A Net-based Object Behaviour Inheritance Modeling Methodology[1]

Shengyuan Wang Jian Yu Chongyi Yuan

Abstract: To make a smooth change under a consistent conceptual framework from one developing stage to the next stage, multi-tier methodology is quite required. We present an object behaviour inheritance modeling methodology based on Petri Nets, in which two net models, STLEN and DCOPN, serve for modeling the behavior of concurrent objects in two separate tiers.

Keywords: Concurrency, Object Orientation, Petri Net, Inheritance, Modeling Methodology.

Extended Abstract

The integration of Petri Nets with object orientation techniques has become promising ([1]-[8]). Parallelism, concurrency and synchronization are easy to model in terms of Petri Nets, and many techniques and software tools are available for the analysis of Petri Nets. These advantages have made Petri nets pretty suitable to model the dynamic behavior of concurrent objects.

A concurrent object-oriented system consists of a dynamically varying configuration of concurrent objects operating in parallel. For each stage in the software development of such sort of systems, diversified Petri Net models may be found in the literature to perform the specification and/or verification of the system behavior respecting that stage. Some net models are used during the early steps of software development ([5][7][8]), and others during the later steps ([1][3][4][6]). However until now, there is a lack of net-based formal methods that can make a smooth change under a consistent conceptual framework from one developing stage to the next stage. This rupture prevents net-based methods from being more widely applied into the modeling of a concurrent object-oriented system, because incrementally developing is one of principles in object-oriented methodology. So we persist the opinion that multi-tier methodology is quite required. Each tier is a significant specification phase in which at least one net-based model may be chosen as the modeling language. A multi-tier methodology should guarantee the system behavior specified in one tier to be preserved in its successor tier, though the specification in the successor tier is usually more detailed.

A practical multi-tier methodology has to concern primary elements of concurrent object-orientation, such as object (class) representation, inheritance, aggregation, cooperation (association), concurrency (intra/inter objects), etc. In this paper, we present a multi-tier inheritance modeling methodology, and two Petri Net models, belonging to two tiers respectively, are invented to illustrate a two-tier methodology. The net model in the super-tier is a modified EN-system, called STLEN, in which both S-elements and T-elements are labeled. And in the successor tier (or sub-tier) is a net model called DCOPN (dynamically configured object net), which has a flavor of concurrent object-oriented programming languages, like net models in [1], [3], [4].

Formally, A *ST-Labeled EN system*, abbreviated as STLEN system, is a tuple $\Sigma = (N, \beta)$, where

(1) $N=(B, E; F, c_{in})$ is a EN system. B and E are the *set of S-elements* and the *set of T-elements* respectively, $F \subseteq S \times T \cup T \times S$ is the *flow relation*, and $c_{in} \subseteq B$ is the *initial case*.

(2) $\beta: B \cup E \to L \cup \{\lambda\}$ is a labeling function such that

$$\forall b \in B, \forall e \in E[\beta(b) \neq \lambda \ \vee \ \beta(e) \neq \lambda \to \beta(b) \neq \beta(e)].$$

[1] Supported by the National Natural Science Foundation of China under grant No. 69973003, and by the China NKBRSF (973) under grant G1999032706.

(3) $\lambda \notin \beta(c_{in})$.

Where L is the set of identifiers that range over a name space, and $\lambda \notin L$ denotes the "unobservable" S-elements or T-elements. β is generally not an injection. Several observable S-elements may be labeled with a single name (identifier), and the same is true for T-elements. For $x \in B \cup E$, x is observable iff $\beta(x) \neq \lambda$. From the definition above, we have $(\beta(B)-\{\lambda\}) \cap (\beta(E)-\{\lambda\}) = \phi$.

For the definition of dynamic behaviors of a STLEN system, one may refer to [9]. (Examples of STLEN systems are given in the Appendix A.)

The formal definition of DCOPN (Dynamically Configured Object Petri Nets) is a little tedious (the details can be found in [10]). (A modified dining philosophers problem may be used to explain the main characteristics of the net model DCOPN, which is presented in the Appendix B.)

Assume DCOPN being the subsequent net model of STLEN in our multi-tier methodology. We hope the (dynamic) behavior specified in one tier to be maintained in its successor tier. Verifying some bisimulation relations between the two tiers usually does this. The bisimulation relations are established by the methodology. For example, we define a bisimulation relation between the STLEN model and the DCOPN model by observing the state changing after the occurrence of associated actions. To establish the bisimulatiom, there should exist a map from object models in the super-tier to those in the sub-tier. The object model *S* in the STLEN tier is mapped to the object model *S'* in the DCOPN tier, such that $S \cong S'$, where $\cong$ denotes the bisimulation relation.

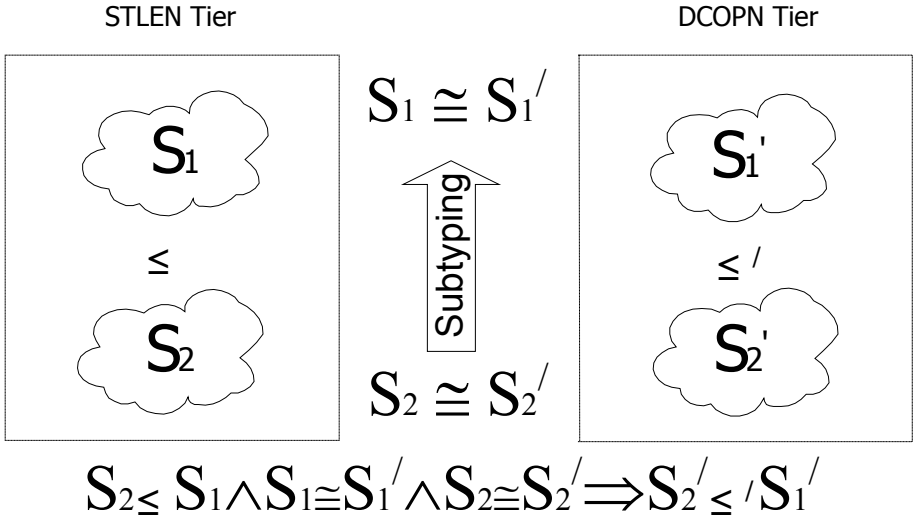
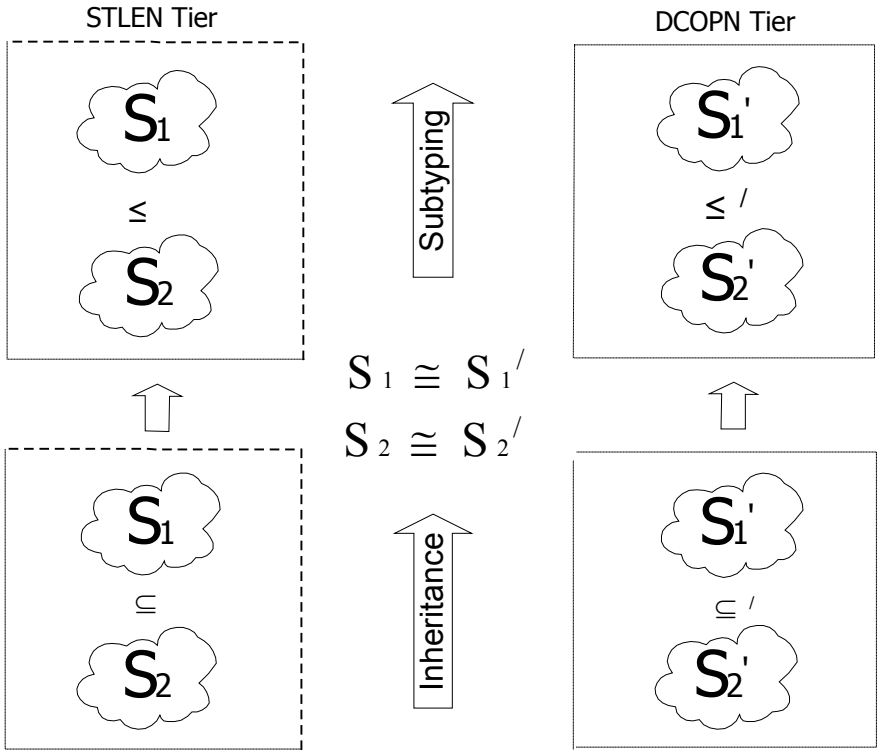


Figure 1

The term *inheritance* has often twofold meanings in the literature: the "code" reuse and the behavior preserving. In many places and also in this paper, the later is the explanation for another term *subtyping*, and the term *inheritance* is only standing for the former. In the situation of sequential object orientation, we used to not distinguishing between *inheritance* and *subtyping*, but it is very helpful to distinguish them in the issues of concurrent object orientation, for example, in the comprehension of *inheritance anomaly*[12].

In our multi-tier inheritance modeling methodology, we suppose that the tier map from the super-tier to the sub-tier can preserve the subtyping relations between net-based object models. This can be illustrated by Figure 1, in which S_1 is mapping to S_1' (there is a bisimulation relation between them), and so is for S_2 to S_2', then the subtyping relation $\leqslant$ between S_2 and S_1 can be preserved to the subtyping relation $\leqslant'$ between S_2' and S_1'.

One of choices for the subtyping relation between STLCN Nets is the one in [9]. The suitable subtyping relations between DCOPN Nets will be discussed in a separate paper, where a weakest requirement defined in the definition 3.2 of [10] have to be satisfied, in which the *contracovariant rule* (for both accept ports and attribute predicates) and the *covariant rule* (for result ports) are assumed (to refer to [13] for these two rules).



$$\text{IF} \quad S_{2}{}' \subseteq {}' S_{1}{}' \wedge S_{1} \cong S_{1}{}' \wedge S_{2} \cong S_{2}{}' \Rightarrow S_{2} \subseteq S_{1} \quad \text{THEN}$$

$$(S_{2} \subseteq S_{1} \rightarrow S_{2} \leq S_{1}) \wedge S_{1} \cong S_{1}{}' \wedge S_{2} \cong S_{2}{}' \Rightarrow (S_{2}{}' \subseteq {}' S_{1}{}' \rightarrow S_{2}{}' \leq {}' S_{1}{}')$$

Figure 2

Subtyping is a behavior preserving relation. Instead, inheritance is used for the code/specification reuse. Practically, to implement the behavior preserving while the code/specification is also highly reused, some incremental inheritance paradigms (capable of implementing the anticipant subtyping relations) need to be developed [8], the more the better. In our multi-tier inheritance modeling methodology, the incremental inheritance capable of implementing the anticipant subtyping relation is required to be reserved by the tier map from the super-tier to the sub-tier, i.e. the THEN part in Figure 2 holds. One of the possibilities to obtain this is to ensure the IF part in Figure 2 to be satisfied on the methodology.

The modeling methodology in this paper is outlined by the following steps:

(1) With the help of any OOA/OOD methodology, develop the behavior model of objects/classes using the net language STLEN and its available tools (in developing). Label both the states (places) and actions (transitions) , and group states on the attributes.

(2) Develop DCOPN nets for STLEN ones by adding details, such as data types, constants, and attribute predicates. Build a map between the STLEN tier and the DCOPN tier. Verify the bisimulation relations between them (by emerging tools). Fulfil the interface specification for each DCOPN net.

(3) For the behaviour inheritance (subtyping) modeling, just consider the deriving in the STLEN tier. Then develop the DCOPN net from the derived STLEN one according to (2). Don't forget that the interface specifications for each super DCOPN class net and its derived DCOPN class net

(developed from STLEN one) have to satisfy some rules (contracovariant / covariant).

(4) Use incremental inheritance paradigms as many as possible. This may substantively save the work. (as illustrated in Figure 2).

Changes in the developing process are allowable, which is simplified for the direct corrections in the DCOPN tier being avoided. This is one of the advantages of the modeling methodology in the paper.

Many aspects for our multi-tier methodology still need to be explored. Researchers who are interested in it may find many new topics about that.

Reference

[1] Charles Lakos. From Coloured Petri Nets to Object Petri Nets. Proceedings of 16th Int. Conference on the Application and Theory of Petri Nets, LNCS 935, Turin, Italy, Springer-Verlag,1995.

[2] E.Batiston, A,Chizzoni, Fiorella De Cindo. Inheritance and Concurrency in CLOWN. Proceedings of the "Application and Theory of Petri Net 1995" workshop on "object-oriented programs and models concurrency", Torino, Italy 1995.

[3] C.Sibertin-Blanc. Cooperative Nets. Proceedings of 15th Int. Conference on the Application and Theory of Petri Nets. LNCS 815, Zaragoza, Spain, Springer-Verlag, 1994.

[4] D.Buchs and N.Guelfi. CO-OPN: A Concurrent Object Oriented Petri Net Approach. Proceedings of 12th Int. Conference on the Application and Theory of Petri Nets, LNCS 524, Gjern, Denmark, 1991.

[5] R.Valk. Petri Nets as Token Objects — An Introduction to Elementary Object Nets. Proceedings of 19th Int. Conference on the Application and Theory of Petri Nets, LNCS 1420, Springer-Verlag, 1998..

[6] U.Becker and D.Moldt. Object-oriented Concepts for Coloured Petri Nets. Proceedings of IEEE Int. Conference on System, Man and Cybernetics, vol. 3, 1993, pp 279-286.

[7] A.Newman, S.M.Shatz, and X.Xie. An Approach to Object System Modeling by State-Based Object Petri Nets. Journal of Circuits, Systems and Computers, Vol.8, No.1(1998) 1-20.

[8] W.M.P. Vander Aalst and J.Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. 18th Int. Conference on the Application and Theory of Petri Nets, LNCS1248, Toulouse, France, Springer-Verlag, 1997.

[9] S.Y.Wang, J.Yu and C.Y.Yuan. A Pragmatic Behavior Subtyping Relation Based on Both States and Actions. To appear in Journal of Computer Science and Technology.

[10] S.Y.Wang, J.Yu and C.Y.Yuan. Modeling Concurrent Object-Oriented Systems with Dynamically Configured Object Petri Nets. Submitted to 22nd International Conference on Application and Theory of Petri Nets (ICATPN 2001). The proceedings will be published by Springer-Verlag in Lecture Notes in Computer Science.

[11] S.Christensen, N.D.Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. Proceedings of 14th Int. Conference on the Application and Theory of Petri Nets, LNCS 691, Chicago, USA, Springer-Verlag, 1993.

[12] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-oriented Concurrent Programming Languages. In Reserch Directions in Concurrent Object-oriented Programming, edited by G.Agha, P.Wegner and A.Yonezawa, The MIT Press, pp.107-150, 1993.

[13] P.America. Designing an Object-oriented Programming Language with Behavioural Subtyping. In Proc. of REX School/Workshop on Foundations of Object-oriented Languages(REX/FOOL), Noordwijkerhout, the Netherlands, May, 1990, LNCS 489, 60-90, Springer-verlag, 1991.

Appendix A. Examples of STLEN systems.

Figure A-1 (a) is an EN system that is the behavior specification of a bounded buffer. The state of a buffer is described with three S-elements "empty", "partial" and "full" (the buffer size is assumed to be greater than 2). The T-elements "p1", "p2" and "p3" are used to represent the possible "put" actions in different cases of the buffer state, and the T-elements "g1", "g2" and "g3" are related to "get" actions in the same meaning.

In building an object model for a bounded buffer, it is natural to use an attribute group named "state" to

hold the current state of the buffer, and to let it have two public methods “put” and “get”. Then we have got an analysis object model for a bounded buffer object (or class), depicted in figure A-3(a).

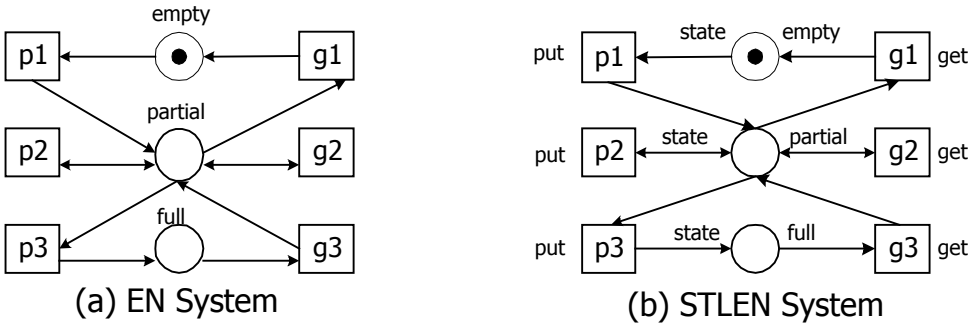


Figure A-1 Bounded Buffer

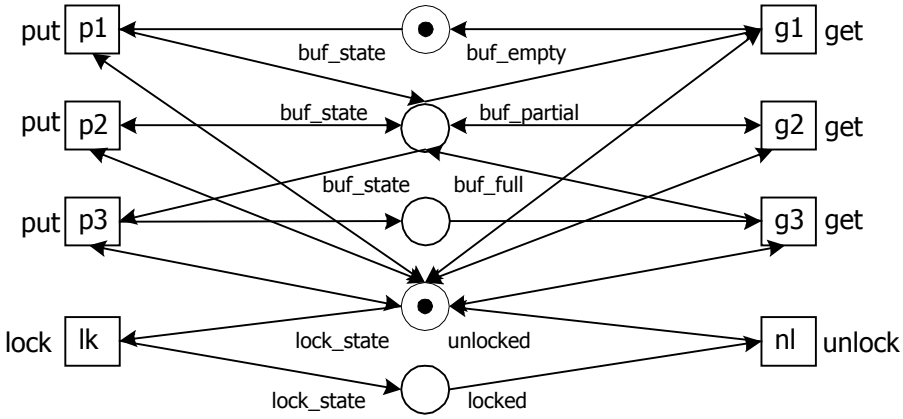


Figure A-2 Lockable Bounded Buffer

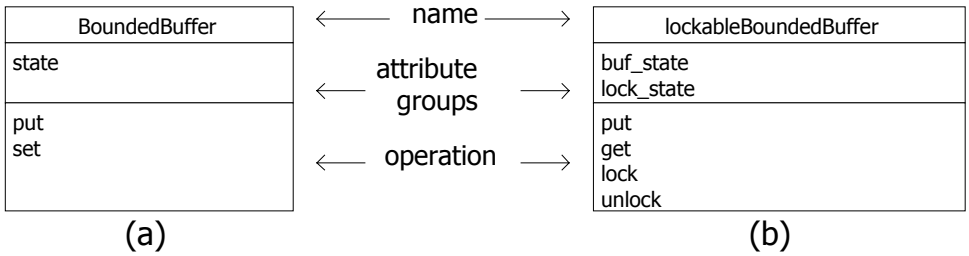


Figure A-3 Analysis Object Models for two Classes of Buffer

Figure A-1(b) is a ST-Labeled EN system whose underlying EN system is the one in figure A-1(a). In this STLEN system, the labeling function is defined by β(“p1”) = β(“p2”) = β(“p3”) = “put”, β(“g1”) = β(“g2”) = β(“g3”) = “get”, and β(“empty”) = β(“partial”) = β(“full”) = “state”.

In diagrams of STLEN systems, the label of a T-element is drawn just outside of the box, and the label of an S-element is depicted closely on the left of the circle, and the label λ is absent.

Associating the STLEN system in figure A-1(b) with the analysis object model in figure A-3(a), the latter is a static object model for a bounded buffer object (or class), and the former is the dynamic object model which can serve as a specification of its dynamic behavior.

Figure A-2 is another example of STLEN systems. A lockable bounded buffer is a bounded buffer that has one more attribute group about the state of its “lock”, and two more methods, which can change the state of this attribute. Figure A-3(b) is the associate (static) analysis object model of a lockable bounded buffer object (or class).

Appendix B. A modified dining philosophers problem specified by the net model DCOPN

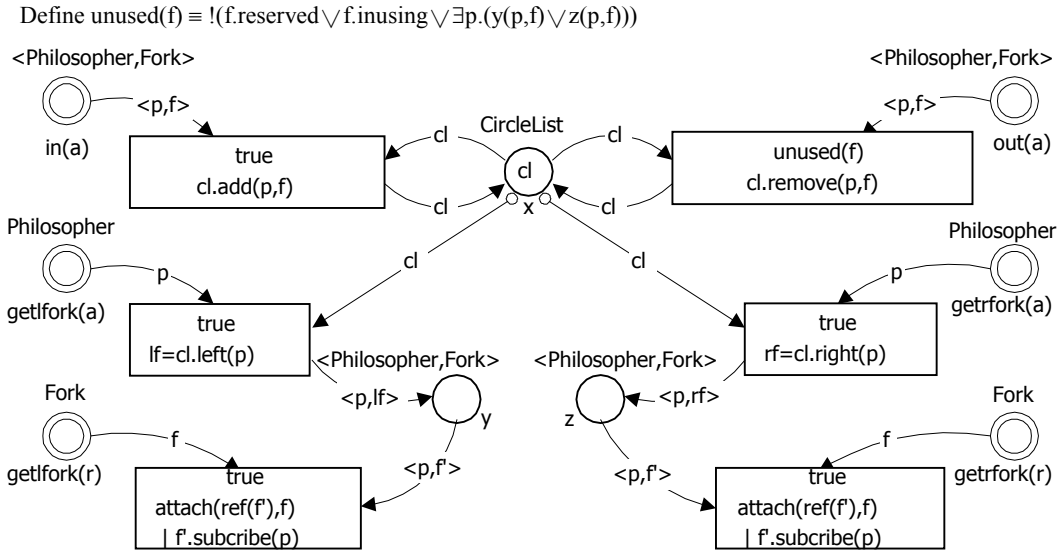


Figure B-1 Class Net of TableManager

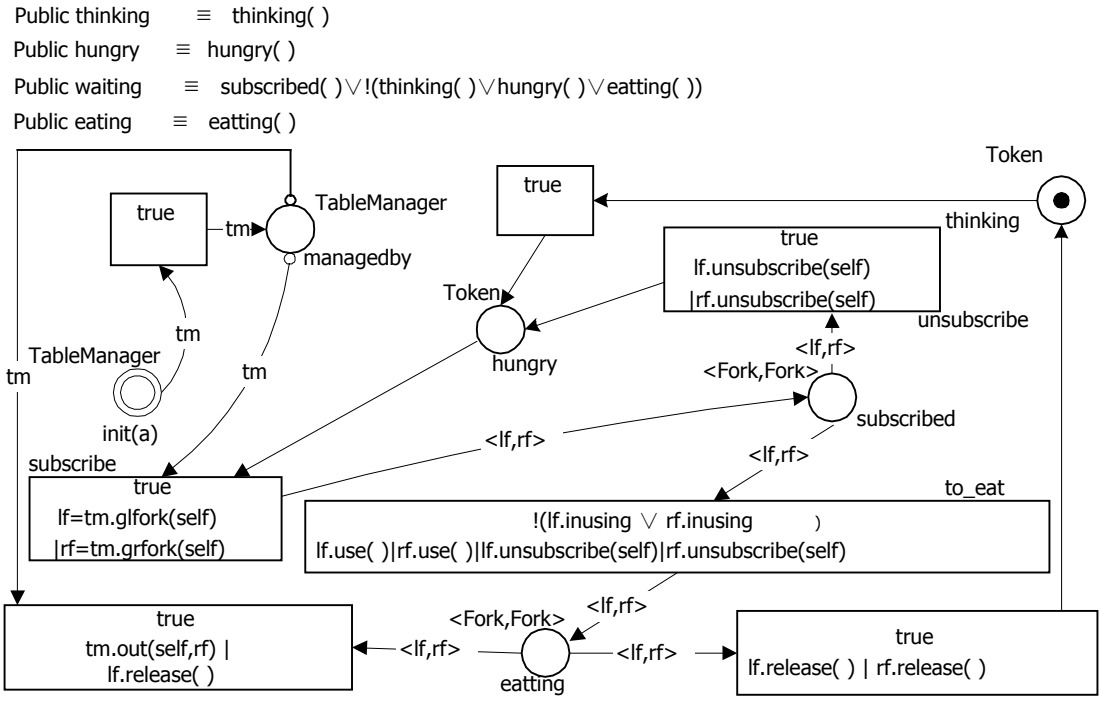


Figure B-2 Class Net of Philosopher

Imagine a big enough table which don't limit the number of dining philosophers. Anytime philosophers can dynamically join in or leave, which is administrated by a table manager. When a philosopher joins in, a new fork is assigned to him, and the table manager serves the philosopher as well as the fork certain places around the table. Then the philosopher begins thinking some thing. When he become hungry, he informs the table manager to get his left hand and right hand forks (both are his neighbor forks) and to subscribe them at the same time. When the forks have been subscribed, he may wait these forks being free, or unsubscribe them to leave them not subscribed by him. When both of the forks he has subscribed are not in using, he can begin eating. After eating, he may release these two forks to the table manager and begins thinking again, or leave the table by releasing the left hand fork and taking the right hand fork away. But he can't leave

whenever his right hand fork is being subscribed or reserved or in the process to be subscribed.

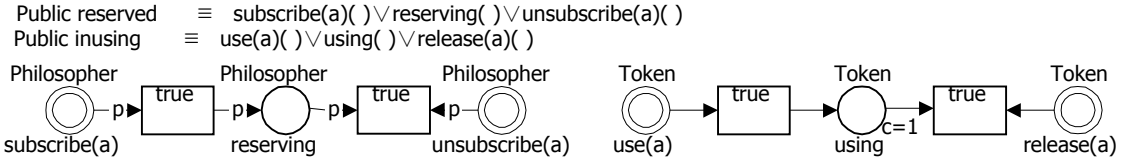


Figure B-3 Class Net of Fork

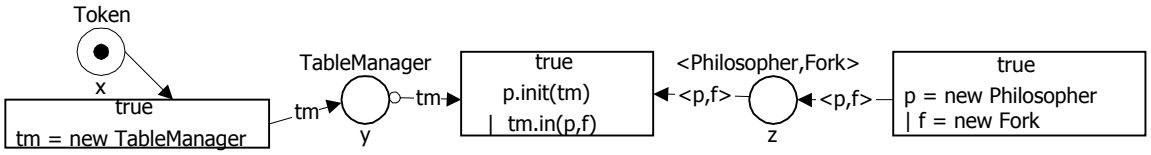


Figure B-4 Initial Net System

The corresponding class nets, depicted in Figure B-2, Figure B-3 and Figure B-1 respectively specify the behaviors of philosophers, forks and the table manager. Doubled circles are *method port places*. A *call* like method has one *accept* port place, and a *call/return* like method has also one *result* port place besides an *accept* port place. For example in Figure B-1, *in* is a *call* like method, and method *getlfork* is a *call/return* like method. Some places serve as *state port places*, which can be use for constructing the attribute predicates and the macro conditions. An attribute predicate may be accessed (read only) by outside of the object (an instance of the class net). For example in Figure B-2, *thinking*, *hungry*, *waiting* and *eating* are all (0-ary) attribute predicates. In Figure B-1, *unused(f)* is a (1-ary) macro condition. In the up part of a transition is its guard expression, and in the down part are listed the actions to be executed if it fires. The colour (type) of a place is written on the near top of the place. A colour type may be a class net such as *Philosopher* in Figure B-1, or a data type such as *CircleList* in Figure B-1, or generally a vector of types such as <*Philosopher*, *Fork*>. Arcs (starting from a place) with a small circle at their beginning ends are *test arcs* ([11]).

The net system in Figure B-4 is the initial picture of a dynamic object net system for the modified philosophers problem. When the system evolves, it may also contain a net system for the table manager and a changeable number of net systems for philosophers and forks, each of which is an instance of its corresponding class net. Each instance net system may be referenced by several *references* (in this case we say the references being attached to the instance net system), which are used as token values (not to use the instance net itself). The current *configuration* of a dynamic object net system consists of all the reachable references from a *root reference*, which is attached to the initial instance net system, such as the one in Figure B-4. The instances not attached by any references in the current configuration are garbage instances.

Referring to Figure B-2, transition *subscribe* includes two call/return like method invocations (transition *unsubscribe* includes two call like ones), in which the transferring of requests (also the parameters) and the accepting of results have not been separated into two transitions. To implement the ordinary call/return work, the execution of the method invocation, for example *lf = tm.glfork(self)*, puts the parameter (the reference given by *self*) into the corresponding accept port place (labeled by *glfork(a)* in its class net) of the instance net referenced by *tm*, and assigns a new reference (initially attached to a *null* instance) to *lf* and simultaneously puts it into the corresponding result port place (labeled by *glfork(r)* in its class net). That is all needed to be done. The transition rules guarantee that the transitions *unsubscribe* and *to_eat* can't fire whenever there are references in the place *subscribed* being attached to *null*. These references can become to attaching to two instances of *Fork* when certain *attach* functions (referring to Figure B-1) are executed. In this way, the *rendezvous* occurring at the accepting of results leaves as the responsibility of the net system, not of the user.