Petri Net-based Thread Composition for Improved System Level Simulation

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 #03-39
Dept. of Information and Computer Science
University of California, Irvine, CA 92697, USA
Dec 19, 2003

ABSTRACT

Shrinking time-to-market requirements and growing system complexity place an ever-increasing pressure on design teams to deliver quality products on time. As simulation often accounts for a considerable amount of the development cycle time reducing it can either allow more time for design improvements or directly reduce the time-to-market. We have shown that the implicit concurrency in a hardware model can be used to improve its simulation performance if we transform the threading model from that imposed by the hardware model structure to one more suitable for efficient simulation. However, as these transformations tend to be nontrivial changes to the structure of a simulation model, it is important to have a rigorous way of applying them. In this paper, we present a Petri net(PN)-based methodology for the application of such transformations using a set of structural reduction rules. The PNs are annotated with code from the system model and transformed as to extract complete code cycles from them. These cycles have better simulation performance due to lower simulation overheads and can also be mapped to threads for further simulation runtime improvements as shown by our experiments.

TABLE OF CONTENTS

1. INTRODUCTION 3
2. PETRI NETS BACKGROUND 4
3. DESIGN ANALYSIS 5
3.1 Process Analysis 6
4. CONVERSION TO PETRI NETS 7
4.1 Free Choice and Well-formed-ness 9
4.2 Reduction rules 10
5. EXPERIMENTS 16
6. CONCLUSIONS AND FUTURE WORK 16
7. REFERENCES 17
8. ANNEX – Complete Reduction 18

LIST OF FIGURES

Figure 1. A Petri net (a) before and (b) after t_1 fires. 4
Figure 2. Example FFT SystemC description 6
Figure 3. Transformation of process *source* 7
Figure 4. APU Graphs for FFT Example 8
Figure 5. Basic Petri net building blocks 9
Figure 6. Mapping APU to Petri net fragment 10
Figure 7. Petri net for FFT Processes 11
Figure 8. Applying reduction FA 12
Figure 9. Applying reduction FS 13
Figure 10. Applying reduction FT 14
Figure 11. Reduction of FFT example 15

1. INTRODUCTION

C++-based high-level modeling frameworks (such as SystemC[14]) are increasing in acceptance for system level modeling.

As designers progress towards using system level models they seek behavioral tools that allow them to simulate and also synthesize hardware from the same models[15] so as to further reduce the time and effort required by modeling. However, developing a model that is efficient for both synthesis and simulation is rather difficult given the different target requirements. These conflicting requirements must be satisfied without altering the modeling style that hardware designers are accustomed to meaning that the software model of a hardware design must be transparently transformed in order to improve its simulation performance. In [12] it was shown that the concurrency in such a model can be exploited to reduce simulation time by using a multiprocessor simulation platform. That was based on observations made in [4] about the efficiency of using user- and kernel-level threads when simulating a model coupled with work in [11] on what modeling paradigm (i.e. task- or message-based) is most efficient in exploiting kernel-level threads.

However, improving simulation performance in this way required a change from user-level to kernel-level multithreading in the simulation engine kernel which brought on the need for transformations of the simulation model that would alleviate the increased overhead[4] of the kernel-level threads. These transformations utilize inter-process communication analysis to restructure the model description from that imposed by the envisioned hardware units (i.e. a task-based paradigm where each task is specialized for a particular function) to a new structure more suitable for the new threading model (i.e. a message-based paradigm where each task performs all or most of processing required).

Experiments with these transformations have shown improvements in simulation performance and their correctness was verified through bisimulation the original and the transformed designs. However, it is desirable to have a more formal approach to the transformations being applied. To that end, in this paper we describe a PN-based approach to applying these transformations that results in the extraction complete code cycles from systems of concurrent processes.

PNs are a well-known formal model for concurrent process systems and have been extensively used both for theoretical and practical purposes in a variety of fields. One such use of PNs has been in software synthesis where Sgroi[13] used a modified coverability result from [6] to enumerate a set of finite, complete cycles for an equal choice PN given an initial marking M_0. Such a set was deemed a valid schedule once the executability of each cycle was verified through simulation. Similarly in [7], Lin presents another software synthesis technique based on the expansion of a safe free choice PN from an initial marking M_0 until a cycle is encountered. The resulting unrolling is scheduled to obtain a C representation of the original net's functionality.

Our approach is to use a similar PN cycle extraction idea to build a restructured representation of the code but we target improving the simulation of software models of hardware designs. Rather than obtaining a cover or unrolling of the PN we apply a set of reduction rules to a code-labeled PN until we reduce it to an atomic PN. We do this by first obtaining a PN from the original system description where transitions are annotated with specific code fragments as dictated by the simulation semantics. We then use a version of the reduction rules in [2] that were augmented to manipulate the code-labeled transitions so as to maintain simulation semantics. The structural nature of the transformations means that we need not rely on an initial marking or on having to verify the executability of each extracted cycle.

The rest of the paper is organized as follows. In Section 2 we give a brief introduction to general PN concepts used in the rest of the paper. We continue with Section 3 presenting the model transformations using a simple example. We introduce our reduction-based methodology in Section 4 then present our experimental setup and results in Section 5. Finally, observations, conclusions and future directions for improving our work are presented in Section 6.

2. PETRI NETS BACKGROUND

PNs originated in C.A. Petri's work[10] and are a well-known mathematical model for describing concurrent systems. We briefly describe just the PN concepts needed to understand our approach to thread composition while referring the reader to [1] and [8] for a more detailed treatise of PNs.

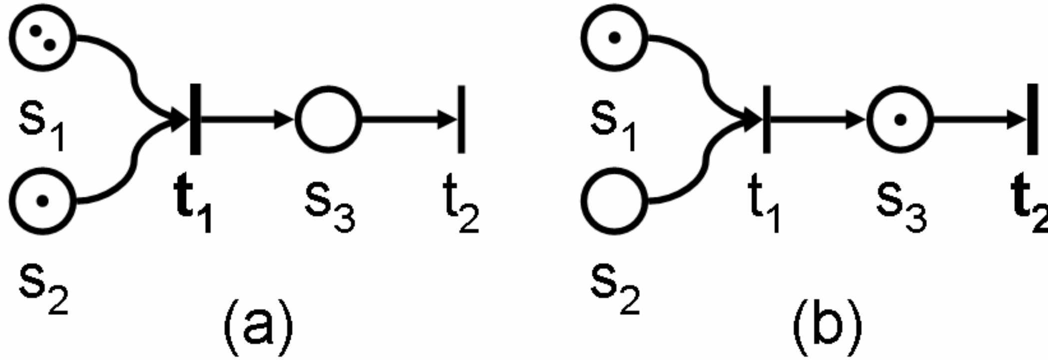

Figure 1. A Petri net (a) before and (b) after t_1 fires.

A *net* N=(S,T,F) is a directed, bipartite graph where S is a set of *places* (represented as circles and denoted as s_i) and T is a set of *transitions* (represented as bars and denoted as t_k). F is a set of arcs between either places and transitions or transitions and places. We assume, without loss of generality, that S, T and F are nonempty and N is connected in order to eliminate trivial nets.

Definition 2-1. The *pre-set* of a node x is the set $\bullet x$ such that if $y\in\bullet x$ then $(y,x)\in F$. Similarly the *post-set* of a node x is defined as the set $x\bullet$ of nodes such that if $y\in x\bullet$ then $(x,y)\in F$.

Related to pre- and post-sets is the notion of cluster.

Definition 2-2. The *cluster* of node x is defined as the set $[x]$ such that $x\in[x]$. Additionally if place $s\in[x]$ then $s\bullet\in[x]$ and if transition $t\in[x]$ then $\bullet t\in[x]$. We also define $C_N=\{[x] | x\in S\cup T\}$ as the *cluster cover* of a net N=(S,T,F).

A *Petri net* is the tuple (N,M_0) composed of a net N and an marking M_0 also known as the *initial marking*. A marking is an assignment of nonnegative integer numbers of tokens to each place in the net. A *PN* is a dynamic entity with additional properties determined by the flow of tokens generated by the firing of enabled transitions. 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. In Figure 1(a) transition t_1 is enabled and fired resulting in the net in Figure 1(b).

Two dynamic properties of interest are liveness and boundedness.

Definition 2-3. A net is *live* if, for every reachable marking M and every transition t, there exists a marking M' from the set markings reachable from M such that t is enabled.

Definition 2-4. A net is *bounded* if for every place s there is a natural number n such that the number of tokens placed in s by any reachable marking is always less than or equal to n.

The flow of tokens through a net can be succinctly captured using an *incidence matrix*.

Definition 2-5. The *incidence matrix* of a net $N=(S,T,F)$ is a matrix with $|S|$ rows and $|T|$ columns defined as $\boldsymbol{I}:(S\times T)\rightarrow\{-1,0,1\}$ such that

$$I(s,t)=\begin{cases}-1 & \text{if } (s,t)\in F\wedge(t,s)\notin F\\ 0 & \text{if } ((s,t)\in F\wedge(t,s)\in F)\vee\\ & \quad (s,t)\notin F\wedge(t,s)\notin F))\\ 1 & \text{if } (s,t)\notin F\wedge(t,s)\in F\end{cases}$$

The net in Figure 1(a) has the following incidence matrix:

$$I=\begin{array}{c|cc} & t_1 & t_2\\ \hline s_1 & -1 & 0\\ s_2 & -1 & 0\\ s_3 & 1 & -1\end{array}$$

Each entry $I(s_i,t_j)$ represents how firing a transition t_j will affect the marking of a place s_i. For example, in the incidence matrix above, we can see that firing transition t_1 will remove one token from place s_1 since $I(s_1,t_1)$=-1.

Despite their dynamic nature, PNs can nonetheless exhibit certain properties that are true at every reachable state. Two such *invariant* properties are referred to as S-invariants and T-invariants.

Definition 2-6. An *S-invariant* of a net is a solution of the system of equations $X\cdot\boldsymbol{I}=\mathbf{0}$. Similarly, a *T-invariant* of a net is a solution to the system of equations $\boldsymbol{I}\cdot X=\mathbf{0}$. An invariant is a called positive if each of it elements is greater than zero.

General PNs have considerable expressive power but their properties are in general either not decidable or computationally too high. Therefore, we will have to restrict our PNs such their algorithms are suitable for practical applications. *State machines* and *marked graphs* are two such restricted PN classes but they lack the ability to model both conflict and synchronization – concepts almost always required by hardware designs. A class of PNs that does satisfy this requirement while maintaining low computational complexity is the class of *free choice* PNs(FCPN). The term *free choice* results from the restriction that *choices* in a conflict be local (i.e. not influenced by *synchronization*). This translates into requiring that the underlying net satisfy if there exists an arc from s to t then there must also exist an arc from any input place of t to any output transition of s.

3. DESIGN ANALYSIS

In **Error! Reference source not found.** we have shown that we can improve the simulation performance of a hardware system description if we change the threading model employed during simulation. We will outline some of those steps to give context to the work being presented here with the help of an FFT example from the SystemC 2.0 distribution[14]. Its simplified pseudo-SystemC description (concurrent processes *source*, *ftt*, and *sink*) is given in Figure 2.

Source

```
init
for ever
 for i=0,15
  wait(in_sample_request)
  in_sample.write
  in_sample_ready.notify
 end for
end for
```

FFT

```
init
for ever
 for i=0,15
  in_sample_request.notify
  wait(in_sample_ready)
  …=in_sample.read
  end for
 fft
 for j=0,15
  out_sample.write
  out_sample_ready.notify
  wait(out_sample_read)
 end for
end for
```

Sink

```
init
for ever
 for i=0,15
  wait(out_sample_ready)
  …=out_sample.read
  out_sample_read.notify
 end for
end for
```

Figure 2. Example FFT SystemC description

3.1 Process Analysis

We start by creating a hierarchical task graph[5] for each process in the hardware design's SystemC description. HTGs allow us to abstract complex, nested constructs (e.g. *if* and *loop* statements) into single graph nodes for easier high level code analysis.

Process restructuring is driven by the observation that if we perform the inter-process communication analysis at a finer level (i.e. less than process level) we can use this information to obtain fewer, larger sections of code that can be simulated relatively independently of each other. This, as we will see later on, will be beneficial twofold. First, it will reduce the computational overhead due to the scheduler having to do bookkeeping for inter-process signal updates during simulation. Second, it will enable us to potentially execute the newly grouped sections of code as kernel-level threads thus allowing for further simulation speed increases through multiprocessing. Therefore an important piece to our process restructuring is the division of each process HTG into so called Atomic Process Units (APU). An APU is defined as the section of an HTG containing the statements that a process may execute (upon invocation by SystemC kernel scheduler[14]) but before relinquishing control to another process.

We assume that each process starts with an initialization step followed by the actual process functionality encapsulated in an infinite loop. This is typical of hardware systems that start by initializing themselves only to then settle in a steady state of waiting for stimuli, processing them and returning to the waiting state.

Since the initialization part is executed only once, we can, without loss of generality, remove it from the process description, as shown in Figure 3(b) for the *source* process, and assume that it will be performed at the beginning of the simulation.

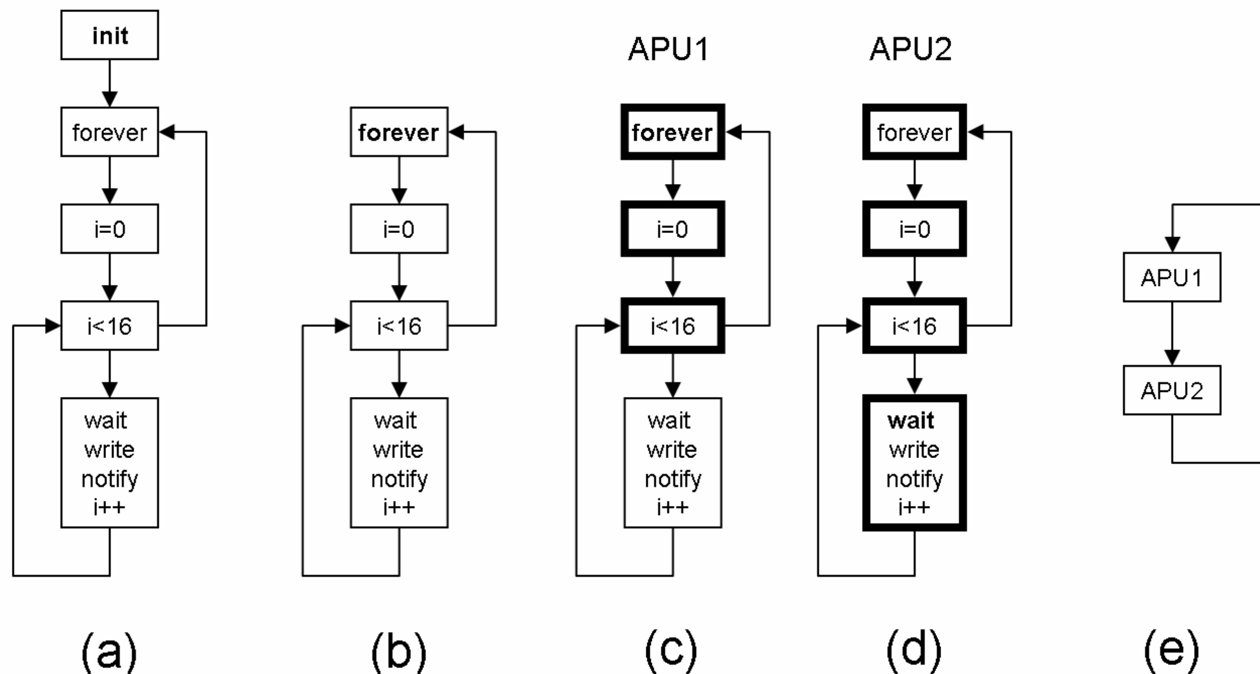

Figure 3. Transformation of process *source*

The APUs are generated from the HTG graph by computing a transitive closure for each synchronization point (e.g. a wait() call in SystemC) where encountering any other synchronization point will block transitive propagation. The highlighted sections of the HTG graphs in Figure 3(c) and Figure 3(d) show the sections that make up the APUs for each of the two synchronization points in the HTG graph of process *source* (i.e. the entry point and the wait() call). The APUs are then interconnected using the existing HTG control flow arcs as shown in Figure 3(e).

Last, but not least, each process is analyzed to determine how it communicates with other processes. This is done by way of determining the synchronization and event generation points present in the source code of the process. The sets of events that an APU is sensitive to (i.e. input events) and that it generates (i.e. output events) are stored with the APU.

By repeating these steps for the *fft* and *sink* processes we obtain the three APU graphs in Figure 4.

4. CONVERSION TO PETRI NETS

Mapping the APU graphs to PNs is a constructive process based on the "basic" building blocks presented in Figure 5(a)-(c).

First each APU is replaced by the building block in Figure 5(a). Note that, while it has only one entry point, it can have multiple exit points (i.e. a conflict or choice point). The reason for this is illustrated in Figure 6 where, starting from the code in Figure 6(a), we derive the HTG graph fragment in Figure 6(b). The APU generation step will assign the outlined HTG fragment to just one APU (given that we generate APUs using wait() synchronization points as boundaries). The PN fragment in Figure 6(c) is the end result with one exit point for each of the possible choices in the source code.

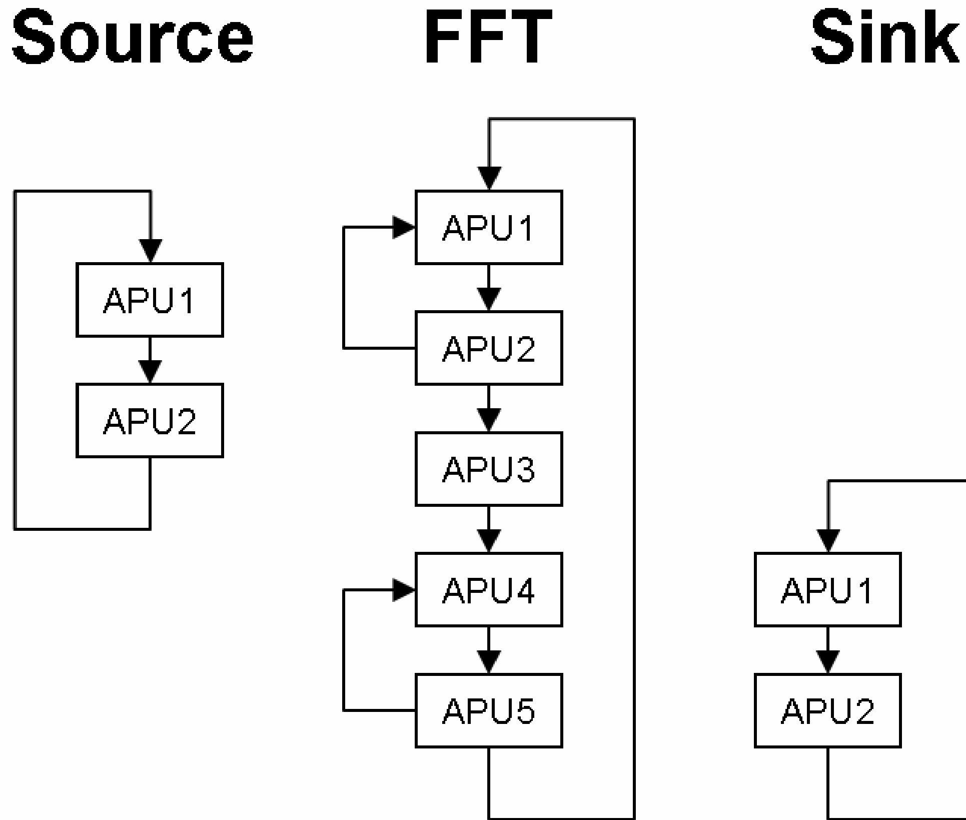

Figure 4. APU Graphs for FFT Example

Transition t_1 in Figure 5(a) serves as a point where any synchronization required for the APU can be performed. For example, the code in Figure 6(a) must synchronize with event e_1. This can be accomplished by introducing a new input place for transition t_1 that will later be connected to the net fragment(s) that produce event e_1 (see Figure 5(d)). The new input place for t_1 ensures that t_1 will not be enabled to fire until both the control flow state and the event state are marked thus achieving the desired synchronization.

As a last step in mapping an APU to the net fragment in Figure 5(a) its transitions are "labeled" with the corresponding HTG graph paths. Transition t_1 is present solely for synchronization purposes and, as such, will not be labeled unlike transitions t_2 through t_n with the respective paths. For example, t_2 will be associated with the path from wait(e_1) to, but not including, wait(e_4).

After the APUs are mapped to PN fragments we must also connect them in such a way as to account for the event flow in the system. This is done by using the previously computed lists of input (i.e. triggers) and output (i.e. triggered) events for each APU.

The input events of an APU were determined by analyzing the type and sensitivity of the process that the APU was generated from as well as the source code that composes the APU. Our current input language,

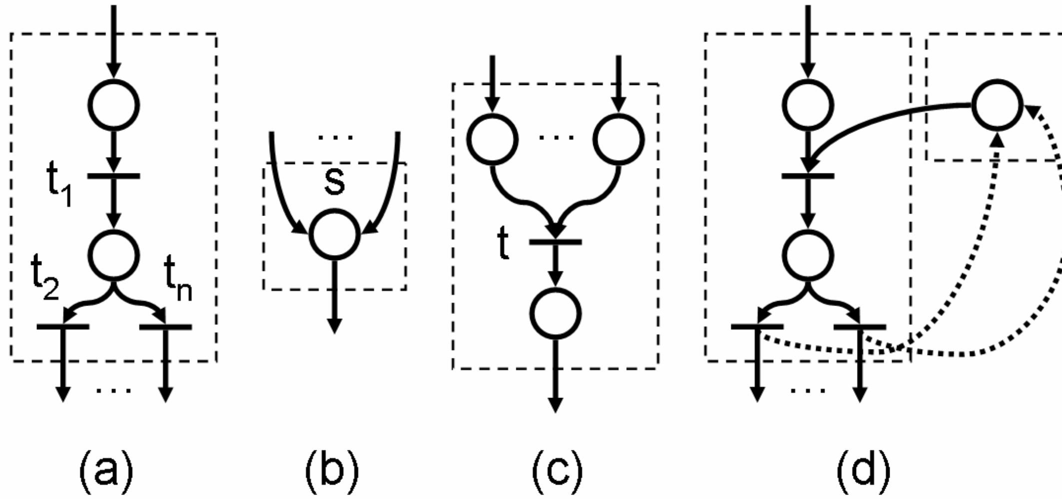

Figure 5. Basic Petri net building blocks

SystemC, allows two such types: an *or-type* (i.e. wait(event$_1$ or ... or event$_n$)) and an *and-type* (i.e. wait(event$_1$ and … and event$_n$)). If *or-type* sensitivity is used then the APU is activated whenever at least one of the input events occurs and this is modeled by the PN fragment in Figure 5(b). As soon as any of the input transitions of place *s* fires it places a token on place s thus enabling the output transition of *s*. Similarly, for *and-type* sensitivity all the input events must occur before the APU is activated. This is modeled in Figure 5(c) by transition *t* requiring that all its input places be marked before it can fire and mark its output place.

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.

The APU in Figure 6(b) has event e_1 in its input list and event e_5 in its output list while Figure 5(d) presents a contrived, but nonetheless valid, way of connecting the input and output events of a PN fragment.

Returning to our FFT example, Figure 7 shows a simplified (for clarity and brevity) overall PN obtained after all the APUs have been mapped to PNs and have been interconnected.

4.1 Free Choice and Well-formed-ness

While we have taken care that our building blocks respect the FCPN tenet of keeping places with multiple output transitions isolated from transitions with multiple input places, we do not yet have a formal proof that the PNs we generate are always free choice. Therefore an extra step is needed in our current methodology to check that the resulting net is free choice. This can be performed in $\boldsymbol{O}(|S|+|T|)$ time using the definition given in Section 2. Another requirement that we place on our PNs is that their underlying nets be well-formed.

Definition 4-1. A net N is *well-formed* if there exists an initial marking M_0 such that (N,M_0) is live and bounded.

More so, every live and bounded net also has the property that there exists a reachable marking M and an occurrence sequence σ such that:

$$\forall t \in T \quad t \in \sigma \wedge M \xrightarrow{\sigma} M$$

This property provided the key 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. We will see in Section 4.2 that such a cycle can be obtained by using an augmented set of reduction rules.

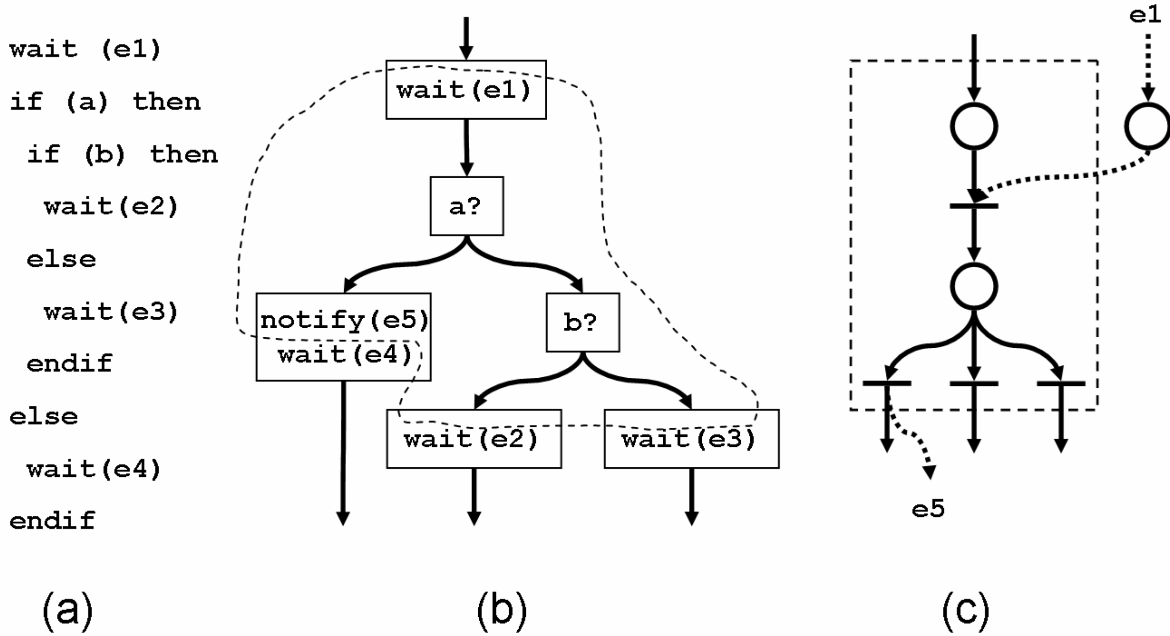

Figure 6. Mapping APU to Petri net fragment

Requiring that a PN be well-formed might seem restrictive but the fact that a well-formed PN is guaranteed to be live and bounded is a desirable trait for reactive concurrent systems. Hardware designs often require that they be deadlock-free and process all the tokens generated within.

Theorem 3.1 Let $N=(S,T,F)$ be a free choice net such that $|S|\geq 1$, $|T|\geq 1$. N is *well-formed* if and only if N is connected, has a positive S- and positive T-invariant and the rank of its incidence matrix is $\text{Rank}(I)=|C_N|-1$.

This theorem allows us to efficiently (polynomial-time complexity algorithm[3]) determine if a FCPN is well-formed.

4.2 Reduction rules

Once well-formed-ness is established can we finally apply a sequence of reductions from a *complete* set of reduction rules with the guarantee that they will reduce the initial PN to an *atomic* PN.

Definition 4.1 A net is called *atomic* if it is isomorphic to the net $N=(S,T,F)$ where $S=\{s\}$, $T=\{t\}$, and $F=\{(s,t), (t,s)\}$.

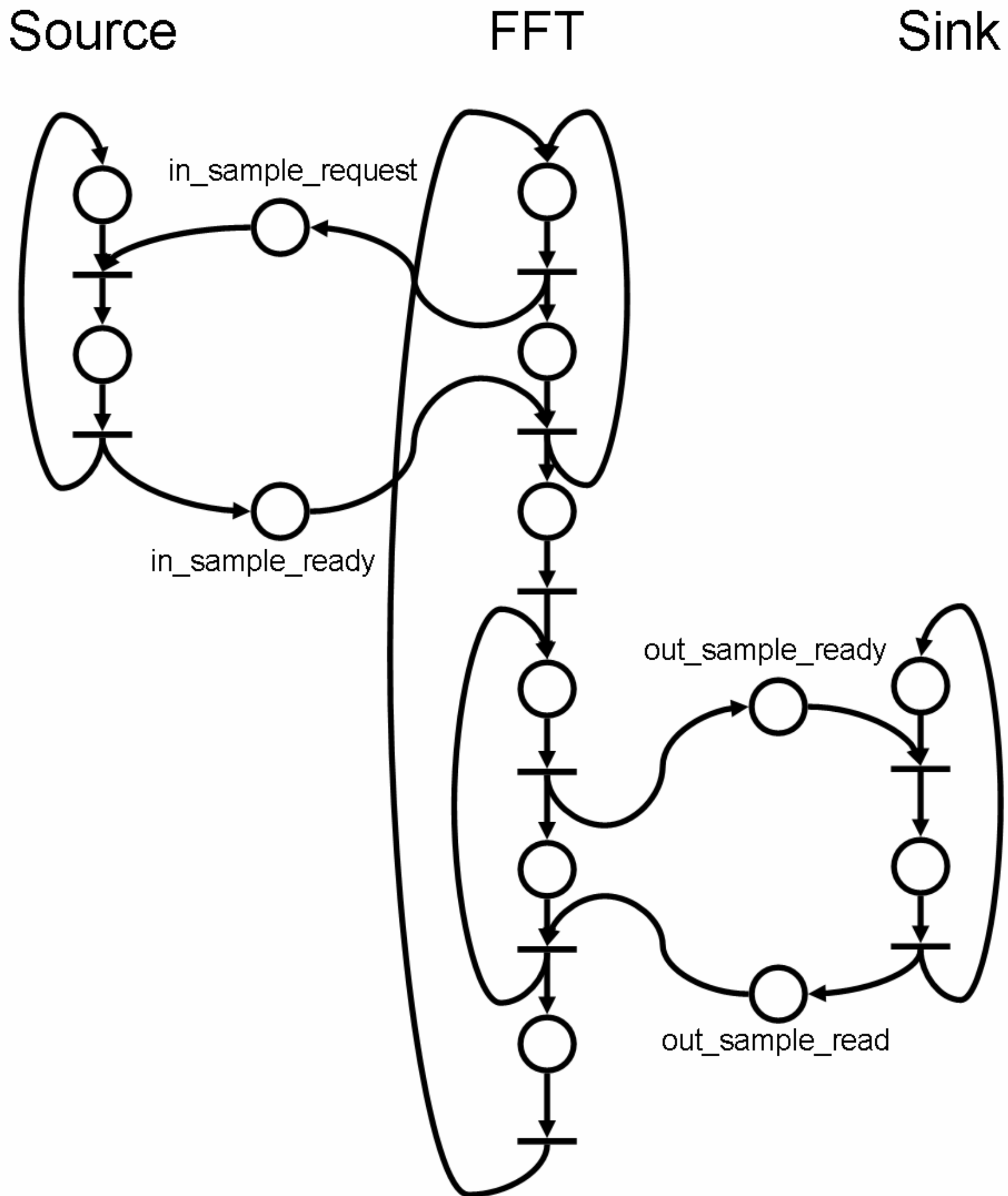

Figure 7. Petri net for FFT Processes

An atomic net has the minimum number of places and transitions required for it to still be live and bounded. The last net in the sequence of reduction steps in Figure 11 is atomic.

Definition 4.2 A set of reductions is *complete* if it can reduce any well-formed net to an atomic net.

Several such sets of reductions have been presented in the PN literature -- most notably the ones in [9] and [2]. We will favor the latter as it can be applied to a more general class of well-formed FCPNs. The set is composed of the following three reductions: ϕ_A, ϕ_S, and ϕ_T. The first reduction deals with places and transitions that are uniquely connected.

Reduction ϕ_A Given a well-formed FCPN N=(S,T,F) where

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

then there exists a 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 rules is presented in Figure 8.

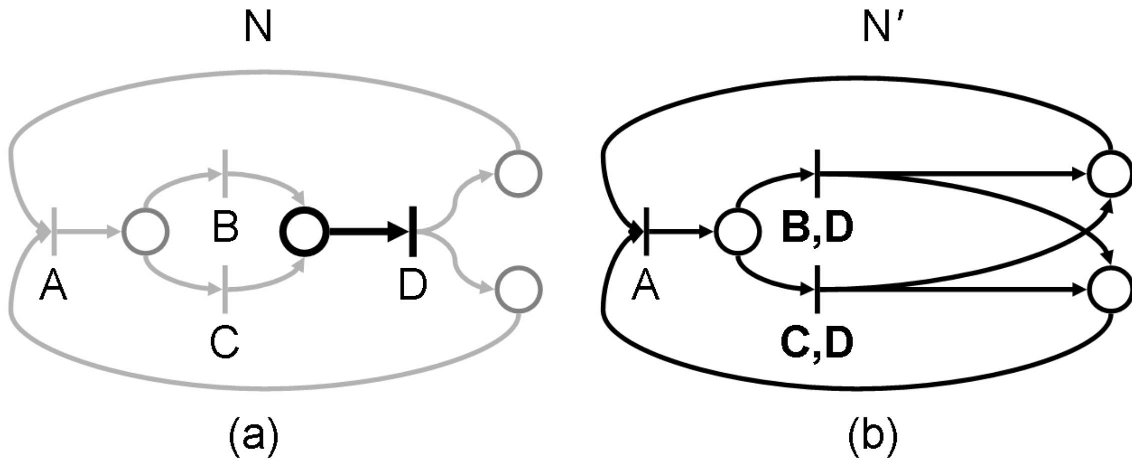

Figure 8. Applying reduction FA

Our augmentation to the original rule is based on the observation that once the highlighted place is marked the highlighted transition is bound to occur. More so, once transitions labeled B and C occur, transition D is also sure to occur. Therefore, when applying ϕ_A, the associated code labels are transformed as follows:

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

Here, $\Lambda(t)$ represents the code associated with transition t while '*,*' represents the concatenation operator. For the example in Figure 8(a) the result of applying ϕ_A is to append the code label D to the code labels of each of the input transitions for place s.

The second rule in the complete set of reductions deals with removing "redundant" places and states the following:

Reduction ϕ_S Given a well-formed FCPN N=(S,T,F) where

$$\begin{aligned} &| S | > 2 \\ &\exists s \in S \quad \textit{nonnegative linearly dependent place} \\ &\bullet s \bigcup s \bullet \neq \varnothing \end{aligned}$$

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

$$S' = S \setminus \{s]$$
$$T' = T$$
$$F' = F \bigcap ((S' \times T') \bigcup (T' \times S'))$$

A graphical representation of this rule is presented in Figure 9. Note that, since only a place (i.e. no code associated with it) is removed, there are no changes to the code associated with the transitions in the net.

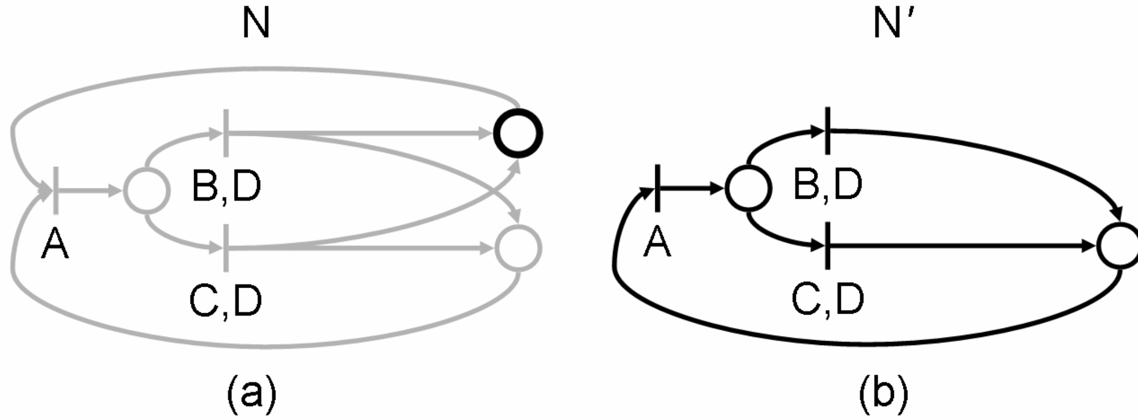

Figure 9. Applying reduction FS

Lastly, the third reduction rule removes "redundant" transitions.

Reduction ϕ_T Given a well-formed FCPN $N=(S,T,F)$ where

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

then there exists a 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 removes a transition and does therefore require changes in the code associated with the remaining transitions. Recall that conflict nodes have conditions associated with them. We can now use these conditions to generate the new code associated with the removal of t

$$\forall s \in ((\bullet t)\bullet) \quad \Lambda(P(s,(t\bullet))) = G(s, \Lambda(P(s,(t\bullet)) | \Lambda(t))$$

In other words, the remaining paths from the predecessors of t to the successor of t will have as associated code the code from both the path and t guarded by the condition governing t.

A graphical representation is given in Figure 10. Note that one more application of ϕ_A would result in an atomic net with the associated code A,cond((B,D),(C,D)) which describes a complete cycle through the initial net.

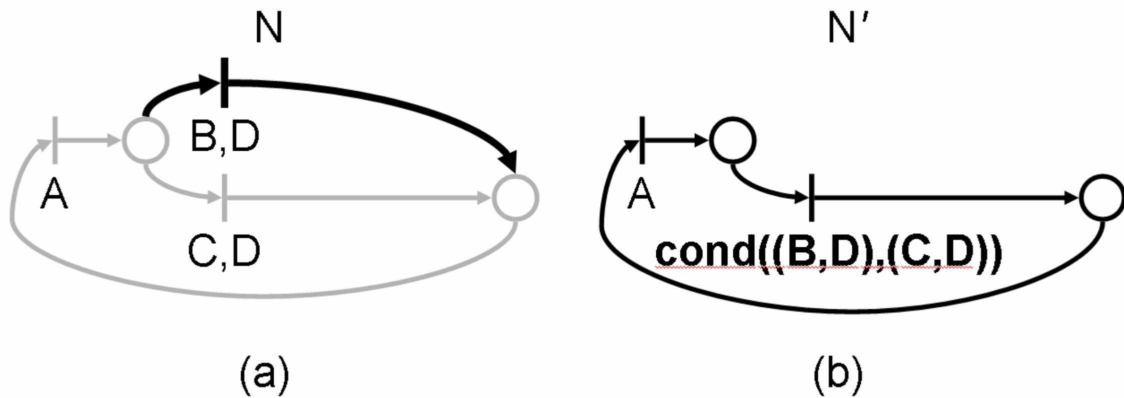

Figure 10. Applying reduction FT

Let us apply the above rules to the sample FFT system. We start with the PN from Figure 11 and present, for simplicity, just the first few reduction steps to be applied to it in order to reach the atomic net. We direct the reader to the Annex for a complete reduction. We first apply ϕ_S to remove the linearly dependent transition highlighted in the initial net. Next we apply ϕ_S once more to remove the place highlighted in the next net. In the third step we apply ϕ_A resulting in the removal of the highlighted place and transition. The code associated with the removed transition is appended to the predecessor transition of the removed place. We can continue applying the appropriate reduction rule until we arrive at the atomic net (i.e. the last net in the chain in Figure 11). The code label associated with the remaining transition will have accumulated a cyclic code path through the initial concurrent processes FFT system.

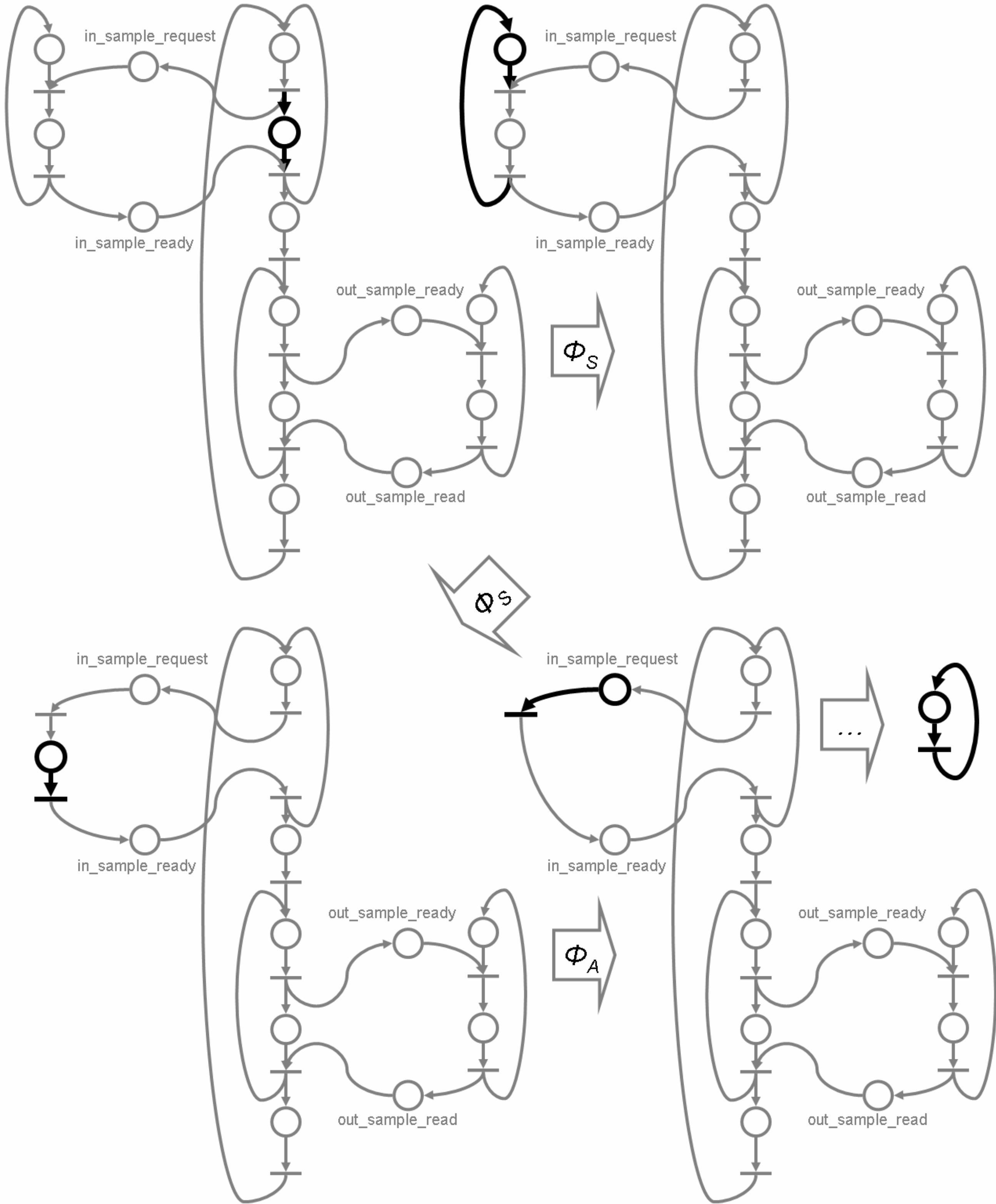

Figure 11. Reduction of FFT example

5. EXPERIMENTS

We have implemented a system that performs the PN conversion, the reduction steps and generates pseudo-SystemC code for the extracted cycles. We then used it to restructure a few of the designs included with the SystemC 2.0 distribution. For comparison we simulated three versions of these systems. First we simulated the original SystemC implementation. We then simulated the generated cycle code where SystemC ports and signals were replaced with our own "light" versions that merely forward the values to the proper process. This was enabled by the restructuring achieved by our transformations that obviate the need to perform signal update computations. Finally, we manually mapped the code from the previous experiment onto threads in a dual processor machine. All simulations were run on a dual PII 300MHz machine running Windows 2000. For single the threaded simulations we have set process affinity to a particular processor so as to avoid any processor switching overhead.

Table 1. Experimental results

Benchmark	Original Time	Rethreaded Time	Multi-Rethreaded Time
FFT	43.22	28.02	14.46
FIR	12.34	2.37	1.67
PKT_SWITCH	38.30	21.65	13.89
RISC_CPU	25.68	19.26	12.32

The columns in Table 1 represent the run times in seconds for each of the three different simulations described above. We can see that significant simulation time improvements are obtained from both reducing the communication bookkeeping overhead as well as from scheduling the extracted cycles as concurrent threads on different processors.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a methodology for extracting complete cycles from code-labeled well-formed FCPNs. This was achieved by applying a sequence of reductions from an augmented set of reduction rules. The extracted cycles were shown to generate simulation models that had lower simulation overhead and can also be simulated as kernel-level threads on multiprocessor machines for added simulation improvement. Our approach is structural by design and therefore did not depend on initial markings or needing to verify that the resulting system's executability. We are currently investigating heuristics to relax the well-formed requirement. We are also in the process of finalizing the implementation of a Verilog front end that would give us access to much larger input designs for more experimental data. Furthermore we are working on a formal proof that our transformations preserve the simulation semantics.

7. REFERENCES

[1] J. Desel, J. Esparza, “Free choice Petri Nets”, Cambridge Tracts in Theoretical Computer Science, Volume 40, Cambridge University Press, 1995.

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

[3] J. Esparza, M. Silva, “A Polynomial-Time Algorithm to Prove Liveness of Bounded Free Choice Nets”, Theoretical Computer Science vol. 102, 1992.

[4] P. Garg, S. Shukla, R. Gupta, “Efficient Usage of Concurrency Models in an Object-oriented Co-design Framework”. In Design Automation and Test in Europe, Designers Forum, 2001.

[5] 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.

[6] M. Hack, "Analysis of production schemata by Petri nets", M.S. Thesis, MIT, February 1972.

[7] B. Lin, “Software Synthesis of Process Based Concurrent Programs”, In Proceedings of the 35th Design Automation Conference, 1998.

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

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

[11] D. C. Schmidt, T. Suda, “The Performance of Alternative Threading Architectures for Parallel Communication Subsystems”, Journal of Parallel and Distributed Computing, submitted 1996.

[12] N. Savoiu, S.K. Shukla, R.K. Gupta, "Efficient Simulation of Synthesis-Oriented System Level Designs", ISSS ’02, October 2002.

[13] M. Sgroi, et al., “Quasi-static Scheduling of Embedded Software Using Equal Conflict Nets”, International Conference on Application and Theory of Petri Nets. ICATPN '99, June 1999.

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

[15] F. Thoen, F. Catthoor, “Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems. Kluwer Academic Publishers, 2000.

8. ANNEX – Complete Reduction