Solving Inheritance Anomaly with OMNets

YU Jian (喻坚), WANG Shengyuan (王生原) and YUAN Chongyi (袁崇义)

Department of Computer Science and Technology, Peking University, Beijing 100871, P.R. China
E-mail: yuj@theory.cs.pku.edu.cn
Received November 2, 2000; revised April 11, 2001.

Abstract This paper presents a concurrent object-oriented modeling language based on Petri nets: OMNets, which helps greatly to avoid the inheritance anomaly problem appeared in concurrent OO languages. OMNets separates the functional part and the synchronization part of objects and uses Petri nets to describe the synchronization part. Both parts are reusable through inheritance.

Keywords inheritance anomaly, concurrency, object-orientation, Petri net

1 Introduction

The integration of concurrency and object-oriented programming has been studied by computer scientists for the past two decades. A big problem which still exists is that concurrency does not work well with inheritance, a key feature of object orientation. It has been pointed out that synchronization code cannot be effectively inherited without non-trivial class re-definitions[1]. When inheriting from a class with the synchronization code, the derived class is forced to reimplement methods of the base class because of the incompatibilities of the synchronization code. This phenomenon is called inheritance anomaly. Many concurrent object-oriented programming languages have been proposed for the attempt to analyze and solve this problem[1-3]. A common opinion that has been widely accepted is that an object should have a clear boundary between its synchronization part and its functional part for easy reuse.

Object Modeling Nets (OMNets for short) is an object-oriented modeling language based on Petri nets. It gives a way to separate the synchronization part and the functional part of an object: it declares the synchronization interface in a method's signature. In OMNets, the synchronization of an object is described by a class composed of Petri nets. Such design eases the inheritance between concurrent objects dramatically: the sequential part of an object can be inherited free of any change; through inheritance of OMNets, the synchronization part can also be reused conveniently.

The following features differ OMNets from other object modeling languages based on Petri nets[4]. First, instead of expressing concurrency and synchronization within an object's provided services directly, OMNets creates a synchronization map of an object and gives its services specific entry and exit. We will describe in Section 3 that such design will help much to avoid inheritance anomaly. Second, OMNets introduces set arc and reset arc to inhibitor-arcs-extended P/T Nets for convenience purpose. It has been proved that inhibitor-arcs-extended P/T Nets has the same simulation power as the Turing machine. Set and reset arcs do not add new constructs to inhibitor-arcs-extended P/T Nets, we will see in Section 2 that they are two simplified marks of specific inhibitor-arcs-extended P/T Nets.

2 OMNets Modeling Language

OMNets modeling language (the formal definition is given in [5]) is composed of two parts: one is the description of a system's data structure and functional aspect using a C++ like syntax. This

Supported by the National Natural Science Foundation of China under grant No.69973003 and by the NKBRSF (973) of China under grant No.G1999032706.
part needs no care about any synchronization and concurrency. The other part is the synchronization class that models the concurrency and synchronization features using OMNets. The keyword `with` establishes a connection between these two parts.

Fig.1 is a simple OMNets of the Reader-Writer problem. The `with` synchronization keyword is used in two situations: as part of a class signature, it points to its synchronization class; and as part of a method signature, it points out this method's synchronization place. The synchronization class is an OMNets being encapsulated (such as the class RW_SYNC in Fig.1). The synchronization place is a special place used to accept the outside request (such as the read_request place in Fig.1) or the release request (such as the read_exit place in Fig.1). The former is called `accept place` and the latter is called `release place`. The synchronization place serves as the interface of synchronization class and is the only visible element part of a synchronization class. So a method's sequential part and synchronization part are separated neatly. When a method that has a keyword `with` is called, a token is put to its accept place before executing the sequential method body. If this token is accepted, the method body begins to run and this token flows to a running place that indicates the state of body running. When the execution of the method body is over, a token will be put to a release place. All deserted tokens flow to a `sink place` with infinite capacity and this scenario of synchronization ends.

Set and reset arcs are used to express Boolean variables in a net. A set arc is from a transition `t` to place `p`, and is drawn as a connector with a cross bar at the place end: `t \rightarrow p`. When `t` fires, `p` will be set (a token is in it) no matter it is empty or already has a token before firing. Reset arc is drawn as a connector with a cross bar at the transition end: `t \leftarrow \neg p`. It does the reverse with set arc, and when `t` fires, the mark of `p` will be zero. Fig.2 is the set arc, the reset arc and their inhibitor-arcs-extended P/T Nets equivalence. Note that the capacity of place `p2` in the set arc equivalence must be bigger than 1 or the transition cannot fire. So an OMNets with set arcs and reset arcs can be transformed into inhibitor arcs extended P/T Nets according to Fig.2.

3 OMNets and Inheritance Anomaly

In this section, we describe how OMNets solves the four kinds of inheritance anomaly identified by Matsuoka: body anomaly, state partitioning, modification of acceptable state and history state sensitivity. Also we take the classic bounded buffer problem and its extended version which are used by Matsuoka in [1] as examples.

3.1 Body Anomaly

Body anomaly is caused by direct putting of the synchronization code in the body of a method. If a derived class wishes to change the synchronization behavior of a base method, there is no choice