

Basic Research in Computer Science

Jeeg: Temporal Constraints for the Synchronization of Concurrent Objects

**Giuseppe Milicia
Vladimiro Sassone**

BRICS Report Series

ISSN 0909-0878

RS-03-6

February 2003

Copyright © 2003,

**Giuseppe Milicia & Vladimiro Sassone.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

**See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK-8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide
Web and anonymous FTP through these URLs:**

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory RS/03/6/

Jeeg: Temporal Constraints for the Synchronization of Concurrent Objects

Giuseppe Milicia, Vladimiro Sassone
milicia@brics.dk, vs@cogs.susx.ac.uk

BRICS
Aarhus University

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.

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* [19] 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 e1) {
    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 [19] 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 `put` to be disallowed. 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 rather than depending on the object's state, as above, depends on its past history, 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.

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 non declarative 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 [24] (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 an aspect oriented language. 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. In §7 we discuss the performance overhead brought forth by the Jeeg methodology. Finally, we discuss related and further work. The appendices provide some optional material, most notably an example of Jeeg to Java translation.

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. 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 :  $\phi$ ;
```

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*. Then, the resumption of `m` 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.

The XML description of synchronization complies with the DTD of Appendix A and is described later in §5.

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, so that the computational overhead does not overcome the expressiveness benefits. A logic that suits our purpose is *linear temporal logic* (LTL) [24]. 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, allowing in

particular the vanishing of history-sensitive inheritance anomaly, and at the same time keeps 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:

Previous $x > 0$

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

$x > 0$ 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.

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

A formula ϕ of **CL** is defined starting from atomic formulas **AP**, denoted by p, q, \dots , 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 [16]. By combining the basic operators it is possible to define two interesting, self-explanatory, auxiliary ones: *always*, *sometime*. Formally, Sometime $\phi \triangleq \text{true Since } \phi$ and Always $\phi \triangleq !\text{Sometime } !\phi$. For the user’s convenience, these operators are predefined in the Jeeg implementation.

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. In order to make this precise, observe that thanks to our assumption that guarded


```

public class Counter {

    private int n = 0;

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

```

Figure 1: A simple counter

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

$$h_0^0 \cdots h_{j_0}^0 o.m_1 h_0^1 \cdots h_{j_1}^1 o.m_2 h_0^2 \cdots h_{j_2}^2 \dots$$

where 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. (Such sequences arises by assignments to public variables or method invocations – either of unguarded and other objects' methods). Formally, $\mathcal{H}_\pi(o)$ can then be defined by induction on k , the number of o 's guarded methods in π ,

$$\mathcal{H}_\pi(o) = \begin{cases} h_{j_0}^0 & \text{for } k = 0, \\ \mathcal{H}_{\pi_k}(o) m_k h_{j_k}^k & \text{for } k > 0 \end{cases}$$

where π_k is the subcomputation of π terminating just before the invocation on $o.m_k$.

Notice that such a definition makes perfect sense under our hypothesis. 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 sensible choice to assume $h_{j_0}^0 h_{j_1}^1 h_{j_2}^2 \cdots$ as the sequence of states of o for the evaluation of temporal guards. Notice also that only the part of $h_{j_k}^k$ containing the values of non-reference private/protected variables of o , say σ_k , is needed for that. We therefore represent object histories by sequences such as

$$\mathcal{H}_\pi(o) \equiv \sigma_0 \xrightarrow{m_1} \sigma_1 \xrightarrow{m_2} \sigma_2 \xrightarrow{m_3} \sigma_3 \dots$$

where the computation π will most often remain implicit. To exemplify the definition consider the simple counter class in Figure 1.

If we execute

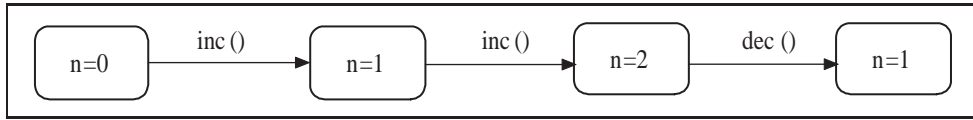


Figure 2: History

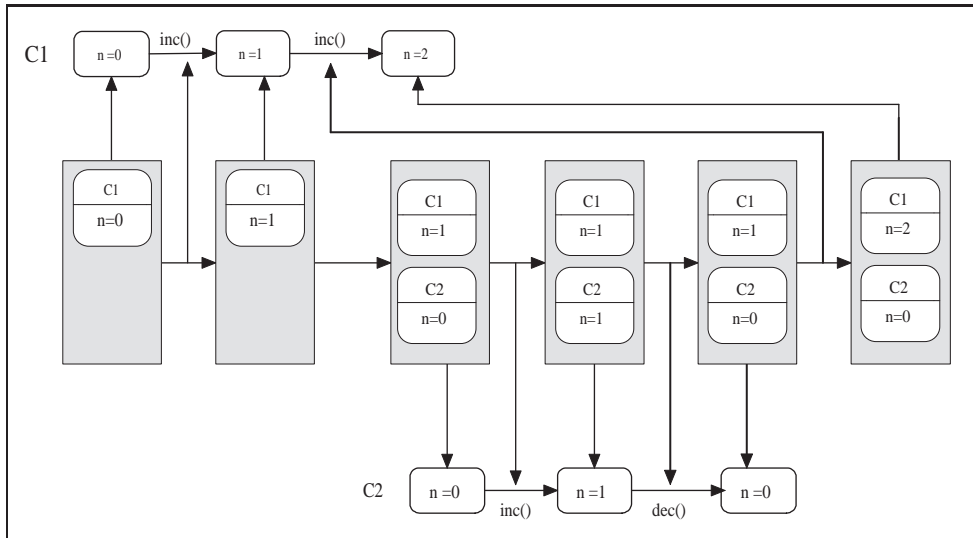


Figure 3: Extracting the history

```

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

we obtain the history in Figure 2. Interwoven executions in the presence of concurrent objects easily get way more complex. Nevertheless, the notion of history of each single object remains relatively simple. Figure 3, for instance, illustrates histories in the case of two concurrent threads executing the code above on two distinct counters.

For practical convenience we will think of event m_i as a reference to a special identifier `event` in σ_i . So, we will write

$$\mathcal{H}(o) \equiv \sigma_0 \sigma_1 \sigma_2 \sigma_3 \dots$$

with the understanding that σ_i binds the identifier `event` to (a value representing method) m_i . (References to `event` are undefined in σ_0 .) For example, in the third state in Figure 2, `event` yields `inc`. Identifier `event` can be used by CL. 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 $\Sigma_k \models \phi$, that is ϕ holds at time k , by structural induction on ϕ as follows.

$$\begin{aligned} \Sigma_k \models p & \text{ iff } \sigma_k \models p \quad (p \text{ is true at } \sigma_k) \\ \Sigma_k \models !\phi & \text{ iff } \text{not } \Sigma_k \models \phi \\ \Sigma_k \models \phi \parallel \psi & \text{ iff } \Sigma_k \models \phi \text{ or } \Sigma_k \models \psi \\ \Sigma_k \models \text{Previous } \phi & \text{ iff } k > 0 \text{ and } \Sigma_{k-1} \models \phi \\ \Sigma_k \models \phi \text{ Since } \psi & \text{ iff } \Sigma_j \models \psi \text{ for some } j \leq k, \\ & \text{and } \Sigma_i \models \phi \text{ for all } j < i \leq k \end{aligned}$$

Finally, we convene that $\Sigma \models \phi$ iff $\Sigma_0 \models \phi$.

3 The inheritance anomaly

A striking example of inheritance anomaly, borrowed from [19] and already mentioned in the Introduction, applies to the class `Buffer` in Figure 4, a simple implementation of a bounded buffer in Java. Consider defining a subclass of `Buffer` that provides an additional method `gget` that removes an element from the buffer only if the last operation performed by the buffer was *not* a `get`. The class `HistoryBuffer` in Figure 5 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* inherited are redefined, resulting in the loss of any code reuse that inheritance should have provided.

The example in Figure 5 follows closely the original presentation seen in [19]. However, it is possible to minimize the amount of code rewriting relying on the implementation of the methods `get` and `put` found in the superclass. We could write:

```
public class HistoryBuffer extends Buffer {
    boolean afterGet = false;

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

```

public class Buffer {

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

    Buffer(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();
    }
}

```

Figure 4: Concurrent bounded buffer in Java

```

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;
}
}

```

This comes at the price of some redundant synchronization. Nevertheless, the problem remains. The addition of the method `gget` forces us to revise the implementation of seemingly unrelated inherited methods.

This kind of anomaly arises from the fact that `gget` is a history-sensitive method. Generally speaking, the inheritance anomaly depends on the synchronization primitives present in the language, and different primitives result in different varieties of anomaly [19]. In particular, languages based on method guards and their cousin technologies run the risk of suffering from 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. It should not come as a surprise now that 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 one we try to quantify it formally.

Consider the Jeeg version of the class `Buffer` as defined in Figure 6. We can define a class `HistoryBuffer` in Jeeg as in Figure 7. 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 expressive power to do without.

As already discussed in the Introduction, a different kind of inheritance

```

public class HistoryBuffer extends Buffer {

    boolean afterGet = false;

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

    public synchronized Object gget() throws Exception {
        while ((current<=0)||afterGet) {
            wait();
        }
        current--;
        Object ret = buf[current];
        afterGet = false;
        notifyAll();
        return ret;
    }
    public synchronized Object get() throws Exception {
        while (current<=0) { wait(); }
        current--;
        Object ret = buf[current];
        afterGet = true;
        notifyAll();
        return ret;
    }
    public synchronized void put(Object v) throws Exception {
        while (current>=MAX) { wait(); }
        buf[current] = v;
        current++;
        afterGet = false;
        notifyAll();
    }
}

```

Figure 5: The class HistoryBuffer in Java

```
public class Buffer {

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

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

    Buffer(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++;
    }
}
```

Figure 6: The Buffer class in Jeeg

```

public class HistoryBuffer extends Buffer {

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

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

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

```

Figure 7: The HistoryBuffer class in Jeeg

anomaly that plagues guard-based languages arises in the case of *mix-in* classes. In [19], the authors use *multiple inheritance* to show this variant of the inheritance anomaly. Java and Jeeg do not provide multiple inheritance, but the use of interfaces results in similar problems. Consider the class `LockBuf` in Figure 8. This is a subclass of the class `Buffer` 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 Figure 8, again by exploiting the temporal operators of the constraint language. Indeed, `lock` and `unlock` 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.mConstr`.


```

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

public class LockBuf extends Buffer 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() { }
}

```

Figure 8: A lockable buffer

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 solve the inheritance anomaly? According to [19], 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 `Buffer` 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 a heavy price. The constraint language of Jeeg, however, allows to describe *sequences of events* and so to ascertain more behaviors from the same state. As long as `CL` can describe a certain sequence, 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 $A \cup \{\epsilon\}$, where ϵ is a special symbol such that $\epsilon \notin A$, are defined by the following grammar.

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

where ϵ denotes the empty word, $a \in A$ denotes the language consisting of a single 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 $\wp(\text{AP})$, 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 [28] 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 that we can identify a certain state of an object (or better, its part expressible in `CL`), by a boolean formula on its (private/protected) field's values. Let A_c be the set of these boolean expressions. It follows that a certain sequence of states can be identified by a set of formulae P in A_c . Note that, in general, P will denote a *set* of sequences of states, that is, all the sequences such that $\Sigma \models P$ (meaning $\sigma_i \models p_i$, for every i). In this context, the following theorem formalizes the correspondence (from the point of view of the class `C`) between sequences of states denoted by `CL` formulae and sequences of states corresponding to star-free regular expressions on A_c .

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_c such that $\Sigma \in X$ iff $\Sigma \models P$ 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.

Suppose we want to define a class `SeizableResource` which allows clients to gain exclusive access to the shared resource. This can be accomplished by providing an additional method `exclusiveRequest`. 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` was 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:

$$M ::= \text{request } M \text{ release} \mid MM \mid \epsilon \mid \dots$$

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)))
                        && (count==0);
    }

    int count = 0;

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

    public void release() {
        count--;
        ...
    }

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

The derived class uses one counter `count` to ascertain whether the resource is currently used by any client. To accomplish this book-keeping 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 again to [6, 27].)

EXAMPLE. The classic `HistoryBuffer` example has been solved using Jeeg in Figure 6. It is interesting to analyze the (simple) temporal constraint used in the example in terms of star-free regular expressions. The constraint relative to the `gget` method is the following:

$$(\text{Previous } (\text{event} \neq \text{get})) \ \&\& \ (\text{current} > 0);$$

For simplicity let us restrict ourselves to its temporal component:

$$(\text{Previous } (\text{event} \neq \text{get}))$$

In light of the previous discussion on the equivalence of (Past) LTL formulae and star-free regular expressions we shall give the same constraint in its regular

expression form. Intuitively, the language the formula describes is that of ‘all the words in the trace alphabet *not* ending with the symbol `get`’. We can define $A^* \equiv \epsilon + \neg\epsilon$. With abuse of notation we shall denote the formula (`event == get`) simply as `get`. In this manner occurrences of the event `get` in the history of the object could be recorded simply as `get`. The corresponding (star-free) regular expression is:

$$\neg(A^* \cdot \text{get})$$

Which formalizes the intuitive set of all the words which do not end with the symbol `get`.

5 Digging deeper into Jeeg

In this section we look deeper into the interaction between Jeeg synchronization primitives and the other available language features.

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 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.

Java (and consequently Jeeg) allow methods and attributes to be declared *static*. Static fields and methods are common to a *class* rather than to each of its instances. To access a static attribute it is, therefore, not enough to own the lock for a certain object instance. Indeed, such lock does not guarantee mutually exclusive access to static attributes. The lock needed to obtain such access must be on the *class* rather than on the *object*. As a consequence, static fields can be only modified by *static* synchronized methods. Conversely static synchronized methods own a lock on the *class* rather than on a specific object instance. For this reason such methods are forbidden to modify non-static object fields.

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 notion is not well-defined. 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.

In the example in Figure 9, the synchronization constraint applies to both definitions of the overloaded method `m`. This choice is motivated by the fact that synchronization constraints relate to the *essential behavior* of a method which we feel should not be changed by overloading. One could certainly define an overloaded method `m` whose definitions access the shared attributes of the object in a completely different manner. It would not be difficult to support these situations by basing synchronization constraints on method signatures rather than identifiers.

Inheritance and method overriding

Consider a subclass `Xbuf` of `Buffer` as defined in Figure 10. There we assume the existence of a support class `Couple` which only wraps up two values as an object. The new class does not override any method of its base class, therefore

```

public class C {
    int i = 0;

    sync {
        m : (event != m) since (i>0);
    }

    public void m(Object o) {
        ....
    }

    public void m(int i) {
        ....
    }
}

```

Figure 9: Method overloading

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.

In Figure 8 we show an example of 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 [26] is based on making synchronization constraints less strict in the derived classes, and other authors argue in favor of this choice [23, 22]. 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 (Figure 11). 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, it models a read-only resource, as a consequence it can be shared without mutual exclusion problems. For this reason it makes sense to allow multiple clients to share the resource, to accomplish this the synchro-

```

public class Xbuf extends Buffer {

    sync {
        get2 : (current > 1);
    }

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

```

Figure 10: Inheriting synchronization constraints

nization constraint of the method `acquire` is made less stringent than in the parent class.

In Figure 12 we see the other extreme, a class which overrides a method but does not override its synchronization constraint which remains the one inherited. Its semantics is straightforward, 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.

XML constraints

In order to favor the separation between method definitions and synchronization code, Jeeg allows for the synchronization constraints to be specified separately in a XML file. When the Jeeg compiler processes a source file `ClassName.j1` it looks for a XML file named `ClassName.xml`. If it finds the file then it validates it against the relevant document type definition (DTD),


```

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;
    }
}

```

Figure 11: A resource hierarchy

which can be found in Appendix A. If the validation is successful the synchronization constraints it describes are *waved* into the resulting class file. If a `sync` section is present in the class definition, it is overridden by the external constraints in the XML file.

To give a quick taste of how to define synchronization constraints using a XML file, consider the bounded buffer example in Figure 7. Its `sync` section is equivalent to the following XML description:

```

<?xml version='1.0' ?>
<!DOCTYPE Jeeg SYSTEM "Jeeg.dtd">

<Jeeg>
  <Class name="HistoryBuffer" super="Buffer" version="j1">

    <Method name="gget">
      <And>
        <Arg>

```

```

public class SerBuf extends Buffer {

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

```

Figure 12: A serializing buffer

```

    <Previous>
      <BooleanExpression>
        event != get
      </BooleanExpression>
    </Previous>
  </Arg>
  <Arg>
    <BooleanExpression>
      current > 0
    </BooleanExpression>
  </Arg>
</And>
</Method>

</Class>
</Jeeg>

```

6 Implementation

The current Jeeg implementation¹ is a pre-processor 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.

¹Available from www.brics.dk/~milicia/Jeeg/

Run time evaluation of CL Expressions

The CL language 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 a recurrent problem. In an 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 appears frequently when trying to apply *model checking* techniques to the verification of Java or C++ programs [11, 25, 8, 10, 5, 15].

Traditionally, LTL model checking is accomplished by first translating the LTL formula in a *Büchi automata* [4] and then proving properties on them [13, 4]. Although in [25, 11] the authors discuss why such a solution is not ideal to the runtime verification on *finite* traces, this approach is nevertheless used by the JPaX runtime analysis tool [8].

Dealing with past tense operators gives us an advantage. The dynamic programming algorithm presented in [25] requires as input the trace of the program to evaluate a certain formula, indeed it traverses the program trace backwards. This implies that the algorithm is not ‘*online*’, i.e. it cannot be executed at the same time as the program it refers to. By duality, 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.

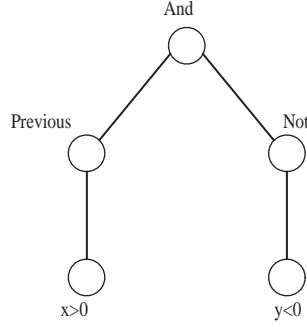
An implementation has thus at least two choices available. The current Jeeg implementation relies on a variation of the dynamic programming algorithm. We found this choice to be the most natural. The algorithm is efficient, indeed weakening the logic would not result in a faster algorithm. Intuitively, 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.

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

$$(\text{Previous}(x > 0)) \ \&\& \ !(y < 0)$$

and its corresponding tree:



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 sub-formula 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 our 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 sub-formula 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. For every node ϕ we use the notation ϕ_0 to refer to its first (left-wise) child and ϕ_1 to its second child, if the node represents a binary operator. Attributes of the children are denoted by $\phi_i.now$ and $\phi_i.before$. The algorithm performs a simple *depth first visit* of the tree and for every node ϕ updates the value of the *before* and *now* fields. First we perform the assignment $\phi.before = \phi.now$, then, depending on the node type, we updated the *now* field according to the following rules:

previous	now = $\phi_0.before$
always	now = before <i>and</i> $\phi_0.now$
sometimes	now = before <i>or</i> $\phi_0.now$
since	now = $\phi_1.now$ <i>or</i> (before <i>and</i> $\phi_0.now$)
and	now = $\phi_0.now$ <i>and</i> $\phi_1.now$
or	now = $\phi_0.now$ <i>or</i> $\phi_1.now$
not	now = <i>not</i> $\phi_0.now$
AP	now = <i>eval</i> (ϕ)

To clarify the working of the algorithm, consider a simple formula:

$$\text{Previous}(x == 1)$$

and a trivial counter class as the one we presented in Figure 1. Suppose we execute the code:

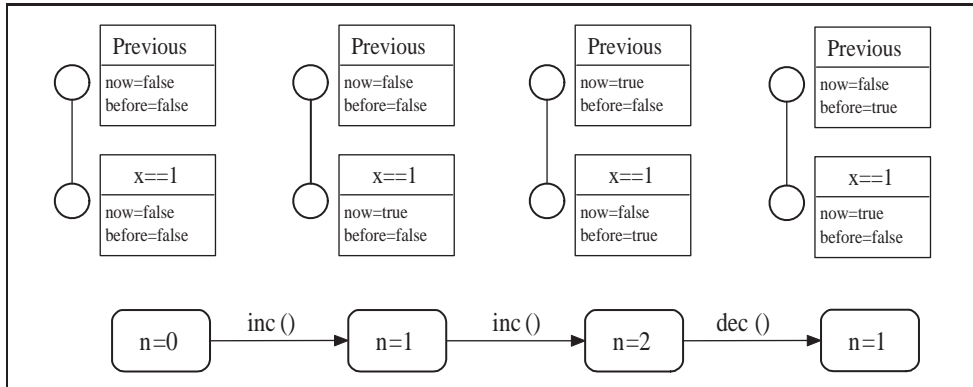


Figure 13: History

```

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

```

then Figure 13 shows how the attributes in the formula tree evolve with respect to the history of the object `c`.

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 issues a `notifyAll()` statement. Blocked methods may then attempt to proceed again.

To perform its tasks the synchronization manager must have access to the private/protected fields of the object. To accomplish this we chose to make 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 Appendix B.

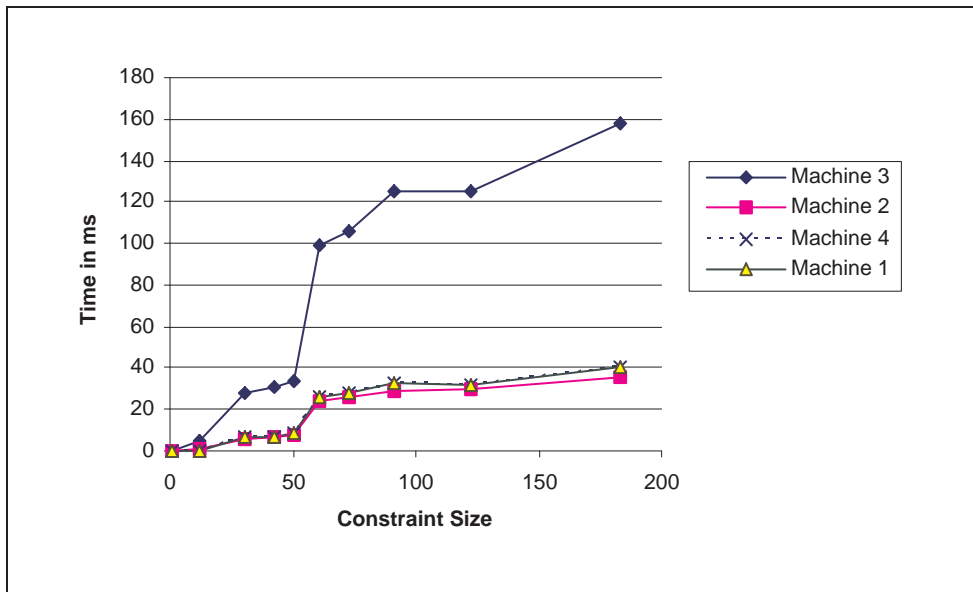


Figure 14: Object creation overhead

7 Benchmarks

To assess the feasibility of our approach we performed some targeted benchmarking on the current prototype implementation of the Jeeg compiler. In this section we outline our results.

7.1 General setting

When benchmarking code running in a JVM care must be taken to avoid interference from the garbage collector. Furthermore a single measurement is no valid indication of the actual time spent during an operation. Multiple measurement of the same experiment must be performed instead. We take their *average* as a fair result of our experiment.

Although Java is designed to be platform independent, different implementations of the virtual machine for different operating systems might perform differently. We chose to perform our tests on two popular operating systems: Linux and Windows 2000.

We chose to run the virtual machine with no optimizations, in particular the code was only *interpreted*, the just in time compiler was turned off. In this manner we could run the same tests a number of times without speed-ups. Our benchmarks are thus a measure of the *worst case* scenario, when the code is executed only once and thus no gain is to be expected by just-in-time compilation. All the programs were compiled and run using the J2SE 1.4 and the `-Xint` option.

To have a better feel of the performance impact in a realistic setting we performed our tests on low-end and high-end machines. Below we list the machines we used:

Machine 1	AMD 1800+XP	256MB	Windows 2000	Jdk 1.4
Machine 2	AMD 1800+XP	256MB	Linux RedHat 6.2	Jdk 1.4
Machine 3	Celeron 300Mhz	192MB	Windows 2000	Jdk 1.4
Machine 4	Pentium 4 1,6 Ghz	512MB	Linux 2.4.18	Jdk 1.4

The code used for the benchmarks is available on the web at: www.brics.dk/~milicia/Jeeg.

7.2 Benchmark results

The overhead introduced by our methodology is felt first at the time of object creation, and then, whenever a call to a synchronized method is performed. We begin by showing the test results in these two situations and conclude with an evaluation of the performance impact of the Jeeg methodology.

Object creation

At object creation time the structures representing the (temporal) formulae of the synchronization constraints must be built. This results in the creation of as many objects as logic operators present in the formulae. As a consequence we expect object creation to become slower as synchronization constraints grow more complex. To quantify the overhead we timed the creation of objects with increasing complex synchronization constraints (in the size of the formulae involved). The constructor of the object was otherwise empty. The results of our tests can be found in Figure 14.

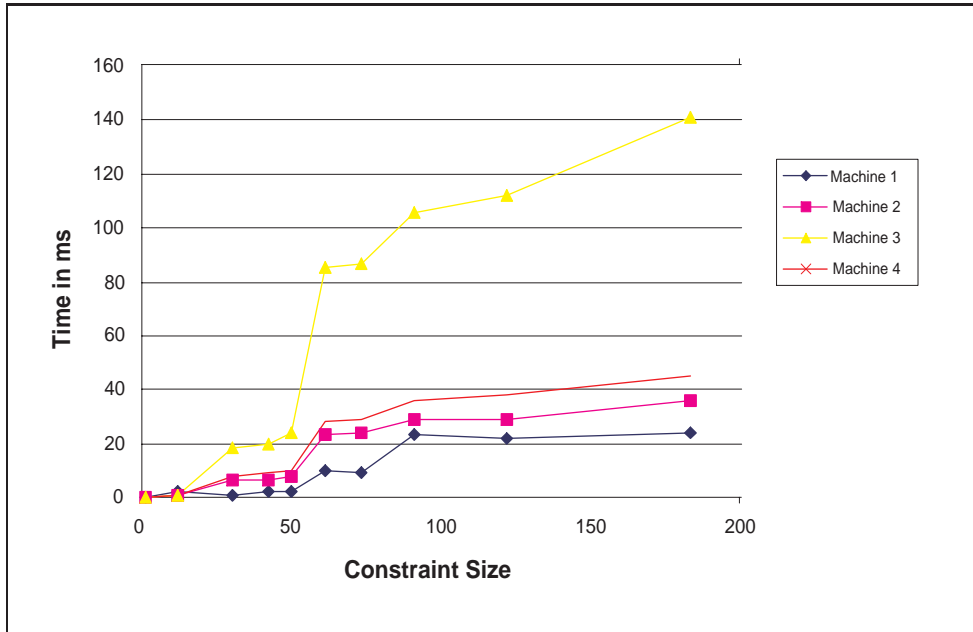


Figure 15: Method call overhead

Method call

Every time a (synchronized) method is called the algorithm described in § 1 must be performed. This results in the evaluation of *all* synchronization constraints. The overhead we face is thus proportional to the *sum* of the sizes of the logic formulae describing the constraints. Clearly every method call will incur in the same overhead regardless of the size of its own synchronization constraint.

To measure the overhead involved in our technique we tested method calls on objects with increasing complex synchronization constraints. We made sure, to avoid any biased result, that the constraints would always evaluate to `true`. Method calls performed no function, in this way we made sure that we only measured the unavoidable overhead brought up by our technique. The results of our tests can be seen in Figure 15.

A different performance problem could result from the fact that the synchronization constraints must be evaluated in mutual exclusion. The object will be locked during the evaluation. If a number of threads are actively accessing the object this could slow down the method calls sensibly. To evaluate this issue we performed the test above with an *increasing* number of threads. The results can be found in Figure 16. We can see that in the presence of large constraints and over 50 threads actively using the objects we face a sensible

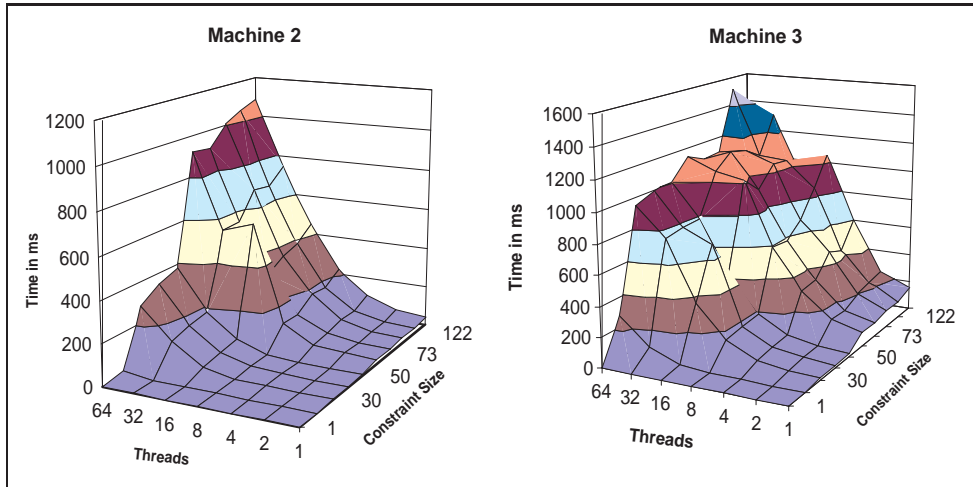


Figure 16: Method Call Overhead

slow-down.

We wish to remark that Jeeg takes care of all the synchronization constraints of the object. An equivalent Java program must accomplish the same results in a different fashion, for example using boolean variables to keep track of its state. An interesting experiment is thus, the comparison of two semantically equivalent Jeeg and Java programs. We use as our test-bed the `HistoryBuffer` example of §3. Figure 17 compares the execution time for a method call to a Java implementation of the class `HistoryBuffer` (as seen in Figure 5) and its Jeeg counterpart (as seen in Figure 7). The high-end machines feel almost no performance loss, on the other hand if many threads are active at the same time, the low-end machine suffers from sever performance losses. However even the low-end machine performs well in the presence of as many as 64 active threads, as Figure 18 shows.

7.3 Evaluation

Our tests show that under low-load (below 70 threads) even the most complex synchronization constraints yield little performance overhead. Low-end machines face worse scalability problems due to the additional time the object is kept locked. If the machine cannot perform the evaluation algorithm fast enough a number of threads will be kept waiting.

Experience shows that the synchronization constraints of an object seldom reach a length of over 10 or 20 logical connectives. Our benchmarks show that for such objects the performance loss is negligible even in case of high-load (> 200 active threads).

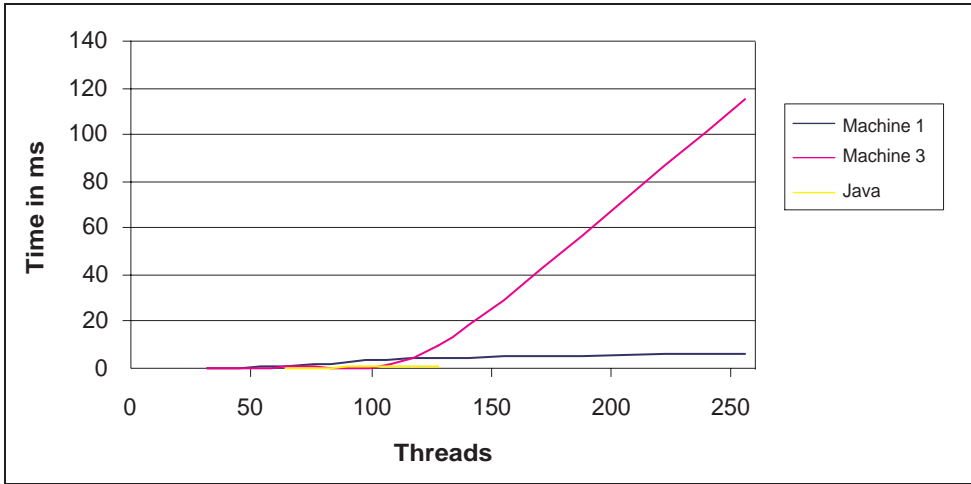


Figure 17: HistoryBuffer performances

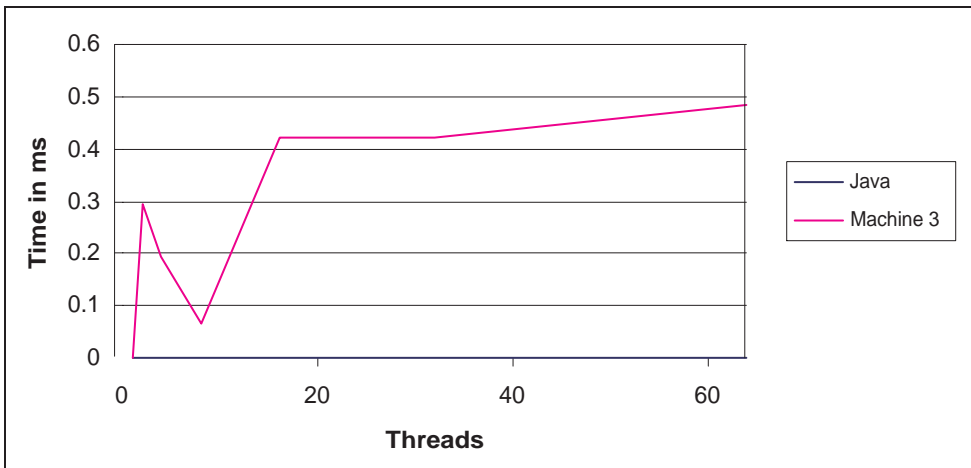


Figure 18: HistoryBuffer performances (details)

We are currently evaluating possible optimization strategies for the formulae evaluation algorithm.

8 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, 11].

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 [19, 18, 17]. 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 regard Jeeg is inspired by the current trends in *component based* and *aspect oriented* programming [14].

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 we pointed out in §5, this manner of synchronization redefinition is not universally accepted. Indeed, some languages [26, 20] take the opposite view and allow the derived class to make the synchronization constraints less stringent, that is *contravariant*. Examples exist in favor of both approaches; as a consequence we decided to allow both manners of overriding. From the point of view of the inheritance anomaly, Frølund’s methodology is subject to the usual problems related to method guards, i.e. the history dependent variants of the anomaly.

Meseguer [20], analyzed the problem of the inheritance anomaly in the context of his rewriting logic based language Maude [21]. 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 any adaption to imperative object oriented languages such as Java.

Different lines of work were taken by Matsuoka and Yonezawa: the first based on the notion of *reflection* [19], the second aiming at reducing the amount of synchronization code to a minimum [19].

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. Similar results were

obtained using the *synchronization patterns* [17] and *synchronization rings* [12] methodologies.

9 Conclusions

We introduced Jeeg, a dialect of Java where 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 us to offer an excellent balance between expressiveness and computational overhead. 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.

With regards to the Jeeg compiler, we are exploring the possibility of optimizing the LTL evaluation procedure by using ad-hoc static-analysis techniques.

The current implementation of the Jeeg compiler is available online at www.brics.dk/~milicia/Jeeg/.

Acknowledgements

We would like to thank Oliver Möeller who extensively reviewed an early version of the paper and answered all our questions about LTL model checking. Thanks to Luigi Santocanale for the early discussions on object traces. A number of people read the original manuscript, we are grateful to all of them for their feedback.

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 pathexplorer. 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] K. Havelund and G. Rosu. Testing linear temporal logic formulae on finite execution traces. Technical Report TR 01-08, RIACS, May 2001.
- [12] D. Holmes. *Synchronization Rings – Composable Synchronization for Object-Oriented Systems*. PhD thesis, Macquarie University, 1999.
- [13] 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.

- [14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Longtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
- [15] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.

- [23] 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.
- [24] 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.
- [25] G. Rosu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical Report TR 01-15, RIACS, May 2001.
- [26] 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.
- [27] Wolfgang Thomas. Languages, automata and logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, Amsterdam, 1990. Elsevier Science Publishers.
- [28] L. Zuck. *Past Temporal Logic*. PhD thesis, Weizmann Institute, 1986.

A The Jeeg Document type definition (DTD)

```
<!ELEMENT Jeeg (Class)>
<!ELEMENT Class (Method*)>
<!ATTLIST Class
    name CDATA #REQUIRED
    super CDATA #IMPLIED
    version CDATA "j1"
>
<!ELEMENT Method (Previous | Sometime | Always | Since
    | And | Or | Implies | BooleanExpression)>
<!ATTLIST Method name CDATA #REQUIRED>
<!ELEMENT Constraint (#PCDATA)>
<!ELEMENT BooleanExpression (#PCDATA)>
<!ELEMENT Previous (Previous | Sometime | Always | Since
    | And | Or | Implies | BooleanExpression)>
<!ELEMENT Sometime (Previous | Sometime | Always | Since
    | And | Or | Implies | BooleanExpression)>
<!ELEMENT Always (Previous | Sometime | Always | Since
    | And | Or | Implies | BooleanExpression)>
<!ELEMENT Since (Arg, Arg)>
<!ELEMENT And (Arg, Arg)>
<!ELEMENT Or (Arg, Arg)>
<!ELEMENT Implies (Arg, Arg)>
<!ELEMENT Arg (Previous | Sometime | Always | Since |
    And | Or | Implies | BooleanExpression)>
```

B The Java code generated from the Buffer example

In this section we show the code generated by the Jeeg compiler from the `Buffer` example presented in §3. Interface and class names referring to the Jeeg runtime-system are normally fully qualified. However, to keep names short here we write, for instance, `PropositionalFormula` instead of the fully qualified `org.brics.gm.jeeg.formulae.PropositionalFormula`.


```

import org.brics.gm.jeeg.formulae.*;
import org.brics.gm.jeeg.events.*;

public class Buffer {

    protected SyncManager _sync = null;
    protected void
    _registerSyncManager(SyncManager s) {
        this._sync = s;
        s.makeStep();
    }

    protected class SyncManager {

        public SyncManager() {
        }

        protected class BufferputProp1
        implements PropositionalFormula {
            public boolean eval() {
                return (current <= MAX);
            }
        }

        protected class BuffergetProp1
        implements PropositionalFormula {
            public boolean eval() {
                return (current > 0);
            }
        }
    }

    protected TemporalPropositionalFormula
    TBufferputProp1 =
        new TemporalPropositionalFormula(
            new BufferputProp1());
    protected TemporalPropositionalFormula
    TBuffergetProp1 =
        new TemporalPropositionalFormula(
            new BuffergetProp1());
    protected TemporalFormula
    getConstr = TBuffergetProp1;
}

```

The include statements above refer the Jeeg run-time system.

Every Jeeg class requires a synchronization manager as described in §1. The method `_registerSyncManager` is used at object creation time to specify which synchronization manager will take care of the class.

The synchronization manager is inserted as an *inner class*.

Propositional formulae ($p \in AP$), are wrapped into classes. Observe that the Java inner class mechanism grants the synchronization manager full access on the private/protected attributes of the surrounding class.

Appropriate temporal formulae representing the synchronization constraints of the object are instantiated. The classes used are taken from the Jeeg run-time system.

```

protected TemporalFormula
    putConstr = TBufferputProp1;

protected int UNKNOWN = -1;
protected int get = 1;
protected int put = 2;
protected Event event =
    new Event(UNKNOWN);

public void makeStep() {
    getConstr.eval();
    putConstr.eval();
}

public boolean getpre() {
    return getConstr.getCurrentValue();
}

public void acquireget()
    throws Exception {
    while (! getpre()) {
        Buffer.this.wait();
    }
    event = new Event(get,
        System.currentTimeMillis());
}

public void releaseget()
    throws Exception {
    event.setEndTime(
        System.currentTimeMillis());
    makeStep();
    Buffer.this.notifyAll();
}

public boolean putpre() {
    return putConstr.getCurrentValue();
}

public void acquireput()
    throws Exception {
    while (! putpre()) {
        Buffer.this.wait();
    }
    event = new Event(put,
        System.currentTimeMillis());
}

```

An unique identifier is generated for each method. This is used to identify the events. At object creation time, when the history of the object is empty the `event` variable takes the UNKNOWN value.

The `makeStep` method evaluates the synchronization constraints. This is done using the algorithm described in §1 after every method execution.

For every method `M` the synchronization manager has two methods `acquireM()` to be called before the execution of `M`'s actual code and `releaseM()` to be called when `M`'s execution is completed.

Synchronization constraints are wrapped into functions name `Mpre()` where `M` is the method the constraint refers to. The conditions are evaluated in the `acquireM()` method.

```

public void releaseput()
throws Exception {
    event.setEndTime(
        System.currentTimeMillis());
    makeStep();
    Buffer.this.notifyAll();
}
} //SyncManager

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

Buffer(int max) {
    MAX = max;
    buf = new Object[MAX];
    this._registerSyncManager(
        new SyncManager());
}

public synchronized Object get()
throws Exception {
    ((SyncManager) _sync).acquireget();
    current--;
    Object ret = buf[current];
    ((SyncManager) _sync).releaseget();
    return ret;
}

public synchronized void put(Object v)
throws Exception {
    ((SyncManager) _sync).acquireput();
    buf[current] = v;
    current++;
    ((SyncManager) _sync).releaseput();
}
}

```

The `releaseM()` method takes care of issuing the `notifyAll()` to wake up any thread waiting. A call to the `makeStep()` method takes care of evolving the object's history.

When the object is created a new synchronization manager is registered. The call to `_registerSyncManager` will take care of initializing the history of the object as well.

Calls to the `acquire` and `release` methods are inserted respectively at the beginning and at the end of the method code. This implements a simple mechanism of *method call interception*.

Recent BRICS Report Series Publications

- RS-03-6 Giuseppe Milicia and Vladimiro Sassone. *Jeeg: Temporal Constraints for the Synchronization of Concurrent Objects*. February 2003. 41 pp. Short version appears in Fox and Getov, editors, *Joint ACM-ISCOPE Conference on Java Grande, JGI '02 Proceedings, 2002*, pages 212–221.
- RS-03-5 Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. *Precise Analysis of String Expressions*. February 2003. 15 pp.
- RS-03-4 Marco Carbone and Mogens Nielsen. *Towards a Formal Model for Trust*. January 2003.
- RS-03-3 Claude Crépeau, Paul Dumais, Dominic Mayers, and Louis Salvail. *On the Computational Collapse of Quantum Information*. January 2003. 31 pp.
- RS-03-2 Olivier Danvy and Pablo E. Martínez López. *Tagging, Encoding, and Jones Optimality*. January 2003. To appear in Degano, editor, *Programming Languages and Systems: Twelfth European Symposium on Programming, ESOP '03 Proceedings, LNCS, 2003*.
- RS-03-1 Vladimiro Sassone and Pawel Sobocinski. *Deriving Bisimulation Congruences: 2-Categories vs. Precategories*. January 2003. To appear in Gordon, editor, *Foundations of Software Science and Computation Structures, FoSSaCS '03 Proceedings, LNCS, 2003*.
- RS-02-52 Olivier Danvy. *A New One-Pass Transformation into Monadic Normal Form*. December 2002. 16 pp. To appear in Hedin, editor, *Compiler Construction, 12th International Conference, CC '03 Proceedings, LNCS, 2003*.
- RS-02-51 Gerth Stølting Brodal, Rolf Fagerberg, Anna Östlin, Christian N. S. Pedersen, and S. Srinivasa Rao. *Computing Refined Bureaucratic Trees in Cubic Time*. December 2002. 14 pp.
- RS-02-49 Mikkel Nygaard and Glynn Winskel. *HOPLA—A Higher-Order Process Language*. December 2002. 18 pp. Appears in Brim, Jančar, Křetínský and Antonín, editors, *Concurrency Theory: 13th International Conference, CONCUR '02 Proceedings, LNCS 2421, 2002*, pages 434–448.