Transitions as Transactions *

Shengyuan Wang, Weiyi Wu, Yao Zhang, and Yuan Dong

Department of Computer Science and Technology
Tsinghua University, Beijing, 100084, China
{wwssyy}@tsinghua.edu.cn
http://soft.cs.tsinghua.edu.cn/~wang

Abstract. As newly developed transactional memory technology has received significant attention as a way to dramatically simplify shared-memory concurrent programming, user-level transactional concurrent programming models have become a very interesting topic in the programming community. However, the fact is that, in existing transactional concurrent programming models, user-level mechanisms have not been well developed. The dilemma is how to make a balance between the performance and correctness of a program. Explicit concurrency among cooperative transactions can undoubtedly decrease the rate of conflicts and improve the performance, but it is harmful to the correctness. In this paper, a transactional concurrent programming approach, based on Petri nets, is presented, which can easily specify concurrency among transactions and do not aggravate programmers remarkably in writing correct transactional concurrent programs. In this approach, a special Petri net system with transition markings is developed. Although such a Petri net system is not defined conventionally, it is shown that its behavior can be simulated through a conventional net, so existing analysis and verification approaches for usual Petri nets can be applied indirectly.

Key words: Concurrent Programming; Transactional Memory; Petri Nets

1 Introduction

Transactional memory mechanism has recently received significant attention as a way to dramatically simplify memory-sharing concurrent programming, in which mutual exclusion and synchronization can be constructed without using any locks [1]. For convenience,a concurrent programming model based on transactional memory mechanism is called a *transactional concurrent programming model* in this paper.

In existing user-level transactional concurrent programming models, there are two major solutions. One of them is to use directly some API's for transactional memory mechanism, which may be implemented by hardware, software or hybrid. For example, programmers can write transactional concurrent programs

* Supported by the National Natural Science Foundation of China under grant No. 90818019

in Java together with the library DSTM2 [2], or in C together with the library TL2-x86 [3]. The advantage of this solution is that one can use existing common languages without changing their compilers, but programmers have to use non-structural library functions carefully.

Another solution is to extend conventional programming languages with some transactional features, such as atomic statement-blocks, as in some new languages Fortress [4], X10 [5], Chapel [6], etc. In this solution, it is easier for programmers to write correct transactional concurrent programs, however, an appropriate compiler must be provided.

To develop a user-level transactional concurrent programming mechanisms, one dilemma is how to make a balance between the performance and correctness of a program. On the one hand, to write an efficient program in the transactional programming paradigm, it still needs programmers' wisdom to build the explicit parallelism among cooperative transactions, in order to decrease the rate of conflicts. On the other hand, however, explicit parallelism is harmful to the correctness of a program, while one of the initial intents of the transactional memory mechanism is to alleviate the burden for a programmer to write concurrent programs.

There have been some contributions in the literature to introduce transactions into existing concurrent programming model. For example, a CCR-based transactional concurrent programming model was proposed by T. Harris and K. Fraser [7], and Baek et al extend the API's of OpenMP [8] to OpenTM [9]. Unfortunately, these approaches still have the usual drawbacks of concurrent programs, that is, not easy to write and not easy to verify.

As well known, Petri nets [10] are useful tools in the specification and verification of concurrent applications. With true concurrency dynamical semantics, a Petri net system has a good opportunity to become a realistic part of a concurrent program for multi-core or multi-thread architecture. In this paper, we present informally a transactional concurrent programming mechanism based on a Petri net, in order to specify concurrency among transactions explicitly while not to aggravate programmers remarkably in writing correct concurrent programs.

Fig.1 shows a simple example described in a typical transactional concurrent program structure, where an atomic statement-block declares a transactional region, and *fork*1, *fork*2, *fork*3 and *cake_ c* are shared objects among cooperative transactions.

In the Petri net system shown in Fig.2, to be explained in more details, each of the transitions declares a transactional region, and *fork*1, *fork*2, *fork*3 and *cake_ c* are shared objects among three cooperative transactions. Since the accesses of *fork*1, *fork*2 , and *fork*3 will never conflict, the rate of access conflicts among transactions is decreased, compared to the program in Fig.1.

Extremely, we can protect all shared objects by the Petri net system, corresponding to the so-called conservative concurrency control. However, if the number of shared objects increases dramatically, the net system may get too big in size. Fortunately, we can leave some shared objects to be protected by the

```
int fork1 = 0, fork2 = 0, fork3 = 0;
int cake_c = 12;

thread ph1:
  while ( true ) {
    atomic {
      read fork1 ;
      read fork2 ;
      write fork1+1 to fork1 ;
      write fork2+1 to fork2 ;
    }
  }

thread ph2:
  while ( true ) {
    atomic {
      read fork2 ;
      read fork3 ;
      write fork2+1 to fork2 ;
      write fork3+1 to fork3 ;
      if ( fork2 mod 10 == 0 ) {
        read cake_c ;
        write cake_c-1 to cake_c ;
      }
    }
  }

thread ph3:
  while ( true ) {
    atomic {
      read fork3 ;
      read fork1 ;
      write fork3+1 to fork3 ;
      write fork1+1 to fork1 ;
      if ( fork1 mod 20 == 0 ) {
        read cake_c ;
        write cake_c-1 to cake_c ;
      }
    }
  }
```

Fig. 1. A simple example of typical transactional concurrent program structure

transactional memory mechanism, if the probability of access conflicts for those shared objects is not that big. For example, we have not made the accesses to *cake_ c* protected by the net system in Fig.2.

We call a shared object to be *critical* or *non-critical* according to its probability of access conflicts. So in the transactional concurrent programming approach suggested in this paper, programmers are encouraged to implement the protection of critical shared objects through Petri net systems, and to leave the non-critical shared objects to be protected automatically by the transactional memory system. In the example shown in Fig.1 and Fig.2, the shared object *cake_ c* is less frequently accessed than *fork*i's, hence, it is assumed that *fork*i's be critical shared objects among *ph*i's, and *cake_ c* be the non-critical shared object among them.

The Petri net model we use is a special colored Petri net model [11], called *resource nets*, which guarantee the access consistency for shared objects. The semantics for a transactional memory mechanism is inspired by the implementation of DSTM2 [2].

The rest of the paper is organized as follows. In Section 2, we make some informal interpretation to the Petri net model, *resource nets*. Further in Section 3, the behavior simulation of a *resource net system* is discussed. Then in Section 4, the program model is briefly presented. Section 5 shows a sample user-level transactional concurrent programming tool, where the concept *resource nets* is applied. Finally, Section 6 gives some remarks and the future work.

2 The Net Model

As stated above, a transactional concurrent program can access two classes of shared objects, *critical* or *non-critical* ones. We use *resource variables* to access critical shared objects, and *global variables* to access non-critical shared objects. In the following, the set of *resource variables* is denoted by V_R, and the set of *global variables* is denoted by V_T.

A *resource net system* is a special colored Petri net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$, where

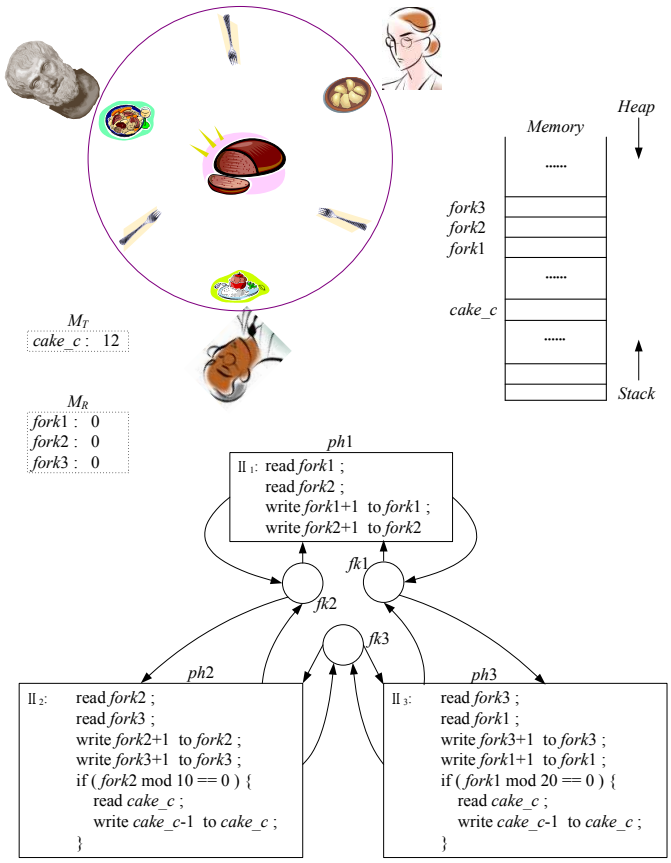

Fig. 2. A resource net system for Dining Philosophers

- $\mathbb{P} \subseteq \{\rho_k \mid k \in \mathbb{N}\}$, and $\mathbb{T} \subseteq \{(\tau_k, \mathbb{I}_k) \mid k \in \mathbb{N}\}$ are the set of places and the set of transitions respectively.
- $\mathbb{A} = (\mathbb{P} \times \mathbb{T}) \cup (\mathbb{T} \times \mathbb{P})$ is the set of arcs.
- $W : \mathbb{A} \to \{\mathbb{S} \mid \mathbb{S} \subseteq V_R\}$ is the inscription function.
- $m_0 \in \mathit{Marking}$ is the initial marking, where $\mathit{Marking} = \{m \mid m : (\mathbb{P} \cup \mathbb{T}) \to \{\mathbb{S} \mid \mathbb{S} \subseteq V_R\}\}$.
- $M_F \subseteq \mathit{Marking}$ is the set of final markings.

A resource net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ has the following features:

- For each transition $(\tau_k, \mathbb{I}_k) \in \mathbb{T}$, a command sequence $\mathbb{I}_k$ is attached. When a transition $(\tau_k, \mathbb{I}_k)$ is fired, it starts a transaction for the command sequence $\mathbb{I}_k$. A command in $\mathbb{I}_k$ can access shared variables in $V_R \cup V_T$, and the variables local to τ_k.
- A transition can hold a token while its transaction is executing, and the token does not return to the net system until the computation is committed or aborted. So we extend the definition of marking with transition markings.

– It is possible that M_F is empty, which is the usual case in conventional Petri net systems..

It is worth to noting that variables in V_R should be disjoined in locations with each other, which are usually implemented by the compiler.

2.1 An Example

***Example* 1** Fig.2 shows a transactional concurrent program with $V_R = \{$ *fork*1, *fork*2,*fork*3$\}$ and $V_T = \{$ *cake_ c* $\}$. The *resource net system* $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$, where

– $\mathbb{P} = \{fk1, fk2, fk3\}$.
– $\mathbb{T} = \{tr1, tr2, tr3\}$, where $tr1 = (ph1, \mathbb{I}_1)$, $tr2 = (ph2, \mathbb{I}_2)$, and $tr3 = (ph3, \mathbb{I}_3)$, where $\mathbb{I}_1$, $\mathbb{I}_2$ and $\mathbb{I}_3$ are command sequences attached to transitions ph1, ph2, and ph3 respectively, as is illustrated in Fig.2.
– $\mathbb{A} = \{(tr1,fk1), (tr1,fk2), (tr2,fk2), (tr2,fk3), (tr3,fk3), (tr3,fk1), (fk1,tr1), (fk1, tr3), (fk2, tr2), (fk2, tr1), (fk3, tr3), (fk3, tr2)\}$.

– W is defined by:
$W(fk1, tr1) = W(tr1, fk1) = W(fk1, tr3) = W(tr3, fk1) = \{fork1\}, W(fk2, tr2) = W(tr2, fk2) = W(fk2, tr1) = W(tr1, fk2) = \{fork2\}, W(fk3, tr2) = W(tr2, fk3) = W(fk3, tr3) = W(tr3, fk3) = \{fork3\}$.
– $m_0 \in$ *Marking* is defined by:
$m_0(fk1) = \{fork1\}$, $m_0(fk2) = \{fork2\}$, $m_0(fk3) = \{fork3\}$, and $m_0(\tau) = \emptyset$ for $\tau = tr1, tr2, tr3$.
– $M_F = \emptyset$.

2.2 Well-Formed Resource Net Systems

A resource net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ is *well-formed*, if

– $\mathbb{P} \cap \mathbb{T} = \emptyset$.
– $\forall \rho \in \mathbb{P}. \forall v_1, v_2 \in m_0(\rho). (v_1 \neq v_2)$, that is, at the initial marking, all tokens owned by a place are corresponding to different resource variables.
– $\forall \rho_1, \rho_2 \in \mathbb{P}. \forall v_1 \forall v_2. (\rho_1 \neq \rho_2 \wedge v_1 \in m_0(\rho_1) \wedge v_2 \in m_0(\rho_2) \rightarrow v_1 \neq v_2)$, that is, at the initial marking, tokens owned by different places have disjoint resource variables.
– $\forall \tau \in \mathbb{T}. (m_0(\tau) = \emptyset)$, that is, at the initial marking, every transition contains no tokens.
– $\forall m \in M_F. \forall \tau \in \mathbb{T}. (m(\tau) = \emptyset)$, that is, at each of final markings, every transition will not contain any tokens.
– $\forall \tau \in \mathbb{T}. \forall \rho_1, \rho_2 \in {}^\bullet\tau. \forall v_1 \forall v_2. (\rho_1 \neq \rho_2 \wedge v_1 \in W(\tau, \rho_1) \wedge v_2 \in W(\tau, \rho_2) \rightarrow v_1 \neq v_2)$, that is, all sets of resource variables on the incoming arcs of the same transition are disjoined with each other.

- $\forall \tau \in \mathbb{T}.\forall \rho_1, \rho_2 \in \tau\bullet.\forall v_1 \forall v_2.(\rho_1 \neq \rho_2 \wedge v_1 \in W(\rho_1, \tau) \wedge v_2 \in W(\rho_2, \tau) \rightarrow v_1 \neq v_2)$, that is, all sets of memory blocks on the outgoing arcs of the same transition are disjoined with each other.
- $\forall \tau \in \mathbb{T}.\forall \rho \in \tau\bullet.(W(\tau, \rho) \subseteq \bigcup_{\rho' \in \bullet\tau}(W((\rho', \tau)))$, that is, no extra shared objects be produced within a transaction associated to each of transitions. For simplification,in this paper, we don't consider the dynamic memory allocation for a shared object within a transaction.

In the above,the pre-set and post-set of a transition $\tau \in \mathbb{T}$ or a place $\rho \in \mathbb{P}$ are used, which are defined as usual: $\bullet\tau = \{\rho \mid (\rho, \tau) \in \mathbb{A}\}$, $\tau\bullet = \{\rho \mid (\tau, \rho) \in \mathbb{A}\}$, $\bullet\rho = \{\tau \mid (\tau, \rho) \in \mathbb{A}\}$, and $\rho\bullet = \{\tau \mid (\rho, \tau) \in \mathbb{A}\}$.

***Example* 2** It is easy to show that the resource net system in Example 1 is well-formed.

2.3 Execution Semantics

To show the execution semantics of a resource net system, we define *TranState* = {*blocked*, *active*, *aborted*, *committed* }, consisting of 4 transaction states of a transition. At the initial marking, every transition has the state *blocked*.

Entering a Transition Whenever a transition τ in the resource net system is in state *blocked*, and the firing condition for τ is satisfied under the current marking m, that is, $\forall \rho \in \bullet\tau.(m(\rho) \supseteq W(\rho, \tau))$, and in the same time, the current marking is not a final state, that is, $m \notin M_F$, the system can enter the transition such that a transaction is started and the transition gets to hold tokens. When it occurs, the state of the τ will be *active*.

Execution of a Command Sequence Whenever a transition τ in the resource net system is in state *active*, and the next command in its remained command sequence is c, the transition can make a progress by executing the command c. We need several separate rules respectively for several cases:

(1) If c reads or writes to a global variable which has been written most recently by some other transition but τ, a *read/write confliction* occurs, and τ will be in the state *aborted*.

(2) If the execution of c has no *read/write confliction* and c has no *write* operation to any global variables, the transition will keep in state *active*.

(3) If the execution of c has no *read/write confliction* and c has a *write* operation to some global variable x, the transition will keep in state *active*, while the system will record τ to be the transition that most recently written to x.

Ready to Commit a Transaction Whenever a transition τ in the resource net system is in state *active*, and there is no next command in its remained command sequence, the system can make a progress to change the state of τ from *active* to *committed*, meaning that the transaction associated to τ is ready to commit.

Committing a Transaction Whenever a transition τ in the resource net system is in state *committed*, the system can make a progress to commit the transaction associated to τ, changing the state of τ from *committed* to *blocked*.

Aborting a Transaction Whenever a transition τ in the resource net system is in state *aborted*, the system can make a progress to return tokens to places in the pre-set of τ, changing the state of τ from *aborted* to *blocked*.

Let $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ be a well-formed resource net system, it is easy to show that for any reachable marking m from the initial marking m_0, the following two properties are satisfied

- $\forall x \in \mathbb{P} \cup \mathbb{T}. \forall v_1, v_2 \in m(x).(v_1 \neq v_2)$,that is, at the marking m, all tokens owned by a place or a transition are corresponding to different resource variables.
- $\forall x_1, x_2 \in \mathbb{P} \cup \mathbb{T}. \forall v_1 \forall v_2.(x_1 \neq x_2 \wedge v_1 \in m(x_1) \wedge v_2 \in m(x_2) \rightarrow v_1 \neq v_2)$, that is, at the marking m, all tokens owned by different places or transitions have disjoint resource variables.

***Example* 3** Since the resource net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ in Example 1 is well-formed. So the above two properties will keep in a well-formed program state during its execution.

3 Behavior Simulation of a Resource Net System

In this section, it will be shown that a well-formed resource net system $\mathcal{N}$ can be reduced to an usual colored Petri net system *desugar*($\mathcal{N}$) so that the behavior of $\mathcal{N}$ can be simulated by *desugar*($\mathcal{N}$), which can be analyzed and verified by using existing approaches in the Petri net community.

The transformation from $\mathcal{N}$ to *desugar*($\mathcal{N}$) is called *desugaring*. Before and after desugaring, the change of net structure can be illustrated by Fig.3.

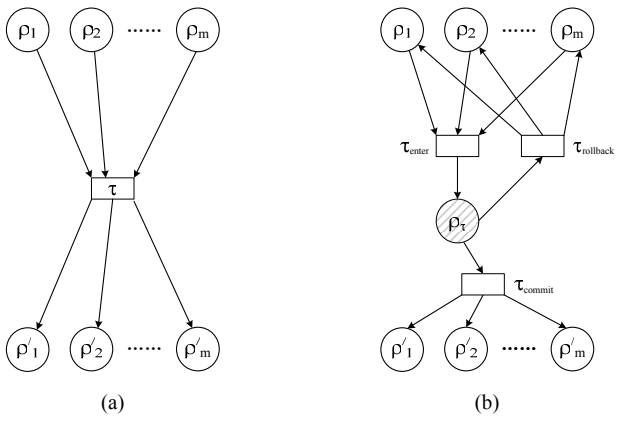

Fig. 3. Net structure before and after desugaring

***Example* 4** Consider the well-formed resource net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ in Example 1. *desugar*($\mathcal{N}$) $= (\mathbb{P}', \mathbb{T}', \mathbb{A}', W', m_0', M_F')$ can be depicted in Fig.4, where

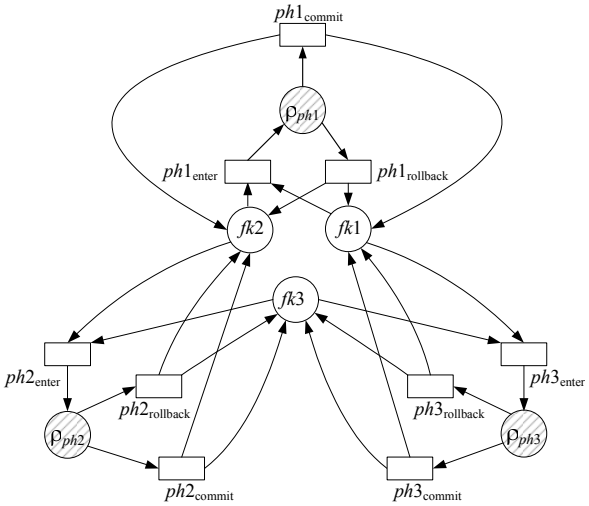

Fig. 4. Example of a desugaring net system

- $\mathbb{P}' = \{fk1, fk2, fk3, \rho_{ph1}, \rho_{ph2}, \rho_{ph3}\}$.
- $\mathbb{T}' = \{ph1_{enter}, ph2_{enter}, ph3_{enter}, ..., ph3_{commit}\}$
- $\mathbb{A}' = \{(ph1_{enter}, \rho_{ph1}), (ph2_{enter}, \rho_{ph2}), (ph3_{enter}, \rho_{ph3}),$
  $...,$
  $(fk1, ph1_{enter}), (fk1, ph3_{enter}), ...,$
  $(ph1_{rollback}, fk1), (ph3_{rollback}, fk1), ...,$
  $(ph1_{commit}, fk1), (ph3_{commit}, fk1), ...,$
  $..., (ph3_{commit}, fk3)\}$.
- The definition of W' is illustrated in Table 1 (partly).
- m'_0 is defined by $m'_0(\rho_{ph1}) = m'_0(\rho_{ph2}) = m'_0(\rho_{ph3}) = \emptyset$, $m'_0(fk1) = \{\{fork1\}\}$, $m'_0(fk2) = \{\{fork2\}\}$, and $m'_0(fk3) = \{\{fork3\}\}$.
- $M'_F = \emptyset$.

It is not difficult to establish a behavior simulation relation between $\mathcal{N}$ and *desugar*($\mathcal{N}$), and show that many behavior properties, including *deadlock-freeness*, for $\mathcal{N}$ can be verified indirectly by verifying those for *desugar*($\mathcal{N}$). For example, it is easy to verify that the usual Petri net system *desugar*($\mathcal{N}$) in Example 4 is deadlock-free, so we can conclude that the resource net system $\mathcal{N}$ is also deadlock-free.

It is worth to noting that the execution semantics can guarantee behaviour consistence between $\mathcal{N}$ and *desugar*($\mathcal{N}$). For every transition τ in $\mathcal{N}$ and ρ_τ in *desugar*($\mathcal{N}$) as illustrated in Fig.3, we have

- If τ is in state *blocked*, there no token in ρ_τ, and τ_{enter} is enabled. If τ_{enter} fires, τ will be in state *active* at the same time.
- If τ is in state *active*, nether τ_{commit} or $\tau_{rollback}$ will fire though there exist tokens in ρ_τ.
- If τ is in state *committed*, τ_{commit} is enabled.
- If τ is in state *aborted*, $\tau_{rollback}$ is enabled.
- τ is in state *active*, *committed*, or *aborted*, iff there no token in ρ_τ.

Table 1. $W' : \mathbb{A}' \to \{\mathbb{L} \mid \mathbb{L} \subseteq 2^{Label}\}$

if $a =$	then $W'(a) =$
$(ph1_{enter}, \rho_{ph1})$	$\{\{fork1\}, \{fork2\}\}$
$(ph2_{enter}, \rho_{ph2})$	$\{\{fork2\}, \{fork3\}\}$
$(ph3_{enter}, \rho_{ph3})$	$\{\{fork3\}, \{fork1\}\}$
...	...
$(fk1, ph1_{enter})$	$\{\{fork1\}\}$
$(fk1, ph3_{enter})$	$\{\{fork1\}\}$
...	...
$(ph1_{rollback}, fk1)$	$\{\{fork1\}\}$
$(ph3_{rollback}, fk1)$	$\{\{fork1\}\}$
...	...
$(ph1_{commit}, fk1)$	$\{\{fork1\}\}$
$(ph3_{commit}, fk1)$	$\{\{fork1\}\}$
...	...
$(ph3_{commit}, fk3)$	$\{\{fork3\}\}$

For the sake of limited space, in this paper, we have not formally defined the desugaring and the behavior simulation relation.

4 The Program Model

In the transactional concurrent programming approach of this paper, a program consists of a set of Petri net systems, which are protected parts in the system, and a set of unprotected threads which contains an initial thread identified *root* and other unprotected threads. Resource variables can only be accessed within protected parts. At the beginning, the thread *root* is initialized to execute at the level which we call *top level*.

A set of Petri net systems can spawn outside a Petri net system, initialized with new allocated resource variables or their references. When a transition in a Petri net system is fired, it becomes a (transactional) transition thread, which will eventually commit, or rollback due to conflicts to access the shared memory.

An unprotected thread except for the thread *root* can be spawned outside a Petri net system.

The program ends if all the Petri net systems achieve one of their final states, and in the same time all the unprotected threads execute to the end.

5 A sample user-level transactional concurrent programming tool

A sample user-level transactional concurrent programming tool has been developing in our lab, based on available software sources, DSTM2 [2], PNK [13] and GJC [14]. In the programming model of this sample programming tool, a program consists of a set of Petri net systems, corresponding to resource net systems in this paper, and other part written in Java Language.

A simple visual IDE for this programming model has been developing.

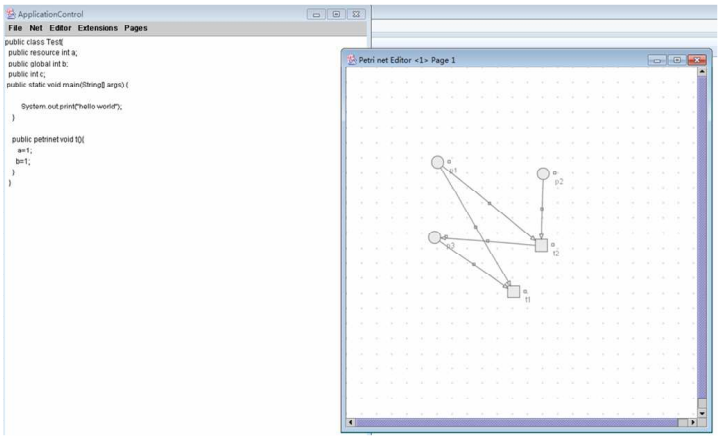

Fig. 5. A basic editing view

Editors In the IDE, each of the elements of a program can be visually edited. Fig.5 shows a basic editing view. A editor for a Petri net system is similar to that provided in PNK source, but some modifications to add code editing area, to make the code editing to be the main input area, and to integrate with associate compilation operations.

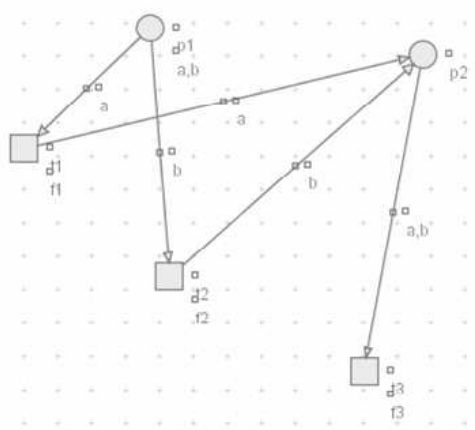

Fig. 6. A visual Petri net system

A visually edited Petri net system will be automatically translated to some code to be fed to the compiler. It actually completes a class inherited from a class PetriNet for the programmer, where PetriNet is the class encapsulated for a special Petri net

virtual machine. For example, for the Petri net system in Fig.6, the result class will be the one in Fig.7.

```
package base_directory.pnml.compile;

public class PNTest extends PetriNet{
   public PNTest(Object objectSource) {
      super(objectSource);
      this.AddTransition("t1","code", "f1");
      this.AddTransition("t2","code", "f2");
      this.AddTransition("t3","code", "f3");
      this.AddPlace("p1", "a,b", "marking");
      this.AddPlace("p2", "", "marking");
      this.AddArc("p1", "t1", "a", "inscription");
      this.AddArc("t1", "p2", "a", "inscription");
      this.AddArc("p1", "t2", "b", "inscription");
      this.AddArc("t2", "p2", "b", "inscription");
      this.AddArc("p2", "t3", "a,b", "inscription");
   }
}
```

Fig. 7. Class for a Petri net system

Associate Compilation The *associate compilation* makes the static check of a Petri net system, then associates its code with all other parts of opened codes and compiles them together with each other. At the early time of the compilation, the Petri net system is translated into its internal form as the Petri net virtual machine instructions, which then is added to its specific class.

Variables corresponding to transaction memory blocks and resource memory blocks are declared with the modifiers *global* and *resource* respectively. Besides, the code for each transition of a Petri net system is defined by a specific member function with the modifier *petrinet*. In the associate compilation, the lexical, syntactical, and semantical analysis associate to the modifiers *global*, *resource* and *petrinet* has been processed carefully.

The compiler has been implemented based on GJC [14], a open Java compiler released by Sun, and kept the original logic of GJC unchanged.

The statical semantic check for a Petri net system is corresponding to the definition of a *well-formed resource net system* in Section 2.2.

STM Integration The transactional memory support is based on DSTM2 [2], with each piece of transition code automatically trisected by invoking provided STM APIs. Hence the variable with modifier Global can be protected by the transactional memory system.

In order to integrate DSTM2's API, we need to make some modification to GJC. We need to change the type of every variable with modifier Global to a wrapper class with a factory. Also we need to change every reference of those variables and every left-value consisted of those variables to corresponding DSTM2's APIs.

Fig.8 illustrates how to transact a global variable in our implementation. Currently we only support basic types or simple "copyable" types.

```
                        @atomic interface _T {
                          T getValue ();
                          void setValue (T value);
                        }
                        Factory<_T> factory_T =
                          Thread.makeFactory(_T.class);
global T a;             _T _a = factory_T.create();
a = x;                  _a.setValue(x);
x = a;                  x = _a.getValue();
```

Fig. 8. Transactional global variables

Petri Net Virtual Machine As stated above, a Petri net system is finally translated to some class inherited from a class PetriNet, which encapsulates interfaces for a special Petri net virtual machine. The Petri net virtual machine is now simply designed with the following instructions:

- AddTransition (name,code)
- AddPlace (name,resource)
- AddArc (Source, Target, Inscription)
- Start ()
- Join ()

where AddTransition, AddPlace, and AddArc are used to construct a Petri net system, and Start and Join used for scheduling the execution of a Petri net system.

Fig.9 shows an example to start the Petri net system defined in Fig.6 or Fig.7. One possible execution result will be "a=1 a=1 b=3", and another possible result is "a=0 a=1 b=3", as is illustrated in Fig.10.

Fig.11 and Fig.12 illustrate the integration of a simple Petri net transition simulator and DSTM routine. The left routine in Fig.11 simulate a Petri net transition to do something, and the right part is the DSTM routine to do the same task. Fig. 12 integrates the functions of two routines in Fig.11, getting a so-called PNTM routine.

The Petri net virtual machine can be implemented on any architecture you like, especially, it will be helpful if the target architecture can efficiently support concurrent programming, such as a CMP system. Until now, we have just implemented the Petri net virtual machine based on JVM.

The latest stable version of source code, in which the invoking of STM API's in DSTM2 has not been packaged, can be downloaded at the URL:
`http://soft.cs.tsinghua.edu.cn/~wang/projects/NSFC90818019/software/pntm.rar`

We are making a research plan to extend the virtual machine and its implementation based on some reconfigurable simulator for multi-core architectures such that some basic performance analysis could be made.

```
package base_directory.pnml.compile;

public class Test {
   private resource int a =0;
   private resource int b =0;
   public static void main(String[] args) {
      Test t= new Test();
      new PNTest(t).start();
   }
   public petrinet  void f1() {
      a=1;
   }
   public petrinet void f2() {
      System.out.print("a="+a);
      b=3;
   }
   public petrinet  void f3() {
      System.out.print("a="+a);
      System.out.print("b="+b);
      a=4;
      b=5;
   }
}
```

Fig. 9. Example for starting a Petri net system

```
<terminated> Test (1) [Java Application] D:\j2sdk1.4.2\jre\bin\javaw.exe
a=1a=1b=3
```

```
<terminated> Test (1) [Java Application] D:\j2sdk1.4.2\jre\bin\
a=0a=1b=3
```

Fig. 10. Two different executions on the simple virtual machine

6 Remarks and Future Work

The paper presents an approach to integrate a Petri net system with a transitional memory mechanism, which has currently been applied to the implementation of a user-level transactional concurrent programming tool in our lab. There is few formalism to play such a role as we have known so far. There exist researches based on Petri nets to model atomic or transactional threads, however the net system is not a part of the program. For example, an approach to check causal atomicity is proposed by modeling programs using Petri Nets [12]. At some extent, concurrent programming models based on Petri nets, such as OPN [15], CLOWN [16], COO [17], CO-OPN/2 [18], and Elementary Object Nets [19] may be extended to support various transaction semantics with the conservative concurrency control.

We observed that the integration of Petri nets with transactional memory can bring benefits to both side, which is the motivation of the paper. On the one hand, with

```
TestAndConsumeToken();
DoThings();
ProduceToken();
```

PN-Transition routine

```
Thread.onCommit(new Runnable() {
  public void run () {
    Commit();
  }
});
Thread.onAbort(new Runnable() {
  public void run () {
    Abort();
  }
});
Thread.doIt(new Callable<Void>() {
  public Void call ()
      throws Exception {
    DoThings();
  }
});
```

DSTM routine

Fig. 11. Petri net transition and DSTM routines

```
Thread.onCommit(new Runnable() {
  public void run () {
    ProduceToken();
  }
});
Thread.onAbort(new Runnable() {
  public void run () {
    ReturnToken();
  }
});
TestAndConsumeToken();
Thread.doIt(new Callable<Void>() {
  public Void call ()
      throws Exception {
    DoThings();
  }
});
```

Fig. 12. PNTM routine

transactional memory, a finer granularity of concurrency can be achieved in a Petri net system, and the scale of the net model can be controlled flexibly. On the other hand, with a Petri net system, the concurrency among cooperative transactions can be built explicitly, which can undoubtedly decrease the rate of conflicts and improve the performance, while the analysis and verification capability of a Petri net model can be inherited.

The main idea in a resource net system, the net system presented in the paper, is to classify shared resources in two classes: (1) resources such that the access policy is driven by the net structure, so that mutual exclusion is guaranteed; (2) resources

whose access policy is driven by a transactional memory model, with possible conflicts, resolved by a commit-rollback protocol.

That is, our approach advocates the methodology that critical objects shared among concurrent transactions will be protected through a resource net system, while non-critical shared objects be left protected automatically by the transactional memory system. Thus the net system can be designed flexibly to keep a moderate size and in a finer granularity than usual net system.

It is shown that the behavior of a well-formed resource net system can be simulated by its desugar net system, which can be analyzed and verified by using existing approaches in the Petri net community. Behavior properties for a well-formed resource net system, such as deadlock-freeness, can be verified indirectly by verifying those for its desugaring net system. For example, INA tool [20] can be directly integrated into our programming tool being developed, as has been done in PNK.

A practical user-level transactional concurrent programming tool, based on DSTM2 [2], PNK [13] and GJC [14], has been developing in our lab. The version so far is not suitable to make a performance analysis because the target virtual machine on which a program with Petri net structures runs is implemented simply based on JVM. We are making a plan to extend the virtual machine and its implementation such that some basic performance analysis could be made.

Certainly, the approach could be extended to other formalisms as well. Furthermore, how to decide critical or non-critical shared objects, we believe, would become an interesting area in software design methodology.

References

1. Tim Harris, et al., Transactional Memory: An Overview, IEEE Micro, vol. 27, no. 3, Pages 8-29, 2007.
2. Maurice Herlihy, Victor Luchangco, Mark Moir, A Flexible Framework for Implementing Software Transactional Memory, In Preceedings of OOPSLA'06, Pages 253-262, 2006.
3. TL2-x86, Stanford Transactional Applications for Multi-Processing, http://stamp.stanford.edu/.
4. E. Allen et al., The Fortress Language Specification, Sun Microsystems, 2005.
5. P. Charles et al, X10: An Object-Oriented Approach to Non-Uniform Cluster Computing, Proc. 20th Ann. ACM SIGPLAN Conf Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA05), ACM Press, pp. 519-538, 2005.
6. Chapel Specification 0.4, Cray Inc., 2005; http://chapel.cs.washington.edu/Specification.pdf.
7. Tim Harris, Keir Fraser, Language Support for Lightweight Transactions, OOPSLA'03, October 26-30, 2003.
8. The OpenMP API specification for parallel programming. URL: http://openmp.org.
9. Baek W., Minh C. C., Trautmann M., Kozyrakis C., and Olukotun K., The OpenTM Transactional Application Programming Interface, In PACT'07: Proceedings of the 16th international conference on Parallel architectures and compilation techniques, Washington, DC, USA: IEEE Computer Society, pp. 376-387, 2007.
10. W. Reisig, Petri Nets, EATCS Monagraphs on Theoretical Computer Science, Springer Verlag, 1985.

11. K.Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 1, EATCS Monographs in Computer Science, Springer verlag, 1992.
12. Azadeh Farzan, P. Madhusudan,Causal Atomicity. In Proceedings of Computer Aided Verification (CAV) 2006, Lecture Notes in Computer Science, volume 4144: 315-328, 2006.
13. Ekkart Kindler, Michael Weber, The Petri Net Kernel: An infrastructure for building Petri net tools, International Journal on Software Tools for Technology Transfer (STTT), Vol 3, No.: 486-497, 2001. PNK available at : http://www2.informatik.huberlin.de/top/pnk/.
14. GJC available at : http://www.sun.com/software/communitysource/j2se /java2/download.xml
15. C.A. Lakos, Object-Oriented Modelling with Object Petri Nets, Concurrent Object-Oriented Programming and Petri Nets, G. Agha, F.D. Cindio, and G. Rozenberg (eds.), Lecture Notes in Computer Science 2001, Springer-Verlag, pages 1-37, 2001.
16. E.Batiston, A.Chizzoni, Fiorella De Cindo, CLOWN as a Testbed for Concurrent Object-Oriented Concepts, Concurrent Object-Oriented Programming and Petri Nets, G. Agha, F.D. Cindio, and G. Rozenberg (eds.), Lecture Notes in Computer Science 2001, Springer-Verlag, pages 131-163, 2001.
17. C.Sibertin-Blanc, CoOperative Objects: Principles, Use and Implementation, Concurrent Object-Oriented Programming and Petri Nets, G. Agha, F.D. Cindio, and G. Rozenberg (eds.), Lecture Notes in Computer Science 2001, Springer-Verlag, pages 216-246, 2001.
18. O. Biberstein, D. Buchs, N. Guelfi, Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism, Concurrent Object-Oriented Programming and Petri Nets, G. Agha, F.D. Cindio, and G. Rozenberg (eds.), Lecture Notes in Computer Science 2001, Springer-Verlag, pages 73-130, 2001.
19. R.Valk. Petri Nets as Token Objects An Introduction to Elementary Object Nets. Proceedings of 19th International Conference on the Application and Theory of Petri Nets, LNCS 1420, Springer-Verlag, 1998.
20. INA:Integrated Net Analyzer, at http://www.informatik.huberlin.de/lehrstuehle/automaten/ina.