DEFINITION OF REUSABLE CONCURRENT SOFTWARE COMPONENTS [1]

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 betweeen objects. Because of the presence of many threads, methods of an object can be concurrently called causing unpredictable results: hence the

[1]This work was initially supported by ESPRIT Project DRAGON, and is continuing under project REBOOT. We also aknowledge support by Italian MURST 40% and CNR.

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 is presented together with an implementation in terms of Ada tasks, with optimization options.

The research is part of DRAGOON, an O.O. variant of Ada designed to support reuse, distribution and dynamic reconfiguration ([Di Maio et al 1989]), but the concepts, notation, formal definition, and implementation are applicable to O.O. languages, such as C++ or Eiffel. The implementation can be adapted to other multi-task environment, e.g. Unix.

2. Specification of concurrent behaviour

In our reference model objects can be active. An active object is an instance of an active class, that is of a class which has a *thread* in addition to methods. A thread is similar to a method, except that it cannot be invoked but is activated at object instanciation time. Here we need not be concerned with object instanciation, but we can assume that in the system there are several concurrent activities, which can simultaneously invoke the methods of an object. This raises the issue of specifying a synchronization rule, also called a *behaviour*, for the activation of methods. There are different scenarios in which the rule could constrain the order of activation of methods:

-methods of a single object;

-methods of different objects of the same class;

-methods of any object of any class.

For simplicity we restrict the scope to the first case: in other words, we do not address the issue of regulating the activations of methods belonging to different objects. The restriction causes no loss of generality, at least in principle: in fact a semaphore can be easily defined as an object with two methods *signal* and *wait*, and any concurrent system can be designed using semaphores.

We call *free* a class *(f-class)* without constraints on method activations: this means that its methods can be executed in parallel on behalf of different active calling objects. When no other concurrent behaviour is indicated, what should the default be? Without a default no class can be instanciated unless the designer provides a behavioural specification: a burden for him when the system to be designed is purely or predominant sequential. The following reasonable alternatives have been considered:

1 -default behaviour is free;

2 -default behaviour is mutually exclusive;

We assume that classes are free by default; this is sometimes a dangerous assumption, since concurrent activation of methods originally intended for serial execution could cause critical races or inconsistencies. But the opposite hypothesis 2 causes inacceptable penalty on run-time efficiency, because every object must be implemented as a task. We prefer to leave to the designer responsibility for the introduction of a mutex constraint when needed.

Concurrent behaviours are specified by special abstract classes, called *behavioural* (shortly *b-class).* A class which can be instanciated, because all of its methods have a body, is called *concrete (c-class);* otherwise it is an *abstract* class (*a-class*). In order to regulate the concurrent behaviour of a free class, we use multiple (actually double) inheritance: the first parent is a f-class (but see later for another possibility), the other is a b-class, and the heir is the result of the prescribed regulation for the methods of the f-class. This heir class is called *behavioured* or *regulated (r-class)*, and this special form of inheritance is called behavioural (*b-inheritance*).

For instance, consider (Ex.2 in Fig 2.) the concrete f-class *Buffer3* (with methods *put, get* and *size*), and the b-class *Mutex:* the result of b-inheritance is an r-class, *BufferMutex3*. An instanciation of *BufferMutex3* is an object interfaced by mutually exclusive methods *put, get* and *size*.

Combination of b-inheritance and inheritance

An important issue is the combination of concurrent and functional specifications within the class hierarchy. In a sequential component catalogue, multiple inheritance relations link a class to its parents and siblings (subclasses).

Moving down an inheritance chain, one usually finds an abstract class progressively made concrete by method bodies, enriched by new methods, and specialized by method redefinitions. Sometimes methods are canceled or hidden. Of course (partially) abstract classes cannot be instanciated. The question is where the concurrent behaviour should be specified, inside the inheritance graph (which is a DAG). The range of possibilities for b-inheritance is presented in Fig.1. We comment each possibility.

	Parent 1 (P1)	Parent 2 (P2)	Result (R)
1	f-class ∩ a-class	b-class	r-class ∩ a-class
2	f-class ∩ c-class	b-class	r-class ∩ c-class
3	r-class ∩ a-class	b-class	r-class ∩ a-class
4	r-class ∩ c-class	b-class	r-class ∩ c-class

Fig.1- Possible domains of parents in behavioural inheritance.

1 - b-inheritance can only be applied to a free class, i.e. at most once along a path in the DAG. This means that the behaviour to be attached to a class P1 must be specified in a single step. If P1 is abstract, the result R is not instanciatable.

2 - same as 1, but in addition P1 must be a concrete class, hence R is instanciatable.

3 - P1 is a behavioured class (abstract), resulting from a previous b-inheritance. Thus an r-class can be obtained by incrementally specifying its behaviour in several steps down the DAG path.

Cases 1 to 3 are illustrated by the examples in Fig.2.

4 - same as 3, but P1 must be concrete; R is thus instan·iatable.

For simplicity and code efficiency we opted for 2, ruling out the possibilities of creating abstract behavioured classes and of superimposing onto a behavioured class another behaviour. Other reasons for this choice are presented later.

Actually, in order to complete the picture, we need to consider (Fig.3) the allowed domains

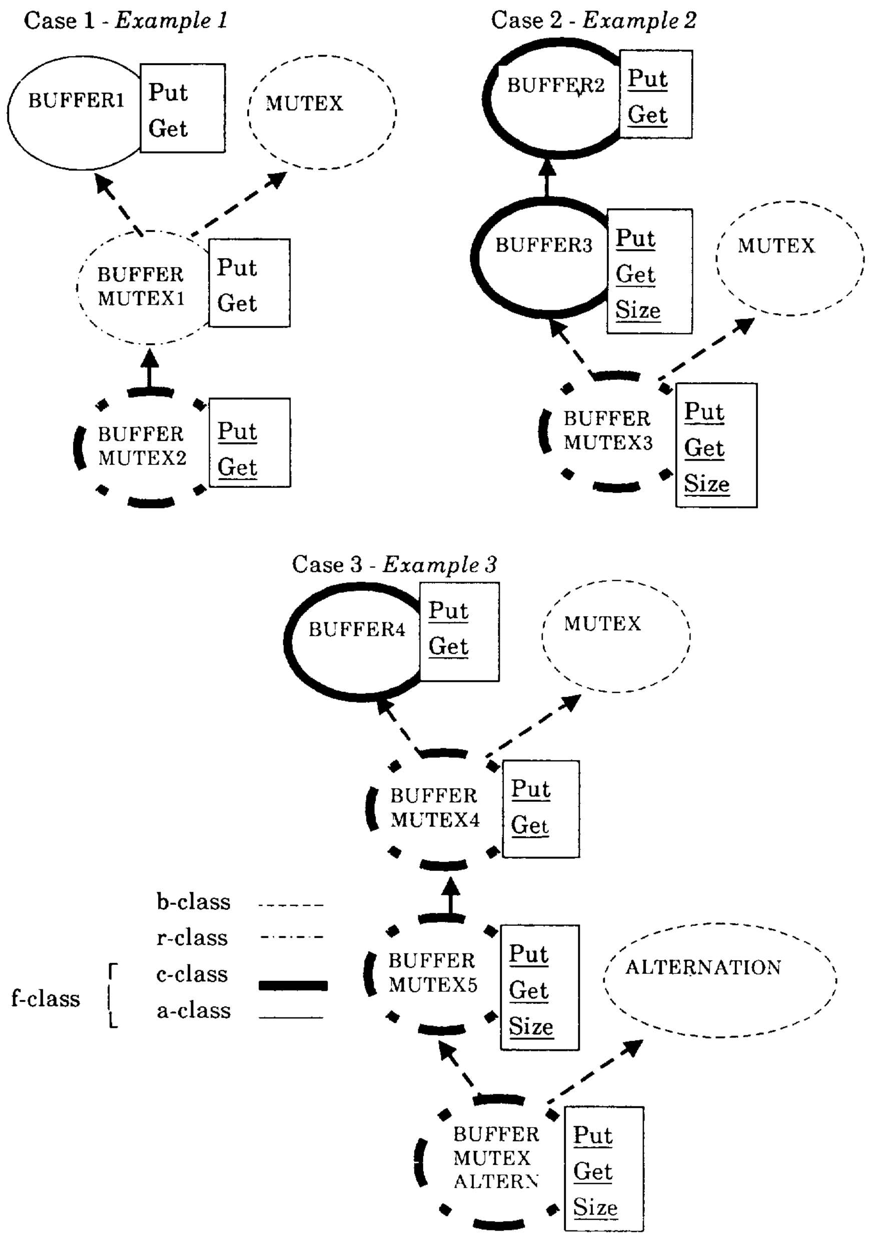

Fig.2 - Inheritance (solid arrows) and behavioural inheritance (dashed arrows) Methods provided with a body are underscored.

of parent classes for normal (non behavioural) inheritance. The cases of one parent or of more than two parents can be treated similarly and are omitted. The central question here is whether normal inheritance should be legal when one (cases 4,5,6) or more (cases 7,8,9) parents have already a behaviour. In principle one could conceive an inheritance relation between b-classes: for instance a b-class *ReaderWriter* with formal methods *Read* and *Write* defines the usual rule (mutex between writing and between writing and reading); then this behaviour could be specialized by inheritance, by means of a second b-class *ReaderWriterWithPrecedence* imposing the constraint that no reading should be allowed when a writing request is pending. These possibilities were excluded on the following grounds: simplicity, code efficiency, the difficulty to treat suppressed methods, and the opinion that composing the behaviours of two classes is not essential, because behaviours are seldom so complex to justify an inheritance taxonomy. Besides, suppose class P12 is the heir of two r-classes P1(M1A,M1B) and P2(M2A,M2B), where M1A, M1B, M2A and M2B represent the corresponding methods. In order to regulate concurrency (e.g. by mutex) of the methods originating from distinct parents, one should then define another class b-inheriting from P12 (which is an r-class), and from a b-class *mutex*. This is case 3 (or 4) of Fig.1, that we intended to exclude. Therefore only cases 1,2,3 of Fig.3 are legal.
In conclusion a behaviour can be attached by b-inheritance only to a concrete, free class. This must be the last step in the chain, since normal inheritance can only be applied to free classes. Since parent 2 of b-inheritance is concrete, we can refer to it as an object, rather than a class, understanding by this term the instance of the P2 class to which behaviour is to be attached. Experience will tell us whether this choice is too restrictive.

	Parent 1 (P1)	Parent 2 (P2)	Result (R)
1	f-class ∩ a-class	f-class ∩ a-class	f-class ∩ a-class
2	f-class ∩ c-class	f-class ∩ a-class	f-class ∩ a-class
3	f-class ∩ c-class	f-class ∩ c-class	f-class ∩ c-class
4	r-class ∩ a-class	f-class ∩ a-class	r-class ∩ a-class
5	r-class ∩ c-class	f-class ∩ a-class	r-class ∩ a-class
6	r-class ∩ c-class	f-class ∩ c-class	r-class ∩ c-class
7	r-class ∩ a-class	r-class ∩ a-class	r-class ∩ a-class
8	r-class ∩ c-class	r-class ∩ a-class	r-class ∩ a-class
9	r-class ∩ c-class	r-class ∩ c-class	r-class ∩ c-class

Fig.3 -Possible domains of parents in normal (non-behavioural) inheritance

Specification of behavioural classes

A b-class abstractly specifies constraints to be imposed on certain events (method activations). It does so independently of the actual methods of any f-class, by referring to formal method names, that will be bound to actual method names at b-inheritance time.

Several possible styles of specification could be adopted: a concurrent programming language, Petri nets or path expressions [Campbell and Habermann 1974]. We chosed the *deontic logic,* a predicative notation [von Wright 1980], which is similar in power to path expressions and quite adequate for the job.

For each formal method parameter MF of a b-class there is a deontic axiom of the form: *permitted*(MF) $\Leftrightarrow$ deontic predicate. Activation of the corresponding actual methods $MA_1, \ldots, MA_n$ is permitted only when the predicate (right-hand side) is true. Notice that MF stands for a set of actual methods, to be ruled by the same constraint.

Deontic expressions use a few *historical* operators returning the activation history of methods. Fig.4 summarizes the basic and derived operators.

Operator	Meaning
Deontic predicate head	
per(MF)	Activation of MF is permitted iff predicate is true.
Historical operators	Historical operators count specific occurrences of events since system start time
req(MF)	No. of requests of method MF
act(MF)	No. of activations of method MF
fin(MF)	No. of terminations of method MF
State operators	State operators return the number of items currently present in run-time system queues
act_now(MF) --derived	No. of current activations of method: act_now(MF) = act(MF) - fin(MF)
req_now(MF) --derived	No. of pending requests of method: req_now(MF) = req(MF) - act(MF)

Fig.4 - Historical and state operators. When MF is a set of methods, the operators return the aggregate result for all methods MA in the set.

Historical and state operators can be extended in the natural way to a set of methods instead that to a single one.

Mutex, mutual exclusion of methods ($MF_1, MF_2, \ldots$), is a common deontic expression:

$per(MF_i) \Leftrightarrow \forall j : act\_now(MFj) = 0$ -- i=j prevents multiple activations of same method

It is convenient to shorten this expression with the notation (> <).

Several possibilities of increasing complexity for the deontic predicate are shown in Fig.5 and discussed on later. We now explain the syntax referring to Ex.1 of Fig.5. Formal method parameters are introduced by the keyword "ruled", for each formal parameter there is exactly one deontic axiom. This specification clearly imposes that the first method to be activated is FOPS (i.e. one of the actual methods that will be bound to FOPS by b-inheritance), since per(FOPS) is true when and only when act(FOPS) = fin(SOPS) = 0 Once FOPS has been activated, act(FOPS) becomes 1, hence per(SOPS) becomes true, and per(FOPS) false, etc. Before discussing the other cases of Fig. 5 we present an explanation

Arguments	Examples
1 -Historical and state operators (see Fig.4) applied to formal methods *Example :* alternation of activations of two sets of methods FOPS and SOPS	**behavioural class** ALTERNATION **is** **ruled** FOPS , SOPS; -- formal methods are -- partitioned into two sets : F(irst)__OP__Set and -- S(econd)__OP__Set **where** per(FOPS) ⇔ act(FOPS) = fin(SOPS) --FOPS activation is permitted iff the no. of past --activations of FOPS and SOPS are equal; per(SOPS) ⇔ fin(FOPS) > act(SOPS) --SOPS activation is permitted iff the no. of --finished activations of FOPS exceeds the no. --of past activations of SOPS; **end** ALTERNATION;

2- Generic b-class w.r.t. a parameter
 	
Example: limits the max number of active methods
	generic
	NUM : POSITIVE;
	behavioural class LIMITER **is**
	ruled OPS;
	where
	per(OPS) ⇔ act.now(OPS) < NUM;
	end LIMITER;

3 -Generic b-class w.r.t the number of method groups *Example:* methods are activated according to their priority	**generic** --a generic b-class K : POSITIVE; -- the no. of priority groups; **behavioural class** PRIORITY **is** **ruled** OP **enum**; --enumeration of sets of formal --methods in order of decreasing priority; **where** per(OP'FIRST) ⇔ (> <); -- > < is mutex; per(OP) ⇔ per(OP'PRED) and req-now(OP'PRED) = 0; --OP activation is permitted iff activation of the --preceding (in the enumeration) formal set of --methods is permitted, and there no pending --requests for it; **end** PRIORITY;

Fig.5 - Arguments of deontic predicates in behavioural classes.

of b-inheritance.

A b-class is used in b-inheritance to provide a behaviour to a c-class. Consider the examples of b-inheritance in Fig.6, which refer to the b-classes of Fig.5.

Case A (Fig.6): The f-class UNI-BUFFER has three methods PUT, PUTLONG and GET, the b-class is ALTERNATION (Fig.5), and the result is the r-class ALTERNATION-UNI-BUFFER. The syntax identifies the parent b-class by the keyword "ruled by". Two actual methods, PUT and PUTLONG, correspond to the formal FOPS, and GET corresponds to SOPS. Thus a possible activation sequence is: PUT, GET, PUT, GET, PUTLONG, GET, ...

Not all kinds of behaviour can be conveniently expressed by such b-classes: for instance the constraint that the number of active methods should be less than a constant, NUM . The generic parameter NUM is introduced in Ex.2 of Fig.5, in order to avoid the need of a separate b-class for each different value of NUM.

Arguments	Examples

4 -Formal Boolean functions (guards) returning a truth value depending on the state of the formal object.
 	
Example: activations of two sets of methods (PT and GT) are permitted in mutex, with additional constraints against underflow and overflow; size of buffer is unknown to the b-class, but a Boolean formal function FULL_GUARD is used to inspect the state of the buffer. This function is activated in mutex.
	generic
	with function FULL_GUARD **return** BOOLEAN;
	behavioural class GUARDED **is**
	ruled PT, GT;
	where
	  per(GT) ⇔ (> <) **and** act(GT) < fin(PT);
	    -- no underflow
	  per(PT) ⇔ (> <) **and not** FULL_GUARD;
	    --no overflow
	  per(FULL_GUARD) ⇔ (> <) ;
	end GUARDED;

5 -Formal functions returning the state of the formal object.
 	
Example: the max number of activations of OP is determined at run-time. The number can be changed at run-time by calling the method SETUP
	generic
	with function MAX **return** NATURAL;
	behavioural class GUARDED_LIMITER **is**
	ruled OP,SETUP;
	(MAX,SETUP);
	where
	per(OP) ⇔ act_now(OP) < MAX **and**
	        act_now(SETUP) = 0 **and**
	        req_now(SETUP) > 0;
	per(SETUP) ⇔ act_now(MAX) = 0 **and**
	        act_now(SETUP) = 0 ;
	per(MAX) ⇔ act_now(SETUP) = 0 **and**
	        req_now(SETUP) > 0;
	end GUARDED_LIMITER;

6 -Parameters of formal methods *Example:*determining the priority level on the value of parameter I	**generic** **type** OP_TYPE **is** (< >) **behavioural class** MULTI_QUEUE **is** **ruled** A(OP_TYPE); **where** per(A(I)) ⇔ act(A(I)) = fin(A(I)) -- I is in the range of OP_TYPE **end** MULTI_QUEUE

Fig.5 (continued) - Arguments of deontic predicates in behavioural classes.

Consider next the activation of methods according to their priorities. If the methods can be partitioned in a fixed number *k* of priority groups, the deontic specification is:

per(MF_1) ⇔ (> <);
per(MF_2) ⇔ (> <) **and** req_now(MF_1) = 0;
....
per(MF_k)) ⇔ (> <) **and** req_now(MF_{k-1}) = 0 **and** ...... req_now(MF_1) = 0;

This is verbose and strictly dependent on *k*. A different b-class would be required for different numbers of priority groups. To solve the problem, *generic* b-classes have been introduced. The use of recursive deontic definitions (case 3 of Fig. 5), permits a single specification of priority behaviour, independently of the number of priority groups. In this case the number of formal methods is determined at inheritance time. Correctness

f class (1st parent)	b-class (2nd parent)	b-inheritance

A)
class UNI_BUFFER **is**
procedure PUT(ITEM : **in**
ELEMENT);
procedure PUTLONG
(LONGITEM:
in LONGELEMENT);
procedure GET(ITEM : **out**
ELEMENT);
end UNI_BUFFER;
	ALTERNATION	**class**
		ALTERNATION_UNI_BUFFER
		is
		inherits UNI_BUFFER;
		ruled by ALTERNATION;
		where
		PUT, PUTLONG ⇒ FOP;
		GET ⇒ SOP;
		end
		ALTERNATION_UNI_BUFFER;

B)
class SERVER **is**
procedure FIRST;
procedure SECOND;
procedure THIRD;
end SERVER;
	PRIORITY	**class** PRIORITY.SERVER **is**
		inherits SERVER;
		ruled by PRIORITY;
		where
		K⇒3;
		FIRST ⇒OP[1];
		SECOND⇒OP[2];
		THIRD ⇒OP[3];
		end PRIORITY SERVER;

C)
class POLY__BUFFER **is**
procedure PUT(ITEM : **in**
ELEMENT) ;
procedure PUTLONG(ITEM :
in LONGELEMENT) ;
procedure GET(ITEM : **out**
ELEMENT) ;
function IS__FULL **return**
BOOLEAN ;
end POLY__BUFFER ;
	GUARDED	**class**
		GUARDED_POLY_BUFFER **is**
		inherits POLY_BUFFER;
		ruled by GUARDED;
		where
		PUT, PUTLONG ⇒ PT;
		GET ⇒ GT;
		IS_FULL ⇒ FULL_GUARD;
		end
		GUARDED_POLY_BUFFER;

D)
class MESSAGE_HANDLER
is
procedure
SEND(M:MESSAGE;
C:PRIORIY)
end MESSAGE_HANDLER
	MULTI_
	QUEUE
		class Q_MESSAGE_HANDLER
		is
		inherits MESSAGE_HANDLER
		ruled by
		MULTI__QUEUE(PRIORITY)
		where
		A(I) ⇒ SEND(M, C ⇒ I);
		end Q_MESSAGE_HANDLER;

Fig.6 - Examples of b-inheritance.

conditions of recursive definitions are presented in [Galli 1991].

A second solution uses universal quantification instead of recursion:

```
generic
K : POSITIVE;  -- the no. of priority groups;
behavioural class PRIORITY is
ruled MF enum;  -- keyword "enum" indicates that there is an ordered set of method groups,
   per(MFk)) ⇔ (><) and ∀1≤i<k: req_now(MFi)=0
end PRIORITY;
```

Going back to the priority problem, suppose now that priority does not depend on the method name *M*, but on the method caller, and that the caller priority is encoded by a parameter of the method: thus M(1) takes priority over M(2). The previous deontic specs are unsuitable since they do not have visibility of actual parameters. A solution to this problem [Goldsack and Atkinson 1990] is described in case 6 of Fig.5.

Next we consider the use of Boolean side-effect free methods (called *guards*) within deontic predicates. Guards are used to inspect part of the private state of an object. The f-class POLY__BUFFER (case C of Fig.6) provides two methods for writing data and one for reading. The behaviour is specified in Ex.4 of Fig.5: mutual exclusion, with the constraints that writers activation must be blocked when the buffer is full; this condition is checked by means of the guard FULL__GUARD. In Fig.6, a correspondence between the guard and the Boolean method IS__FULL is established. Note that, in the case of guards (or more generally state-inspecting functions), a formal method parameter cannot be bound to more than one actual method (Case C of Fig.6).
A further step is represented by the use of non-boolean functions in deontic expressions. Similar to guards, such functions could be used to inspect the state of the object.

Discussion

Fig.7 summarizes the pros and cons of each form of b-class specification. Case 1 suffers from two limitations: any constant in the deontic predicates is fixed at b-class definition time. If Ex.2 of Fig.5 were to be specified by a non generic b-class, different b-classes would have to be written for each value of NUM. The same problem occurs w.r.t. Ex.3 of Fig.5 which requires a varying number of method groups, one for each priority level. The advantages of case 1 are simplicity and reduced run-time overhead. Case 2 is perhaps the optimum w.r.t. expressivity, efficiency and reusability. Case 3 adds expressive power at the cost of increased abstraction and run-time overhead. Cases 4 and 5 permit to specify state variable dependent behaviour, (Ex.4 Fig.5) which cannot be simulated by previous cases. The use of state inspecting functions within the predicates makes somehow the b-class dependent on the signature of the f-class, thereby reducing reusability. The translation for this case is shown in section 3. Case 6 [Goldsack and Atkinson 1990] is also dynamic, since it allows to specify behaviours wich depend on actual values of methods parameters before they are stored into the object's state. Such message oriented specification is incomparable with cases 4 and 5. We have not developed the translation because we feel that this case is hard to understand in complex situations. In conclusion we recommend the use of genericity for b-classes, but we defer until feedbacks from experimentation will be available a judgement on the convenience of dynamic classes.
Notice that in b-classes guards, as other formal methods, are regulated by a deontic predicate. The safest solution is to activate a guard in mutual exclusion, but weaker conditions still ensuring consistency are proposed in [Galli 1991]. For instance in case 5 of Fig.5 the function MAX can be activated in parallel with any method of the set OP.
Since a guard is usually defined only for synchronization purposes and is not invoked by other objects, the question is when it should be evaluated. Conceptually a guard value

b-class	Limits	Pros/Cons
Non-Generic b-class *1 -Historical and state operators (see Fig.4) applied to formal methods*	Number of method groups: fixed at b-class definition time; Number of methods within a group fixed at b-inheritance time; Deontic predicates non-parametric;	Low reusability High run-time speed Simplicity
Generic b-class		
2- with respect to a parameter	Number of method groups: fixed at b-class definition time; Number of methods within a group fixed at b-inheritance time; Deontic predicates parametric w.r.t. a generic parameters	High reusability High run-time speed Simplicity
3- enumeration of method groups	Number of method groups: fixed at b-inheritance time; Number of methods within a group fixed at b-inheritance time; Deontic predicates parametric w.r.t. a generic parameters	Highest reusability Low run time speed High abstraction reduces readibility
Dynamic b-class *4,5- Formal functions (guards) returning the state of the formal object.*	Number of method groups: fixed at b-class definition time; Number of methods within a group fixed at b-inheritance time; Deontic predicates non-parametric	Lower reusability. The b-class is closely coupled to the features of the c-class. Some overhead introduced by the evaluation of the guards Possibility of run-time varying behaviour
Parametric b-class *6- Parameters of formal methods*	Number of method groups: fixed at b-class definition time; Number of methods within a group fixed at b-inheritance time Deontic predicates - parametric	Lower reusability High run-time speed The b-class is strictly coupled to the signature of the f-class.

Fig.7 - Analysis of features of b-classes. Combinations of cases are also possible.

should be continously monitored. In practice, evaluation is only required when the state of the object may change, that is each time a method activation terminates.

3. Translation of behavioural inheritance

Semantics of behavioural inheritance

The semantic of behavioural inheritance can be expressed using high level Petri Nets (E/R nets [Ghezzi et al 1991]). Tokens can have values associated to them. A transition is enabled depending on the value of a predicate associated to the input tokens. Thus we associate to each transition a predicate, and some actions that specify the value of the output tokens. If no actions are specified, a dummy (without a value) token is created.

In the initial configuration (Fig.8) there are several valued tokens denoted Y, but no dummy tokens. For instance we consider the b-class 4 of Fig.5, and b-inheritance with c-class (*C* of Fig.6) POLYBUFFER exporting the methods PUT, PUTLONG, GET and IS-FULL. The E/R net of b-inheritance is represented in Fig.8. The historical operators are translated into places containing a token with an associated value. The values are updated according to the actions when a transition fires. A bidirectional arrow stands for two simple ones. The places containing dummy tokens are used to represent the state of the graph and to enable the correspondent transitions. In Fig 9 we list the predicates and actions for the transitions in Fig.8. Notice that the resulting net has a behaviour which depends on the state of the sequential class, through the marking of the place STATE GUARD. For simpler, unguarded b-classes there is no such dependency. For further examples and discussion of E/R models of behaviour refer to [Galli 1991].

Translation into ADA

For Ada translation of a b-class, we examined the following options:

- Ada code is produced separately for b-classes, and for free c-classes and at b-inheritance time the two parts are linked together.
- at first only the Ada code for the free c-class is produced, and at b-inheritance time the code for the r-class is produced; no code is produced for the b-classes, which are not present in the library.

To compare the options, suppose that the c-classes C1 and C2 have to be regulated by the same b-class B, producing the r-classes R1 and R2. With the first option, there is one Ada component B●Ada for B (using tasks and possibly generics). There are no Ada components for R1 or R2, and each instance of R1 (or R2) is the link of C1●Ada (a purely sequential code) and B●Ada.

With the second option, there is no Ada component for B, but only a descriptor. After b-inheritance, two different Ada components for R1 and R2 are created.

Advantages of the first option are: consistency with the library organisation of DRAGOON; economy of components, since there is only one component per b-class; and reduced number of Ada compilations. The basic advantages of the second option is that code is more specific, hence more efficient. The decision is stil open, but we present an example of the first option, which has been analyzed in more detail. Notice that the hypotheses on domains of parents

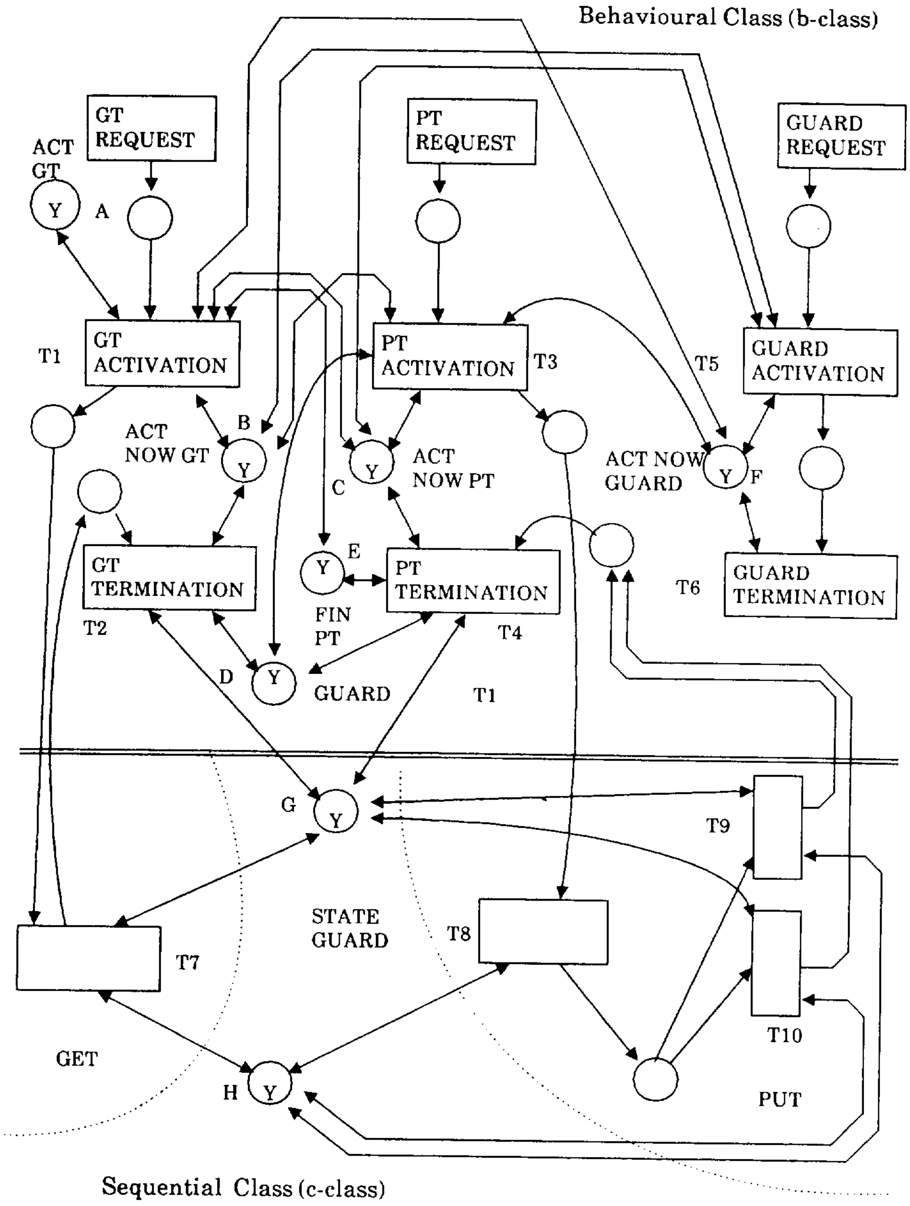

Fig.8 -Modeling a guarded behavioural class with Petri nets (see Ex.4 fig.5).

in behavioural inheritance (*vs* Fig.1 and 3 and previous discussion) are based on the first option.

A second choice that has been investigated concerns the assignment of a value to the formal parameters of a generic b-class; we examined the following options:

P T1:
A.Y<E.Y and
B.Y= 0 and
C.Y= 0 and
F.Y= 0
AC T1:
A.Y:=A.Y+1;
B.Y:=B.Y+1;
C.Y:=C.Y;
F.Y:=F.Y;
E.Y:=E.Y+1;
	P T3:
	not D.Y=True
	and
	B.Y= 0 and
	C.Y= 0 and
	F.Y= 0
	AC T3:
	D.Y=D.Y;
	B.Y:=B.Y;
	C.Y:=C.Y+1;
	F.Y:=F.Y;
		P T5:
		B.Y= 0 and
		C.Y= 0 and
		F.Y= 0
		AC T5:
		B.Y:=B.Y;
		C.Y:=C.Y;
		F.Y:=F.Y+1;

AC T2:
D.Y:=G.Y;
B.Y:=B.Y -1;
G.Y:=G.Y;
	AC T4:
	D.Y:=G.Y;
	C.Y:=C.Y -1;
	G.Y:=G.Y;
	E.Y:=E.Y+1;
		AC T6:
		F.Y:=F.Y -1;

AC T7:
G.Y:=False;
H.Y:=H.Y -1;
 	
AC T8:
H.Y:=H.Y+1;
	P T9:
	H.Y<Buffer_dim
	AC T9:
	G.Y:=False;
	H.Y:=H.Y;
		P T10:
		H.Y=Buffer_dim
		AC T10:
		G.Y:=True;
		H.Y:=H.Y;

Fig.9 - Predicates (P) and actions (AC) in the E/R net of Fig.8.
Transition without predicates are assumed TRUE by default.

-Assignment of values at b-inheritance time.

-Assignment of values at creation time of an instance variable of the corresponding r-class.

The second solution is attractive because it permits to have, after b-inheritance, one library component (generic Ada package) for every r-class. This component is customized w.r.t. the values of the formal parameters of the b-class (e.g. NUM in Ex.2 Fig 5) at object creation time. On the other hand, this forces the normal user to consider the structure of the b-class at every object creation action. Consider b-class PRIORITY (Ex.3 Fig 5) and a c-class C with methods that must be grouped into three priority groups. When creating an instance of C regulated by PRIORITY, the user must correctly specify the value 3 for the number of groups. Due to this drawback, we chose the first alternative, which increases the number of library components, but not the size of object code.

Next we describe the structure of the code. First recall the translation of sequential classes. As usual in O.O. languages, the translation of a method call, say PUT, for a free concrete class must account for run-time selection of the method body. In DRAGOON this takes the following form (called a *shell*):

```
procedure SHELL_PUT( ... ) is
begin
```

```
    ...
  PUT( ... );
    ...
 end SHELL__PUT;
```

where SHELL__PUT is a runtime system procedure which takes care of dynamic binding (polymorphism). Next consider the translation of the r-class:

```
 class ALTERNATION__UNI__BUFFER is
 inherits UNI__BUFFER;
 ruled by ALTERNATION;  --see Fig. 5 Ex.1
 where
  PUT, PUTLONG ⇒ FOP;
  GET ⇒ SOP;
 end ALTERNATION__UNI__BUFFER;
```

For r-classes, the previous scheme has to be modified, by enclosing a method call between two statements: the first is a call to a system procedure START__METHOD, the last one is a call to a system procedure END__METHOD.

```
  procedure SHELL__PUT( ... ) is
   begin
    GUARDED.START__METHOD(PT, ... );
    ...
    PUT(... );
    ...
    GUARDED.END__METHOD (PT, ... );
   end SHELL__PUT;
```

The code is organized as a generic package (since the b-class ALTERNATION is generic) GUARDED, defining the following major entities:

A procedure START__METHOD that interfaces to the entry of each method, via evaluation of the deontic predicate.

A procedure END__METHOD which is invoked by a method upon termination.

A task type BEHAVIOUR; an instance of the task is created at object creation time, for each instance of the r-class. The task offers two kinds of entries. In order to achieve synchronisation, the entries belonging to the first group are invoked by START__METHOD and the entries belonging to the second group by END__METHOD.

```
generic
 with FULL__GUARD return BOOLEAN;
 package GUARDED is
  type METHOD__SET is (PT, GT, FULL__GUARD);
  procedure START__METHOD(OP: METHOD__SET, REF: in DUMMY );
  procedure END__METHOD  (OP: METHOD__SET, REF: in DUMMY );
  private
   task type BEHAVIOUR is
     ...............
     entry START__PT;
     entry START__GT;
     entry START__FULL__GUARD;
     entry END__PT;
     entry END__GT;
     entry END__FULL__GUARD;
  end BEHAVIOUR;
end GUARDED
```

```
package body GUARDED is
 procedure START__METHOD(OP:METHOD__SET, -......) is
  begin
   case OP is
    when PT       = > REF.START__PT;
    when GT       = > REF:START__GT;
    when FULL__GUARD = > REF.START__FULL__GUARD;
   end case;
  end ;
 procedure END__METHOD(OP:METHOD__SET, ....) is
  begin
   case OP is
    when PT       = > REF.END__PT;
    when GT       = > REF.END__GT;
    when FULL__GUARD = > REF.END__FULL__GUARD;
   end case;
  end;
  .................
 task body  BEHAVIOUR is
    ACT__GT       : NATURAL : =0;      --the numbers correspond to the
    FIN__PT      : NATURAL : =0;       --historical functions
    ACT-NOW       : NATURAL : =0;
    FULL__GUARD__FLAG: BOOLEAN : = FALSE;
 begin
  accept INIT(.......) do
   .............
  end;
  FULL__GUARD__FLAG : = FULL__GUARD;
  loop
   select
    when  ACT-NOW=0 and ACT__GT < FIN__PT = > --translation of the first axiom
       accept START__GT do
         ACT-NOW : = ACT-NOW + 1;          --update of the historical functions
         ACT__GT : = ACT__GT + 1;
     end;
   or
    when  ACT-NOW=0 and not FULL__GUARD__FLAG = > --translation of the
       accept START__PT do                                  --second axiom
         ACT-NOW : = ACT-NOW + 1;          --update of the historical functions
       end;
    or
    when  ACT-NOW=0 = >                   --translation of the third axiom
       accept START__FULL__GUARD do
         ACT-NOW : = ACT-NOW + 1;          --update of the historical functions
       end;
    or
       accept END__GT do                 --termination of the method GET
         ACT-NOW : = ACT-NOW - 1;         --update of the historical functions
         FULL__GUARD__FLAG : = FULL__GUARD;
       end;
    or          accept END__PT do          --termination of the method PUT
         ACT-NOW : = ACT-NOW - 1;         --update of the historical functions
         FIN__PT : = FIN__PT + 1;
         FULL__GUARD__FLAG : = FULL__GUARD;
       end;
    or
       accept END__FULL__GUARD do   --termination of the evaluation of
         ACT-NOW : = ACT-NOW - 1;         --the guard
       end;
   end select;
```

```
      end loop;
     end BEHAVIOUR;
  end GUARDED;
```

Historical operators (see Fig.4) are naturally mapped onto counters. For each method group in the b-class there is an entry for START and another for END. Each entry of kind START is guarded by a deontic predicate; upon acceptance, historical counters are updated. Entries of kind END are called when a method finishes; no predicate is needed, but only historical counters are updated and values returned by guards or state-inspecting functions (if any) are updated. Notice that the task accesses FULL_GUARD without going through the method selection shell.

The previous translation present a serious problem for run-time performance: the task is always busy waiting inside the loop containing the *select* statement. This causes much overhead for systems with many behavioured objects, since each one corresponds to an active task. To avoid the problem, the *select* statement is expanded with an *else* branch, which is entered when no other entry call is accepted. The *else* branch put the task to sleep. The task is awakened by the next incoming call to START or to END a method.

The translation of behaviours with a generic number of method groups (e.g. Ex. 3 in Fig.5) is more complex, and the reader should refer to [Galli 1991] for more details. Essentially, for each group there are two families of task entries in the task BEHAVIOUR. An analysis of the translation for the various b-classes (Fig.7) shows that generic b-classes with a varying number of formal method groups introduce the highest run-time overhead.

The above translation is not essentially dependent on the Ada tasking model, and a similar solution could be worked out for other multi-task systems (e.g. Unix).

4. Conclusion

In our opinion the separate specification of functional aspects and method synchronization constraints (behavioural aspects) is necessary for the orderly construction of large, reusable collections of components. Behavioural inheritance provides a coherent, "divide and conquer" approach. We have investigated a range of abstract notations for expressing behaviour which differ in genericity, degree of reuse, and run-time efficiency; they are all based on deontic logic, a rigorous yet not too cryptical notation which we found suitable for the representative cases we considered.

The proposal needs now to be validated by experience. In particular the cost effectiveness of the deontic notations incorporating guards or method parameters has to be assessed. In fact on one hand it certainly enlarges the range of expressible behaviours, but on the other hand it introduces a tighter coupling with the functional class interface. In the worst case this could defeat the very objective of having reusable abstract behaviours.

Another critical issue to be further investigated is performance of implementation. The generated code attempts to minimize tasking overhead, but this could prove insufficient for large, heavily constrained real-time systems.

Finally we mention a problem to be considered: specification of time constraints.

Aknowledgement

Most of the ideas presented have emerged from joint work on DRAGOON. In particular we thank C.Atkinson, R.Bayan, M.De Michele, C.Destombes, A.Di Maio, S.Goldsack; S.Morasca for the E/R net model, and M.Paci for helpfull suggestions.

5. References

America P. : "POOL-T: A parallel object oriented language", in *Object oriented concurrent programming*, MIT Press, pp. 199-220, 1989

Atkinson C. : "An object-oriented language for software reuse and distribution", *Ph.D. Thesis,* Imperial College, London, 1990.

Campbell, R.H. and Habermann A.N. : "The specification of process syncronisation by path expression," *ACM Computer Survey*,17(4), 1974.

Cardigno C. et al : "Object Oriented Concurrency and Behavioural Inheritance", *Proc. ECOOP'89 Workshop on Object-Based Concurrent Programming*, Nottingham, July,1989.

Di Maio A. et al. : "Dragoon: An Ada-Based Object Oriented Language for Concurrent, Real-Time, Distibuted Systems", *Proc. Ada-Europe Conference*, Madrid 1989.

Galli de' Paratesi G. : "Specifiche di concorrenza per ADA orientato ad oggetti", *Thesis* Dipartimento di Elettronica - Politecnico di Milano(Draft), 1991.

Ghezzi C. et al. : "A unified high level Petri net for time-critical system", *IEEE Transaction on Software Engineering,* February 1991.

Goldsack S.J.,Atkinson C.: "Separating concerns for synchronisation and functionality in an Object-Oriented Language", submitted for publication, 1990.

Von Wright G.H.: "Problems and Prospects of Deontic Logic: A Survey", in *Modern Logic - A Survey: Historical, Philosophical and Mathematical Aspects of Modern Logic and its Applications*, (Agazzi E. ed.), Reidel Publishing Company, 1980.