Electronic Notes in Theoretical Computer Science 82 No. 8 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 3 pages

Temporal Constraints and Concurrent Objects

Extended Abstract

Giuseppe Milicia[1] and Vladimiro Sassone[2]

BRICS[1], *University of Aarhus, Denmark*⋆

COGS[2], *University of Sussex – UK*

1 Introduction

It is well known that the coexistence of inheritance and concurrency in object-oriented languages is not without problems [2,1]. Matsuoka and Yonezawa coined the term *inheritance anomaly* to refer to such problems [3].

Commonly, in object oriented code, the set of messages accepted by an object is not uniform in time. Depending on the object's state, some of its methods will be unavailable, as e.g., `pop` from a empty stack, or `put` on a full buffer. To avoid inconsistencies in a concurrent setting, programmers use *synchronization code* to implement the required constraints. However, the redefinition of synchronization behaviour in sub-classes often forces the redefinition of entire methods. The inheritance anomaly nullifies the code-reuse promised by inheritance. The situation can be exemplified in a simple case by the following idealized pseudo-code of a buffer.

```
class Buffer {
 ...
    void put(Object el) { if ("buffer not full") ... }

    Object get() { if ("buffer not empty") ... }
}
```

If we wish to derive a class `HistoryBuffer` providing the additional method `gget` which removes an element only if the last operation *was not a* `get`, we are most likely forced to redefine the inherited methods to do some book-keeping. We have an instance of *history-sensitive* inheritance anomaly.

Generally speaking, the inheritance anomaly has been classified in three broad varieties [3]: *partitioning of states*, *modification of acceptable states* and *history-sensitiveness of acceptable states*. This latter is relevant to languages based on *method guards* such as Java and C# and it is the variety of the anomaly we wish to address.

⋆ Center of the Danish National Research Foundation

Although modern programming languages provide concurrency and inheritance, the inheritance anomaly is most commonly ignored. Indeed, Java and C# are mainstream concurrent object oriented languages whose synchronization primitives are based exclusively on (a non declarative use of) locks and monitors.

2 A bit of Jeeg

Jeeg [4] is a dialect of Java related to the aspect oriented programming paradigm. Synchronization constraints, expressed declaratively as guards, are totally decoupled from the body of the method, so as to enhance separation of concerns.

A typical Jeeg program has the following form:

```
public class MyClass {
   sync { m : φ; ...  }

   // Standard Java class definition
   public ... m(...) { ... }
  ...
}
```

Intuitively, `m` : ϕ means that at a given point in time a method invocation `o.m()` can be executed if and only if the guard ϕ evaluated on object `o` yields *true*. Otherwise, the execution of `m` is *blocked* until ϕ becomes *true*. The novelty of the approach is that guards are expressed in (a version of) Linear Temporal Logic [5] (LTL), so as to allow expressing properties based on the history of the computation. Exploiting the expressiveness of LTL, Jeeg is able to single out situations such as the `HistoryBuffer` class described in the introduction, thus ridding the language from the corresponding anomalies. The syntax of the LTL variation we use is the following:

$$\phi ::= \texttt{AP} \mid !\phi \mid \phi \&\& \phi \mid \phi \| \phi \mid \mathit{Previous}\ \phi \mid \phi\ \mathit{Since}\ \phi$$

We can then code the `HistoryBuffer` example as:

```
public class HistoryBuffer extends Buffer {
   sync { gget: (Previous (event != get)) && (! empty); }

   public HistoryBuffer(int max) { super(max); }

   public Object gget() throws Exception { ...  }
}
```

Expressiveness

Due to the nature of the problem, it is of course impossible to claim formally that a language avoids the inheritance anomaly or solves it. The matter

depends on the synchronization primitives of the language of choice, and new practice in object oriented programming may at any time unveil shortcomings unnoticed before and leading to new kinds of anomalies. Nevertheless, since the expressive power of LTL is clearly understood, one of the pleasant features of Jeeg is to come equipped with a precise characterization of the situations it can address. More precisely, all anomalies depending on sensitivity to object histories expressible as star-free regular languages can, in principle, be avoided in Jeeg.

Implementation

The current implementation of Jeeg relies on the large body of theoretical work on LTL, that provides powerful model checking algorithms and techniques. Currently, each method invocation incurs an overhead that is linear in the size of the guards appearing in the method's class. Also, the evaluation of the guards at runtime requires mutual exclusion guarantees that have a (marginal) computational cost. When compared with the benefit of a substantially increased applicability of inheritance, we feel that this is a mild price to pay, especially in the common practical situations where code overriding is infeasible or cost-ineffective.

We are currently working on alternative ways to implement the ideas of Jeeg, aiming both at a lower computational overhead and at more expressive logics.

The reader interested in the details is referred to [4]. The current Jeeg implementation is available as open-source from `http://sourceforge.net/projects/jeegc/`.

References

[1] P. America. POOL: Design and experience. *OOPS Messenger*, 2(2):16–20, Apr. 1991.

[2] J. P. Briot and A. Yonezawa. Inheritance and synchronization in concurrent OOP. In *European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *LNCS*, pages 32–40. Springer-Verlag, 1987.

[3] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming language. In A. Gul, W. Peter, and Y. Akinori, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[4] G. Milicia and V. Sassone. Jeeg: A Programming Language for Concurrent Objects Synchronization. In *Proceeding of JavaGrande/ISSCOPE 2002*, pages 212–221, November 2002.

[5] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, Oct. 31–Nov. 2 1977. IEEE Computer Society Press.