Eiffel ||

Inheritance of operations and inheritance of synchronisation constraints in Eiffel || [2] are independent of each other. In other words, a change in synchronisation constraints does not necessitate a rewrite of the operations, and visa-versa. Unfortunately, the monolithic approach of specifying all synchronisation constraints in a single routine, *Live*, means that there is no independence of constraints: changing one synchronisation constraint requires rewriting them all. The net result is that Eiffel || cannot handle the column 3 of the inheritance matrix. Whenever any synchronisation constraints change, the programmer has to rewrite the *Live* routine completely. This is quite unfortunate since the *Live* routine is often long and it is tedious not being able to reuse it.

Summary

Enabled-Sets was shown to have very poor inheritability. We feel that this is because Enabled-Sets mixes synchronisation code in with the sequential code of operations. All the other mechanisms we evaluated have a clean separation between synchronisation constraints and operations and consequently have better inheritability. Of these, the best inheritability was found in Guide and Scheduling Predicates, both of which allow individual constraints to be inherited/rewritten independently of each other.

6 Conclusions

The conflict between synchronisation and inheritance has too many forms for it to be comprehensively presented in a single example. This paper has defined an *inheritance matrix* which allows us to view the conflict in all of its many forms. We then discussed how the inheritance matrix can be used to evaluate the inheritability of synchronisation mechanisms. Finally, we presented some preliminary results from evaluating several synchronisation mechanisms in this manner.

What we have learned suggests two important requirements for a synchronisation mechanism:

1. Synchronisation constraints should be separated from the sequential code of operations. This facilitates their independent inheritance/redefinition.
2. It should be possible to inherit/redefine individual synchronisation constraints independently of one another.

References

[1] Colin Atkinson. *An Object-Oriented Language for Software Reuse and Distribution.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, London SW7 2BZ, England, February 1990.

[2] Denis Caromel. Concurrency: An Object-Oriented Approach. In Jean Bézivin, Bertrand Meyer, and Jean-Marc Nerson, editors, *TOOLS 2*, pages 183–197. Angkor, 1990.

[3] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin, and X. Rousset de Pina. A Synchronisation Mechanism for an Object Oriented Distributed System. In *Proceedings of the 11th International Conference on Distributed Computer Systems*, Arlington, Texas, February 1990.

[4] Ciaran McHale, Bridget Walsh, Seán Baker, and Alexis Donnelly. Scheduling Predicates. Technical Report TCD-CS-91-24, Department of Computer Science, Trinity College, Dublin 2, Ireland, October 1991. Presented at the workshop on Object-Based Concurrent Programming at ECOOP '91, Geneva, Switzerland. Proceedings to appear in a volume of *Lecture Notes in Computer Science*, Springer Verlag.

[5] Chris Tomlinson and Vineet Singh. Inheritance and Synchronisation with Enabled-Sets. In *OOPSLA '89 Proceedings*, pages 103–112, October 1989.

- The impression that retyping is better than reuse arises because synchronisation counters are a compact way of expressing constraints. There would be more incentive to reuse code in a more verbose synchronisation mechanism.
- A Eiffel-like "flat" utility could be used to allow programmers to easily see the effective synchronisation constraints on a class, even when some of the constraints are inherited from ancestor classes.

DRAGOON

The behavioural classes of DRAGOON [1] are, in effect, *generic* classes. (As far as we know, DRAGOON is the first language to support the concept of generic synchronisation policies.) One potential advantage of having generic synchronisation classes is that a core of such classes can form part of the standard library for a language. In this way, the synchronisation code of a class can be self-documenting. Unfortunately, the syntax used in DRAGOON to instantiate a synchronisation policy is too verbose—there are as many lines of code used to instantiate behavioural class as there are in the behavioural class itself.

DRAGOON handles all the cells in the inheritance matrix in an oblique manner: it maps the entire matrix into a single column.[5] This can prove tiresome for the programmer and lead to DRAGOON programs being verbose but, other than that, there are no severe difficulties.

An interesting result of not being able to inherit from a synchronised class is that, although **student** may be a subclass of **person**, **synchronised_student** is not a subclass of **synchronised_person**, according to the class hierarchy. We do not know if this irregularity has any ramifications for program development.

Of a more practical nature, having to write a sequential class and then inherit from it simply to apply synchronisation constraints leads to name-space pollution. We overcame this by giving all classes a standard prefix to indicate if they were a sequential or synchronised class.

Enabled-Sets

Synchronisation constraints are embedded in the sequential code of operations and this results in several conflicting interactions between the inheritance of synchronisation constraints and inheritance of operations:

Column 4 Inheriting from an unsynchronised class and applying synchronisation constraints requires an incremental re-implementation of each operation to allow the constraints to be added.

Column 2 Changing the synchronisation policy on a class results in having to rewrite all the operations, even if the functionality of the operations remains unchanged.

Row 2 Similarly, a total re-implementation of an operation requires rewriting the synchronisation constraint for that operation.

Row 3 Incremental change in the code of an operation is sometimes possible. But it is just as likely that the embedded synchronisation constraint will interfere with this, resulting in the operation having to be totally re-implemented.

In fact, the only cells in the inheritance matrix which Enabled-Sets provides support for are $(1, 1)$, which represents the trivial case where nothing changes, and the combination of cells $(1, 3)$ and $(4, 4)$ which is represented by the *ExtendedBoundedBuffer* example.

[5] Depending on your point of view, DRAGOON either maps everything into column 2 (synchronisation constraints must be re-implemented from parent to child class) or column 4 (synchronisation constraints are never re-implemented on top of an existing synchronised class; rather they are (re)applied to the unsynchronised equivalent).

5 Results

We have used the inheritance matrix to evaluate the inheritability of several synchronisation mechanisms, by attempting to solve a set of examples in each mechanism. This section summaries the initial results we have obtained from these evaluations.

Guide

Guide [3] has no conflicting interaction between inheritance of operations and inheritance of synchronisation constraints. In other words, a change in synchronisation constraints does not necessitate a rewrite of the operations, and visa-versa. Also, the synchronisation constraints are independent of each other. Thus, changing one constraint does not necessitate rewriting the other constraints.

The only limitation of Guide is that while synchronisation constraints can be inherited without modification or changed totally, they cannot be *incrementally* changed. Thus, Guide cannot handle column 3 of the inheritance matrix directly. Instead, the programmer must effect incremental changes in a synchronisation constraints by rewriting the affected constraints totally. Thus, column 3 of the matrix is handled by shifting it onto column 2.

Scheduling Predicates

Scheduling Predicates [4] can be thought of as an extension of Guide's synchronisation mechanism. In particular, Scheduling Predicates provides support for incremental changes to synchronisation constraints. This is achieved by use of the keyword **inherit** in constraints. For example:

Foo: **inherit and** *exec*(Bar) = 0;

Frequently, synchronisation constraints on existing operations are incrementally modified to take into account a new operation. In such cases, the constraint on the new operation will frequently be similar to a constraint on one of the existing operations. In such cases, it is useful for the new operation to inherit a constraint from an existing operation. The following example will show how Scheduling Predicates allows this.

Consider a class which has two operations, X and Y, which execute in mutual exclusion. The synchronisation constraints would therefore be:

X, Y: $exec(\mathrm{X}, \mathrm{Y})^4 = 0$;

Now consider a subclass which introduces a new operation, Z. A first attempt at writing the synchronisation constraints for this subclass would be:

X, Y: **inherit and** *exec*(Z) = 0;
Z: **inherit** X **and** *exec*(Z) = 0;

This can be more compactly expressed as:

X, Y, Z: **inherit** X **and** *exec*(Z) = 0;

This support for incremental change of synchronisation constraints means that, unlike Guide, Scheduling Predicates can easily handle column 3 of the inheritance matrix.

A problem we found with both Guide and Scheduling Predicates is that if some constraints are expressed in a class but others are expressed in an ancestor class then it can be difficult to understand the overall synchronisation policy. We sometimes found it easier to restate all the constraints explicitly in a class, rather than inherit some of them. In other words, *sometimes it is better to retype than to reuse.* This leads us to then wonder if inheritance of synchronisation constraints is of much importance. Some points to note about this are:

[4] Note that the form $exec(A, B, \ldots, Z)$ is a shorthand for $exec(A) + exec(B) + \cdots + exec(Z)$.

From Figure 3 we see that the *ExtendedBoundedBuffer* example illustrates only a small subset of the interactions between synchronisation and inheritance. Hence we see the limitation of using a single example to explain the problem.

4 The Inheritance Matrix as an Evaluation Tool

The preceding sections have introduced the inheritance matrix as a graphical way to define the, potentially conflicting, interaction between inheritance of operations and inheritance of synchronisation constraints. However, the matrix can be used for more than just defining the problem: it can also be used as a tool to evaluate how complete a particular synchronisation mechanism's solution to the problem is.

Leaving aside the three impossible cases, the matrix has thirteen cells. The *ExtendedBoundedBuffer* (discussed in section 3) occupies just two of these. If we can fill the remaining eleven cells with other examples then we will have a set of examples which illustrate *all* the possible interactions between inheritance of operations and inheritance of synchronisation constraints.

Attempting to implement such a set of examples with a particular synchronisation mechanism will show up the strength and weaknesses of that mechanism's ability to interact with inheritance. Repeating this exercise for several different synchronisation mechanisms allows us to compare them on an uniform basis.

Filling the Matrix

Initially we had hopes of filling the matrix with a single set of examples which could be used to evaluate the inheritability of all synchronisation mechanisms. However, the expressive power of synchronisation mechanisms varies widely. Also, some mechanisms support intra-object concurrency while others only work with single-threaded active objects. Because of these two factors, we have not yet been able to find a universal set of examples which can be used to evaluate all mechanisms.

Instead, we have chosen examples on an as-needed basis for the different synchronisation mechanisms we have evaluated. We had some reservations about taking this approach, reasoning that evaluating, say, Guide with one set of examples and Enabled-Sets [5] with another might not provide a common basis to compare the two mechanisms. However, this reservation proved to be without foundation. In practice, evaluating a mechanism with one set of examples will yield the same results as evaluating it with a different set.[3]

Unfortunately we have space to present just one of the examples which we used to evaluate the inheritability of synchronisation mechanisms. (We will be happy to discuss the other examples at the workshop.) Figure 4 shows a base class, *Foo*, whose operations execute in mutual

```
class Foo is
        Write() is ...
        Read() is ...
synchronisation
        Write: exec(Read, Write) = 0;
        Read: exec(Read, Write) = 0;
end Foo;
```

```
class Bar is inherit Foo;
        // no change to operations
synchronisation
        // total change in constraints
        Write: start(write) = term(Read);
        Read: term(write) > start(Read);
end Bar;
```

Figure 4: Example for cell (1, 2) of the inheritance matrix

exclusion. Class *Bar* inherits *Foo* and applies different synchronisation constraints (to implement an alternation policy) while retaining the original code of the operations. This example fills in cell (1, 2) of the inheritance matrix.

[3] This is assuming that the synchronisation mechanism has enough expressive power to implement all the examples in either set.

Synchronisation Constraints / Operations	No Change In Constraint	Total Change In Constraint	Incremental Change of Constraint	New Constraint
No Change in Operation	(1, 1)	(1, 2)	(1, 3)	(1, 4)
Total Reimplementation Of Operation	(2, 1)	(2, 2)	(2, 3)	(2, 4)
Incremental Reimplementation Of Operation	(3, 1)	(3, 2)	(3, 3)	(3, 4)
New Operations				(4, 4)

Figure 2: The Inheritance Matrix

The introduction mentioned that attempts to use synchronisation and inheritance together often leads to problems. These problems can be said to be *conflicting interactions* between inheritance of operations and inheritance of synchronisation constraints. The inheritance matrix is an ideal tool for visualising the different types of interaction.

An Example: The Extended Bounded Buffer

Consider a class which inherits from a *BoundedBuffer* class and introduces a new operation, *GetRear* [5]. There is no change in the code of the two inherited operations, *Put* and *Get*, but their synchronisation constraints must be modified to take into account the new operation. Thus, *Put* and *Get* belong in cell $(1, 3)$ in the inheritance matrix (Figure 3). The new operator, *GetRear*, is given a new constraint and so it belongs in cell $(4, 4)$.

Synchronisation Constraints / Operations	No Change In Constraint	Total Change In Constraint	Incremental Change of Constraint	New Constraint
No Change in Operation			Put, Get	
Total Reimplementation Of Operation				
Incremental Reimplementation Of Operation				
New Operations				GetRear

Figure 3: The Inheritance Matrix for the *ExtendedBoundedBuffer* example

Inheritance of Synchronisation Constraints

The previous section examined the sequential operations of a class without regard to synchronisation. In this section, we will do the opposite, i.e., examine synchronisation in isolation from sequential code. (We will discuss the interaction of sequential code and synchronisation constraints in Section 3.)

Just as a *class* is a collection of related operations (and data), so one can consider a *synchronisation policy* to be a collection of related synchronisation constraints. The following example shows a mutual exclusion policy defined over two operations:

sync_policy Mutex {
    W: "no executing R's or W's";
    R: "no executing R's or W's";
}

Just as we can inherit from sequential classes and, optionally, re-implement operations, so too can we inherit from a synchronisation policy and re-implement individual constraints at will. For example, the following *ReadersWriter* policy is obtained by inheriting from *Mutex* and changing the constraint on *R*:

sync_policy ReadersWriter **inherit** Mutex {
    // no change to the constraint on W (1)
    R: "no executing W's"; (2)
}

We could again inherit from this to make the synchronisation policy take account of a second read operation:

sync_policy ExtendedRW **inherit** ReadersWriter {
    // no change to the constraint on R (1)
    W: inherit **and** "no executing R_2's"; (3)
    R_2: "no executing W's"; (4)
}

In brief, inheritance of synchronisation constraints results in the following possibilities:

(1) No change in a constraint

(2) A total change in a constraint

(3) Incremental change in a constraint

(4) Introduction of a new constraint

We note that the possibilities for inheritance of constraints in a hierarchy of synchronisation policies mirror those for the inheritance of operations in a class hierarchy.

3 Problem Definition

Since the previous sections have listed the different possibilities for these two types of inheritance, we can easily express their interaction in the form of an *inheritance matrix* as shown in Figure 2. (The cells in the matrix are numbered for ease of reference.) For example, cell (1, 2) represents the case where the functionality of inherited operation does not change but it is given a new synchronisation constraint. The three shaded cells in the matrix represent impossible cases.[2]

[2]If a subclass introduces a new operation, ***Foo***, then it must be given a synchronisation constraint (if not ***explicitly*** given a constraint then it will ***implicitly*** adopt the default constraint of the synchronisation mechanism, whatever that may be). It is impossible for ***Foo*** to have it's constraint changed since ***Foo*** did not exist in the super class.

Evaluating Synchronisation Mechanisms: The Inheritance Matrix

Ciaran McHale[1], Bridget Walsh,
Seán Baker, Alexis Donnelly

1 Introduction

It is recognised that attempts to use synchronisation and inheritance together often leads to problems. In most systems reported in the literature, the problem is illustrated by the *Extended Bounded Buffer* example [5].

Naturally, a single example cannot illustrate the conflict in all its varied forms, and thus it gives us with only a *partial* understanding of the problem; and if one has only a partial understanding of a problem then one can only hope to find a partial solution.

We accomplish several tasks in this paper. First, we will define the conflict between synchronisation and inheritance. We then show how this definition can be used to derive a set of examples which can be used to test the *inheritability* of a synchronisation mechanism. Finally, we present some preliminary results obtained from using these examples to evaluate several synchronisation mechanisms.

2 Inheritance

Inheritance of Operations

In a sequential object-oriented language, operations may be inherited and, optionally, reimplemented. The classes in Figure 1 illustrate the various ways in which this might happen. In brief, inheritance of operations results in the following possibilities:

- No change in an operation
- A total re-implementation of an operation
- Incremental re-implementation of an operation (i.e., *Foo* invokes *super.Foo*)
- Introduction of a new operation

```
class C1 {
        A() {...};
        B() {...};
        C() {...};
}
```

```
class C2 inherit C1 {
        // no change to A()
        B() {...}; // total change
        C() { // incremental change
                ...
                super.C();
        }
        D() {...}; // new operation
}
```

Figure 1: Possibilities for inheritance of operations

[1]Authors' address: Department of Computer Science, Trinity College, Dublin 2, Ireland.
Tel: +353-1-7021539 Email: {cjmchale,bwalsh,baker,donnelly}@cs.tcd.ie