A Semantics-Preserving Reduction of Code-Annotated Well-formed Free Choice Petri Nets

Nick Savoiu
UC Irvine
savoiu@ics.uci.edu

Sandeep K. Shukla
Virginia Tech
shukla@vt.edu

Rajesh K. Gupta
UC San Diego
gupta@cs.ucsd.edu

Center for Embedded Computer Systems
University of California, Irvine, CA, USA
http://www.cecs.uci.edu/

Technical Report #04-07
Dept. of Information and Computer Science
University of California, Irvine, CA 92697, USA
Feb 12, 2004

ABSTRACT

Software models of hardware designs are commonly used for simulation during system design and verification. They provide excellent control over and visibility into the designs being simulated but complex design simulations are very time consuming. Improving the run-time performance of such models would have a direct impact on critical design cycle metrics such as time-to-market. In our previous work we have shown that we can exploit the implicit concurrency in such system models to significantly improve their simulation run-time. The key to that was a concurrency reassignment technique that reduced the overhead of emulating hardware parallelism in software. In the early stages of experimentation we have simulated both the original and restructured models and compared the results to verify the correctness of our transformations. However, a stronger proof is always desirable and, therefore, in this paper we formally prove that our Petri net-based reduction approach to software model restructuring preserves the SystemC simulation semantics.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Petri nets are a well known model of concurrent systems routinely used for both theoretical and practical applications to due to their analyzability and expressive power. Concurrent systems can often be characterized by the sequences of actions that they allow to occur and Petri nets are a natural fit given that they can model the occurrence of actions by the firing of transitions. The set of allowable transition sequences characterizing the Petri net can be used to characterize the system that it models.

The target of analyzing a Petri net is often that of determining the modeled system's properties such that another Petri net that is deemed "better" by a certain metric (e.g. smaller, faster, etc.) can be obtained. In our previous work[15] we have improved the simulation performance of SystemC[16] models of hardware designs using Petri nets as an underlying model. To do this we exploited the implicit concurrency in the model to change the threading mechanism employed in the simulation engine from user-level to kernel-level. Our metric for improvement was the amount of code that could be executed without explicit synchronization between threads. A larger size code improved simulation performance through a combination of reduced simulation bookkeeping overhead and more efficient use of multiprocessing.

However, such a new, "better" Petri net is only useful if one can show that it is equivalent to the original Petri net. The equivalence of two models can generally be tested in one of two ways. One method takes an empirical approach whereby the *before* and *after* models are simulated for a given set of tests and, if the simulation results agree, the systems are then deemed equivalent. The advantage of such a technique is its relative simplicity and we have used it in the early stages of experimentation in our work. However, a critical shortcoming of this method is that it fails to prove equivalence beyond a doubt. A more rigorous, and generally more difficult, approach is to formally prove the equivalence of such models. Recall that the behavior of a concurrent system modeled by a Petri net can be characterized by the set of transition sequences that the Petri net allows. By labeling each transition in the net with symbols from an alphabet these sequences can be regarded as words from a language over that alphabet. This leads to the possibility of formally proving the equivalence of two Petri nets through language equivalence if we can show that the transformations producing the "better" Petri net are language-preserving.

This can be difficult to show for general Petri nets due to the intractability of most of their properties. However, since the Petri nets we consider are not general in structure (i.e. we extracted them from structured SystemC code) we can prove this for a smaller class of Petri nets and still obtain results with practical applications. Thus, we have concentrated our attention on well-formed free choice Petri nets (FCPN) due to the extensive body of theoretical work which has yielded efficient algorithms for determining most of their properties -- a critical aspect for a practical application. We transform the Petri nets using a well known set of reduction rules for FCPNs[4] that we have augmented to also manipulate the transition labels so as to extract code cycles through the original system that can then be used to speed up simulation. Nonetheless, such cycles must preserve the initial simulation semantics to be useful. Therefore, in this paper we show that augmented reduction rules set is language-preserving and that this translates into the preserving the simulation semantics of the original SystemC model.

The rest of the paper is organized as follows. Section 2 introduces some general Petri net and Petri net language concepts. We continue with section 3 by presenting the set of Petri net reductions for well-formed FCPN followed by briefly introducing our approach to Petri net modeling for SystemC design descriptions and showing a simple reduction of such a description in section 4. We then devote section 5

to proving that these transformations are language-preserving. We conclude in section 6 with some observations regarding the applicability of such reduction sets and future directions for our work.

2. PETRI NETS AND NET LANGUAGES

We make only a brief introduction to Petri net concepts, as required to present our work, while referring the reader to [3], [10] or [14] for more in depth coverage.

Let S be a set of *places* represented as circles and denoted as s_i. Let T be a set of *transitions* represented as bars and denoted as t_k. Let F be a set of arcs between either places and transitions or transitions and places.

Definition 2-1. A directed, bipartite graph $N=(S,T,F)$ is called a *net*.

We assume, without loss of generality, that S, T and F are nonempty and N is connected in order to eliminate trivial nets.

A *marking* is an assignment of nonnegative integer numbers of tokens to each place in a net. We define $M(s)$ as the number of tokens that marking M attributes to place s of net N.

Definition 2-2. A *Petri net* is defined as the tuple (N,M_0) composed of a net N and a marking M_0 also known as the *initial marking*.

Unlike a net, a Petri net is a dynamic entity driven by the flow of tokens generated as enabled transitions are fired. A transition is *enabled* if each of its input places is marked with at least one token. Firing a transition removes one token from each of its input places and places one token in each of its output places. For example, in Figure 1(a), transition t_1 is enabled and its firing will remove a token from places s_1 and s_2 and add one token to place s_3.

Definition 2-3. Let $N=(S,T,F)$ be a Petri net. Then the net $N^d=(T,S,F^{-1})$ is the *dual net* of N.

A Petri net (N,M_0) is *bounded* if $\exists k \in \boldsymbol{N}$ such that $\forall s \in S$ then $M(s) \leq k$ for every reachable marking M. A Petri net (N,M_0) is *live* if $\forall M$ a reachable marking and $\forall t \in T$ then $\exists M'$ a reachable marking such that t is enabled by M'. A net N is *well-formed* if the Petri net (N,M_0) is live and bounded.

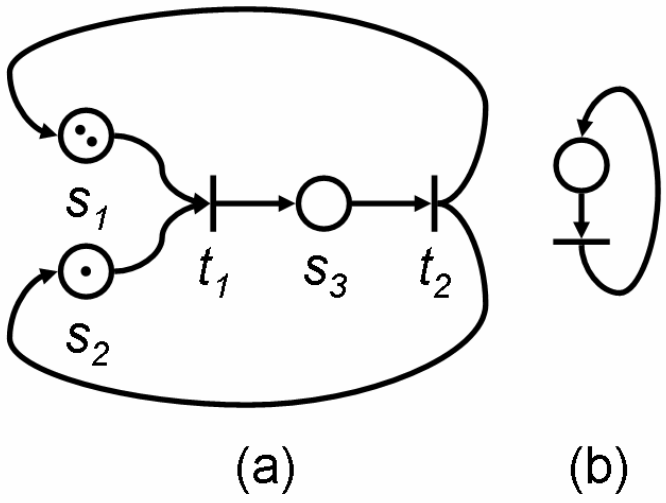

Figure 1. Example Petri Nets

Definition 2-4. A net is called *atomic* if it is isomorphic to the net $N=(S,T,F)$ where $S=\{s\}$, $T=\{t\}$, and $F=\{(s,t), (t,s)\}$.

An atomic net, such as the one in Figure 1(b), has the minimum number of places and transitions required for it to still be live and bounded.

We define an *alphabet* as a finite set of symbols while a *word* is defined as a finite length sequence of symbols from an alphabet. The word containing no symbols, λ, is known as the *null* or *empty* word. A *language* is defined as a potentially infinite set of words over an alphabet.

Languages are commonly generated by means of a grammar or automaton. Defining a language by means of a *grammar* (i.e. a set of productions) implies that any word in the language can be generated by successive applications of the productions. Similarly, specifying a language by means of an *automaton* implies that any word in the language is generated by some execution of the automaton.

Definition 2-5. A *Petri net language* is the language generated by a Petri net.

Petri net languages have received significant attention especially in the area of discrete event systems[1][7][13] while good descriptions of various classes of Petri net languages can be found in [8] and [11]. In [11], Peterson presents Petri net languages in a fashion similar to that of other areas of formal language theory[1]. In particular, similarly to other automaton-generated languages, different classes of Petri net languages have been defined depending on the choices made from various *initial state*, *labeling function* and *terminal state* schemes.

We will briefly discuss these concepts to better understand the effect these choices have on the languages generated by Petri nets that are thus restricted.

2.1 Initial State

The initial state of a Petri net determines how tokens are assigned to its places before its execution begins. There are several ways of representing the initial state of a Petri net.

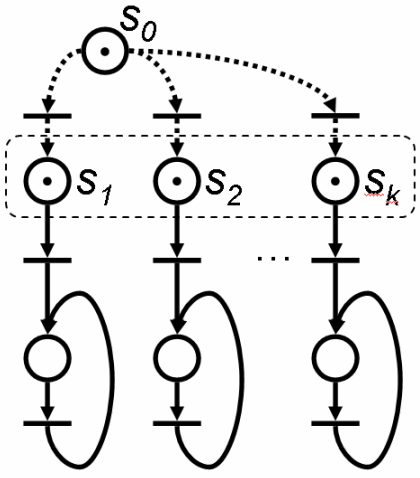

Figure 2. Petri Net initial state

One such way is to allow any arbitrary marking μ to be the initial state. A similar way, particularly useful when several Petri nets are combined in order to be executed together, is to require that the initial state be restricted to a marking with only one token in a place designated as the *start place*.

This can be easily implemented in the context of our concurrent processes system by simply introducing a global start place s_0 (initially marked with one token) as shown in . We then add transitions that connect s_0 with the start places s_1 to s_k for each of the concurrent processes. This ensures that, after the first step in the execution of the Petri net, all the process start places will be marked by a token.

2.2 Labeling Function

In order for a Petri net to generate words as a result of its execution we must associate a symbol from an alphabet with each transition in the Petri net. Let $N=(S,T,F)$ be such a net.

Definition 2-6. A *labeling function* $f : T \rightarrow \sum$ for the net N if a function that associates an element of $\sum$ with each transition in N.

Just as for the initial state, there are several variations on the labeling function. For example, a Petri net is called *free-labeled* if all its transitions are distinctly labeled (i.e. $f(t_1)=f(t_2)$ implies that $t_1=t_2$). Alternatively one could relax the requirement that labels be distinct and/or even allow λ-labeled transitions. These λ transitions can be thought to be of an "internal" nature and therefore do not affect the words generated by the Petri net. For our purpose, however, we will chose the free-labeled case as it best fits our associating code fragments with each Petri net transition.

2.3 Terminal State

In [11] four common choices for selecting the final marking(s) are presented. Since the terminal state(s) of a Petri net determine which executions result in valid words in the Petri net language choosing different sets of terminal states may lead to different languages being generated.

Terminal states are closely related to the notion of *next state function*.

Definition 2-7**.** A function $\delta: M\times T \rightarrow M$ is called a *next state function*.

A next state function defines terminal states for the Petri net by identifying markings that allow no further advancement of the execution of a Petri net given a certain marking.

Given a net $N=(S,T,F)$, a labeling function $f : T \rightarrow \sum$, a next state function $\delta: M\times T \rightarrow M$, and an initial state with marking μ we can define the four major classes of Petri net languages as follows.

Definition 2-8**.** A Petri net language $L(N)$ is an *L-type* language if it has a finite set of final markings M_f such that $L(N)=\{f(t)\in \sum^* | t\in T^* \wedge \delta(\mu,t) \in M_f\}$.

However defining a finite set of markings as the set of final states can be deemed too restrictive since, if $\delta(\mu,t_i)$ is defined then $\delta(\mu',t_i)$ is defined for any $\mu' \geq \mu$. This leads to the following class of Petri net languages.

Definition 2-9**.** A Petri net language $L(N)$ is a *G-type* language if it has a finite set of final markings M_f such that $L(N)=\{f(t)\in \sum^* | t\in T^* \wedge \exists\ \mu_f\in M_f \text{ s.t. } \delta(\mu,t) \geq \mu_f\}$.

Another way of designating a terminal state is to require that the next step function, from a particular state, be undefined for any alphabet symbol. This leads to the following class of Petri net languages.

Definition 2-10**.** A Petri net language $L(N)$ is a *T-type* language if it has a finite set of final markings M_f such that $L(N)=\{f(t)\in \sum^* | t\in T^* \wedge \delta(\mu,t)$ is defined but $\forall t_j\in T,\ \delta(\delta(\mu,t),tj)$ is undefined$\}$.

Finally, and of particular interest to us, is the class of languages that have as their final states all the reachable states of their underlying Petri net. Note that if $w\in \sum^*$ is a word in such a language L then

$\forall w' \in \sum^*$ such that $w=w'x$ and $x \in \sum^*$ then w' is also a word in the language L (i.e. L is a *prefix closed* language).

Definition 2-11**.** A Petri net language $L(N)$ is a *P-type* language if it has a finite set of final markings M_f such that $L(N)=\{f(t) \in \sum^* | t \in T^* \wedge \delta(\mu,t)$ is defined$\}$.

Given that we use Petri nets to model the simulation of concurrent processes such a model should to be in a valid state at any time during its execution rather than at certain *terminal states*. Thus for the remainder of the paper we assume that when we refer to a *Petri net language* we actually mean a *P-type Petri net language*.

3. REDUCTION RULES

Sets of reductions rules for Petri nets have been presented, among others, in [4], [5], and [9]. We will, however, use the set presented in [4] as it is applicable to a wider class of FCPNs thus extending the reach of our technique. The set has two properties that made it suitable for our application. First of all, it is a *complete* set of reductions.

Definition 3-1. A set of reductions is *complete* if it can reduce any well-formed net to an atomic net.

Second, it has the property that applying any of its reductions rules to a well-formed FCPN results in an FCPN that is still well-formed.

Furthermore, consider the following lemma from [3]:

Lemma 3-1. For every well-formed FCPN N there exists a reachable marking M and an occurrence sequence σ such that any t in T occurs at least once in σ and σ returns N to marking M.

The above properties and lemma provided the insight into the existence of a cycle through a well-formed FCPN that will exercise all its transitions while returning it to the initial state and to the possibility of extracting such a cycle using such a complete set of reductions.

We have adapted the rules to our particular application (i.e. language preservation) by augmenting each transformation with transition label adjustments.

The set is composed of reductions ϕ_A, ϕ_S, and ϕ_T. The first reduction deals with the removal of a place s and a transition t that are uniquely connected.

Reduction ϕ_A Given a free choice net $N=(S,T,F)$ where

$$\begin{aligned} &\exists s \in S \quad \bullet s \neq \varnothing \wedge s\bullet = \{t\} \\ &\exists t \in T \quad t\bullet \neq \varnothing \wedge \bullet t = \{s\} \\ &\qquad\quad (\bullet s \times t\bullet) \bigcap F = \varnothing \end{aligned}$$

there exists a free choice net $N'=(S',T',F')$ such that

$$\begin{aligned} S' &= S \setminus \{s\} \\ T' &= T \setminus \{t\} \\ F' &= (F \bigcap ((S' \times T') U (T' \times S'))) \bigcup (\ \bullet s \times t \bullet\ ) \end{aligned}$$

A graphical representation of this reduction rule is presented in Figure 3. We augmented the original rule based on the observation that once place s is marked transition t is bound to occur.

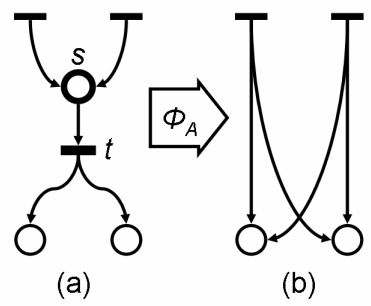

Figure 3. Applying the Φ_A reduction

Thus, when applying ϕ_A, the labels are transformed as follows given that transition t is being removed from the net

$$\forall t' \in \bullet s \quad \Lambda(t') = \Lambda(t') + \Lambda(t)$$

where $\Lambda(t)$ represents the transition label associated with transition t and '+' represents the label concatenation operator.

The second rule, ϕ_S, deals with the removal of a "redundant" place and is stated as follows:

Reduction ϕ_S Given a free choice net $N=(S,T,F)$ where

$$|S| > 2$$
$$\exists s \in S \quad \textit{nonnegative linearly dependent place}$$
$$\bullet s \cup s \bullet \neq \emptyset$$

there exists a free choice net $N'=(S',T',F')$ such that

$$S' = S \setminus \{s\}$$
$$T' = T$$
$$F' = F \cap ((S' \times T') \cup (T' \times S'))$$

A graphical representation of this rule is presented in Figure 4. This reduction required no augmentation since only a place is removed. As a place has no label associated with it, its removal does not affect any of the transitions labels.

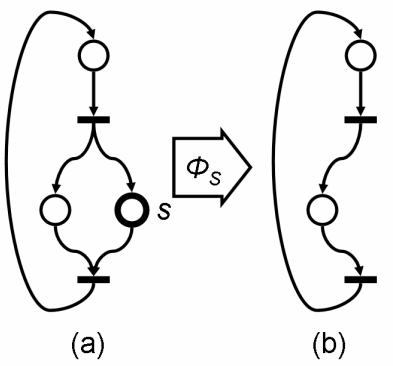

Figure 4. Applying the Φ_S reduction

Finally the last reduction rule, ϕ_T, has to do with the removal of a "redundant" transition.

Reduction ϕ_T Given a free choice net $N=(S,T,F)$ where

$$|T|>2$$
$$\exists t \in T \quad \textit{nonnegative linearly dependent transition}$$
$$\bullet t \bigcup t\bullet \neq \emptyset$$

there exists a free choice net $N'=(S',T',F')$ such that

$$S' = S$$
$$T' = T \setminus \{t\}$$
$$F' = F \bigcap ((S' \times T') \bigcup (T' \times S'))$$

Intuitively, the transition being removed does not contribute any tokens to the net that are not already contributed by other transitions (due to the linear dependence requirement).

This reduction, as depicted in Figure 5, removes a transition and we must make changes to other transition labels if the language being generated by the Petri net is to be preserved.

As we will see later on, conflict nodes (such as the input place of t in Figure 5) will have conditions associated with them. We use these conditions to generate new labels for the remaining transitions due with the removal of t. Thus, $\forall s_1 \in \bullet t$ and $\forall s_2 \in t\bullet$ then

$$\Lambda(P(s_1,s_2))=c(\Lambda(P(s_1,s_2))|\Lambda(t)).$$

In other words, the remaining paths between $\bullet t$ and $t\bullet$ (i.e. $P(\bullet t, t\bullet)$) will have as the associated code the guarded code from both it and transition t. The guards c is created based on the conditions associated with the places in $\bullet t$.

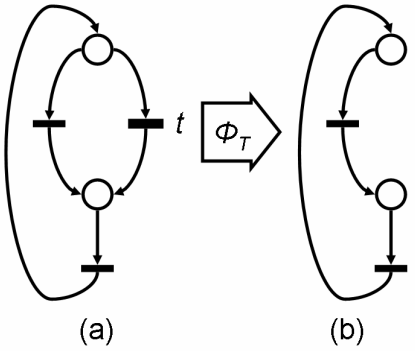

Figure 5. Applying the Φ_T reduction

However, for these transformations to be useful from a practical point of view we must show that they are language-preserving. Let us start by presenting the modeling of a SystemC design as a Petri net and its reduction using the above rules.

4. PETRI NET MODELING

We will only briefly describe how Petri nets are obtained from a process description referring the reader to [15] for more details. Let us consider the system consisting of three concurrent processes as presented in .

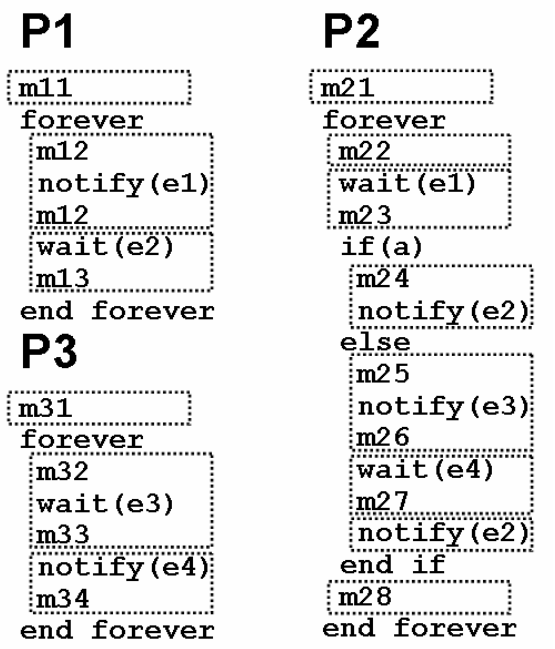

Figure 6. Example concurrent processes system

We start by creating a hierarchical task graph(HTG)[6] for each process in the system. The hierarchical nature of HTGs allows us to consider or ignore the inner workings of complex code segments depending on whether or not event they do any event processing (i.e. generation or consumption). For example, a loop node will be black-boxed and treated as a basic block if it contains computation code but no event processing. On the other hand, the *if* statement in process P2 from cannot be ignored due to the presence of event processing statements (e.g. *wait* and *notify* for SystemC) as thus its control flow must be considered when generating a Petri net from it.

We assume that the processes present in a SystemC description are composed of an initialization section

followed by an infinite loop containing the process functionality. This is generally the case for hardware designs which are our intended target. To further refine the system analysis, we divide each process HTG graphs into atomic process units (APUs). This identifies the HTG sections that correspond to the code executed each time a process is invoked by the SystemC scheduler. We do this by computing a transitive closure for each synchronization point (i.e. the *wait* statements in). Encountering any other synchronization point will block transitive propagation. The outlined code in shows the sections that make up the APUs for each of processes in our example. Additionally, we also determine the sets of events that each APU is sensitive to (i.e. input events) and that it generates (i.e. output events).

Next, the APU graphs are mapped to Petri nets through a constructive process based on the "basic" building blocks presented in (a)-(c). First each APU is replaced by the building block in (a). Transition t_1 serves both as an entry point for control flow as well as a synchronization point. The latter is accomplished by introducing a new input place for transition t_1 that will be connected to the net fragment(s) that produce the required event(s). The new input place ensures that t_1 will not be enabled until both the control flow state and the event state are marked. Any conflicts (i.e. *if* statements in the APU) are resolved to the transitions t_2 through t_n. As a last step in mapping an APU to the net fragment in (a) each transition is "labeled" with the corresponding HTG graph sections outlined in .

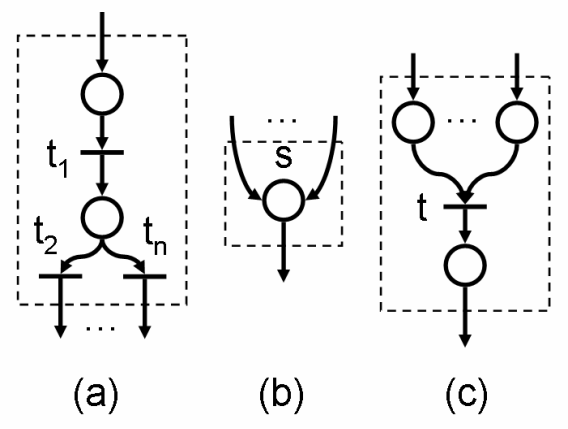

Figure 7. Petri net building blocks

The above Petri net fragments are then connected into Petri nets that account for the control and event flow in the system. For that we use the control flow information from the HTGs together with the previously computed lists of input and output events for each APU. SystemC allows for two types of synchronization based on input events: an *or-type* (i.e. *wait*(*event*$_1$ or ... or *event*$_n$)) and an *and-type* (i.e. *wait*(*event*$_1$ and … and *event*$_n$)).

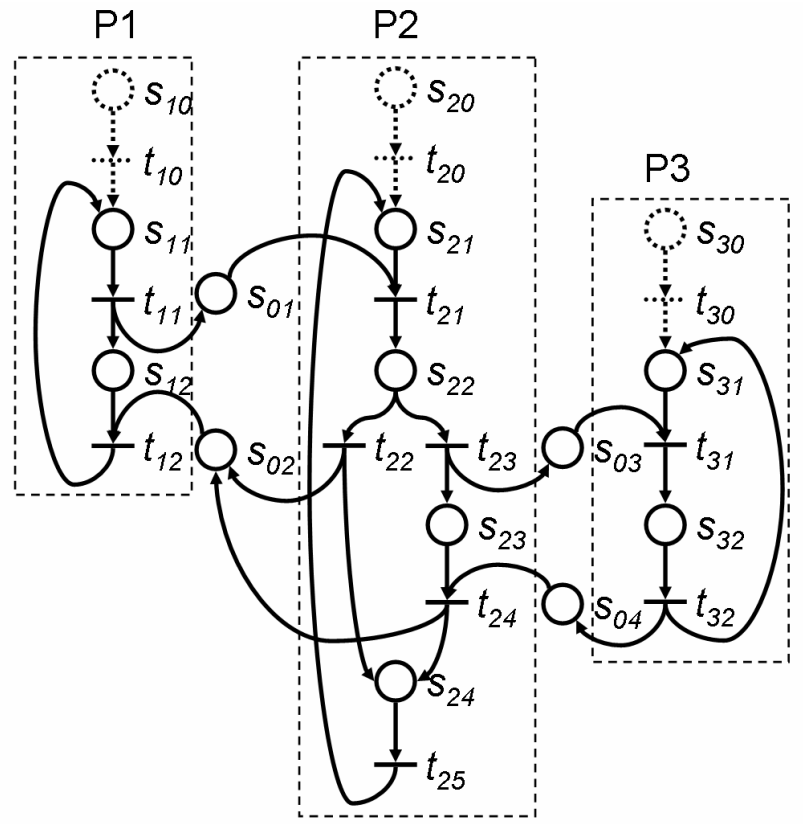

Figure 8. Initial Petri net

For *or-type* sensitivity the APU is activated whenever at least one of the input events occurs and this can be modeled by the Petri net fragment in (b). As soon as any of the input transitions of place *s* fires it

places a token on place *s* thus enabling the output transitions of *s*. In the case of *and-type* sensitivity all the input events must occur before the APU is activated. Transition *t* in (c) requires that all its input places be marked before it can fire and mark its output places. The output events of an APU are determined by identifying any calls to event producing statements (e.g. *notify*() for SystemC) in the APU code. For example, P3 in has such a statement.

Figure 8 presents a slightly simplified (for clarity and brevity) overall Petri net obtained as a result of converting the processes in .

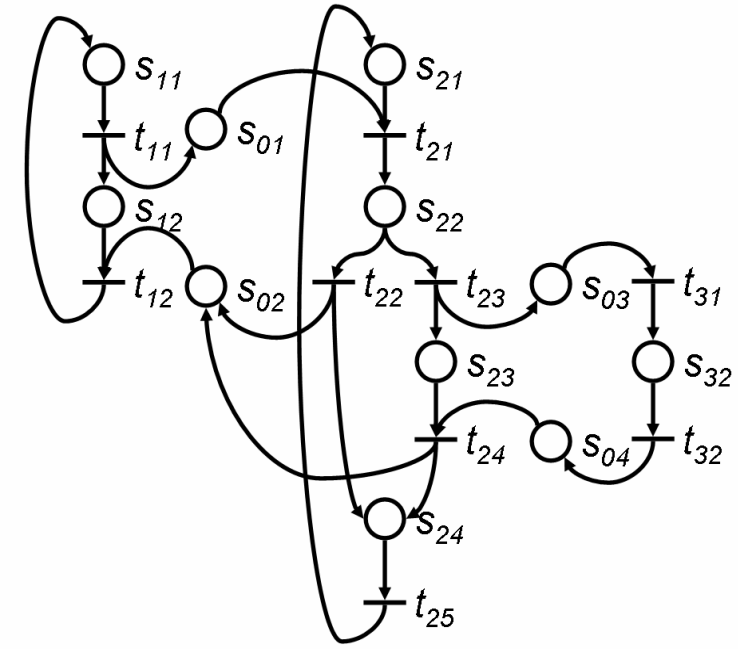

Figure 9. Petri net after removing s_{i0}, t_{j0} and applying ϕ_S

Once the net has been generated we must verify that it is a well-formed FCPN before we can apply the reduction rules. Since our conversion algorithm generates FCPNs by construction for SystemC processes composed of structured code, we need only check that the resulting net is well-formed. This can be efficiently done using theoretical results (that originated in [1]) which allow us to check for the well-formedness of FCPNs in polynomial time. Since well-formedness implies strong connectedness we first remove all the initialization sections which would prevent the nets from being strongly connected.

After the well-formedness was determined we can start applying the reduction rules. For the remaining net (i.e. the net in Figure 8 with dotted places and transitions removed) we can determine that $s_{31}=s_{11}+s_0+s_{02}+s_{22}+s_{03}+s_{04}$. This means that s_{31} is a nonnegative linear combination and we can thus apply the ϕ_S reduction to remove s_{31} resulting in the Petri net in Figure 9. Next, since (s_{03},t_{31}) and (s_{32},t_{32}) are uniquely connected, we can apply ϕ_A to remove both pairs. shows the intermediate state of the net after the (s_{03},t_{31}) pair was removed. The label from each removed transition is appended to that of the input transitions of its respective input place. These reductions now cause s_{04} to be a nonnegative linear combination of other places in the net (i.e. $s_{04=}s_{23}$). Therefore it can be removed with no changes to the labels required in this case. Next, (s_{23},t_{24}) become uniquely connected and can be removed by applying ϕ_A once more. Finally, we encounter the first occasion to apply ϕ_T as now transition t_{22} is a nonnegative linear combination of other transitions in the net (i.e. $t_{22}=t_{23}+t_{24}$) and this results in the net in Figure 11.

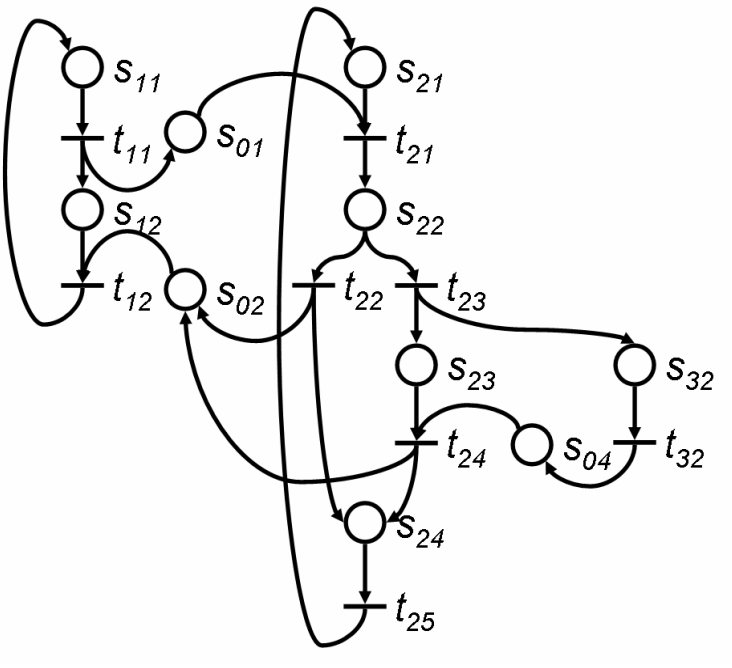

Figure 10. Petri net after applying ϕ_A

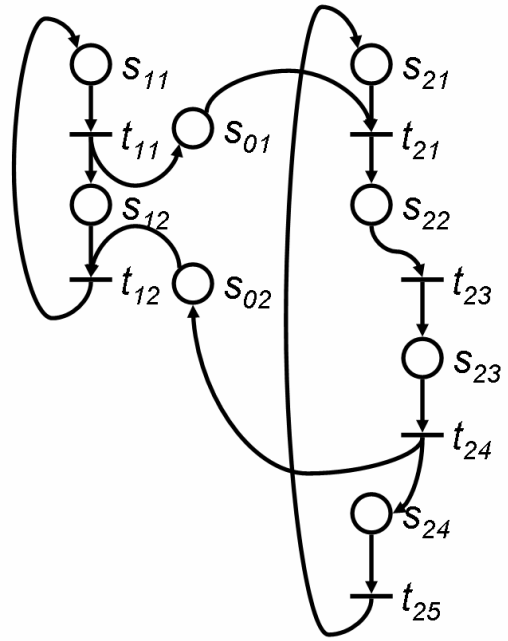

Figure 11. Petri net after applying ϕ_T

In this case removing t_{22} requires that its code label be added as previously explained to the code label for transition t_{23}. For brevity we will skip over some steps in the reduction and only cover the last few steps to be applied before reaching the atomic net. As can be seen from Figure 12, applying the sequence

of ϕ_S to remove s_{21}, ϕ_A to remove s_{01} and t_{21}, ϕ_S to remove s_{02}, and, finally, ϕ_A to remove s_{12} and t_{12} will result in an atomic net like the one in Figure 1(b) composed of s_{11} and t_{11}.

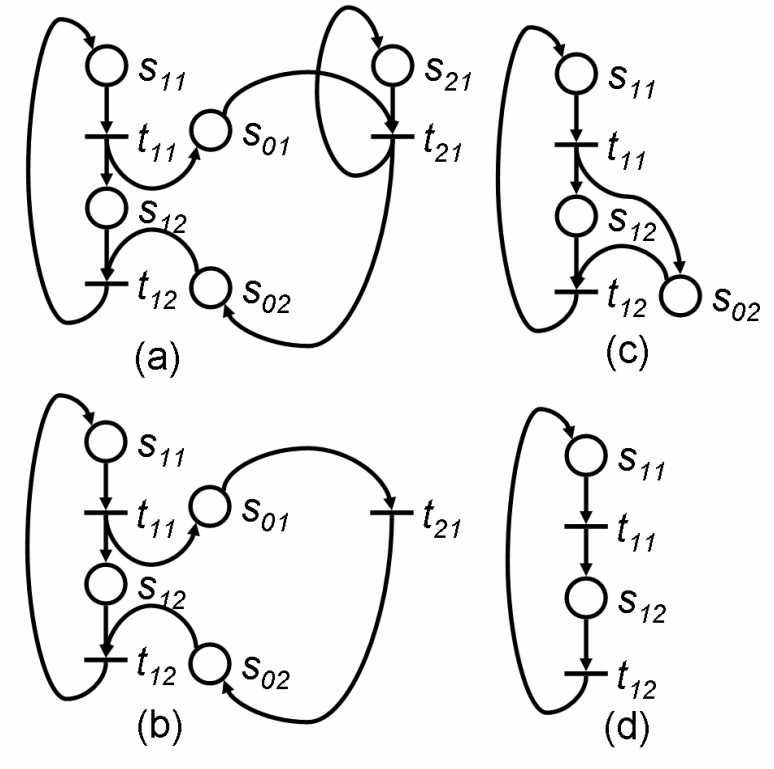

Figure 12. Final Petri net reduction steps

Most notably, the code label associated with this last remaining transition (i.e. t_{11} in our case) will have accumulated a cyclic code path through the initial system of concurrent processes. Thus, starting from the processes in , we arrive to a code label for t_{11} as presented in Figure 13.

```
forever
 m11
 notify(e1)
 m12
 wait(e1)
 m22
 if (a)
  m23
  notify(e2)
 else
  m24
  notify(e3)
  m25
  wait(e3)
  m32
  notify(e4)
  m33
  wait(e4)
  notify(e2)
  m26
 end if
 m27
 wait(e2)
 m13
end forever
```

Figure 13. Code label for t_{11}

5. SEMANTICS RETENTION

In this section we will address each of the transformations and show that the language generated by the reduced Petri net will be equal to the language of the original Petri thus proving the language-preserving properties of the transformations.

Our equivalence proofs are based on a two step approach. First we show that the reduction does not change the markings in $S \cap S'$ (i.e. the common places between the original and the reduced Petri net). This implies that the two nets will also agree on enabled transitions in $T \cap T'$ for a given occurrence sequence σ. The second step will account for the removed transitions (i.e. those in $T \setminus T'$) by showing how our augmentations guarantee that the words generated by N' will be the same as the words generated by N.

Let $N=(S,T,F)$ and $N'=(S',T',F')$ be free-labeled Petri nets. For the remainder of this section N will be the net before a reduction is applied while N' will be the net after the reduction is applied. We also assume that N satisfies all the requirements for the application of the particular reduction.

Let us start with the Φ_A reduction.

Theorem 5-1. The augmented reduction Φ_A is language preserving (i.e. $L(N)=L(N')$).

Proof. Let s and t in Figure 14 be the place and transition, respectively, involved in the Φ_A reduction. We start by proving that given a starting marking M_0 and a valid occurrence sequence σ, N and N' will coincide in place markings for the place set $S \cap S'$.

Case 1. s is marked by σ in N

Since s is marked by σ we can infer the following two statements. First, in the immediately previous step $\exists t' \in \bullet s$ such that t' was enabled. Second, t is enabled since $s\bullet=\{t\}$ and $\bullet t=\{s\}$ and therefore t will fire in the immediately following step leading to $\forall s' \in t\bullet$ to be marked.

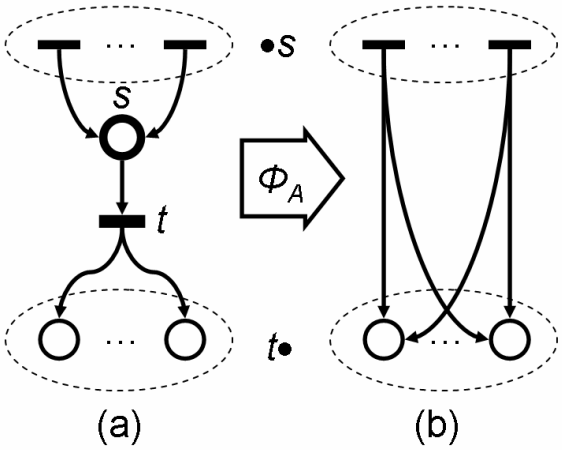

Figure 14. Φ_A Proof

Now, let us consider that Φ_A is applied (i.e. s and t are removed). Then each $t' \in \bullet s$ is connected to each $s' \in t\bullet$ (according to the $\bullet s \times t\bullet$ term in F'). However, since $\bullet s \in T'$ and Φ_A does not remove any transitions from $\bullet s$ it means that there still exists $t' \in \bullet s$ that is enabled by σ and therefore, due to the connections in $\bullet s \times t\bullet$, then all $s' \in t\bullet$ are still being marked by σ in S' just as they were previously marked by t.

Case 2b. s is not marked by σ in N

The proof is similar to Case 2a. Since s is not marked $\forall t' \in {}^\bullet s$, t' was not enabled in the immediately previous step. Also since s is not marked and s is only connected to t ($s^\bullet = \{t\}$ and ${}^\bullet t = \{s\}$) we can infer that t is not enabled. Therefore, in the immediately following step, t will not fire and will not contribute any tokens to any state in $t^\bullet$ although transitions other than t might place tokens in those places.

Let us again assume that Φ_A is applied and s and t are removed and that each $t' \in {}^\bullet s$ is connected to each $s' \in t^\bullet$. We have shown before that no transitions are added to nor removed from ${}^\bullet s$ by Φ_A and therefore none of the transitions in ${}^\bullet s$ will contribute any tokens to places in $t^\bullet$ although other transitions (not in ${}^\bullet s$) may do so.

Thus we have shown that even after removing s and t from N, N and N' still reach the same markings in $S \cap S'$ given a starting marking M_0 and a sequence σ. This means that the sequences of transitions fired will also be identical (in $T \cap T'$). To show that $L(N)=L(N')$ we have to show however that the words generated by σ are the same in N and N'. Recall that our augmentation to Φ_A called for the code label for t to be appended to the code labels of all the transitions in ${}^\bullet s$. This means that, if s is marked by σ, then t would fire in N. However, only one of the transitions in ${}^\bullet s$ will be enabled (the Petri nets are bounded and derived from SystemC code) and therefore only one instance of the code label of t will be generated by the labeling function in N' just as in N. Alternatively, if s is not marked by σ then t does not fire in N and we have shown that none of the transitions in ${}^\bullet s$ would be enabled in N'. Therefore the code label for t would also not be generated by σ in N' either.

This concludes the proof that the augmented reduction Φ_A is language preserving. Next we address the Φ_S reduction.

Theorem 5-2. The augmented reduction Φ_S is language-preserving.

Proof. Consider an occurrence sequence σ in N such that given a starting marking M_1, M_2 is reachable from M_1 via σ. First we will show that the removal of s by Φ_S does not change the markings reached in $S \cap S'$.

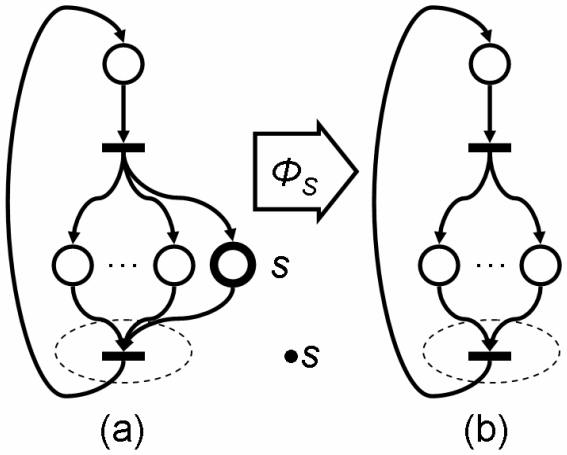

Figure 15. Φ_S Proof

Case 1. $\forall t \in {}^\bullet s \;\; t \notin \sigma$

Since no t in $\bullet s$ occurs in the sequence σ then s is not enabled by σ and therefore its removal does not affect the markings reached by σ.

Case 2. $\exists t \in \bullet s$ such that $t \in \sigma$

Let t' be a transition such that $t' \in s\bullet$. Since s is marked we have two distinct possibilities: t' is enabled or t' is not enabled by s. Let us show that removing s does not change whether t' is enabled or not. We will consider the enabled case first.

Case 2a. t' is enabled in N

Since t' is enabled we can deduce that $\forall s' \in \bullet t'$ then s' is enabled (s is such a place). Therefore $\forall s'' \in \{\bullet t' \mid s'' \neq s\}$ then s'' is marked. This means that if we were to remove s from N we would still have all the places in $\bullet t' \setminus \{s\}$ being marked and therefore t' will still be enabled in N'.

Case 2b. t' is not enabled in N

Since t' is not enabled this implies that there exists s such that $s' \in \bullet t'$ and s' is not marked. However, since s is marked in N, this means that $s \neq s'$. Therefore if we remove s from N s' will still be in $\bullet t' \setminus \{s\}$ and thus t' will still not be enabled in N' since not all its input places are marked.

We have shown that removing s from N as a result of applying Φ_S does not change the markings that are reachable via occurrence sequence σ in N'.

Showing that $L(N)=L(N')$ is simplified by the fact that no transitions are removed as a result of applying Φ_S. Since the markings are not affected either this means that the two nets will also agree on the sequences of transitions.

This concludes the proof that reduction Φ_S is language-preserving. Finally, we address the Φ_T reduction.

Theorem 5-3. The augmented reduction Φ_T is language-preserving.

Proof. The first part of the proof, that N and N' will reach the same markings in $S \cap S'$, can be easily shown using a dual proof. To obtain the dual net (see Definition 2-3) one has only to replace the places in N with transitions and vice versa. This also means that $(N^d)^d = N$. We can thus restate a problem for N in terms of N^d, solve it then revert back to N thus showing that the proof holds also for N. We can apply this mechanism to show that N and N' will agree on markings for all the common places for Φ_T. Restating the problem for Φ_T using the dual net converts it into a problem for Φ_S which we have already proved in **Theorem 5-2**.

We now only have left to show that $L(N)=L(N')$ after the augmented Φ_T transformation is applied. Let us first start by showing that even after t is removed there exists a path from s_1 to s_2 for every $s_1 \in \bullet t$ and for every $s_2 \in t\bullet$. To do that we will use the following lemma from [3]:

Lemma 5-1. Reduction Φ_T can be applied to a non-atomic well-formed FCPN $N=(S,T,F)$ if and only $\exists t \in T$ such that N'$=(S,T\setminus\{t\},F)$ is strongly connected.

This means that in our case the underlying net for N' is strongly connected and thus there exists a path from s_1 to s_2 in N'.

Since after removing t any state s in $\bullet t$ is still connected to the net this means that before removing t $\forall s \in \bullet t \; |s\bullet| > 1$ and thus all s in $\bullet t$ are actually conflict (choice) states. However, since the Petri net was generated from HTG graphs (i.e. we have generated balanced *if* statements) then, initially, for any state s

$|\bullet s| \leq 2$ and $|s\bullet| \leq 2$. But none of the transformations increase the number of output transitions of a place thus $|s\bullet| \leq 2$ holds at any time. Since in the case above we have the additional restriction that $|s\bullet| > 1$ we can deduce that $|s\bullet| = 2$ before t is removed and $|s\bullet| = 1$ after and, therefore, $k=1$. This simplifies our task of identifying a transition that is a likely candidate to receive the code label from the removed transition t.

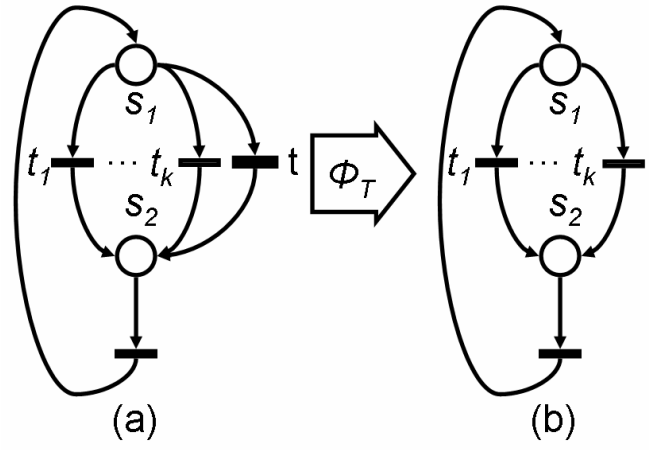

Figure 16. Φ_T Proof

Recall that, as we generated the Petri nets from APU graphs, we associated conditions with each conflict place in the Petri net. Since t will be removed we must ensure that its code label is still executed when the proper conditions are met. Our augmentation is reminiscent of *if-conversion* from compiler technology: the branches of an *if* statement are transformed into straight line code by guarding the statements with the proper conditions. Assume that if the condition c associated with s was true then execution would have been passed to t_1 and then to the places in $t\bullet$. When t is removed we must "graft" the path $s_1 \rightarrow t \rightarrow s_2$ onto the remaining path as follows. First we change the code label for t_1 from $\Lambda(t_1)$ to $c(\Lambda(t_1)|\Lambda(t))$ meaning that if c is true execute $\Lambda(t_1)$ otherwise $\Lambda(t)$. Next we update the remaining code labels for all paths leading to s_2 such that they all check condition c and, if false, we ignore their code labels (i.e. $c(\Lambda(t_1)|\lambda)$). This ensures that, if c is false, only the code label for t is executed and control is then directed to s_2 without any other code labels being generated.

The above code label manipulations ensure that N' will generate the same code labels when c is false as N. Therefore $L(N')=L(N)$ and this concludes the proof that reduction Φ_T is language-preserving.

We have shown that each transformation is language-preserving leading to the entire set being language-preserving and thus, from a simulation point of view, semantics-preserving. Consider this together with the guarantee that the set of reductions $\{\Phi_A, \Phi_S, \Phi_T\}$ will reduce any well-formed FCPN to an atomic Petri net (i.e. one place and on transition) and we obtain a powerful mechanism of obtaining a complete cycle through the code labels encapsulated in the Petri net.

6. CONCLUSION

In this paper we have presented a proof that a set of reduction rules are semantics-preserving when applied to well-formed FCPN derived from a SystemC software model of a hardware design. The reduction rules were augmented to manipulate the code labels associated with the Petri net transitions. The end result of applying the reduction rules was that of obtaining a code label (for the last transition in the Petri net) that constitutes a complete cycle through the SystemC code used to generate it. These cycles were used in our previous work to improve the simulation performance of SystemC models.

Future directions of our work are aimed at better determining what Petri net classes can be generated from SystemC descriptions given our modeling method. Also, despite the fact that liveness and boundedness (as a result of requiring well-formedness) are desirable properties for a concurrent systems, we will investigate the relaxation of some of the requirements placed on the Petri nets that these rules can reduce given that, for example, rule Φ_A can be applied to well-formed Petri nets that are not necessarily free choice.

7. REFERENCES

[1] J. Campos, G. Chiola, M. Silva, "Properties and performance bounds for closed free choice synchronized monoclass queueing networks", IEEE Transactions on Automatic Control, vol. 36, no. 12, pages 1368-1381, 1991.

[2] R. David, "GRAFCET - a powerful tool for specification of logic controllers", IEEE Transactions on Control Systems Technology, 3(3), pages 253-268, 1995.

[3] J. Desel, J. Esparza, "Free-choice Petri Nets", Cambridge Tracts in Theoretical Computer Science, Volume 40, Cambridge University Press, 1995.

[4] J. Esparza, “Reduction and Synthesis of Live and Bounded Free Choice Petri Nets”, Information and Computation, 1991.

[5] H.J. Genrich, P.S. Thiagarajan, “A Theory of Bipolar Synchronization Schemes”, Theoretical Computer Science 30, pages 241-318, 1984.

[6] M. Girkar, C.D. Polychronopoulos, "The Hierarchical Task Graph as a Universal Intermediate Representation", International Journal of Parallel Programming, vol. 22, no. 5, October 1994.

[7] L. E. Holloway, B. H. Krogh, A. Giua, "A survey of Petri net methods for controlled discrete event systems", Discrete Event Dynamic Systems: Theory and Applications, Vol. 7, pages 151-190, 1997.

[8] M. Jantzen, “Language theory of Petri nets”, Advances in Petri Nets, LNCS 254-I, pages 397-412, Springer-Verlag, 1987.

[9] A.V. Kovalyov, “On Complete Reducibility of Some Classes of Petri Nets”, 11th International Conference on Application and Theory of Petri Nets, 1990.

[10] T. Murata, “Petri nets: Properties, Analysis and Applications”, Proceedings of IEEE, 77(4), pages 541–580, April 1989.

[11] J. L. Peterson, “Petri Net Theory and the Modeling of Systems”, Prentice-Hall, 1981.

[12] C. A. Petri, “Communications with Automata”, Griffiths Air Force Base Technical Report RADC-TR-65-377, 1966.

[13] P. J. Ramadge, W. M. Wonham, "The control of discrete event systems", Proceedings of IEEE 77(1), pages 81-98, 1989.

[14] W. Reisig, "Petri Nets: An Introduction", Springer-Verlag, 1985.

[15] N. Savoiu, S.K. Shukla, R.K. Gupta, "Petri Net-based Thread Composition for Improved System Level Simulation", University of California Technical Report 03-39, 2003.

[16] SystemC , http://www.systemc.org.

[17] R. G. Taylor, "Models of Computation and Formal Languages", Oxford University Press, 1997.