which provides an unambiguous mathematical specification of the module and a VHDL description which has a one-to-one correspondence with the CPN defined behavior of the module. Thus, a model built from the primitive modules can be simulated using a VHDL simulator and at the same time analyzed using CPN theory by converting the model into its corresponding CPN representation. The system model can be simplified by reducing the CPN model using a set of CPN reduction rules [4]. The reduced CPN model can be simulated in the same VHDL environment if we can convert the reduced CPN model to VHDL. By simulating all our models in the same environment, we eliminate the errors due to changes in the environment and thus, get consistent simulation results. Also, the reduced CPN model speeds up the simulation.

In this paper, we concentrate on how we use VHDL in our methodology. First, we give an overview of the UVA design environment in Section II. In Section III, we briefly describe the CPN representations for the primitive modules. In Section IV, we present the CPN to VHDL translation algorithm. In Section V, we present the CPN reduction rules which reduce the CPN model. In Section VI, we illustrate how the same performance model can be
used to study other aspects of the system like reliability. Finally, we present some experimental results in Section VII.

II UVA design environment

The UVA environment allows the user to capture the primitive element model of the system using Mentor Graphics' Design Architect schematic editor. Once the primitive element model is created using the schematic editor, the user can study the performance of the system through simulation. We can simulate the hierarchical VHDL obtained by directly substituting each primitive element by its corresponding VHDL or we can reduce the equivalent CPN and then, translate the reduced CPN to VHDL and simulate the resulting VHDL. We use CPN reduction in the CPN to VHDL path in Figure 1 to simplify the PN model in the hope of reducing the simulation time.

In order to accommodate both VHDL and Petri Nets (PN), we use an internal representation of the primitive modules called mr. A UVA module can be represented in VHDL, in Petri Nets, or as a netlist of other modules in mr. No other format lets the mixing of Petri Nets and VHDL as mr lets us to do. Once a primitive element model of a system is translated into the internal mr representation, it can be further translated into VHDL through two different paths as described before.

III Colored Petri Net models

The UVA modules are divided into three categories — control, color, and delay. The control modules are used to control the flow of data. The color modules are used to modify the state information. The delay modules help simulate the time delays in the system. The fault modules are special color modules that are used to study the fault tolerant characteristics of the system.

We use Jensen's Colored PNs (CPN) to model the modules. The CPN models are more succinct than other types of PNs like ordinary PNs and stochastic PNs [3]. They achieve their succinctness by distributing their complexity into a) net structure, b) descriptions, and c) net inscriptions.

The CPN structure is a bipartite directed graph. Figure 2 shows the CPN model of an example control module. The places (circles), the transitions (bars), and the arcs connecting them form the structure of the CPN. The CPN places can contain tokens. Unlike other PN tokens, the CPN tokens can have complex data types called colors. The data types of the tokens and the variables used in the net inscriptions are described in CPN descriptions (see the declaration of x and y variables in Figure 2 for an example).

The CPN inscriptions called arc expressions written by the side of an input (output) arc indicate what tokens to remove (add) when the transition associated with the arc fires. The inscriptions written by the side of a transition (called guard) describe the additional conditions that need to be satisfied for the transition to fire. Normally, a transition will fire if all the input places have the tokens indicated by the corresponding arc inscription.

The switch module, shown in Figure 2, sends the ready token arriving at input i1 to output o1 if a control token is present at the control input c1. The control token in place c1 is not removed as n is 0 in n'y. If n is 1, it is omitted as in x. The switch module then waits for an acknowledge token to arrive on the output o1. When the acknowledge

```
var x,y: Token;
```

![Figure 2: CPN model of the switch control module.](image)
token arrives on the output o1, the switch module acknowledges its input i1.

For a detailed description of each of the modules and the CPN model for each, the reader is referred to [5].

IV Petri Net to VHDL translation algorithm

In this section, we present an algorithm to translate a PN into VHDL. We also analyze the space and time complexity of the algorithm.

The arc expressions of the PN will evaluate to the enum constant a) minus if the arc is (place, trans) and the arc inscription is n'y and n > 0 and the place has at least n tokens, b) zero if the arc is (place, trans) and the arc inscription is 0'y and the place has at least 1 token, c) not if the arc is (place, trans) and the arc inscription is -1'y and the place has no tokens in it, d) plus if the (trans, place) arc's inscription is n'y, e) repl if the (trans, place) arc's inscription is =n'y, and f) nop otherwise.

Simulation rule: A transition is enabled if none of its input arc expressions evaluate to nop. An enabled transition fires by removing n tokens (if the arc inscription is n'y) from the places whose arc expressions evaluate to minus, adding n tokens (if the arc inscription is n'y) to the places whose arc expressions evaluate to plus, and replacing the tokens in the places whose arc expressions evaluate to repl with n tokens (if the arc inscription is =n'y).

From the simulation rule, it is clear that we consider at most one color type per place. Also, if we use the following color for the PN token, we need to use only one color type for all the places in the PN:

type pntoken is record
  num: integer;
  tk: token; -- uva primitive module token
end record;

Since, each data link and each control link in a network of primitive modules hold at most one UVA primitive module token, the PN models of the primitive modules can be built with at most one token per place. We also assume the same in the algorithm.

A place activates a VHDL signal of type pntoken whenever it gets or gives away its token. A transition also activates a VHDL signal of type pntoken whenever it fires. We make each place and each transition a separate process. Each process will update only its corresponding signal.

A place gets a new token when one of its input transitions with plus or repl arc expression fires, and it gives away its token when one of its output transitions with minus arc expression fires. So, a place process depends on all its input transitions, and all the output transitions with minus arc expression. Example: The place process for the place p in the PN of Figure 3a is given in Figure 3b. The process p depends on the input transitions a and b and

\[
\begin{align*}
\text{p: process } (a, b, e, f) & \text{ begin} \\
\text{if a'active then} & \text{ p <= (p.num + 1, a.tk);} \\
\text{elsif b'active then} & \text{ p <= (1, b.tk);} \\
\text{elsif c'active} & \text{ or f'active} \\
\text{ p.num <= p.num - 1;} & \text{ end if;} \\
\text{ end p;} \\
\end{align*}
\]

(b)

Figure 3: Place process example.

output transitions e and f. Since, output transition arcs for c and d are not minus, the process p does not depend on them. If one of the input transition signals is active, then the process p updates the pntoken signal p with the active transition signal. If any one of the output transition signals whose arc evaluates to minus is active, then the process p resets the token count of the pntoken signal p.

A transition is enabled when all its input places with minus or zero arc expressions have tokens and all input places with not arc expressions have no tokens. An enabled transition may or may not fire. For example, consider the PN shown in Figure 4a wherein if the places with minus arc expressions a, b, and d have tokens and the place with the not arc expression c does not have any token, then transitions t, u, and v will all be enabled. But, because of conflict only two of them can fire. We assign a priority to the firing of the transitions to resolve conflicts. In Figure 4a, we have assigned transition t the highest priority followed by u and v. So, the transition process u has to check whether transition signal t is active before it activates signal u. In a similar manner, process v will check whether transition signal u is active before it activates signal v. Whereas, process t does not have to check any transition process as it has the highest priority to fire.

112
for each node n in the PN
    declare a ptoken signal as follows:
    signal n::ptoken;
    if (n is a place) then
        generate_place_process (n);
    else
        generate_trans_process (n);
    end if;
end for;

Figure 5a: PN to VHDL algorithm—main function.

The main function that translates our PN to VHDL is presented in Figure 5a. It declares a signal of type ptoken for each PN node and then, if the node is a place, it generates a place process by calling generate_place_process() or if the node is a trans, it generates a transition process by calling generate_trans_process().

The generate_place_process() is presented in Figure 6b. The generated VHDL code checks whether each node in the input node list is active. If a node is active and the arc expression is plus, then it increments the token count of the place's ptoken by 1 and copies the UVA token of the input node to the place's ptoken. After that, if any of the output node with the minus arc expression is active, it decrements the token count of the place's ptoken by 1.

The generate_trans_process() function is presented in Figure 7c. It generates a VHDL process which first checks whether the transition is enabled or not. Once the transition is enabled and none of the transitions involved in the conflict that has higher firing priority than the current transition node has fired, the generated VHDL process copies the ptoken of the first input place with the minus arc expression in the input place list to the transition signal.

Since the translation algorithm generates one signal and one process for each node, it takes O(n) time. If the degree of a node (the total number of input and output nodes connected to the node) is a constant, then the algorithm takes O(n) space.

V Petri Net reduction rules

The PN reduction rules presented here are given in terms of ordinary PN. They are used to reduce the PN for the control modules part of the complete model. The PNs for the control modules do not manipulate color and hence, they can be treated as ordinary PNs.

Figure 8 illustrates the application of one of our rules.
function generate_trans_process
  (node t)
  t1 = transitions involved in conflict with t
  htl = a \in t1 and priority of a higher than t
  impl = list of input places with minus arc
        expression
  inpl = list of input places with not arc
        expression
  generate:
  t: process (htl, impl, inpl)
    if \forall a \in impl check a.num = 1
      and \forall a \in inpl check
        a.num = 0 then
      if \forall a \in htl check
        a.active = false then
        t <= head(impl);
      end if;
    end if;
  end function;

Figure 7c: PN to VHDL—transitions process
generator.

Figure 8: Application of a reduction rule.
The rule that is applied states that if transition \( t_c \) can fire
only after transitions \( t_a \) and \( t_b \) fire, and transition \( t_b \) can fire
only after transition \( t_a \) fires, then remove any places that
exclusively connect transitions \( t_a \) and \( t_c \).

In addition to our rules, we also use the rules given in
[6]. We describe all the rules in great detail in [4].

VI Reliability analysis using the primitive
modules
A subset of the primitive modules referred to as the
Fault modules are used to model several dependability
characteristics of a system including failure, fault
detection, error correction, and reconfiguration strategies.
We have a set of abstraction rules which can be effectively
used to extract the analytical reliability model from the
primitive element model of a system [8]. For example,
Figure 9 illustrates the application of the abstraction rules
to the voter of a Triple Modular Redundancy (TMR)
system. The abstraction rules simply throw away all the
information that are not needed for the reliability analysis.
Figure 10 illustrates how the abstracted TMR CPN model
is converted into traditional Markov model for reliability
analysis. The reader is referred to [8] for more information
on the reliability analysis in the UVA methodology.

VII Experimental results
PN models for a three computers system that share a
common bus and the ATAMM multiprocessor system
described in [7] are built and reduced using the application
of our rules. We have obtained significant reductions as is
evident from Table 1.

<table>
<thead>
<tr>
<th>Example</th>
<th>Nodes before reduction</th>
<th>Nodes after reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-computers</td>
<td>166</td>
<td>68</td>
</tr>
<tr>
<td>ATAMM</td>
<td>735</td>
<td>360</td>
</tr>
</tbody>
</table>

Even though we have implemented the PN to VHDL
translation algorithm, we have not merged the
implementation into our main tool illustrated in Figure 1.
So, we manually translated the PN for one of the computers
in the 3-computers example into VHDL using the
translation algorithm. The time to simulate both the
3-computers UVA model in which all the computers are
modeled using the UVA modules and the 3-computers
model in which one of the three computers is a PN model
is recorded in Table 2. As can be seen in Table 2, the

<table>
<thead>
<tr>
<th>Example</th>
<th>UVA model (sec)</th>
<th>PN model (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu time</td>
<td>17.8</td>
<td>6</td>
</tr>
<tr>
<td>real time</td>
<td>20.9</td>
<td>15.8</td>
</tr>
</tbody>
</table>

Table 2: 3-computers example simulation results.
savings are 17% in cpu time and 24% in real time (wall
clock time).
Figure 9: Reduction of the Voter module

(a) Voter Primitive Element Model

(b) Snapshots in the Reduction Process

(c) Reduced CPN Model

Figure 10: Obtaining the Markov Model

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
</tr>
</thead>
<tbody>
<tr>
<td>m0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m5</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m7</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m8</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m9</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m10</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m11</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Possible markings of the system CPN

(c) Corresponding Markov model

(d) Reduced Markov model
VIII Conclusion

In this short paper, we briefly described the VHDL based system level design environment. We used a set of predefined primitive elements, each of which has both an underlying Petri Net representation and a VHDL representation, to model a system. The PN models not only formalizes the UVA methodology but also lend themselves to transformations like PN reductions so that the complexity of the system level model is greatly reduced. We also presented an algorithm to translate the PN model to VHDL so that the PN model can be simulated. Finally, we gave a couple of examples which showed the reduction rules does indeed greatly simplify the original model.

IX References


