Typestate-Oriented Design

A Coloured Petri Net Approach

Jorge Luis Guevara Díaz

Department of Computer Science
Institute of Mathematics and Statistics
São Paulo University,São Paulo-Brazil
jorjasso@vision.ime.usp.br

Abstract

Typestate Oriented programming is a extension of the object oriented paradigm, where objects are modeled in terms of changing states. This paper propose the use of coloured petri nets as technique of design typestates for Typestate Oriented Programming.

Categories and Subject Descriptors D.3.2 [*Programming languages*]: Language Classifications—Object-oriented languages; D.2.2 [*Software Engineering*]: Design Tools and Techniques—Object-oriented design methods; D.2.2 [*Software Engineering*]: Design Tools and Techniques—Petri nets

General Terms Design, Object Oriented Programming, Petri Nets.

Keywords keyword1, keyword2

1. Introduction

Typestate-oriented programming [1] is a novel approach where objects are modeled in terms of changing states, this approach allows that each state may have its own representation and methods which may transition the object into a new state.

Petri nets are a powerful modelling technique for systems, Petri nets originate from the early work of Carl Adam Petri[8]. Coloured Petri nets [7] are a extension to basic petri nets that make them more useful for practical modelling.

The current design of the typestates[1] is made with simple state machines (Petri nets are more sophisticated state machines). This work propose the use of non-hierarchical petri nets to design typestates for typestate oriented programming.

This paper is organized as follows: Section 2 briefly reviews typestate, typestate for objects and typestate oriented programming concepts. Section 3 describe two petri nets concepts such place/transition nets and non hierarchical coloured petri nets, this section gives an formal definition of Petri nets and non-hierarchical coloured petri nets and gives an example to understand the concepts. Section 4 show how use non-hierarchical coloured petri nets to design typestates for typestate oriented programming, this section shows three examples: iterator, graphical user interface and Files.

MAC5714 '10 June 20, Sao Paulo, Brazil.

Reprinted from MAC5714 '10, [Unknown Proceedings], June 20, Sao Paulo, Brazil., pp. 1–6.

2. Typestate: A programming language concept

2.1 Typestates for Objects

Typestate [9] is a refinement of concept of type of the data object, while type of the data object determines the set of operations ever permitted on the object, typestate determine the subset of these operations which is permitted in particular context and captures the notion of an objects begin in appropriate or inappropriate state for the application of a particular operation.

Each type has an associated set of typestates. An object of a given type is at each point in a program in a single one of the typestates associated with its type. In each typestate, it is legal to apply some operations of the type, but not others.

The typestate transition is defined by a typestate **precondition**, which may hold in order for the operation to be applicable, and one or more typestate **postcondition** reflection the possible typestates of the operand after the operation is applied.

Objects change state over time. providing the programmer with logic for writing precondition, postcondition, and objects invariants quickly run into decidability problems [4]. Typestates capture aspects of the state of an object, when an objects state changes its typestate may change as well.

2.2 Typestate Oriented Programming

Typestate Oriented Programming [1] is a new paradigm that extend object-oriented programming with typestates. Plaid [10] is a new programming language designed to support Typestate oriented Programming (Figure 1). In this approach a typestate is like a class in that it has its own interface (a set of method signatures), representation (fields), and behavior (method implementations).

Characteristics of Typestate Oriented Programming:

- The programs are made up of dynamically created objects,
- Each object has a typestate that is changeable.
- Each typestate has an interface, representation, and behavior.

As an example consider the Figure 1. A File can have different states, for example closeFile or openFile. This approach is different from object oriented programming in the sense that an object has encapsulated all behavior. State of a file can change, for example a closedFile can be open (state of the file is now openFile) file, this natural behavior is sintactically represented in PLAID as $[ClosedFile >> OpenFile]$.

3. Petri Nets

3.1 Place/Transition Nets

Petri Nets also called Place/Transition Nets (PT-nets), are used for many different practical purposes, they have a graphical represen-

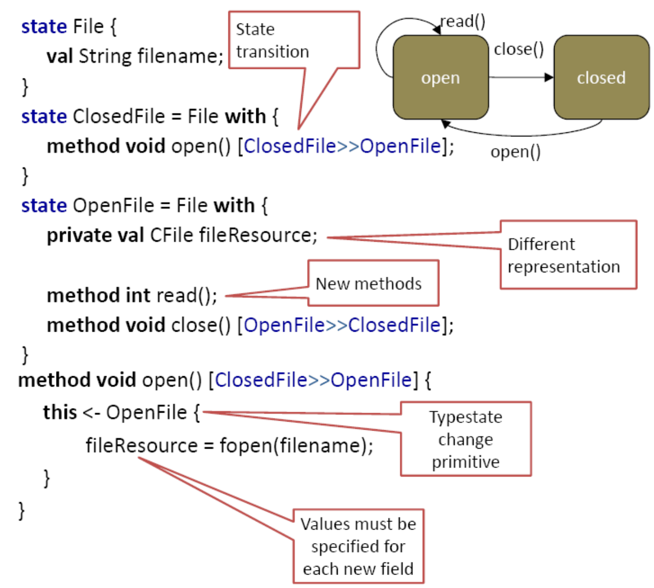

Figure 1. Example of Typestate Oriented Programming in PLAID.

tation and a well-defined semantics allowing formal analysis. They are useful for modelling concurrent, distributed, asynchronous behavior in a system.

A **net** is a bipartite graph $G(V, E)$ where:

- $V = P \cup T$, P is the set of **places** represented with circles and T is the set of **transitions** represented with vertical bars.

Definition A PT-net is a tuple $PTN = \{P, T, A, W, M_0\}$ satisfying the requirements below:

- P is a finite set of **places**
- T is a finite set of **transitions**
- $A \subseteq (P \times T) \cup (T \times P)$ is a finite set of **arcs**
- $W : A \longrightarrow \{1, 2, 3...\}$ is a weighting function
- $M_0 : P \longrightarrow \{1, 2, 3...\}$ is the initial marking
- $(P \cap T) = \phi$ and $(P \cup T) = \phi$

A transition $t_j \epsilon T$ is **enabled** if there is a token in each $p_i \epsilon P$ that has and edge to the transition. An enabled transition may or may not **occur** (Figure 2).

3.2 Non-hierarchical CP-net

A PT-net has no types and no modules, with Coloured Petri Nets (CP-nets) it is possible to use data types and complex manipulation. CP-nets are a extension of Petri Nets where each token has attached a data value called the token colour, the token colours can be manipulated by the occurring transitions. With CP-nets it is possible to make hierarchical descriptions (hierarchical CP-nets) for example a large model can be obtained by combining a set of sub-models. Hierarchical CPN-nets allow well-defined interfaces between sub-models, well-defined semantics of the combined model and also sub-models can be reused.

3.3 Definition of non-hierarchical CP-nets: [7]

Definition A non-hierarchical CP-net is a tuple
$CPN = \{\Sigma, P, T, A, N, C, G, E, I\}$ satisfying the requirements below:

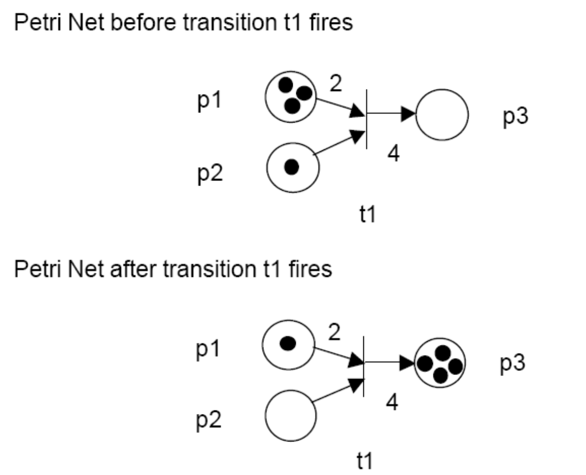

Figure 2. Place/Transition Nets.

1. Σ is a finite set of non-empty types, called **colour set**.
2. P is a finite set of **places**.
3. T is a finite set of **transitions** .
4. A is a finite set of **arcs** such that:
   $P \cap T = P \cap A = T \cap A = \phi$.
5. N is a **node** function. It is defined from A into $P \times T \cup T \times P$.
6. C is a **colour** function. It is defined from P into Σ.
7. G is a **guard** function. It is defined from T into expressions such that:
   $\forall t \epsilon T : [Type(G(t)) = Boolean \wedge Type(Var(G(t))) \subseteq \Sigma]$
8. E is an **arc expression** function, It is defined from A into expressions such that
   $\forall a \epsilon A : [Type(E(a)) = C(p(a))_{Bag} \wedge Type(Var(E(a))) \subseteq \Sigma]$, where $p(a)$ is the place of $N(a)$
9. I is the **initialization** function. It is defined from P into closed expressions such that:
   $\forall p \epsilon P : [Type(I(p)) = C(p)_{Bag}]$

As a example consider a CPN-net from Figure 3. The **colour sets** determines the types. operations and functions that can be used in the net inscriptions i.e., arc expressions, guards, initialization expressions, etc. In the example:

- $\Sigma = \{U, I, P, E\}$.

The **places**, **transitions** and **arcs** are described by three sets P, T and A, In the example:

- $P = \{A, B, C, D, E, R, S, T\}$
- $T = \{T1, T2, T3, T4, T5\}$,
- $A = \{(A, T1), (T1, B), (B, T2), (T2, C), (C, T3), (T3, D), (D, T4), (T4, E), (E, T5), (T5, A), (T5, B), (R, T1), (S, T1), (S, T2), (T, T3), (T, T4), (T3, R), (T5, S), (T5, T)\}$.

The **node** function maps each arc into a pair where the first element is the source node and the second the destination node. In the example:

- N((A,T1))=(source, dest).
- N((T1,B))=(dest, source).

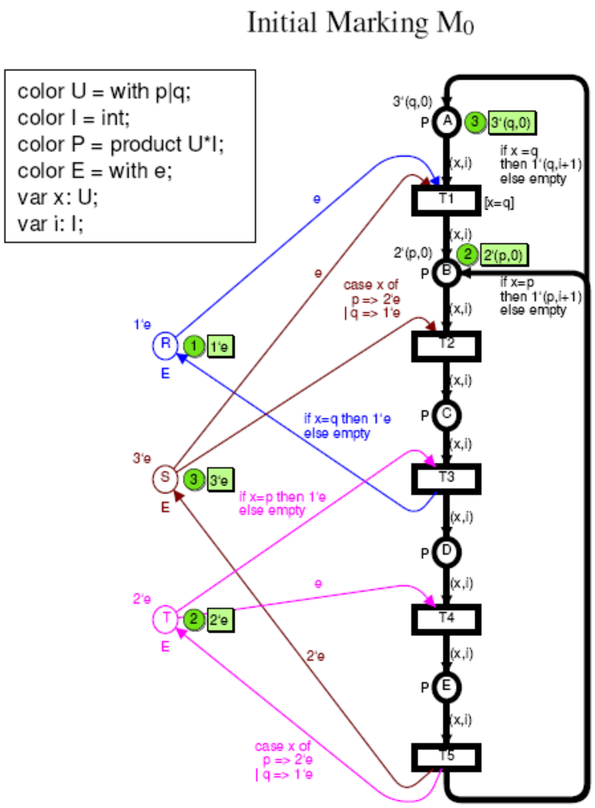

Figure 3. CPN-net.

The **colour** function C maps each place p, to a colour set $C(p)$. In the example:

- if $p\epsilon\{A,B,C,D,E\}$ then $C(p)=P$, otherwiese $C(p)=E$.

The **guard** function G maps each transition t to an expression of type boolean ($Type(G(t)) = Boolean$) i.e, a predicate. Moreover, all variable in $G(t)$ must have types that belongs to Σ, (that is to say $Type(Var(G(t)))$. In the example:

- $G(t) = [x = q]$ if $t = T1$, otherwiese $G(t) = true$.

The **arc expression** function E maps each arc into an expression which must be of the type $C(p(a))_{Bag}$. (where $p(a)$ is the place of $N(a)$ and $C(p(a))_{Bag}$ is a bag). Moreover, all variable in $E(a)$ must have types that belongs to Σ, (that is to say $Type(Var(E(a)))$. In the example:

- $E(a) = 2$ if $a\epsilon\{(R,T1),(S,T1),(T,T4)\}$.
- $E(a) = 2'e$ if $a = (T5,S)$.
- $E(a) =$ case x of $p \Rightarrow 2'e|q \Rightarrow 1'e$ if $a\epsilon\{(S,T2),(T5,T)\}$
- $E(a) =$ if $x = q$ then $1'e$ else empty if $a = (T3,R)$
- $E(a) =$ if $x = p$ then $1'e$ else empty if $a = (T,T3)$
- $E(a) =$ if $x = q$ then $1'(q,i+1)$ else empty if $a = (T5,A)$
- $E(a) =$ if $x = p$ then $1'(p,i+1)$ else empty if $a = (T5,B)$
- $E(a) = (x,i)$ otherwiese

The **initialization** function I maps each place p into a expression which must be of type $C(p)_{Bag}$. In the example:

- $I(p) = 3'(q,0)$ if $p = A$

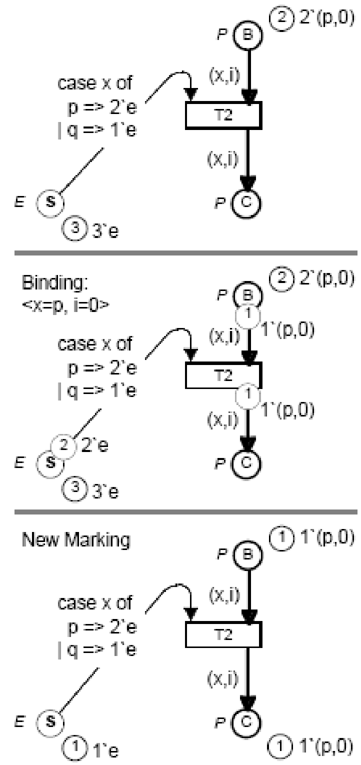

Figure 4. Ocurrence of enabled binding.

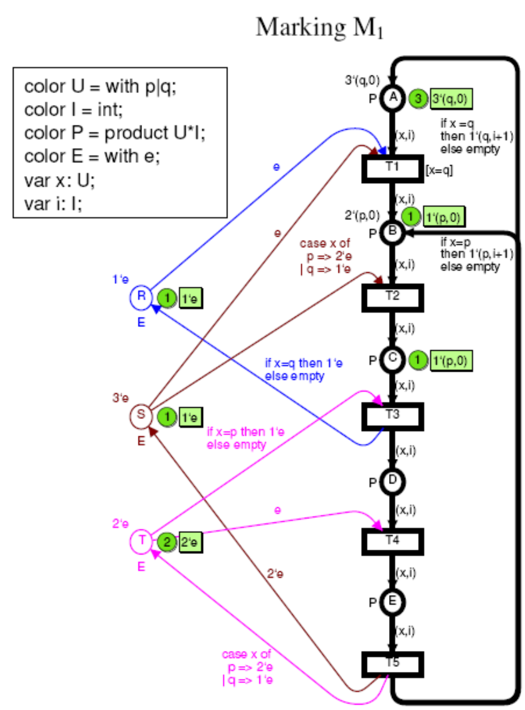

Figure 5. CPN-net.

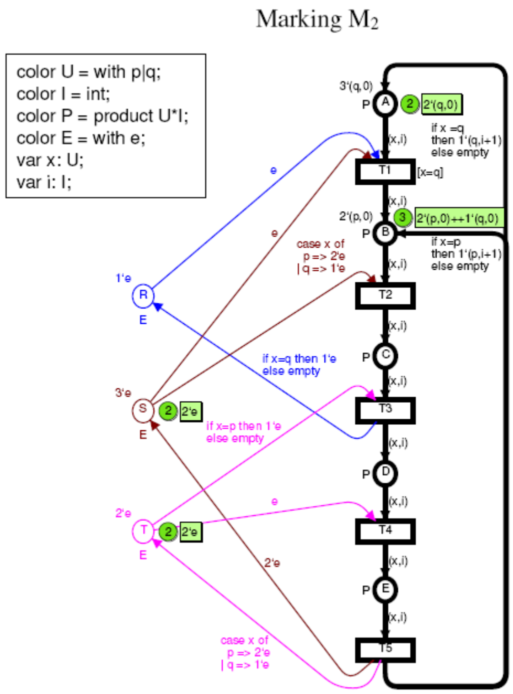

Figure 6. CPN-net.

- $I(p) = 2'(p,0)$ if $p = B$
- $I(p) = 1'e$ if $p = R$
- $I(p) = 3'e$ if $p = S$
- $I(p) = 2'e$ if $p = T$
- $I(p) = \phi$ otherwiese

A **token** element is a pair (p, c) where $p \epsilon P$ and $c \epsilon C(p)$. In the example are three $(q, 0) - tokens$ (of colour P) on A, two $(p, 0) - tokens$ (of colour P) on B, the places C, D and E have no tokens, R has one $e - token$, S has three $e - tokens$, T has two $e - tokens$ (of colour E).

A **marking** is a distribution of tokens on the places. The **initial marking** M_0 (Figure 3) is obtained by evaluating the **initial expressions** (the underlined expressions next to the place). The marking of the each place is a bag. In the example:

- $M_0 = 3'(A, (q, 0)) + 2'(B, (p, 0)) + 1'(R, e) + 3'(S, e) + 2'(T, e)$:

For each **binding** (i.e $b_1 =< x = p, i = 0 >$, for the transition $T2$) we can check whether the transition is **enabled** in the current marking. When a transition is enabled for a certain binding, this transition may **occur** (Figure 4), and it then remove tokens from its input places and add tokens to its output place In the example:

- if $b_1 =< x = p, i = 0 >$ then the **binding element** $(T2, b_1)$ is enabled in the initial M_0 and transform the marking M_0 into the marking M_1 (Figura 5).
- if $b_1 =< x = q, i = 0 >$ then the **binding element** $(T2, b_1)$ is enabled in the initial M_0 and transform the marking M_0 into the marking M_2.(Figura 6).
- if $b_2 =< x = q, i = 100 >$ then the **binding element** $(T2, b_2)$ is not enabled in the initial M_0

The markings M_0 and M_2 is **directly reachable** from M_0 (see the figure).

Its possible that two transitions in a CP-net can be **concurrently enabled**. In the example:

- $1'T1, < x = q, i = 0 > +1'(T2, < x = p, i = 0 >)$

3.4 Benefits and advantages of CP-nets

Some benefits [7]) of CP-nets are:

- Description and analysis become more compact and manageable , the complexity is divided between the net structure, the declarations and the net inscriptions.
- Is possible describe data manipulations in a direct way using the arc expressions.
- It becomes easier to see the similarities and diferences between similar systems parts
- It is possible to create hierarchical descriptions

Some advantages [7]) of CP-nets are:

- Graphical representation
- Well-defined semantics
- Generality
- Few, but powerfull primitives
- Explicit description of states and actions
- Hierarchical descriptions
- Computer tools supporting their drawing, simulation and analysis.

4. Typestate-Oriented Design: Coloured Petri Nets

This section, shows the use of CPN-nets as design tool for typestate programming. The procedure for using CPN-nets is as follows:

- The places are the typestates.
- The transitions are the methods that makes typestate change.
- The colour set determine the types handled by the CPN-net.
- The guards specifies some conditions for change the typestates.
- The arcs determine the data values.
- The tokens carries the data value that belongs to the type associated with the place.
- The initial marking is the starting point for the CPN-net operation.

The CPN-net approach (Figure 10, Figure 14, Figure 15) is more concise than the state machine described (Figure 9, Figure 13).

I show the use of a CPN-net through a series of examples .(all examples was taken from the article *Typestate-oriented programming* [1]). Examples were modeled with CPNTOOLS [11] software.

4.1 Example:Iterators

This example shows a iterator. The Iterator state has two states: avail and end. Figure 7 shows the source code in Plaid. Figure 8 shows how a client may use an iterator in Plaid. Figure 9 shows a iterator state machine. The figure 10 shows the CPN-net approach.

- Places. (Avail, End)
- Transitions. (next())

```
state Iterator {
    conserved type TElem;
    final immutable Collection<TElem> coll;
}

state Avail extends Iterator {
    TElem next() [Avail >> (Avail || End)];
}

state End extends Iterator {
}
```

Figure 7. Iterators in Plaid.

```
Collection<String> c = ...
Iterator<String> i = c.iterator();

while(i instate Avail) {
  String o = i.next();
}
```

Figure 8. Iterator client code in Plaid.

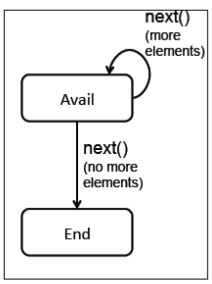

Figure 9. Iterator State Machine.

- Colour set. (Element)
- Guards. ($n > 1, n = 1$)
- Arcs. ($empty, e$)
- Tokens. (ne)
- Initial marking. ($10'e$)

4.2 Example:Graphical Interface

This example shows a graphical interface with two states: idle and runing. Figure 11 shows the source code in Plaid. Figure 12 shows how a client may use an iterator in Plaid. Figure 13 shows an iterator state machine.

The figure 13 shows the CPN-net approach.

- Places. (Running, Idle)
- Transitions. (start() and stop())
- Colour set. (Element)
- Guards. (null)
- Arcs. (e)
- Tokens. ($1e$)
- Initial marking. ($1'e$)

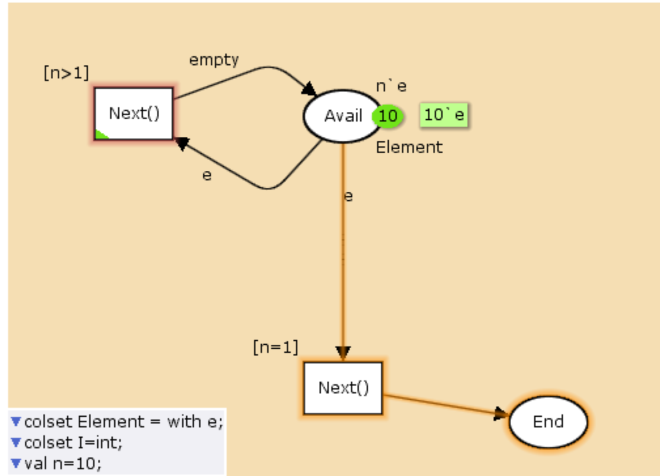

Figure 10. Iterator with CPN-net.

```
state Idle {
  void start() [Idle >> Running];
}
state Running {
  void stop() [Running >> Idle];
  void run(InputEvent e);
}
```

Figure 11. Graphical interface code in Plaid.

```
state MoveIdle extends Idle {
  GraphicalObject go;
  void start() [Idle >> Running] {
    this <- Running {
      void run(InputEvent e) {
        go.move(e.x,e.y);
      }
      void stop() [Running >> Idle] {
        this <- MoveIdle{}
      }
    }
  }
}
```

Figure 12. Graphical interface client code in Plaid.

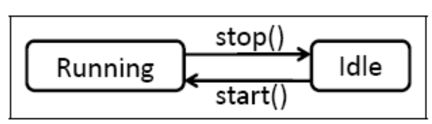

Figure 13. Graphical interface state machine.

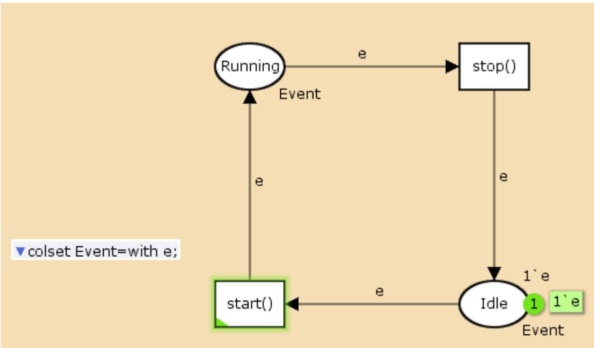

Figure 14. Graphical interface with CPN-net.

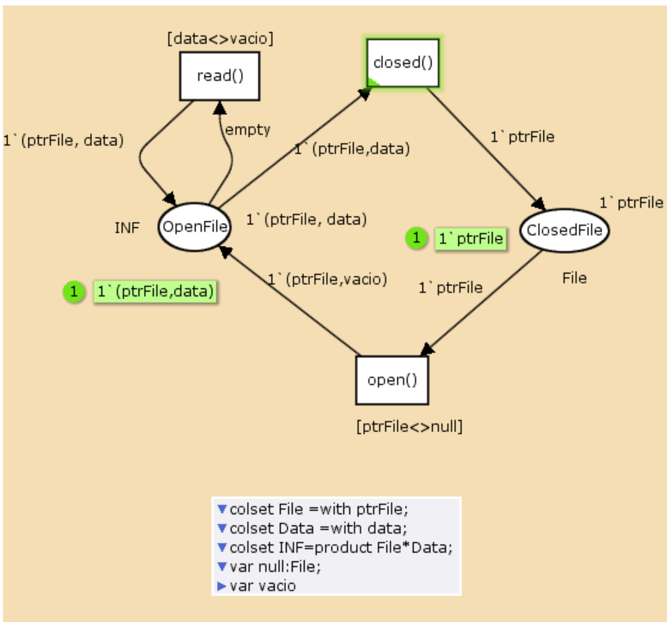

Figure 15. Files with CPN-net.

4.3 Example:Files

This example shows a File with two states: OpenFile and ClosedFile. Figure 1 shows the source code in Plaid.

The figure 15 shows the CPN-net approach.

- Places. (OpenFile, ClosedFile)
- Transitions. (read(), open() and closed())
- Colour set. (File, Data, INF)
- Guards. $(ptrFile <> null, data <> vacio)$
- Arcs. $(1ptrFile)$
- Tokens. $(1ptrFile, 1'(ptrFile, data))$
- Initial marking. $(1ptrFile + 1'(ptrFile, data))$

5. Conclusions

CPN-nets are a powerful modelling technique that have a graphical representation which allows a concise modelling of typestates, also CPN-nets have a well-defined semantics which unambiguously defines the behavior of each CPN-net. CPN-nets are very general and can be used to describe a range from informal systems to formal systems. For this reason, CPN-nets are suitable for modelling typestates. CPN-nets have a number of formal analysis methods (not described in this paper) by which properties of CPN-nets can be proved.

Typestate programming is a new programming paradigm that is actually developed in Carnegie Mellon University. Plaid (currently developing) is a language for Typestate programming. I show that CPN-nets can be used in successfully manner for design typestates.

References

[1] Aldrich, J., Sunshine, J., Saini, D., and Sparks, Z. 2009. Typestate-oriented programming. In Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (Orlando, Florida, USA, October 25 - 29, 2009). OOPSLA '09. ACM, New York, NY, 1015-1022. DOI= http://doi.acm.org/10.1145/1639950.1640073

[2] Bierhoff, K. and Aldrich, J. 2005. Lightweight object specification with typestates. In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Lisbon, Portugal, September 05 - 09, 2005). ESEC/FSE-13. ACM, New York, NY, 217-226. DOI= http://doi.acm.org/10.1145/1081706.1081741

[3] Bierhoff, K. and Aldrich, J. 2007. Modular typestate checking of aliased objects. In Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (Montreal, Quebec, Canada, October 21 - 25, 2007). OOPSLA '07. ACM, New York, NY, 301-320. DOI= http://doi.acm.org/10.1145/1297027.1297050

[4] Robert Deline and Manuel Fahndrich. Typestates for Objects. In Proc. European Conference on Object-Oriented Programming, 2004.

[5] Janneck, J. W. and Esser, R. 2002. Higher-order petri net modelling: techniques and applications. In Proceedings of the Conference on Application and theory of Petri Nets: Formal Methods in Software Engineering and Defence Systems - Volume 12 (Adelaide, Australia). C. Lakos, R. Esser, L. M. Kristensen, and J. Billington, Eds. ACM International Conference Proceeding Series, vol. 145. Australian Computer Society, Darlinghurst, Australia, 17-25.

[6] Pankratius, V. and Stucky, W. 2005. A formal foundation for workflow composition, workflow view definition, and workflow normalization based on petri nets. In Proceedings of the 2nd Asia-Pacific Conference on Conceptual Modelling - Volume 43 (Newcastle, New South Wales, Australia). S. Hartmann and M. Stumptner, Eds. Conferences in Research and Practice in Information Technology Series, vol. 107. Australian Computer Society, Darlinghurst, Australia, 79-88.

[7] K. Jensen: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997. ISBN: 3-540-60943-1.

[8] C. A. Petri. Kommunikation mit Automaten. Schriften des IIM nr. 2, Institut fur Instrumentelle Mathematik, Bonn, 1962.

[9] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng., 12(1), 1986.

[10] http://www.cs.cmu.edu/ aldrich/plaid/

[11] http://wiki.daimi.au.dk/cpntools-help/