Jeeg: A Programming Language for Concurrent Objects Synchronization

Giuseppe Milicia
BRICS - Basic Research in Computer Science
University of Aarhus, Denmark
milicia@brics.dk

Vladimiro Sassone
School of Cognitive and Computing Sciences
University of Sussex, United Kingdom
vs@susx.ac.uk

ABSTRACT

We introduce Jeeg, a dialect of Java based on a declarative replacement of the synchronization mechanisms of Java that results in a complete decoupling of the 'business' and the 'synchronization' code of classes. Synchronization constraints in Jeeg are expressed in a linear temporal logic which allows to effectively limit the occurrence of the inheritance anomaly that commonly affects concurrent object oriented languages. Jeeg is inspired by the current trend in aspect oriented languages. In a Jeeg program the sequential and concurrent aspects of object behaviors are decoupled: specified separately by the programmer these are then weaved together by the Jeeg compiler.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures, Inheritance*; D.1.3 [**Programming techniques**]: Concurrent Programming

General Terms

Languages, Design

Keywords

Java, Inheritance Anomaly, Temporal Logic

1. INTRODUCTION

In the late eighties, the first experiments in mixing object oriented programming languages and concurrency unveiled serious difficulties in merging the two concepts [1, 3]. Typically, the code for concurrency control, interwoven in the business code of classes, represented an obstacle to code inheritance, making it essentially impossible even in simple, common situations. The term *inheritance anomaly* [16] was coined to refer to the issue. Indeed, the problems arising from the interaction of inheritance and concurrency were considered so severe as to suggest removing inheritance from concurrent object oriented languages entirely [1].

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. In sequential situations, it is sometimes conceivable for clients to keep track of which methods are enabled and which are not. For instance, it could be required of the stack user to know at any given point in time whether or not the stack is empty. In a concurrent scenario, however, this is clearly not an option. Clients have no way of knowing about other clients, and any cooperation in this respect requires non-trivial, specific protocols. Our only option is to interweave the stack code with code that controls access from clients. Concurrent objects must take direct control of their synchronization code, and the phenomenon of inheritance anomaly sets in, forcing programmers to override inherited code in order to refine the synchronization code therein. 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") ...
 }
}
```

Suppose now that to enhance `Buffer` we wish to add, for instance, method `freeze` that makes it read-only. Whatever the original chunks of code for "*buffer not ...*", chances are that they must be totally rewritten to take into account the new enabling condition.

Generally speaking, the inheritance anomaly has been classified in three broad varieties [16] that we review below.

Partitioning of states. Inspired by the example above, one may disentangle code and synchronization conditions by describing methods enabling according to a partition of the object's states. To describe the behavior of class `Buffer`, for instance, the state can be partitioned in three sets: `empty`, `partial`, and `full`, the former containing the states in which the buffer is empty – so that `get` is inhibited – the latter those in which it is full – so that is `put` to be disallowed.

JGI'02, November 3–5, 2002, Seattle, Washington, USA.

One can then specify

```
put: requires not full
get: requires not empty
```

and refine the code of `get` and `put` to specify the state transitions. For instance, `get` would declare the conditions under which the buffer becomes `empty` or `partial`:

```
Object get() {
  ...
  if ("buffer is now empty") become empty;
  else become partial;
}
```

The inheritance anomaly here surfaces again, as derived classes may force a refinement of the state partition. As an example, consider adding a method `get2` that retrieves two elements at once. Alongside `empty` and `full`, it is necessary to distinguish those states where the buffer contains exactly one element. Clearly, the state transitions specified in `get` and `put` must be re-described accordingly.

History-sensitiveness of acceptable states. When method enabling depends on a object's past history rather than, as above, on its object's state, a different form of inheritance anomaly occurs. Suppose for instance that we want to refine our buffer with a method `gget` that works like `get` but that cannot be executed immediately after a `get`. Clearly, that can only be achieved in Java adding code to `get` to keep track of its invocations. That is, we have to rewrite the entire class. We will revisit this problem later on. A similar situation arises enforcing a password expiration policy whereby the method `login` cannot be executed k times without an intervening call to `passwd`.

Modification of acceptable states. A third kind of anomaly happens with mix-in classes, that is classes created to be mixed-into other classes to add to their behavior. The typical situation arises when one wishes to enrich a class with a method that influences the acceptance states of the original class' methods. Our previous example of the method `freeze` belongs essentially to this category of anomaly. Similarly, it is reasonable to expect to be able to design a

```
class Lock {
  ...
  void lock()   { ...; }
  void unlock() { ...; }
}
```

to be used to add lock capabilities to clients classes by means of the standard inheritance mechanism. But, clearly enough, (multiple) inheritance of `Lock` and `Buffer` does nothing towards creating a lockable buffer, unless we completely recode `get` and `put` to keep into account the state of the `Lock` component of the object.

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 procedural use of) locks and monitors.

Although no generally accepted solution has emerged so far, several approaches have appeared in the literature that mitigate the inheritance anomaly. Our proposal, Jeeg, focuses on Java. Jeeg is a dialect of Java based on method guards whose particularity is to address history-sensitive inheritance anomaly. As in guard based languages, methods are labeled by formulae that describe their enabling condition. The novelty of the approach is that we use (a version of) Linear Temporal Logic [22] (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 those described in the examples above, thus ridding the language from the corresponding anomalies. 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, we will see that all anomalies depending on sensitivity to object histories expressible as star-free regular languages can, in principle, be avoided in Jeeg.

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

Jeeg is related to the aspect oriented programming (AOP) paradigm. Synchronization constraints, expressed declaratively, are totally decoupled from the body of the method, so as to enhance separation of concerns.

The structure of the paper is as follows: §2 presents the language, while §3 cures the classical inheritance anomalies with it; §4 treats the expressive power of Jeeg. More details on the language and its current implementation are provided respectively in §5 and §6. Finally, we discuss related and further work.

2. A TASTER OF JEEG

Jeeg differs from Java for the use of new synchronization primitives which replace the `wait()`, `notify()`, and `notifyAll()` constructs. In Jeeg the synchronization code of a class *is not* inlined in its methods; rather it is specified separately. This can be done either via a `sync` section of the class definition or via an XML file associated with the class (see [19]). In the former case, a Jeeg class has the following structure:

```
public class MyClass {
   sync {
      ....
   }
   // Standard Java class definition
  ...
}
```

The `sync` section consists of a sequence of declarations of the form:

```
m : φ;
```

where `m` is a method identifier and ϕ, the *guard*, is a formula in a given *constraint* language to be described shortly. Methods associated with a guard are said *guarded*. 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 resumption of `m` at that point follows the familiar rules of the Java `notifyAll` primitive. Guarded methods are executed in mutual exclusion at the level of objects. Indeed, from a Java perspective, every guarded method is implicitly synchronized. Synchronization constraints in Jeeg are thus exclusively at the *method level*: there is no `synchronized` keyword and it is not possible to define guarded regions. Although it may appear that our synchronization mechanism becomes too coarse, please note that synchronized regions can always be refactored into synchronized methods with an overhead which is negligible in most situations.

The expressive power of this model of synchronization depends of course on the choice of the constraint language. Indeed, if we limit ϕ to Java boolean expressions we obtain a declarative version of the standard synchronization mechanism of Java.

2.1 The constraint language

Choosing the constraint logic is a trade-off between expressiveness and efficiency, as the truth of formulae must be verified at every method invocation. We need a logic more expressive than Java boolean expressions that, however, does not substantially worsen the computational cost of formula evaluation. A logic that suits our purpose is *linear temporal logic* (LTL) [22]. As we shall see, (a variation of) LTL used in the context of Jeeg gives a substantial improvement on the expressiveness of Java boolean expressions, tackling in particular the history-sensitive inheritance anomaly, while keeping the overhead on evaluation time on the linear scale.

LTL introduces time in propositional and first order logic. It becomes possible to reason about dynamic, evolving systems by expressing properties referring to what happened in the past or to what will happen in the future. For example, one can write:

$$\mathit{Previous}\ (x > 0)$$

which holds of those system states whose preceding state validates the proposition 'x is greater than 0.' Or also:

$$(x > 0)\ \mathit{Since}\ (y < 0),$$

true if at some point in time y was less than 0 and at all subsequent instants (that is *since* then) x has been positive.

The syntax of our constraint language of choice, `CL` is as follows.

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

A formula ϕ of `CL` is defined starting from atomic formulas `AP`, denoted by $p, q, \ldots$, which are Java boolean expressions. We consider exclusively *pure* boolean expressions, with no side-effects, method invocations, or references to objects (other than the implicit references to `self`); also, ϕ can only refer to private/protected fields of the class it belongs to. Note that we could allow particular methods which can be assumed to have no side-effects, e.g. `Object.equal()`, in an ad-hoc manner. `CL` has the obvious conjunction `&&`, disjunction `||` and negation `!` connectives. In addition to these, it provides two temporal *past* operators: *previous* and *since*, whose informal meaning we described before. This logic is a variation of LTL known as *past tense* LTL [13]. By combining the basic operators it is possible to define two interesting, self-explanatory, auxiliary ones: *Sometime* $\phi \triangleq$ `true` *Since* ϕ and *Always* $\phi \triangleq\ !\mathit{Sometime}\ !\phi$. For the programmer's convenience, these operators are predefined in the Jeeg implementation.

```
public class Counter {

    private int n = 0;

    public void inc() { n++; }
    public void dec() { n--; }
}
```

Table 1: A simple counter

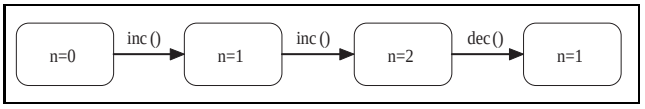

Figure 1: History

All this would not be very helpful in our attempt to tackle the history-sensitive anomaly without a way to refer to the history of object method invocation. The notion of event introduced below serves this purpose.

Definition (Event). An *event* for object o is the execution of one of its methods.

From this basic notion we can define $\mathcal{H}_\pi(o)$, the *history* of object o in (a concurrent) computation π. Informally, this is the sequence of the events of o in π, in the order they occur, together with the states they connect. Thanks to our assumption that guarded methods run in mutual exclusion, each computation unambiguously defines a sequence of method invocations for each object involved. So, without loss of generality, as far as o is concerned, the generic computation π will have the shape

$$h_0^0 \cdots h_{j_0}^0 o.\mathtt{m}_1 h_0^1 \cdots h_{j_1}^1 o.\mathtt{m}_2 h_0^2 \cdots h_{j_2}^2 \ldots$$

where $\mathtt{m}_i$'s are all the activations of a guarded methods of o in π and $h_0^i \cdots h_{j_i}^i$ are sequences of Java heaps (which arise by assignments to public variables or method invocations – either of unguarded and other objects' methods.) As guards may only refer to private/protected variables, their value can only be affected by invocation of methods of o. It is therefore a possible to assume $h_0^0 h_0^1 h_0^2 \cdots$ as the sequence of states of o for the evaluation of temporal guards, forgetting the other intermediate states. Also, only the part of h_0^k containing the values of non-reference private/protected variables of o, say σ_k, is needed. Therefore, for the simple counter class in Table 1, the execution of

```
Counter c = new Counter();
c.inc();c.inc();c.dec();
```

gives rise to the history in Figure 1. Interwoven executions

```
public class Bbuf {

    protected Object[] buf;
    protected int MAX;
    protected int current = 0;

    Bbuf(int max) {
        MAX = max;
        buf = new Object[MAX];
    }
    public synchronized Object get()
    throws Exception {
        while (current<=0) { wait(); }
        current--;
        Object ret = buf[current];
        notifyAll();
        return ret;
    }
    public synchronized void put(Object v)
    throws Exception {
        while (current>=MAX) { wait(); }
        buf[current] = v;
        current++;
        notifyAll();
    }
}
```

Table 2: Concurrent bounded buffer in Java

```
public class Gbuf extends Bbuf {
    boolean afterGet = false;

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

    public synchronized Object gget()
    throws Exception {
        while ((current<=0)||(afterGet)) {
            wait();
        }
        afterGet = false;
        return super.get();
    }
    public synchronized Object get()
    throws Exception {
        Object o = super.get();
        afterGet = true;
        return o;
    }
    public synchronized void put(Object v)
    throws Exception {
        super.put(v);
        afterGet = false;
    }
}
```

Table 3: The class Gbuf in Java

of concurrent objects can easily get way more complex than this. Nevertheless, the notion of history of each single object remains relatively simple.

It will be convenient to think of event $\mathtt{m}_i$ as a reference to a special identifier `event` in σ_i. So, we end up writing

$$\mathcal{H}_\pi(o) \equiv \sigma_0\sigma_1\sigma_2\sigma_3\ldots$$

with the understanding that σ_i binds the identifier `event` to (a value representing method) $\mathtt{m}_i$. (References to `event` are undefined in σ_0.) For example, in the third state in Figure 1, `event` yields `inc`. Identifier `event` can be used in `CL` formulas. In this way, history information finds its way into our constraint language.

Next, we give a formal semantics to `CL` by defining the relation $\mathcal{H}_\pi(o) \models \phi$ expressing that property ϕ holds of object o after a computation π. Let Σ denote $\mathcal{H}_\pi(o)$. For all indexes k in Σ, we define $\Sigma \models_k \phi$, that is ϕ holds at time k, by structural induction on ϕ as follows.

$$\begin{aligned}
\Sigma \models_k p \quad &\texttt{iff} \quad p \texttt{ is true at } \sigma_k \\
\Sigma \models_k !\phi \quad &\texttt{iff} \quad \text{not } \Sigma \models_k \phi \\
\Sigma \models_k \phi \texttt{ || } \psi \quad &\texttt{iff} \quad \Sigma \models_k \phi \texttt{ or } \Sigma \models_k \psi \\
\Sigma \models_k \textit{Previous } \phi \quad &\texttt{iff} \quad k > 0 \text{ and } \Sigma \models_{k-1} \phi \\
\Sigma \models_k \phi \textit{ Since } \psi \quad &\texttt{iff} \quad \Sigma \models_j \psi \text{ for some } j \leq k, \\
&\qquad \texttt{and } \Sigma \models_i \phi \text{ for all } j < i \leq k
\end{aligned}$$

Finally, we convene that $\Sigma \models \phi$ iff $\Sigma \models_0 \phi$.

3. THE INHERITANCE ANOMALY

An example of inheritance anomaly, borrowed from [16] and already mentioned in the Introduction, applies to class `Bbuf` in Table 2, a simple implementation of a bounded buffer in Java. Consider defining a subclass of `Bbuf` that provides an additional method `gget` that removes an element from the buffer only if the last operation performed by the buffer is *not* a `get`. The class `Gbuf` in Table 3 is a possible solution. It illustrates a characteristic occurrence of the inheritance anomaly. Ideally, we would expect method `gget` to be independent from the methods defined in the parent class. A deeper analysis shows that `gget` can only be implemented if *both* the inherited methods are redefined. Our solution in Table 3 rewrites less code than the one in [16], where the example was introduced. This is due to a careful use of delegation to the `super` class (and comes at the price of some extra synchronization). Notice, however, that the essence of the problem is unchanged: the addition of the method `gget` forces us to revise the implementation of seemingly unrelated inherited methods.

This kind of anomaly arises because `gget` is a history-sensitive method. In general, the inheritance anomaly depends on the synchronization primitives present in the language, and different primitives result in different varieties of anomaly [16]. In particular, languages based on method guards and their cousin technologies (such as Java monitors) are prone to history-sensitiveness of acceptable states. This is indeed the case of Jeeg, as its synchronization mechanisms are based on a variation of method guards. Therefore, a good test of expressiveness for Jeeg is given by handling subclassing by history sensitive methods, and `gget` above. The additional expressive power added to method guards by the temporal aspects of `CL` suffices to solve several occurrences of the inheritance anomaly. In this section we exemplify such expressiveness, while in the following we quantify it formally.

Consider the Jeeg version of the class `Bbuf` as defined in Table 4. We can define a class `Gbuf` in Jeeg as in Table 5. This example shows how the use of the temporal operator *Previous* avoided the occurrence of the inheritance anomaly. We no longer need to introduce an instance variable to keep track of the last operation performed. `CL` gives us enough

```
public class Bbuf  {

    sync {
        put : (current < MAX);
        get : (current > 0);
    }

    protected Object[] buf;
    protected int MAX;
    protected int current = 0;

    Bbuf(int max) {
        MAX = max;
        buf = new Object[MAX];
    }
    public Object get() throws Exception {
        current--;
        Object ret = buf[current];
        return ret;
    }
    public void put(Object v) throws Exception {
        buf[current] = v;
        current++;
    }
}
```

Table 4: The Bbuf class in Jeeg

```
public class Gbuf extends Bbuf {

   sync {
      gget: (Previous (event != get)) && (current > 0);
   }

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

   public Object gget() throws Exception {
       current--;
       Object ret = buf[current];
       return ret;
   }
}
```

Table 5: The Gbuf class in Jeeg

expressive power to do without.

As already discussed in the Introduction, a kind of inheritance anomaly that plagues guard-based languages arises in the case of *mix-in* classes. In [16], the authors use *multiple inheritance* to show this variant of the anomaly. Java and Jeeg do not provide multiple inheritance, but the use of interfaces results in similar problems. Consider the class `LockBuf` in Table 6. This is a subclass of the class `Bbuf` that implements the `Lock` interface resulting in a *lockable* buffer. A locked buffer must not accept any other message than `unlock`. One would expect the newly introduced methods to be orthogonal to the inherited ones (this would seem even more natural if they were inherited by multiple inheritance). Naturally, in Java, we cannot simply implement the `Lock` interface to have a lockable buffer, as methods `put` and `get` need to be redefined to account for the new locked and unlocked states, possibly introducing a new boolean variable `locked` to distinguish the two states the buffer can be into. Jeeg solves the problem elegantly, as can be seen in Table 6, again by exploiting the temporal operators of the constraint language. Indeed, `lock` and `unlock`

```
public interface Lock {
    public void lock();
    public void unlock();
}


public class LockBuf extends Bbuf implements Lock {

    sync {
        get : (super.getConstr) &&
              (! Previous (event==lock));
        put : (super.putConstr) &&
              (! Previous (event==lock));
        lock : (! Previous (event==lock));
        unlock : true;
    }

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

    public void lock() { }

    public void unlock() { }
}
```

Table 6: A lockable buffer

are history-sensitive methods. Note that the synchronization constraints of the inherited methods are overridden, while the method definitions are not. As explained in §5 below, in Jeeg method definitions and their synchronization constraints are orthogonal and can be overridden/inherited separately. As expected, the syntax `super.getConstr` allows us to refer to the synchronization constraint of a given method, `get` in this case, as defined in the super class. In general, for the constraint attached to method `m` in the super class, we write `super.mCostr`.

4. EXPRESSIVENESS OF JEEG

When introducing a new synchronization primitive in a concurrent object oriented language, it is often difficult to assess its impact on the inheritance anomaly in a *quantitative* manner. Building on the large body of results on LTL, such analysis is however possible for Jeeg. In particular, we will adapt to our context a characterization of LTL expressiveness in term of 'star-free' regular languages. (For a thorough introduction to LTL the reader is referred to [6].)

The question we are interested in is: to what degree does Jeeg rule out the inheritance anomaly? According to [16], in a language like Java the anomaly arises when the *observable behavior* of an object is more complex than what can be ascertained from its *internal state*. For instance, the internal state of a `Bbuf` object cannot account for the information of whether or not the last method to be executed was a `get`. Therefore, in order to define `gget`, we need to refine the internal state of the object, which comes at the price of code rewriting. The constraint language of Jeeg, however, allows to describe *sequences of events* and so to ascertain more behaviors from the same state. In general, as long as `CL` can describe a certain sequences of events, we can write a constraint that avoids the need of state refinement. A measure of how much of the inheritance anomaly disappears in Jeeg can thus be obtained by measuring which sequences of states are definable in `CL`. For the purpose of this section, we assume `AP` finite.

Definition (General Regular Expressions). Given a finite alphabet A, the regular expressions over A are defined by the following grammar:

$$re ::= a \mid re \cdot re \mid re + re \mid \neg r \mid re*$$

where $a \in A$ denotes the language consisting of the string a, and $\cdot$, $+$, $\neg$ and $*$ represent respectively language concatenation, union, negation with respect to A^* and Kleene closure. The star-free regular expressions are the regular expressions with no occurrence of $*$.

A classical result about LTL says that the sets of state sequences definable by LTL formulae on atomic propositions `AP` coincide with the star-free regular languages on the alphabet $\wp$(`AP`), the powerset of `AP`. Spelling this out, a set of state sequences X is the set of all Σ that satisfy a given ϕ of LTL if and only if X is a star-free regular language. (The reader is referred to [26] for the details.)

Applied to our framework, this result gives a first answer to our question above: `CL` can define the sets of sequences of states that are star-free regular languages on finite subsets of `AP`. To refine this statement, let us observe we can replace a finite set of atomic propositions $\{p_1, \cdots, p_k\}$ by a single atomic proposition: the conjunction p_1 `&&` $\cdots$ `&&` p_k. Let $A_{\mathtt{C}}$ be the subset of `AP` consisting of atomic propositions well-formed for a class `C`. For $\Sigma = \sigma_0 \cdots \sigma_i$ a sequence of states of the private/protected fields of `C` and $P = p_0 \cdots p_i$ a sequence in $A_{\mathtt{C}}$, we say that Σ satisfies P if p_k is true in σ_k for all $0 \leq k \leq i$.

Theorem (Characterizing `CL`). Let `C` be a class and X a set of state sequences. Then, for a given `CL` formula ϕ on `C`, $X = \{\Sigma \mid \Sigma \models \phi\}$ if and only if there exists a star-free regular expression re on $A_{\mathtt{C}}$ such that $\Sigma \in X$ iff Σ satisfies P, for some $P \in re$.

It is interesting to specialize this result when `AP` is restricted to conjunctions of atomic formulae of the kind `event == m`. In such case, `CL` expresses properties of sequences of events – as states are only distinguishable in that respect – and captures precisely those sets of sequences of *events* that are star-free regular languages on the alphabet of method identifiers.

The characterization in terms of regular languages provides also intuition about what *cannot* be expressed in `CL` and, therefore, will result in the occurrence of inheritance anomalies. We show an admittedly contrived example below.

Example. Consider a class representing a simple shared resource which can be simultaneously held by multiple clients:

```
public class SharedResource {
    sync {
       request: true;
       release: true;
    }
    public void request() {  ...  }
    public void release() {  ...  }
    ...
}
```

Before using the resource, clients are supposed to call the method `request`. When the client does not need the resource anymore, it should call the method `release`. To keep the example simple, we assume clients to respect this protocol.

We want to define a class `SeizableResource` which allows clients to gain exclusive access to the shared resource. An additional method `exclusiveRequest` must be provided. Clearly, this method should be allowed to execute only when no other client is using the resource. To accomplish this we must make sure that any call to the method `request` is followed by a call to the method `release`. Unfortunately this constraint cannot be expressed by LTL. Indeed from a language point of view, we want to know whether the history of the object is a word in the language:

```
M ::= request M release | MM |  ϵ | ...
```

where the dots stand for any method identifier in the class `SharedResource`. It is well known that this language, a language of balanced parentheses, is not star-free nor regular. As a consequence, it is not possible to write a synchronization constraint for the method `exclusiveRequest` in `CL`, that is to find a formula that describes the states where `exclusiveRequest` is enabled. What we need to do is to keep track manually of whether the resource is being used or not:

```
public class SeizableResource extends SharedResource {
     sync {
        request : ! (Previous
                  (event==exclusiveRequest));
        exclusiveRequest :
                  (! (Previous
                       (event==exclusiveRequest))
                  ) && (reqCount==relCount);
     }

     int reqCount = 0;
     int relCount = 0;

     public void request() { reqCount++; ... }

     public void release() { relCount++; ... }

     public void exclusiveRequest() { ... }
}
```

The derived class uses two counters, `reqCount` and `relCount`, to ascertain whether the resource is currently used by any client. To accomplish this bookkeeping it is necessary to redefine the base-class methods `request` and `release`.

The example above is typical. A constraint which cannot be expressed in LTL must involve some form of recurrent counting. (For an in depth discussion on these issues we refer to [6, 25].)

5. DIGGING DEEPER INTO JEEG

In this section we look deeper into the interaction between Jeeg synchronization primitives and the other available language features. The reader is referred to [19] for the details.

Synchronized and unsynchronized methods

In Jeeg, methods for which a synchronization constraint is specified are executed in mutual exclusion. In Java terms, they are synchronized. On the other hand, methods for which no synchronization constraint is specified have no mutual exclusion guarantee. Clearly, an undisciplined use of unsynchronized methods may lead to mutual exclusion problems. This is particularly relevant in our setting as the evaluation of a guard must be atomic in order to be meaningful. If an unsynchronized method attempts to modify an

```
public class XBuf extends Bbuf {

    sync { get2 : (current > 1); }

    public Pair get2() {
        current--;
        Object ret1 = buf[current];
        current--;
        Object ret2 = buf[current];
        return new Pair(ret1, ret2);
    }
}
```

Table 7: Inheriting synchronization constraints

attribute of the object while a guard is being evaluated we may end up with an inconsistent result. A trivial example will clarify the situation.

```
public class Counter {
    sync { process : count%20==0; }
    protected count=0;
    ...
    public inc() {count++;...}
    public process() {...}
}
```

In the example above the method `inc` is not executed in mutual exclusion, as a consequence it can modify the value of `count` during a call to the method `process` and the evaluation of its guard. Naturally, a call to `inc` can change the value of the guard for `process` *after* its evaluation, and this would leave the method `process` to be executed in an inconsistent state. A similar situation would occur if guards were allowed to use `public` attributes. To avoid these situations, the attributes occurring in a guard must be accesses in mutual exclusion with the evaluation of the guard. Therefore in Jeeg attributes used in guards can only be modified by synchronized methods. In the case of *static* attributes the lock must be on the *class* rather than on the *object*. As a consequence, they can be only modified by *static* synchronized methods.

Another issue related to unsynchronized methods is that the step-wise history of the object is not well defined as regards to their execution order. Indeed, there can be two methods active at the same time. To force an ordering between unsynchronized methods we adopt the policy of accounting for methods in the history according to the moment their execution finishes. Notice however that in a multiprocessor system, this may depend on the relative speeds of threads. It is therefore bad programming practice in such systems to rely on guards whose truth values depend on the relative ordering of unsynchronized methods.

Method overloading

From a synchronization point of view, Jeeg does not distinguish between different versions of an overloaded method. The synchronization granularity stops at the method identifier level. This distinction could be easily introduced by basing synchronization constraints on method signatures rather than identifiers.

Inheritance and method overriding

Consider a subclass `XBuf` of `Bbuf` as defined in Table 7. There we assume the existence of a support class `Pair` which

```
public class Resource {
    int ownerID;
    boolean busy;
    ....
    sync {
        acquire : ! busy;
        release : true;
    }
    public void acquire(int ID) {
        ownerID = ID;
        busy = true;
    }
    public void release() { busy = false; }
}

public class ReadOnlyResource extend Resource {
    sync { acquire : true; }
}
```

Table 8: A resource hierarchy

```
public class SerBuf extends Bbuf {

    public Object get() {
       current--;
       // A byte representation of buf[current]
       byte[] b = ...
       return b;
    }
}
```

Table 9: A serializing buffer

only wraps up two values as an object. The new class does not override any method of its base class, therefore the methods are inherited together with their synchronization constraints. The additional constraint for the new method is independent of the existing ones.

In Jeeg method definitions and their synchronization constraints are completely decoupled. This scales up to method overriding and indeed it is possible to *selectively* override the method definition, its synchronization constraint, or both.

The example in Table 6 shows a class which does not override the bodies of its `get` and `put` methods but overrides their synchronization constraints, making them stricter. In this case we say that the synchronization constraint for the super-class has been *covariantly* redefined. In [7], the author favors this manner of synchronization overriding. There is, however, no general agreement on this issue. As an example the language Rosette [24] is based on weakening the synchronization constraints in the derived classes, and other authors argue in favor of this choice [21, 20]. In Jeeg both manners of synchronization overriding are possible, indeed we believe that both techniques have their use in different situations. As an example of a derived class which makes the synchronization constraints of the parent *less* stringent consider the a simple class representing a resource (Table 8). The base class `Resource` allows the `acquire` method to be called only when the resource is not already taken. The derived class `ReadOnlyResource` must adopt a less stringent policy: as it models a read-only resource, it can be shared without mutual exclusion problems. It makes then sense to allow multiple clients to share the resource.

In Table 9 we see the other extreme, a class which over-

rides a method but does not override its synchronization constraint which remains the inherited one. The method `get` returns the object stored in the buffer as a chunk of bytes. Clearly this does not affect its concurrent behavior and it is safe to keep its synchronization constraint unchanged.

Jeeg and Exceptions

Method execution might be stopped by the occurrence of a *unhandled exception*. With respect to the object history two possibilities arise. We could choose to keep the method in the history or to ignore it. It is possible to provide examples favoring one or the other approach. Both solutions pose no implementation challenges. In the current implementation, we chose to put in the history only methods which completed their execution.

6. IMPLEMENTATION

Currently, Jeeg is implemented as a pre-processor[1] which given a Jeeg source file generates an equivalent `.java` file and compiles it to byte-code. The resulting class files rely on a runtime system which must be in the `CLASSPATH` of the Java Virtual Machine (JVM). A requirement on the JVM is that it must be Java 2 compliant. The purpose of the runtime system is to implement a *run-time* evaluator for the `CL` formulae used in the program.

Run time evaluation of Constraints

Language `CL` is essentially a variation of LTL based on past-tense temporal operators. Every time a guarded method is called its execution depends on the truth value of a certain temporal formula: its synchronization constraint. If the constraint evaluates to true the method is executed, otherwise it is blocked until the condition becomes true.

Run-time evaluation of LTL formulae is not uncommon. In a wider context the problem can be stated as follows:

Given a finite trace Σ *and a LTL formula* ϕ*, does* $\Sigma \models \phi$ *?*

This problem occurs frequently when trying to apply *model checking* techniques to the verification of Java or `C++` programs [23, 8, 10, 5].

Traditionally, LTL model checking is accomplished by first translating the LTL formula into a *Büchi automaton* [4] and then proving properties on them [11, 4]. Although [23] discusses why such a solution is not ideal to the runtime verification on *finite* traces, this approach is used by the JPaX runtime analysis tool [8].

Dealing with past tense operators gives us an advantage. The dynamic programming algorithm presented in [23] requires as input the trace of the program to evaluate a certain formula, indeed it traverses the program trace backwards. This means that the algorithm is not '*online*', i.e. it cannot be executed together with the program it refers to. By duality, however, the same algorithm becomes online for the past fragment [9]. The algorithm has complexity $O(m)$ where m is the size of the LTL formula. An alternative approach would rely on modifying the automata-based algorithm proposed in [8] to adapt them to past tense operators.

An implementation has thus at least two choices available. The current Jeeg implementation relies on a variation of the dynamic programming algorithm. We found this to be the most natural choice. The algorithm is efficient, and indeed we discovered that several weakenings the logic would not result in a faster algorithm. Intuitively, given a Jeeg program and its set of synchronization constraints the compiler generates a run-time evaluation algorithm for them and *weaves* it into the business code of the program. At every step in the object history, i.e. method execution, the evaluator updates the truth values of the synchronization constraints.

[1]Available from `www.brics.dk/~milicia/Jeeg/`

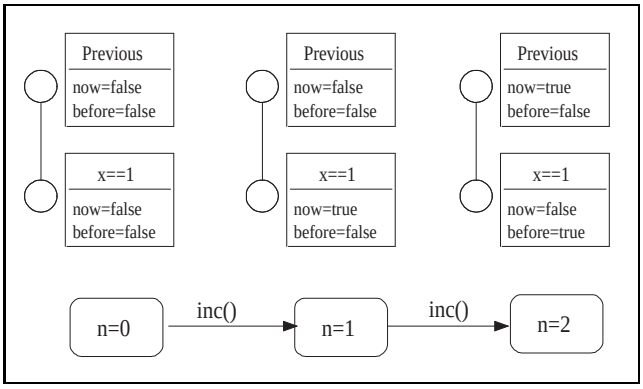

Figure 2: Evaluation algorithm

The evaluation algorithm consists of (repeated) visits to the syntax tree of the formula. To focus the ideas, let us consider the example of the temporal formula

$$Previous\ (x == 1)$$

and its corresponding tree (Figure 2). Every node of the tree represents a subformula of the original temporal formula and is *labeled* by two attributes, '*now*' and '*before*' which respectively hold the truth value of the corresponding subformula at the current time and one step before. The task of the algorithm is to visit the tree and update the values of the two attributes for every node.

In §2.1 we adopted a *strong* semantics for the temporal operators, that is we assumed that *Previous* can only be applied at times greater than zero. As a consequence, at the initial instant the '*before*' attribute of every subformula is set to `false`. The truth value of the '*now*' attribute is initialized when the object is created and depends on the initial state of the object. The algorithm performs a simple depth first visit of the tree and for every node ϕ applies, depending on its type, an appropriate update rule. In the description of the update rules that follows, we use l and r to refer, respectively, to the current node's left- and right-wise child. The attributes of the children are denoted by the standard dot-notation, and we use a multiple assignment operator with the obvious meaning.

previous	*before*, *now* := *now*, *l.before*
always	*before*, *now* := *now*, *before* and *l.now*
sometimes	*before*, *now* := *now*, *before* or *l.now*
since	*before*, *now* := *now*, *r.now* or (*now* and *l.now*)
and	*before*, *now* := *now*, *l.now* and *r.now*
not	*before*, *now* := *now*, not *l.now*
AP	*before*, *now* := *now*, *evalNode*()

To clarify the working of the algorithm, consider a simple formula *Previous* $(x == 1)$ and a trivial counter class as the one we presented in Table 1. Figure 2 shows the evolution of the attributes in the formula tree with respect to the history of the object `c` when we execute the following code.

```
Counter c = new Counter();
c.inc(); c.inc(); c.dec();
```

It is easy to see that the complexity of the run-time evaluation algorithm is linear in the size of the formula tree. The run-time overhead involved is thus linear in the size of the synchronization constraints.

Synchronization Manager

For the evaluation algorithm to be sound, formulae must be evaluated at every step in the program history, i.e. after every method execution. This is accomplished by a *synchronization manager* through a mechanism of method call interception (MCI), typical of the implementation of aspect oriented languages.

The synchronization manager takes control after a method call. Then it checks whether the synchronization constraint for the method is verified. Note that the constraint must not be evaluated at this stage, its truth value is already available. This is the case as the truth value of synchronization constraints is updated *after* the execution of a method. If the constraint is `true` the control goes back to the method code, otherwise the synchronization manager performs a `wait()` and put the method on hold. After the execution of a method is accomplished, the control shifts back to the synchronization manager. At this point the synchronization constraints are evaluated. Since the execution of a method may change the state of the object, after updating the value of the synchronization constraints, the manager takes care of notifying the blocked methods which may then attempt to proceed again.

To perform its tasks the synchronization manager must have access to the private/protected fields of the object. We accomplish this by making the synchronization manager an *inner class* of the object it manages.

A complete example showing the Java code generated from a Jeeg source file can be found in [19].

A note on performance

We performed extensive benchmarks on our prototype implementation. The results are encouraging. The overhead introduced by Jeeg is felt first at object creation time, when the synchronization manager is initialized, and then every time a synchronized method is called. As an example consider the Java and Jeeg implementation of the `Gbuf` class (see Tables 3 and 5). Using the JDK 1.4 under Windows 2000 and Linux 2.4.18 the performance difference between the two implementations is negligible. Under heavy load (¿100 threads using the object) on a low-end machine (Celeron 300Mhz) method calls were about 5ms slower for the Jeeg program. This is due to the longer time the Jeeg object remains locked while the synchronization manager performs its duty. Better performing machines did not exhibit this behavior even in the presence of more than 1000 threads.

More complex constraints result in slower performances. However, even in presence of large constraints (size over 64), performances were still acceptable. We refer the reader to [19] for a thorough discussion.

7. RELATED WORK

The idea of specifying synchronization constraints in programming (as opposed to verifying) using a *temporal* logic has, to the best of our knowledge, not been explored before. Indeed, only recently the problem of run-time evaluation of LTL formulae has come to the attention of the research community [5, 23].

The idea of a complete separation between the definition of a method and its synchronization constraints is known to be helpful in avoiding the inheritance anomaly [16, 15, 14]. In this work, we uphold the concept by making synchronization code and method definitions totally independent, to the degree that they need no be specified in the same file. In this respect Jeeg is inspired by the current trends in *component based* and *aspect oriented* programming [12].

Frølund proposed a methodology for selective inheritance of synchronization constraints [7]. His proposal, based on method guards, favors the *covariant* redefinition of synchronization constraints in derived classes. As remarked in §5, this way to synchronization redefinition is not universally accepted. Indeed, some languages [24, 17] take the opposite view and allow the derived class to make the synchronization constraints less stringent, that is *contravariant*. Examples exist supporting both approaches; Jeeg allows both manners of overriding. From the point of view of the inheritance anomaly, Frølund's methodology is subject to the common problems of method guards, i.e. the history dependent variants of the anomaly.

Meseguer [17], analyzed the problem of the inheritance anomaly in the context of his rewriting logic based language Maude [18]. Meseguer's work aimed at removing the need for synchronization code in the first place. This technique, based on rewriting logic, is closely tied to the Maude system and we are not aware of adaptions to imperative object oriented languages such as Java.

Different lines were pursued by Matsuoka and Yonezawa: the first based on the notion of *reflection* [16], the second aiming at reducing the amount of synchronization code to a minimum [16].

An approach more in line with aspect oriented programming is presented in [2]. Although their use of *Abstract Communication Types* (ACT) does provide a way to tackle the history sensitive anomaly in a modular fashion, it is still based on ad-hoc coding. Every instance of the anomaly requires the programmer to write a specific ACT to solve it. The problem is thus *moved* from the object to the ACT rather than solved.

8. CONCLUSIONS

We introduced Jeeg, a dialect of Java were synchronization constraints are written in linear temporal logic and are specified in a declarative manner. We showed by examples that the additional expressive power of our synchronization language, `CL`, is helpful in treating the inheritance anomaly. Also, we provided a characterization of the expressiveness of `CL` in terms of regular languages that yields a precise description of the sequences of events we can express. Finally, we described the current implementation of Jeeg.

Propositional linear temporal logic seems to offer us the best balance between computational overhead and expressiveness. It would indeed be interesting to base Jeeg on quantified linear temporal logic (QLTL) or monadic second order logic (MSOL), 'second order' variations of LTL of greater expressiveness. In particular, QLTL and MSOL stay to regular languages as LTL stays to star-free regular languages. However, while giving us the power to express synchronization policies as complex as regular languages or

more, these options would present an increased computational cost that we are currently investigating.

Concerning the implementation, we are exploring the possibility of optimizing the LTL evaluation procedure by using ad-hoc static-analysis techniques and sophisticated scheduling protocols. The current implementation of the Jeeg compiler is available at `http://www.brics.dk/~milicia/Jeeg/`.

9. REFERENCES

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

[2] L. Bergmans. *Composing concurrent objects*. PhD thesis, University of Twente, 1994.

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

[4] E. Clarke, O. Grumberg, and S. Peled. *Model checking*. The MIT press, 1999.

[5] D. Drusinsky. The temporal rover and the ATG rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer-Verlag, 2000.

[6] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[7] S. Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *ECOOP '92, European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 185–196. Springer-Verlag, 1992.

[8] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering 2001 (ASE'01)*, San Diego, California, November 2001. IEEE Computer Society.

[9] K. Havelund and G. Rosu. Monitoring Java programs with Java Path Explorer. In *First Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, July 2001.

[10] K. Havelund and G. Rosu. Monitoring programs using rewriting. In *Automated Software Engineering 2001 (ASE'01)*, San Diego, California, November 2001. IEEE Computer Society.

[11] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[13] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In R. Parikh, editor, *Proceedings 3rd Workshop on Logics of Programs, Brooklyn, NY, USA, 17–19 June 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer-Verlag, Berlin, 1985.

[14] C. V. Lopes and K. J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. *Lecture Notes in Computer Science*, 821:81–99, 1994.

[15] S. Matsuoka, K. Wakita, and A. Yonezawa. Synchronization constraints with inheritance: What is not possible — so what is? Technical Report TR 90-10, Department of Information Science, the University of Tokyo, 1989.

[16] 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.

[17] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 220–246. Springer-Verlag, July 1993.

[18] J. Meseguer and T. Winkler. Parallel programming in Maude. In *Proceedings of Research Directions in High–Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 253–295, Berlin, Germany, June 1992. Springer.

[19] G. Milicia and V. Sassone. Jeeg: A programming language for concurrent objects synchronization. Technical report, BRICS, 2002. Available from `http://www.brics.dk/~milicia/Jeeg`.

[20] O. Nierstrasz and M. Papathomas. Viewing objects as patterns of communicating agents. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 38–43, Oct. 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[21] O. Nierstrasz and M. Papathomas. Towards a type theory for active objects. *ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 Workshop on Object-Based Concurrent Systems*, 2(2):89–93, Apr. 1991.

[22] 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. IEEE Computer Society Press, Oct. 31–Nov. 2 1977.

[23] G. Rosu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical Report TR 01-15, RIACS, May 2001.

[24] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of the OOPSLA '99 Conference on Object-oriented Programming Systems, Languages and Applications*, 1989.

[25] Wolfgang Thomas. Languages, automata and logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, Amsterdam, 1990. Elsevier Science Publishers.

[26] L. Zuck. *Past temporal logic*. PhD thesis, Weizmann Institute, 1986.