Extending G-Nets to Support Inheritance Modeling in Concurrent Object-Oriented Design

Haiping Xu and Sol M. Shatz
The University of Illinois at Chicago
Chicago, IL, 60607

Abstract

G-Nets are a type of Petri net defined to support modeling of system as a set of independent and loosely-coupled modules. The modular features of G-Nets provide support for incremental design and successive modification, however the G-Net formalism is not fully object-oriented due to a lack of support for inheritance. In this paper, we introduce extensions to G-Nets to support explicit modeling of inheritance. Bounded buffer examples are used, which we define as subclasses of an unbounded buffer, to illustrate the expressive power of the extended G-Net models. Various forms of inheritance are formalized and discussed in the context of concurrent object-oriented design. In addition, the inheritance anomaly problem is examined and discussed.

1 Introduction

A widely accepted software engineering principle is that a system should be composed of a set of independent modules, where each module hides the internal details of its processing activities and modules communicate through well-defined interfaces. The G-Net model (a form of Petri net) provides strong support for this principle [1][2]. A G-Net system is composed of a number of G-Nets, each of them representing a self-contained module or object. A G-Net is composed of two parts: a special place called *Generic Switch Place (GSP)* and an *Internal Structure (IS)*. The *GSP* provides the abstraction of the module, and serves as the only interface between the G-Net and other modules. The *IS*, a modified Petri net, represents the detailed design of the module. An example of G-Net is shown in Figure 1. Here the G-Net model represents an unbounded buffer. The generic switch place is represented by *GSP(UB)* enclosed by an ellipsis, and the internal structure of this model is represented by a rounded box which contains the detailed design of four methods: *isEmpty(), put(e), get()* and *who()*. The functionality of these methods are defined as follows: *isEmpty()* checks if the buffer is empty and return a boolean value, *put(e)* stores an item *e* into the buffer, *get()* removes an item from the buffer and returns that item, and *who()* prints the object identification of the unbounded buffer UB. A *GSP* of a G-Net *G* contains a set of methods *G.MS* (listed in the rectangle beside the *GSP* place) specifying the services or interfaces provided by the module, and a set of attributes *G.AS* as attributes/state variables (we do not show them in Figure 1). In *G.IS*, Petri net places represent primitives; while transitions, together with arcs, represent connections or relations among those primitives. These primitives may be actions or method calls, represented by special places called *Instantiated Switch Place (ISP)*. A primitive becomes *enabled* if it receives a token, and an enabled primitive can be executed. Given a G-Net *G*, an *ISP* of *G* is a 2-tuple *(G'.Nid, mtd)*, where *G'* could be the same G-Net *G* or some other G-Net, *Nid* is a unique identification of G-Net *G'*, and $mtd \in G'.MS$. Each *ISP(G'.Nid, mtd)* denotes a method call *mtd()* to G-Net *G'*. An example of *ISP* is shown in the method *get()* (denoted as an ellipsis), where the method *get()* makes a method call *isEmpty()* to the G-Net module/object itself to check if the buffer is empty. Note that we have extended G-nets to allow the use of the keyword **self** to refer to the module/object itself.
From the above description, we can see that a G-Net model essentially represents a module or an object rather than an abstraction of a set of similar objects. To introduce inheritance, it is required that the concept of *class* can be modeled and a new way for instantiation of G-Nets needs to be defined. As a result, we redefine the semantic of *ISP* to represent a call to an object (i.e., an instantiation of a G-Net) rather than a G-Net[1]. Furthermore, we discuss extensions to support inheritance modeling and inheritance anomaly issues.

2 Extending G-Nets for Class Modeling

To support inheritance, we use G-Nets to model classes rather than objects. The instantiation of a G-Net *G* generates a unique object identification *G.Oid* and initializes the state variables in *G.AS*. We call the instantiated G-Net *G* as G-Net object *G_obj*. An *ISP* method invocation is no longer represented as the 2-tuple

[1] In the rest of this paper, we follow the convention that a *G-Net* refers to a *G-Net class* model and a *G-Net object* refers to an instance of a *G-Net class*.

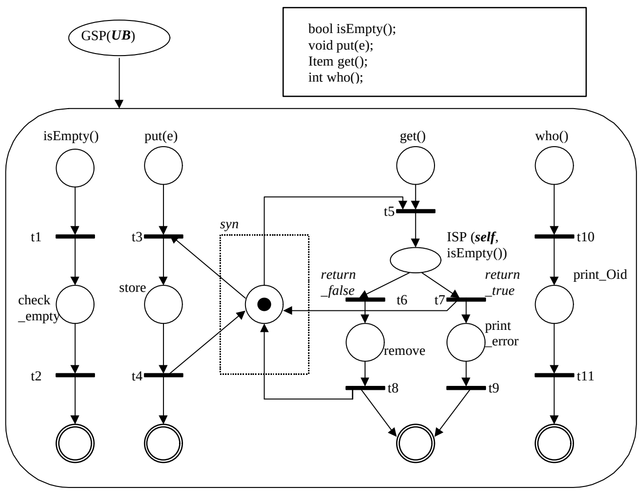


Figure 1 G-Net Model of Unbounded Buffer UB

(G'.Nid, mtd), instead it is the 2-tuple *(G'.Oid, mtd)*, where different object identifications could be associated with the same G-Net class model.

The token movement in a G-Net object is similar to that of original G-Nets [1]. The only difference is that we allow two types of tokens, namely *sTkn* tokens and *gTkn* tokens. An *sTkn* token is a colored or colorless token used in synchronization modules, which we will introduce shortly. A *gTkn* token is a triple *(seq, sc, msg)*, where *seq* is the propagation sequence of the token, *sc* ∈ (**before**, **after**) is the status color of the token and *msg* is a triple *(mtd_name, para_list, result)*. For ordinary places, tokens are removed from input places and deposited into output places by firing transitions. However, for the special place *ISP*, whenever a method call is made to a G-Net object, the token in the *ISP* place is processed (by attaching information for the method call) and removed, and an identical token is deposited into the *GSP* place of the called G-Net object. Through the *GSP* of the called G-Net object, the token is then dispatched into an *entry place* of the appropriate called method. After the method call, the token will reach a *goal place* (denoted by double circles) with the result attached to the token. As soon as this happens, the token will return to the *ISP* place of the caller and the information related to this completed method call will be detached.

More specifically, when a G-Net object *G_obj* with *G.Oid* makes a method call *ISP(G'.Oid, mtd(para_list))* in its thread/process with *G.Pid*, the procedure for updating the *gTkn* token is as follows:

1. Call_before: *gTkn.seq* ← *gTkn.seq* + *<G.Oid, G.Pid, mtd>*; *gTkn.msg* ← *(mtd, para_list, NULL)*; *gTkn.sc* ← **before**.
2. Transfer the *gTkn* token to the *GSP* place of the called G-Net object with *G'.Oid*.
3. Wait for the result to be stored in *gTkn.msg.result*, and the *gTkn* token to be returned.
4. Call_after: *gTkn.seq* ← *gTkn.seq* – *LAST(gTkn.seq)*; *gTkn.sc* ← **after**.

Notice that in the unbounded buffer class model we introduced a synchronization module *syn* to synchronize the methods *get()* and *put(e)*. This mechanism is necessary because these methods need to access the same unbounded buffer and they should be mutually exclusive. Generally, to design the synchronization module, we can either fulfill all synchronization requirements in one synchronization module or distribute them in several synchronization modules. To simplify our model, we follow the second option. Therefore, each class model may contain as many synchronization modules as necessary, and each synchronization module can be used to synchronize among a group of methods. As we will see, the synchronization module can not only be used to synchronize methods defined in a class model, but also can be used to synchronize methods defined in a subclass model and methods defined in its superclass (ancestor) model.

We now provide a few key definitions for our extended G-Net class models.

Definition 2.1 *Internal Structure (IS)*

The *internal structure* of G-Net *G* (representing a class) is a 2-tuple (M, S), where M is a set of *methods* and S is a set of *synchronization modules*. The arcs connecting M and S belong to S.

Definition 2.2 *Method*

A method is a triple (P, T, A), where P is a set of places with three special places called *entry place*, *ISP place* and *goal place*. Each method can have only one *entry place* and one *goal place*, but it may contain multiple *ISP places*.

T is a set of transitions, and each transition can be associated with a set of guards. A is a set of arcs defined as: ((P-{*goal place*}) x T) ∪ ((T x (P-{*entry place*}).

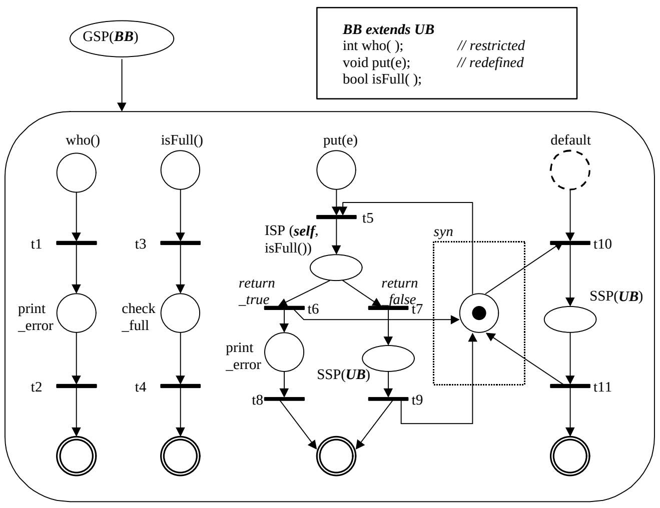


Figure 2 G-Net Model of Bounded Buffer BB

Definition 2.3 *Synchronization Module*
A synchronization module is 4-tuples (P, A, I, O), where P is a single place used to hold an *sTkn* token, and A is a set of arcs defined as: (P x IS.M.T) ∪ (IS.M.T x P). I is a set of arc inscriptions on place incoming arcs, and O is a set of arc inscriptions on place outgoing arcs.

3 Extending G-Nets to Support Inheritance

3.1 Introducing Inheritance into G-Nets

With inheritance, when we instantiate a G-Net *Sub_G* (a subclass), it is not enough to just associate an *Oid* with *Sub_G* and initialize the state variables defined in *Sub_G* class. We must associate the same *Oid* with all of *Sub_G*'s superclasses (ancestors) and initialize all state variables defined in those classes. The initialized part corresponding to the subclass and each of the superclasses (ancestors) is called *primary subobject* and *subobject* respectively [4][3]. When a method call is made to the object *Sub_G_obj* (i.e., an instantiation of class *Sub_G*), it is always the case that only the *GSP* place of the primary subobject is marked. The subobjects corresponding to the superclasses (ancestors) of *Sub_G* are not activated unless the method call to *Sub_G_obj* is not defined in the subclass model *Sub_G*.
When a method call is not found in a subclass model, we need to resolve the problem by searching the methods defined in the superclass models. To do this, we define a new mechanism called a *default place*. A *default place* is a default *entry place* defined in the *internal structure* of a subclass model and is drawn as a dash-lined circle, as shown in Figure 2. When a method is dispatched in a subclass model, the methods defined in the subclass model are searched first. If there is a match, one of the *entry places* of those methods is marked; otherwise, the *default place* is marked instead. After the dispatching, necessary synchronization constraints are established by the *synchronization modules*. If the *default place* is marked, the method call is then forwarded to a named superclass model. At first, it may seem that we can use the *ISP* method invocation mechanism to forward an existing method call. However this is not quite proper. Note that the initial method call will attach information associated with the call to the *gTkn* token. Now the subsequent call to the superclass would again attach the same information to the token, and the method call will actually be invoked more than once. To solve this problem, we introduce a new mechanism called a *Superclass Switch Place (SSP)*.
An *SSP* (denoted as an ellipsis in Figure 2) is similar to an *ISP*, but with the difference that the *SSP* is used to forward an existing method call to a *subobject* (corresponding to a superclass model) of the object itself rather than to make a new method call. Essentially, an *SSP* does not update the *gTkn* token because all the information for the method call has already been attached by the original *ISP* method call. In the context of multiple inheritance, we represent an *SSP* mechanism in subclass *Sub_G* as *SSP(G')*, where *G'* is one of the superclasses of *Sub_G*. Note that the object identifier is not necessary, as in the case of *ISP* method invocation, because the method call will be forwarded to the object itself (i.e., its subobject). When the method call is forwarded to the subobject corresponding to the superclass model *G'*, the *GSP* place of the superclass model *G'* is marked, and the methods defined in the superclass model are searched. If a method defined in the superclass model is matched, as in the case of *ISP* method invocation, the matched method is executed, and the result is stored in *gTkn.msg.result* and the *gTkn* token returns to the *SSP* place. Otherwise, the *default place* (if any) in the superclass

is marked, and the methods defined in the grandparent class model are searched. This procedure can be repeated until the called method is found. If the method searching ends up in a class with no methods matched and no *default place* defined, a "method undefined" exception should be raised. This situation can be avoided by static type checking.

Now consider a bounded buffer example as shown in Figure 2. We define a bounded buffer class BB as a subclass of an unbounded buffer class UB. Since the buffer has a limited size of *MAX_SIZE*, when there is a *put (e)* method call, the size of the buffer needs to be checked to make sure that the buffer capacity is not exceeded. In this case, the method *put (e)* defined in the class model UB is no longer correct, and it needs to be redefined in the subclass model BB. A simple way to redefine the method *put (e)* in subclass BB is to first make an *ISP* method call *isFull()* to the bounded buffer object itself. The method *isFull()* is used to check if the bounded buffer is full and it is added to the BB class model as shown in Figure 2. If it returns true, i.e., the bounded buffer has already been full, an error or exception will be generated; otherwise, the method call *put(e)* will be forwarded to its superclass UB by using an *SSP* mechanism. Here we use an *SSP* to allow reuse of the original method *put(e)* defined in class UB. As we will explain later, we call this situation refinement inheritance. Note that if we use *ISP(**self**, put(e))* in this situation, a dead loop will occur. This is because the methods defined in the subclass will always be searched first; and consequently, the method *put(e)* defined in subclass BB will be called recursively. Again we see the value of introducing the *SSP* mechanism.

It is also important to notice that a synchronization module can be used to synchronize methods defined in a subclass model and methods defined in the superclass model. However, in this case, all methods defined in superclass (ancestor) models must be synchronized as a whole. For instance, in Figure 2, the refined method *put(e)* defined in subclass BB is synchronized with all methods defined in the superclass UB, yet the synchronization between the method *put(e)* and the inherited method *isEmpty()* is unnecessary.

To formally define extended G-Nets with inheritance, we need to redefine the *internal structure* and define the concept of *Abstract Superclass Module*.

Definition 3.1 *Internal Structure*
The *internal structure* of G-Net is a triple (M, S, A), where M is a set of *methods*, S is a set of *synchronization modules*, and A is an optional *Abstract Superclass Module*. The arcs connecting M and S, or connecting S and A belong to S. There are no direct arcs between M and A.

Definition 3.2 *Abstract Superclass Module*
An *Abstract Superclass Module* is a triple (P, T, A), where P is a set of places includes three special places: *default place*, *goal place* and *Superclass Switch Place (SSP)*. T is a set of transitions with optional guards. A is a set of arcs defined as: ((P – {*goal place*}) x T) ∪ (T x (P – {*default place*})).

3.2 Modeling Different Forms of Inheritance

3.2.1 Augment Inheritance

Augment inheritance is straightforward - new protocols, which are not defined in the superclass model, are added to a subclass model. For instance, consider the design of the subclass BB as shown in Figure 2. We require a service to check if the buffer is already full. This can be done by adding a new method *isFull()* to the subclass BB. Since the method *isFull ()* does not override any methods in class UB, we have used augment inheritance.

3.2.2 Restrictive Inheritance

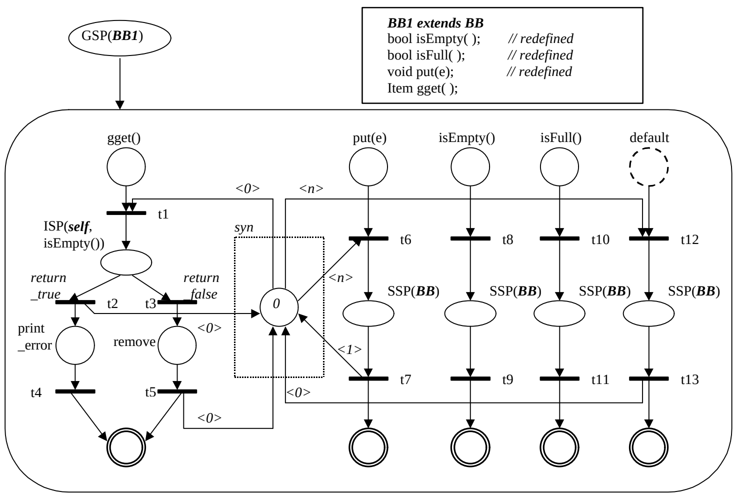
In some cases, we regard a class as a specialization of another class, with some superclass methods absent from the protocol of the subclass. This type of inheritance actually runs counter to the semantics and intentions of inheritance, because the "IS-A" relationship between superclass and subclass is broken. However, restrictive inheritance may be necessary when using an existing class hierarchy that can not be modified. Usually, restrictive inheritance is implemented in the subclass by overriding the disallowed superclass methods to produce error messages or signal exceptions [3]. Here we use a trivial example to illustrate how to model restrictive inheritance. Suppose we need to disallow the inherited method *who()* in our subclass BB. This can be simply done by redefining method *who()* in class BB; the redefined method *who()* does nothing but prints an error message to indicate that the method call for *who()* is disallowed in subclass model BB.

3.2.3 Replacement Inheritance

A subclass can completely redefine the behavior of its superclass for a particular method defined in the superclass. With this form of method overriding, we say that the method in the subclass replaces the method defined in the superclass. Replacing a superclass method generally occurs when the subclass can define a more efficient method or needs to define a method in a different way [3]. An example of replacement inheritance would be possible in the bounded buffer example, if we redesign the method *get()* in subclass BB to make the "remove" action more efficient.

3.2.4 Refinement Inheritance

More frequently, the semantics of a subclass demand that the subclass respond to a method call by a method that includes the behavior of its superclass, but extends it in the same way. In this case, we say that the subclass method



```
BB1 extends BB
bool isEmpty( );      // redefined
bool isFull( );       // redefined
void put(e);          // redefined
Item gget( );
```

Figure 3 G-Net Model of Bounded Buffer BB1

refines the superclass method. Practically, method refinement is more common than method replacement because it provides a semantic consistence with specialization. When implementing method refinement, we may simply refine the method by *copying* the relevant superclass method into the subclass model. However, we would like our extended G-Net formalism to provide a mechanism that supports automatic sharing of the superclass method. This capability is supported by the *SSP* mechanism and it has been illustrated by the method refinement of *put(e)* in bounded buffer BB as shown in Figure 2.

3.3 Modeling Inheritance Anomaly Problem

Inheritance anomaly refers to the phenomenon that synchronization code can not be effectively inherited without non-trivial re-definitions of some inherited methods [5][6]. As a consequence, some well-known proposals for concurrent object-based languages, such as families of Actor languages, POOL/T, Procol and ABCL/1, chose to not support inheritance as a fundamental language feature [5]. Also some languages like Concurrent Smalltalk or Orient84/K do provide inheritance but do not support intra-object concurrency - that is there is only a single thread of control within an object [7].

There have been previous efforts to solve the inheritance anomaly problem [5][6], but most of the proposals are based on quasi concurrency, where only one thread at a time is allowed to execute. As stated in [7], this type of inheritance anomaly seems to be almost solved. “True” concurrency refers to cases that more than one thread can be executed in an object at the same time. Reference [7] talked about solutions in this context. The inheritance anomaly problem has usually been approached in terms of analyzing the causes. The causes have been classified as partitioning of acceptable states, history-only sensitiveness of acceptable states, and modification of acceptable states [5]. Here, we analyze the inheritance anomaly problem based on clarifying the terminology of “synchronization constraints”, and we always view a concurrent system as a “true” one.

As we will see, synchronization constraints among methods can be specified explicitly or implicitly. An explicit synchronization constraint refers to the concurrent/mutual exclusive execution between two methods in an object. For instance, in the unbounded buffer example, method *get()* and method *who()* can be executed concurrently, however the execution of method *get()* and method *put(e)* must be mutually exclusive. This type of synchronization constraint creates the inheritance anomaly problem when a method *m1* defined in a subclass module needs to be mutually exclusive with a particular inherited method *m2* that is defined in its superclass (ancestor) module. A simple way to deal with this situation is to refine the method *m2* (e.g., to use the SSP mechanism in our extended G-Net model) and to establish mutual exclusion between *m1* and *m2* in the subclass module. In this case the method defined in the superclass (ancestor) module can be reused by a refinement inheritance.

An implicit synchronization constraint refers to cases where acceptance of a method in an object is based on that object’s state. The state of an object can be changed by executing a method in that object. For instance, when a buffer is in a state of “empty”, the method *get()* is not allowed to execute; however, after executing the method *put(e)*, the state of the buffer is changed from “empty” to “partial,” and at this time, the method call of *get()* becomes acceptable. Since the methods *get()* and *put(e)* are indirectly synchronized through the state of the buffer, we called this type of synchronization constraint an implicit synchronization constraint. The implicit constraints can be further classified in terms of two different views of an object’s state, namely internal view and external view. Under an internal view, the state of an object can be captured by the evaluation of state variables of the object

[5]. For example, the state "empty" of a buffer can be captured by checking if the state variable of *buffer_size* evaluates to "0". This type of synchronization can always be added to a subclass module without redefining inherited methods because it can be easily maintained by checking state variables before allowing the execution of a method.
Another view is the external view, where the state is captured indirectly by the externally observable behavior of the object [5]. For example, a state under external view could be the state of a buffer object when the last executed method is *put(e)*. When synchronization constraints with respect to the external view of an object's state are added to a subclass module, some methods defined in a superclass (ancestor) module must be redefined. Fortunately, in most cases, as long as no deadlocks are introduced, we can again use refinement inheritance to reuse the original method defined in the superclass (ancestor) module. We use the classic example of *gget()* to illustrate this situation. Consider a new bounded buffer BB1, defined as a subclass of bounded buffer BB, and add a new method called *gget()*. The behavior of *gget()* is almost identical to that of *get()*, with the sole exception that it can not be executed immediately after the invocation of *put(e)* [5]. The design of the new bounded buffer BB1 is illustrated in Figure 3. To establish the synchronization between methods *gget()* and *put(e)*, the method *put(e)* must be redefined in the subclass module BB1. Suppose we have an object *bb1*, an instance of class BB1. Initially, the token in the synchronization module *syn* is "0". Whenever there is a method call other than *put(e)* to object bb1, the token will be removed and deposited back to the synchronization module with the same value of "0". However, if there is a method call for *put(e)*, the token in the synchronization module *syn* will be removed first, and then the method call *put(e)* will be forwarded to its superclass BB by using the *SSP(BB)* mechanism. After the method call of *put(e)*, a token with value "1" will be deposited into the synchronization module *syn*. At this time, if there is a method call for *gget()*, the call must wait because a token with value "0" is necessary to enable the transition *t1*. Thus the synchronization between methods *gget()* and *put(e)* is correctly established. Note that we cannot reuse the method *get()* when designing the method *gget()* by using the *SSP(BB)* mechanism. This is inapplicable because *gget()* and *get()* are two different methods. In addition, we need to redefine the methods *isEmpty()* and *isFull()* to avoid deadlocks.

4 Conclusion and Future Works

Inheritance has been introduced into several object-oriented net models, such as LOOPN++ [8] and CO-OPN/2 [9]. However, those methods do not use net-based extensions to capture inheritance properties. Our approach explicitly models inheritance at the net level to maintain an underlying Petri net model that can be exploited during design simulation or analysis. In future work, we will explore an algorithmic basis for synthesis of subclass models as well as investigate how to analyze extended G-Nets at an abstract level, with consideration for the state explosion problem. Issues like design consistency and deadlock avoidance will be of primary concern.

References

[1] Y. Deng, S. Chang, A. Perkusich and J. de Figueredo, "Integrating Software Engineering Methods and Petri Nets for the Specification and Analysis of Complex Information Systems", *Proc. of The 14th International Conference on Application and Theory of Petri Nets*, Chicago, June 21-25, 1993, pages 206-223.

[2] A. Perkusich and J. de Figueiredo, "G-nets: A Petri net based approach for logical and timing analysis of complex software systems", *Journal of Systems and Software*, 39(1):39-59, 1997.

[3] Caleb Drake, *Object-oriented programming with C++ and Smalltalk*. Upper Saddle River, New Jersey, Prentice Hall, 1998.

[4] J. G. Rossie Jr., D. P. Friedman and M. Wand, "Modeling Subobject-Based Inheritance", *Proceedings of ECOOP'96*, Vol. 1219, Lecture Notes in Computer Science, pages 248-274, Springer-Verlag, 1996.

[5] Satoshi Matsuoka and Akinori Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages". In Gul Agha et. al., editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107-150. MIT Press, 1993.

[6] S. Mitchell and A. Wellings, "Synchronization, Concurrent Object-Oriented Programming and the Inheritance Anomaly", *Computer Languages*, 1996, Vol. 22, No. 1, pages 15 - 26.

[7] Laurent Thomas, "Inheritance Anomaly in True Concurrent Object Oriented Languages: A Proposal", *IEEE TENCON'94*, August 1994, pages 541-545.

[8] Charles Lakos, "Pragmatic Inheritance Issues for Object Petri Nets", In *Proc. of Technology of Object-Oriented Languages and Systems (TOOLS) Pacific 1995*, Melbourne, Australia, Prentice-Hall (1995).

[9] Olivier Biberstein, Didier Buchs and Nicolas Guelfi, "CO-OPN/2: A Concurrent Object-Oriented Formalism," In *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, July 21-23, 1997, Chapman and Hall, London, pages 57-72.