DEFINITION OF REUSABLE CONCURRENT SOFTWARE COMPONENTS

S. Crespi Reghizzi, G. Galli de Paratesi
Dipt. Elettronica - Politecnico di Milano, Piazza Leonardo, 32 - Milano, Italy 20133.
S. Genolini
TXT Ingegneria Informatica SpA
Via Socrate 41, Milano, Italy 20128

Abstract

In O.O. languages with active objects, a constraint (or behaviour) on method activations is needed to avoid inconsistencies and to meet performance requirements. If the constraint is part of a class definition, the class population grows with the product of the number of behaviours. As pointed out in [Goldsack and Atkinson 1990] this undesirable growth may be controlled by separating the specification of the functional characteristics and the behavioural characteristics of a class. This work extends the concept of behavioural inheritance (b-inheritance) which provides a behaviour to a sequential class. Furthermore, the interaction between b-inheritance and inheritance is discussed. Deontic logic notation for specifying behaviour is extended to deal with the definition of more complex constraints and to improve reusability characteristics of components. The proposal is formalized by extended Petri nets and the translation into a concurrent language is outlined. The project is under development within the O.O. ADA extension DRAGOON [Di Maio et al 1989].

1. Introduction

This work addresses the specification of software components, for concurrent systems in the specific perspective of software reuse. A first, more conservative, approach to concurrent component design assumes an existing collection of sequential components, which have to be used in a concurrent setting; this can cause inconsistencies in state variables, saturation of resources or other problems, unless suitable restrictions are imposed on concurrent activations.

A second, more organic approach, not investigated in this paper, assumes that components are designed from the beginning with concurrent use in mind. This approach is strongly recomandable in the design of highly parallel systems, since the very structure of algorithms differs from the sequential case.

A typical O.O. language with classes, multiple inheritance, and objects is taken into consideration; classes can be active, i.e. endowed with a control thread. Method invocation is the protocol for communication between objects. Because of the presence of many threads, methods of an object can be concurrently called, causing unpredictable results: hence the...
need to specify a constraint on their activations. Constraints are also motivated by the need to control computer resource usage (e.g. by limiting the number of concurrent activations of a reentrant method).

There are essentially two basic strategies for introducing concurrency features [America 1989]. The first approach is to encapsulate sequential and concurrent features within the same class specification. The second is to superimpose concurrency constructs as an extra layer, orthogonal to the object-oriented paradigm. Specifying sequential and concurrent features at the same time may raise two kinds of problems:

- **First:** there may be a conflict between the use of inheritance to support software adaptability, and the inclusion of synchronization constraints in the class, to ensure correctness. In fact, modification of a class functionality may involve adding new methods or removing existing ones, thereby making the synchronization constraints inconsistent w.r.t. the new class interface.

- **Second:** class population increases by a large factor. For instance a class SymbolTableManager can have a variety of behaviours, such as mutual exclusion on all methods, concurrent activation of methods performing a read operation but mutual exclusion of methods involving updates, concurrency limited by a constant $k$ in order to avoid task proliferation, various priority constraints, etc. The definition of a separate class for each combination of functional and behavioural specifications besides being unpractical, moves in the opposite direction of software reuse.

As a consequence it was argued ([Goldsack and Atkinson 1990], [Di Maio et al 1989]) that synchronization constraints, called *behaviours*, should not be a part of class specification, but should be superimposed using an orthogonal construct. Class behaviour must be specified separately and independently of functionality: a *behavioural class* (b-class) is an abstract, generic, specification of behaviour. Multiple inheritance, called *behavioural inheritance*, is exploited to associate a synchronization constraint, specified by a behavioural class, with the methods of a sequential class.

This approach is consistent with the hypothesis that the design of concurrent behaviours and the reuse of existing classes are the concerns of two different kinds of persons dealing with a software component base. The normal user is not expected to design new abstract behaviours, but only to use library's b-classes, whereas the expert user can specify new behaviours to be added to the component base.

This research focuses on the notation for specifying concurrent behaviour, on the formalization of behavioural inheritance by extended Petri nets, and on the automatic generation of concurrent code for behavioured objects.

In Sect. 2 the notion of concurrent behaviour, behavioural inheritance and its relation to inheritance is discussed. Furthermore, a gamut of constructs for expressing generic synchronisation constraints is analyzed using the method of deontic predicates (a notation related to path expressions [Campbell and Habermann 1974]). For each construct, expressive power, degree of reusability and runtime efficiency are evaluated. In Sect.3 the formalization of the behavioural heir by means of Petri nets extended with firing predicates...