# PNTM Manual

WWY

September 4, 2011

This is a brief manual of using and modifying PNTM model.

## 1 Usage

The package includes two eclipse projects and two jar files. The PNK project is an IDE, and the PNRE project is a runtime.

To run the demo, please follow instructions below:

1. Build PNK and PNRE projects.
2. Open a terminal applications, move to `PNRE/test/`.
3. input `java -cp ../../crimson.jar:../../PNK/bin de/huberlin/informatik/pnk/appControl/ApplicationControl` to launch IDE.
4. input `java -cp ../../crimson.jar:../../PNK/bin cn/edu/tsinghua/cs/soft/pnml/compile/Main Test.java -net net.pnml` to compile `Test.java` with net file `net.pnml`.
5. input `java -cp ../../bcel-5.2.jar:../bin/:. Test` to run compiled byte code with virtual machine.

## 2 Current Design

The GUI part of PNTM environment is based on PNK, the compiler is a modified openJDK, and the runtime is based on DSTM2.

### 2.1 Editor

The IDE provides a simple GUI with a code editor and a net editor. The editor for Petri net system is modified from PNK. All elements can have extra fields and be edited visually. The extra fields such as resources in places and code in transitions all correspond special variables and functions in the code, as in Table 1.

### 2.2 Virtual Machine

The code editor can compile code along with Petri nets. Before compilation, the Petri nets are interpreted to internal representation for static check. In order to guarantee correctness at the level of Petri nets, the Petri nets must meet constraints below:

1. All resources must have different names.
2. There should always be no more than one resource with a certain name at one time.

Table 1: Extra fields in Petri nets' elements and corresponding elements in code

| code in transition | `petrinet` function name |
|---|---|
| resource in place | `resource` variable |
| inscription in arc | `resource` variable |

Table 2: Equivalent code to transformed code

| pre-defined interface | equivalent code |
|---|---|
| `@atomic interface AtomicInt {`<br>`int getValue ();`<br>`void setValue (int value);`<br>`}`<br>`Factory<AtomicInt> factory_AtomicInt =`<br>`Thread.makeFactory(AtomicInt.class);` | `AtomicInt a =`<br>`factory_AtomicInt.create();` |

3. The arcs to one transition should have no common resource.
4. The arcs from one transition should have no common resource. Besides, the resources in outgoing arcs must be the subset of resources in incoming arcs.

After checking the correctness of Petri nets, the editor will append a piece of code for building Petri Net simulator at runtime. Hence the simulator is created and initialized at runtime. The runtime provide 5 APIs as below:

1. `AddTransition(`transition`,` code`)` adds transition.
2. `AddPlace(`place`,` resource`)` adds place.
3. `AddArc(`source`,` target`,` inscription`)` adds arc.
4. `Start()` starts simulation.
5. `Join()` terminates simulation.

The runtime is a Petri nets VM using DSTM2. The first 3 API can build up a simulator of a Petri net and the last 2 API control the simulator. The simulator allocate a DSTM2 `Thread` for every transition in the net. The `Thread`s are all waiting for notification. Only notified `Thread` can consume resources, call function and produce new resources. A global lock is used to protect all resources in order to make manipulation on resources atomic and prevent dead-lock. Some optimizations accelerate the check-and-consume process. Hence the overhead is relatively low. When the transition consumes resources and is ready to fire, corresponding `petrinet` function is called using reflective mechanism. If all `global` variables protected by STM successfully commit, the transition will produce new resources. Otherwise the transition will revert all state and return consumed resources.

## 2.3 Compilation

At the early stage of compilation, the compiler will recognize and mark new keyword `petrinet`, `resource`, `global`. The `global` variables need to transform to instances of pre-defined interfaces in order to meet requirement of DSTM2's APIs. For example, The equivalent code to transformed ``` `global int a;` '' is shown in Table 2.

After AST is built, the compiler will check the semantic correctness. The functions with `petrinet` modifier and Petri nets themselves should meet constraints below:

1. `petrinet` function must have function body.
2. `petrinet` function should have no parameter.
3. Only `resource`, `global` and local variables can be used in a `petrinet` function.
4. Only `petrinet` function can be called in transition.
5. Resources in incoming arcs of a transition must be a superset of all `resource` variable used in corresponding `petrinet` function.

In addition, all references of `global` variables and all left-values consisted of `global` variables must be transformed to proper getter and setter in order to meet requirement of DSTM2's API. The equivalent code to getter and setter is shown in Table 3.

The rest of compilation is the same with openJDK.

Table 3: Equivalent code to getter and setter

| transform type | original code | equivalent code |
|---|---|---|
| initializer | `global int a = 0;` | `...;a.setValue(0);` |
| reference | x = a + 1; | x = a.getValue() + 1; |
| left-value | a = x + 1; | a.setValue(x + 1); |
| self-operation | a++; | a.setValue(a.getValue() + 1); |

# 3 Current Implementation

This section contains some details of implementation.

## 3.1 Virtual Machine

### 3.1.1 Runtime

Runtime can build and simulate a PNTM model at runtime. Several java classes describe the structure of PNTM net system.

1. `PetriNet` is the interface of PNTM and provides 5 APIs.
2. `Net` is the actual PNTM net system, consisting of lists of `Place`s, `Transition`s and `Arc`s.
3. `Node` is the abstract class of node in net system, which can derive two specific types of node.
   (1) `Transition` is the class of transitions in net system.
   (2) `Place` is the class of places in net system.
4. `Arc` is the class of arcs in net system.
5. `Resource` is the class of token sets and inscription sets in net system.
6. `TransactionManager` is the controller of one certain transaction attached to one transaction.

### 3.1.2 Software Transactional Memory

Some minor changes are applied to DSTM 2 to fit PNTM semantics. The Durability part of ACID property of DSTM 2 is eliminated. Thus when conflict occurs, one thread will abort right away. The `PriorityManager` and `shadow.Adapter` are used for Contention Manager and Adapter instead of others.

```
dstm2.manager.PriorityManager.resolveConflict(...)
dstm2.Thread.doIt(...)
```

For convenience of compilation process, the return type of Setters is changed.

```
dstm2.factory.AtomicFactory.createPrimitiveSetMethod(...)
```

## 3.2 Compilation

Compilation consists of 3 stages, and each stage does different transformation of code and net system. The API of javac is changed to integrate net system with java code.

### 3.2.1 Pre-Processor

The pre-processor is called before any kind of compilation. It analyzes net system, transforming it to easier-understandable format and insert additional code for virtual machine. Static checking of net system also occurs here.

### 3.2.2 Parser

Two tokens TRANSACT and PETRINET are added representing keyword `transact` and `petrinet`.

```
com.sun.tools.mod.javac.parser.Token
com.sun.tools.mod.javac.code.Flags
com.sun.tools.mod.javac.parser.Parser.modifiersOpt(...)
```

TRANSACT token will add TRANSACT flag to modifier, while PETRINET token will add both TRANSACT and PETRINET flags to modifier. TRANSACT flag means that variable need to be protected by transactional memory for rolling back, while PETRINET flag means that variable or function is connected to net system structures.

```
com.sun.tools.mod.javac.parser.Parser.variableDeclaratorRest(...)
com.sun.tools.mod.javac.parser.Parser.fakeTransactType(...)
com.sun.tools.mod.javac.parser.Parser.fakeTransactInit(...)
com.sun.tools.mod.javac.parser.Scanner.fakeName(...)
```

When parsing declaration of variable with TRANSACT flag, type and initializer should be transformed.

### 3.2.3 Compiler

Since javac applies visitor scheme, doing transformation on AST becomes difficult and more redundant laboring.

All references of TRANSACT variable should be replaced by a actual Getter function call, and all assign operation of TRANSACT variable should be replaced by series of Getter and Setter function calls. The compiler should detect whether one TRANSACT variable is only the reference or already transformed pre-defined interface. Some special cases should be handled carefully that the type of one TRANSACT variable should be the original reference's basic type.

```
com.sun.tools.mod.javac.comp.Attr.isTransact(...)
com.sun.tools.mod.javac.comp.Attr.needSetter(...)
com.sun.tools.mod.javac.comp.Attr.canonicalType(...)
com.sun.tools.mod.javac.comp.Attr.fakeGetter(...)
com.sun.tools.mod.javac.comp.Attr.fakeGetter(...)
```

Almost every visit methods in `Attr` should be modified to check if one component of tree structure should be transformed.

New scope property is added to check if one function or variable is TRANSACT. If the function declaration or definition violates the constraints of PNTM model, errors will be reported.

```
com.sun.tools.mod.javac.comp.AttrContext
```

# 4 Future Work