Temporal Constraints for Concurrent Object Synchronisation

WOOD, Warsaw 12.04.03

Vladimiro Sassone

with Giuseppe Milicia

University of Sussex, MYTHS

≪ ⋘ ⋙ ≫

What does Inheritance do, after all?

```
class Buffer {
    void put(Object v) { ...; }
    void get() { ...; }
    ...
}

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

What does Inheritance do, after all?

```
class Buffer {
    void put(Object v) { ...; }
    void get() { ...; }
    ...
}
class Lock {
    ...
    void lock() { ...; }
    void unlock() { ...; }
}
```

Influence Buffer inheriting behaviour from Lock.

```
class LockableBuffer extends Buffer, Lock{ }
```

What does Inheritance do, after all?

```
class Buffer {
    void put(Object v) { ...; }
    void get() { ...; }
    ...
}

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

Influence Buffer inheriting behaviour from Lock.

```
class LockableBuffer extends Buffer, Lock{ }
```

Do you expect a Buffer which is locked/unlocked via Lock?

Objects and Concurrency

Objects: A fundamental, state-of-the-art concept for engineering complex software systems. (Design Patterns, Refactoring, . . .)

Concurrency: A fundamental technology to meet today's demands on software functionalities. (Internet, Mobile and Embedded Devices, Software Agents, . . .)

≪ ⋘ ⋙ ≫

Objects and Concurrency

Objects: A fundamental, state-of-the-art concept for engineering complex software systems. (Design Patterns, Refactoring, . . .)

Concurrency: A fundamental technology to meet today's demands on software functionalities. (Internet, Mobile and Embedded Devices, Software Agents, . . .)

Alas, a difficult marriage

Synchronisation of concurrent activities and inheritance do not mix:

Inheritance Anomaly (Yonezawa [1987])

≪ ⋘ ⋙ ≫

Objects and Concurrency

Objects: A fundamental, state-of-the-art concept for engineering complex software systems. (Design Patterns, Refactoring, . . .)

Concurrency: A fundamental technology to meet today's demands on software functionalities. (Internet, Mobile and Embedded Devices, Software Agents, . . .)

Alas, a difficult marriage

Synchronisation of concurrent activities and inheritance do not mix:

Inheritance Anomaly (Yonezawa [1987])

So bad to justify banning inheritance from OO languages! (America [1991])

Objects and Concurrency

Objects: A fundamental, state-of-the-art concept for engineering complex software systems. (Design Patterns, Refactoring, . . .)

Concurrency: A fundamental technology to meet today's demands on software functionalities. (Internet, Mobile and Embedded Devices, Software Agents, . . .)

Alas, a difficult marriage

Synchronisation of concurrent activities and inheritance do not mix:

Inheritance Anomaly (Yonezawa [1987])

So bad to justify banning inheritance from OO languages! (America [1991])

The plan

- *Explain the phenomenon via examples;*
- *Illustrate the driving lines of the main existing approaches;*
- *Design and implementation of the programming language Jeeg.*

Concurrency and Interference

The problem: $x := 0; (\ x := x + 1 \| x := x + 2\ )$. Then, $x \in \{1, 2, 3\}$.

≪ ⋘ ⋙ ≫

Concurrency and Interference

The problem: $x := 0; (\ x := x + 1 \| x := x + 2\ ).$ Then, $x \in \{1, 2, 3\}$.

The solutions:

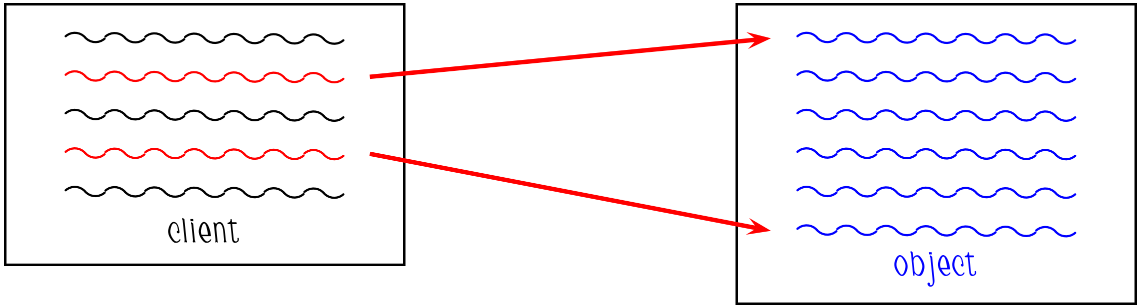
- Operational Mechanisms: Semaphores and Locks, ...
- Linguistic Constructs: Critical Regions and Monitors, ...
- Alternative Models: Message Passing, Resource-Based, ...

Their relevance: In the end the problem is in the concurrency model

The Java Concurrency Model

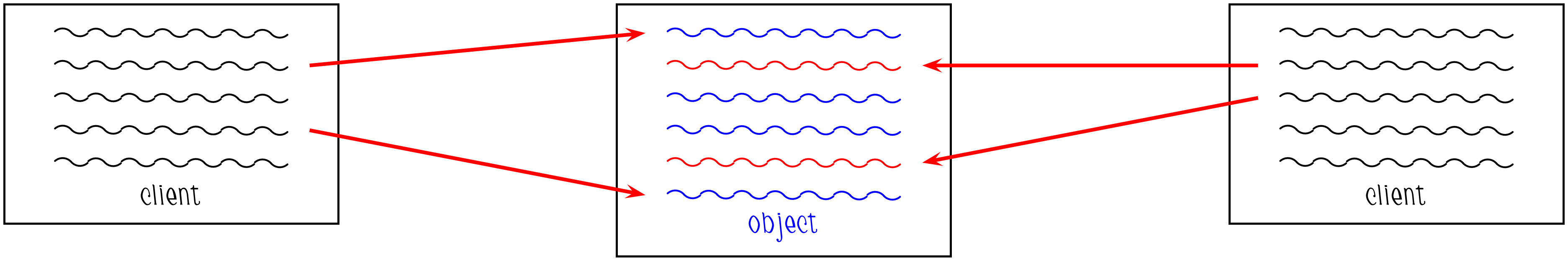
```java
public class Buffer {
    protected Object[] buf;
    protected int MAX, current = 0;

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

Business and Synchronisation Code



In sequential programming, clients can be asked to behave well. E.g., don't get unless you have put. (Synchronisation code and Business code.)

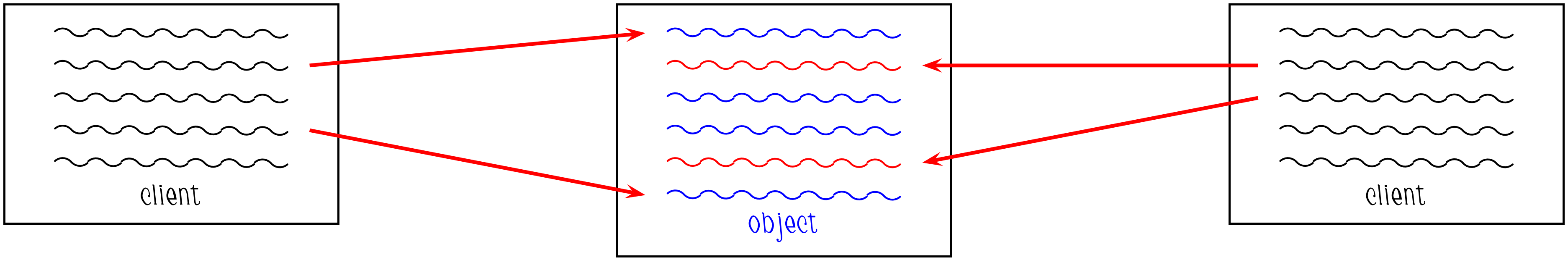
Business and Synchronisation Code



In sequential programming, clients can be asked to behave well. E.g., don't get unless you have put.

In concurrency, the resource must contain synchronisation code. This results essentially in methods not being available at certain moments in time.

Business and Synchronisation Code



In sequential programming, clients can be asked to behave well. E.g., don't get unless you have put.

In concurrency, the resource must contain synchronisation code. This results essentially in methods not being available at certain moments in time.

Concurrent object oriented programs in common programming languages consist of business code inextricably interwoven with synchronisation code.

The Inheritance Anomaly

Inheritance Anomaly: Adding a new method morally unrelated, forces the redefinition of all other methods of a class.

≪ ⋘ ⋙ ≫

The Inheritance Anomaly

Inheritance Anomaly: Adding a new method morally unrelated, forces the redefinition of all other methods of a class.

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

≪ ⋘ ⋙ ≫

The Inheritance Anomaly

Inheritance Anomaly: Adding a new method morally unrelated, forces the redefinition of all other methods of a class.

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

Add a method freeze.

Chances are that the synchronisation code in Buffer must be totally rewritten for that.

All approaches to the anomaly so far consist of disentangling business and synchronisation code. None is very successful.

Partitioning of States

State Partition: Introduce an explicit partition of the object's state, and explicit enabling conditions for methods.

Example. In the case of Buffer, choose empty, partial, full and the declarations:

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

≪ ⋘ ⋙ ≫

Partitioning of States

State Partition: Introduce an explicit partition of the object's state, and explicit enabling conditions for methods.

Example. In the case of Buffer, choose empty, partial, full and the declarations:

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

Then

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

≪ ⋘ ⋙ ≫

Partition of States

This solves the problem only very partially.

Partition of States

This solves the problem only very partially.

> Consider adding get2 which retrieves two elements at once. Then, the partition empty and full is not enough anymore.

Need to distinguish those states where there is exactly one element: single.

Partition of States

This solves the problem only very partially.

> Consider adding get2 which retrieves two elements at once. Then, the partition empty and full is not enough anymore.

Need to distinguish those states where there is exactly one element: single.

Correspondingly, refine it to be:

```
get2: requires not empty or single

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

History-Sensitiveness of Acceptable States

When methods' enabling depends on the history of objects, we have a form of the anomaly so-called history-sensitive.

For instance, a method withdraw available only after a method authenticate has been completed.

History-Sensitiveness of Acceptable States

When methods' enabling depends on the history of objects, we have a form of the anomaly so-called history-sensitive.

For instance, a method withdraw available only after a method authenticate has been completed.

> To exemplify, we want to add to Buffer a method gget enabled only if the last method invoked of Buffer was other than get.

History Buffer

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

History Buffer, again

```
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();
        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;
    }
}
```

Modification of Acceptable States

Anomaly when mix-in classes are used to add behaviour to object via multiple inheritance.

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

Trying to influence the enabling conditions of a class, by inheritance.

```
class LockableBuffer extends Buffer, Lock{ }
```

≪ ⋘ ⋙ ≫

Modification of Acceptable States

Anomaly when mix-in classes are used to add behaviour to object via multiple inheritance.

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

Trying to influence the enabling conditions of a class, by inheritance.

```
class LockableBuffer extends Buffer, Lock{ }
```

Of course, this does no much towards having a lockable buffer, in any language I know of.

Question: Is the Inheritance Anomaly nonsense or a genuine problem?
If you look at it from the OO standpoint, it is genuine.

JEEG

Jeeg tackles the (History-Sensitive) Inheritance Anomaly. It is:

- an aspect-oriented superimposition of two separate languages
  - Java (no synchronized(), wait(), notify(), and notifyAll() for business code);
  - Linear Time Temporal Logic for synchronisation code (method guards).

JEEG

Jeeg tackles the (History-Sensitive) Inheritance Anomaly. It is:

- an aspect-oriented superimposition of two separate languages
  - Java (no synchronized(), wait(), notify(), and notifyAll() for business code);
  - Linear Time Temporal Logic for synchronisation code (method guards).

```
public class MyClass {
  sync {
    m : φ;
    ....
  }
  ...// Standard Java class definition
}
```

m is a method id and ϕ, the guard, is a formula in a given constraint language. When m is invoked, the thread is kept on hold unless ϕ. When the condition is true, all waiting threads are awaken. m is implicitly synchronized.

JEEG

Jeeg tackles the (History-Sensitive) Inheritance Anomaly. It is:

- an aspect-oriented superimposition of two separate languages
  - Java (no synchronized(), wait(), notify(), and notifyAll() for business code);
  - Linear Time Temporal Logic for synchronisation code (method guards).

```
public class MyClass {
  sync {
    m : φ;
    ....
  }
  ...// Standard Java class definition
}
```

m is a method id and ϕ, the guard, is a formula in a given constraint language. When m is invoked, the thread is kept on hold unless ϕ. When the condition is true, all waiting threads are awaken. m is implicitly synchronized.

If ϕ is a boolean expression, this is just a declarative version of Java concurrency.

The logic

Logic: a trade-off between expressiveness and efficiency: its formulae must be verified at every method invocation!

The logic

Logic: a trade-off between expressiveness and efficiency: its formulae must be verified at every method invocation!

Linear temporal logic (past tense)

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

AP are pure boolean expressions with no:

- side-effects,
- references to objects.
- method invocations,
- and it only refers to private/protected fields of the class it belongs to.

Derived connectives:

$$\phi \;\&\&\; \psi \triangleq !(!\phi \;||\, !\psi); \quad \text{Sometime } \phi \triangleq \text{true Since } \phi; \quad \text{Always } \phi \triangleq \text{!Sometime } !\phi.$$

The logic

Logic: a trade-off between expressiveness and efficiency: its formulae must be verified at every method invocation!

Linear temporal logic (past tense)

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

AP are pure boolean expressions with no:

- side-effects,
- references to objects,
- method invocations,
- and it only refers to private/protected fields of the class it belongs to.

Derived connectives:

$$\phi \,\&\&\, \psi \triangleq !(!\phi \mid\mid !\psi); \quad \text{Sometime } \phi \triangleq \text{true Since } \phi; \quad \text{Always } \phi \triangleq !\text{Sometime } !\phi.$$

This yield a rather expressive language CL, yet easy to implement.

An Object's History

A generic computation π from o's perspective.

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

≪ ⋘ ⋙ ≫

An Object's History

A generic computation π from o's perspective.

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

Here only the part of $h_{j_k}^k$ containing the values of private/protected, non-reference variables of o, say σ_k, can affect evaluation. Therefore, we take

$$\mathcal{H}_o(\pi) \equiv \sigma_0 \stackrel{m_1}{\rightarrow} \sigma_1 \stackrel{m_2}{\rightarrow} \sigma_2 \stackrel{m_3}{\rightarrow} \sigma_3 \ldots$$

≪ ⋘ ⋙ ≫

An Object's History

A generic computation π from o's perspective.

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

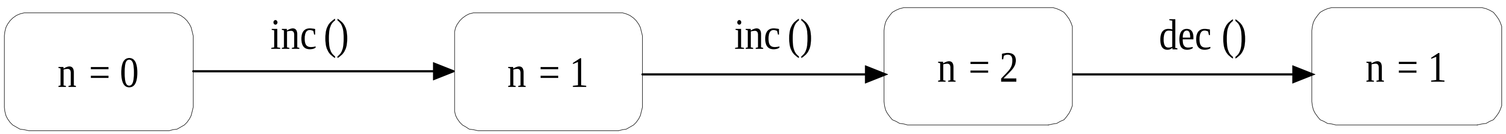
Here only the part of $h_{j_k}^k$ containing the values of private/protected, non-reference variables of o, say σ_k, can affect evaluation. Therefore, we take

$$\mathcal{H}_o(\pi) \equiv \sigma_0 \stackrel{m_1}{\rightarrow} \sigma_1 \stackrel{m_2}{\rightarrow} \sigma_2 \stackrel{m_3}{\rightarrow} \sigma_3 \ldots$$

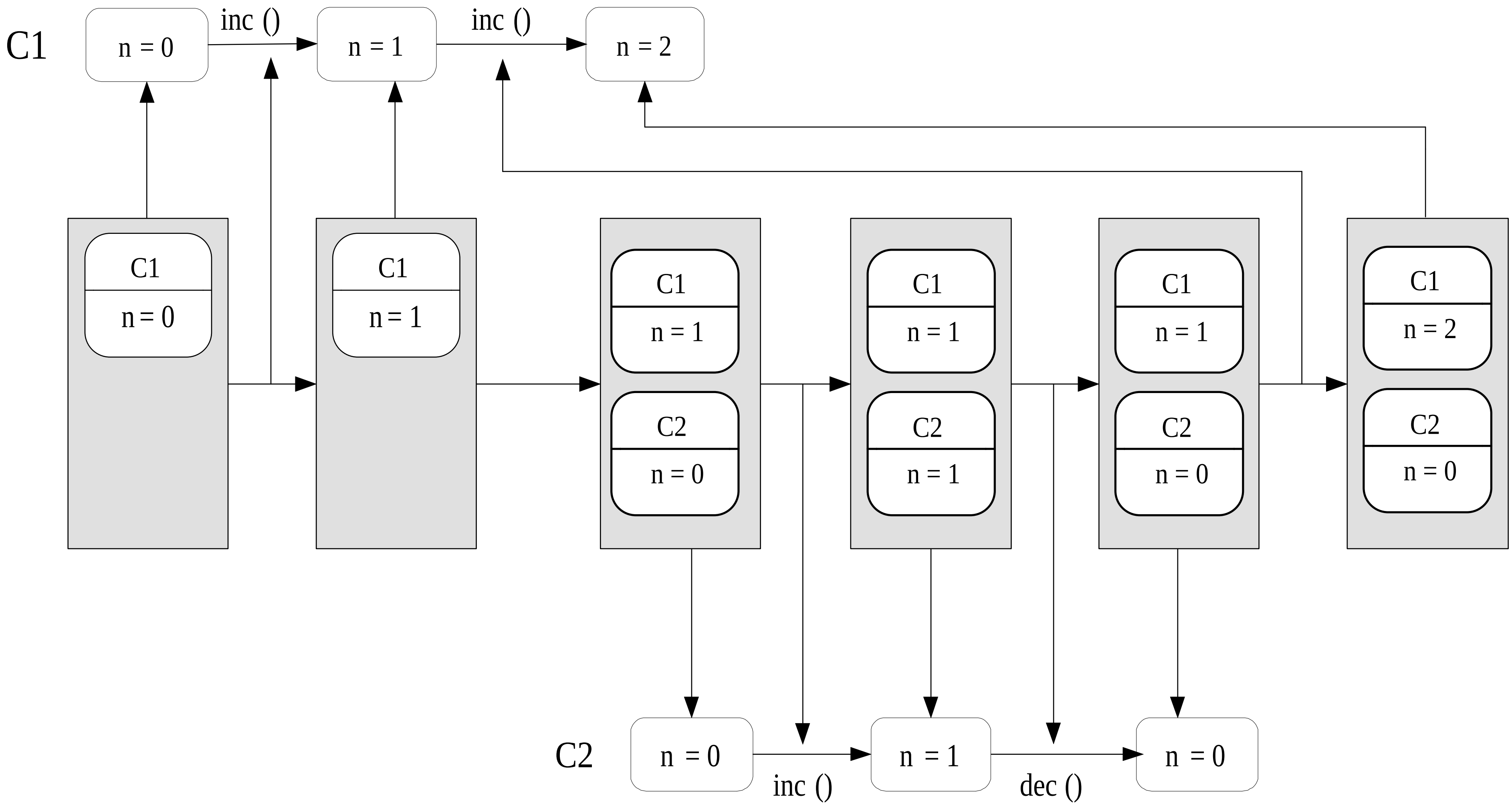
We think of $H_o(\pi)$ as

$$\mathcal{H}_o \equiv \sigma_0 \sigma_1 \sigma_2 \sigma_3 \ldots$$

where σ_i binds the special identifier `event` to (a value representing method) $\mathtt{m}_i$.



Concurrent Objects' Histories



Interpretation of Formulae on Object Histories

Let Σ denote $\mathcal{H}_o(\pi)$. 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.

$$\Sigma_k \models p \quad \texttt{iff} \quad \sigma_k \models p \quad (p \texttt{ is true at } \sigma_k)$$

$$\Sigma_k \models !\phi \quad \texttt{iff} \quad \text{not } \Sigma_k \models \phi$$

$$\Sigma_k \models \phi \mid\mid \psi \quad \texttt{iff} \quad \Sigma_k \models \phi \texttt{ or } \Sigma_k \models \psi$$

$$\Sigma_k \models \text{Previous } \phi \quad \texttt{iff} \quad k > 0 \text{ and } \Sigma_{k-1} \models \phi$$

$$\Sigma_k \models \phi \text{ Since } \psi \quad \texttt{iff} \quad \Sigma_j \models \psi \text{ for some } j \leq k, \text{ and } \Sigma_i \models \phi \text{ for all } j < i \leq k$$

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

Buffer in JEEG

```
public class Buffer {
    sync {
        put :  current < MAX;
        get :  current > 0;
    }
    protected Object[] buf;
    protected int MAX, current = 0;
    Buffer(int max) {
        MAX = max; buf = new Object[MAX];
    }

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

    public void put(Object v) throws Exception {
        buf[current++] = v;
    }
}
```

History Buffer 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 {
        Object ret = buf[--current];
        return ret;
    }
}
```

Lockable Buffer in JEEG

```
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() { }
}
```

Expressiveness of JEEG

It is generally hard to formalise to what extent the anomaly is removed.
Nicely, Jeeg allows for a "quantitative" analysis.

Expressiveness of JEEG

It is generally hard to formalise to what extent the anomaly is removed. Nicely, Jeeg allows for a "quantitative" analysis.

Expressiveness of LTL: A set of state sequences X is the set of all Σs that satisfy a given ϕ if and only if X is a star-free regular language. (Zuck [1986])

Star-free Regular Languages:

$$re ::= \epsilon \mid a \mid re \cdot re \mid re + re \mid \neg r \qquad (\mid re*)$$

Expressiveness of JEEG

It is generally hard to formalise to what extent the anomaly is removed. Nicely, Jeeg allows for a "quantitative" analysis.

Expressiveness of LTL: A set of state sequences X is the set of all Σs that satisfy a given ϕ if and only if X is a star-free regular language. (Zuck [1986])

Star-free Regular Languages:

$$re ::= \epsilon \mid a \mid re \cdot re \mid re + re \mid \neg r \qquad (\mid re*)$$

State for C: $p \in \mathtt{A}_\mathtt{C} \subset \mathtt{AP}$; Sequence of states: $P \in \mathtt{A}_\mathtt{C}^*$. $(\Sigma \models P \text{ iff } \Sigma_k \models P_k)$

Theorem (Characterizing CL). For ϕ a formula on C, $X = \{\Sigma \mid \Sigma \models \phi\}$ iff there exists re on $\mathtt{A}_\mathtt{C}$ such that $\Sigma \in X$ iff $\Sigma \models P$ for some $P \in re$.

Expressiveness of JEEG

It is generally hard to formalise to what extent the anomaly is removed.
Nicely, Jeeg allows for a "quantitative" analysis.

Expressiveness of LTL: A set of state sequences X is the set of all Σs that satisfy a given ϕ if and only if X is a star-free regular language. (Zuck [1986])

Star-free Regular Languages:

$$re ::= \epsilon \mid a \mid re \cdot re \mid re + re \mid \neg r \qquad (\mid re*)$$

State for C: $p \in \mathtt{A_C} \subset \mathtt{AP}$; Sequence of states: $P \in \mathtt{A_C^*}$. $(\Sigma \models P \text{ iff } \Sigma_k \models P_k)$

Theorem (Characterizing CL). For ϕ a formula on C, $X = \{\Sigma \mid \Sigma \models \phi\}$ iff there exists re on $\mathtt{A_C}$ such that $\Sigma \in X$ iff $\Sigma \models P$ for some $P \in re$.

Special case: Only atomic propositions of the kind `event == m`.

Then CL would capture precisely those sequences of events which are star-free regular languages (i.e., enforce synchonisation policies so expressible).

Examples

HistoryBuffer: the temporal constraint

`Previous event != get`

can be expressed by the following star-free regular expressions.

$$\neg(A^* \cdot \texttt{get}) \qquad \text{where } A^* \triangleq \epsilon + \neg\epsilon.$$

The temporal constraint

$$\texttt{Sometime m} \triangleq \texttt{true Since m.}$$

corresponds to

$$A^* \cdot \texttt{m} \cdot A^*.$$

Limitations of LTL: No Counting

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

Define a class SeizableResource which allows exclusive access to the shared resource: An additional method exclusiveRequest must be provided.

Clearly, this leads to identify a pattern of events such as:

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

It is well known that this language is not regular. Methods request and release will have to be redefined. The anomaly surfaces again here.

Runtime Evaluation of CL Expressions

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

Traditionally: build a Buchi automata to 'model-check' sequences. Dealing with past tense operators gives us an advantage: an 'online' algorithm.

≪ ⋘ ⋙ ≫

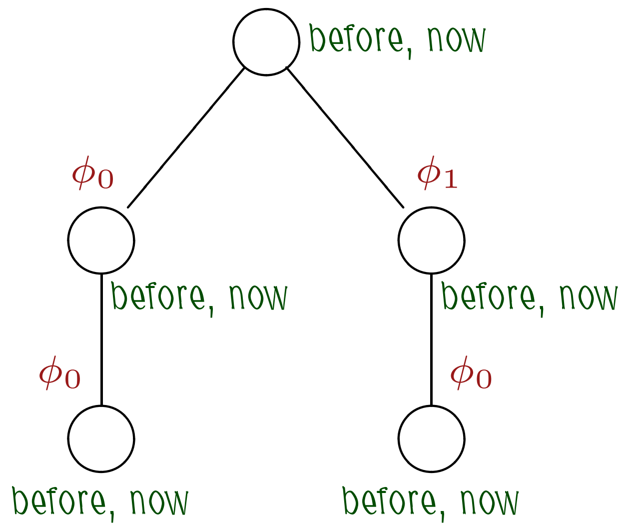
Runtime Evaluation of CL Expressions

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

Traditionally: build a Buchi automata to 'model-check' sequences. Dealing with past tense operators gives us an advantage: an 'online' algorithm.

- Build the syntax tree of the formula;
- Associate variables before and now to every node, initially set to `false`;
- Visit the tree depth-first and simultaneously assign ϕ.before := ϕ.now and ϕ.now as follows.

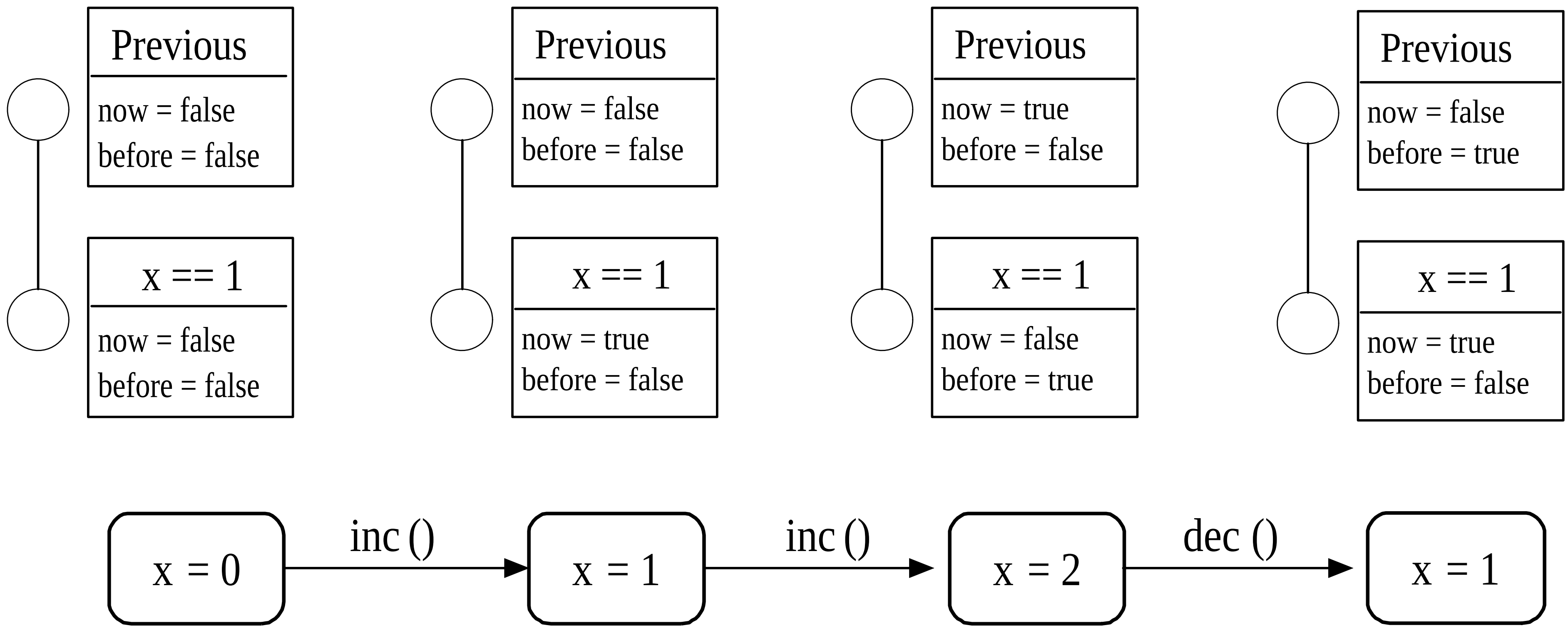
previous	now := ϕ_0.before
since	now := ϕ_1.now *or* (before *and* ϕ_0.now)
or	now := ϕ_0.now *or* ϕ_1.now
not	now := *not* ϕ_0.now
AP	now := *eval*(ϕ)



An Example

Example: Let us consider the evaluation of the temporal formula

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



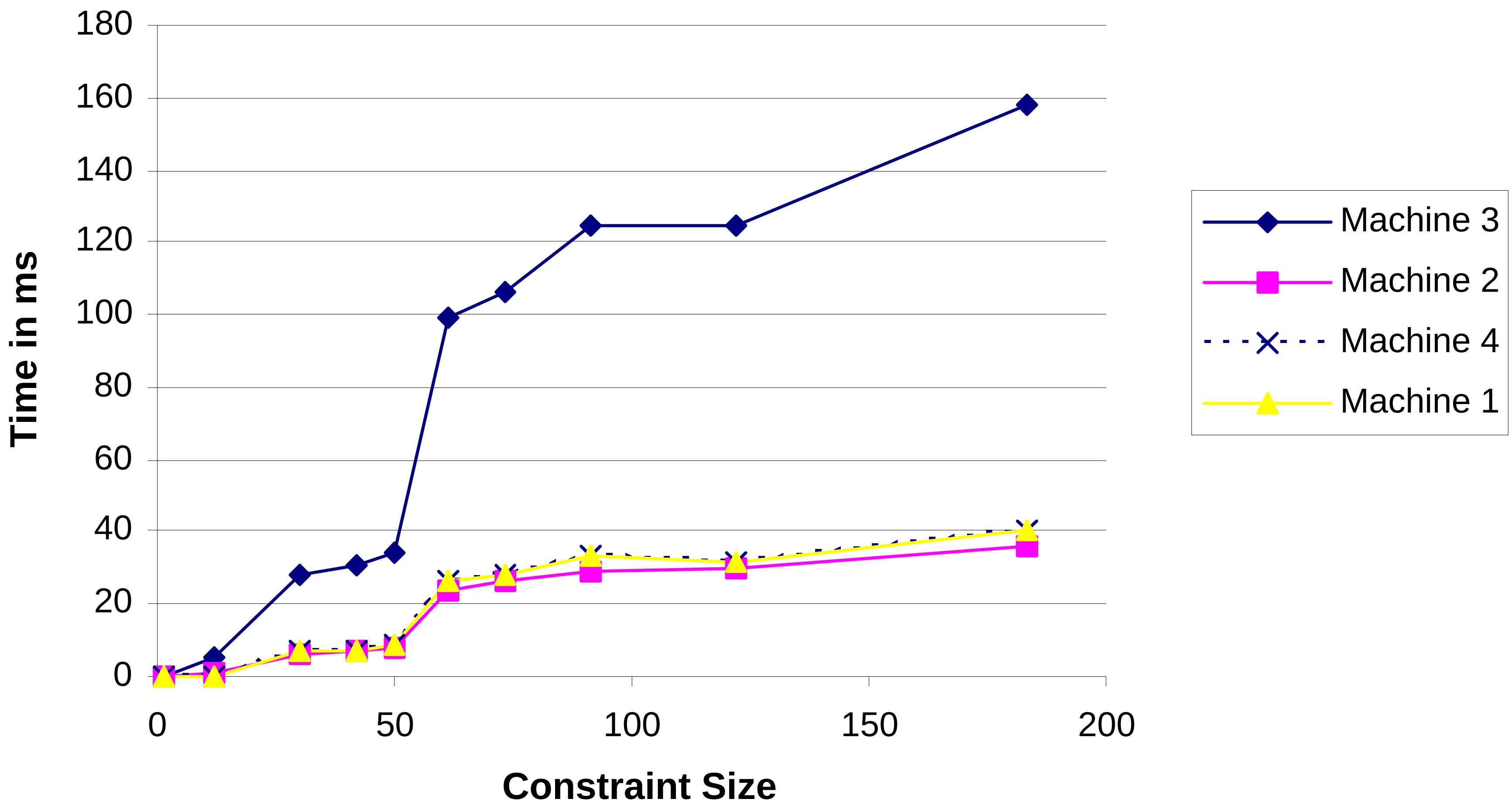
The Synchronisation Manager

Formulae must be evaluated after every method execution. This is done by a synchronization manager via Method Call Interception. It

- takes control at method call and checks (not evaluates) the constraint for the method.
- If it holds, control goes to the method code; otherwise the synchronization manager performs a wait(), putting the object to sleep.
- After the method execution, control shifts back to the manager, which now re-evaluates the synchronization constraints.
- After updating the formulae logic value, the manager issues a notifyAll() statement. Blocked methods may then attempt to proceed again.

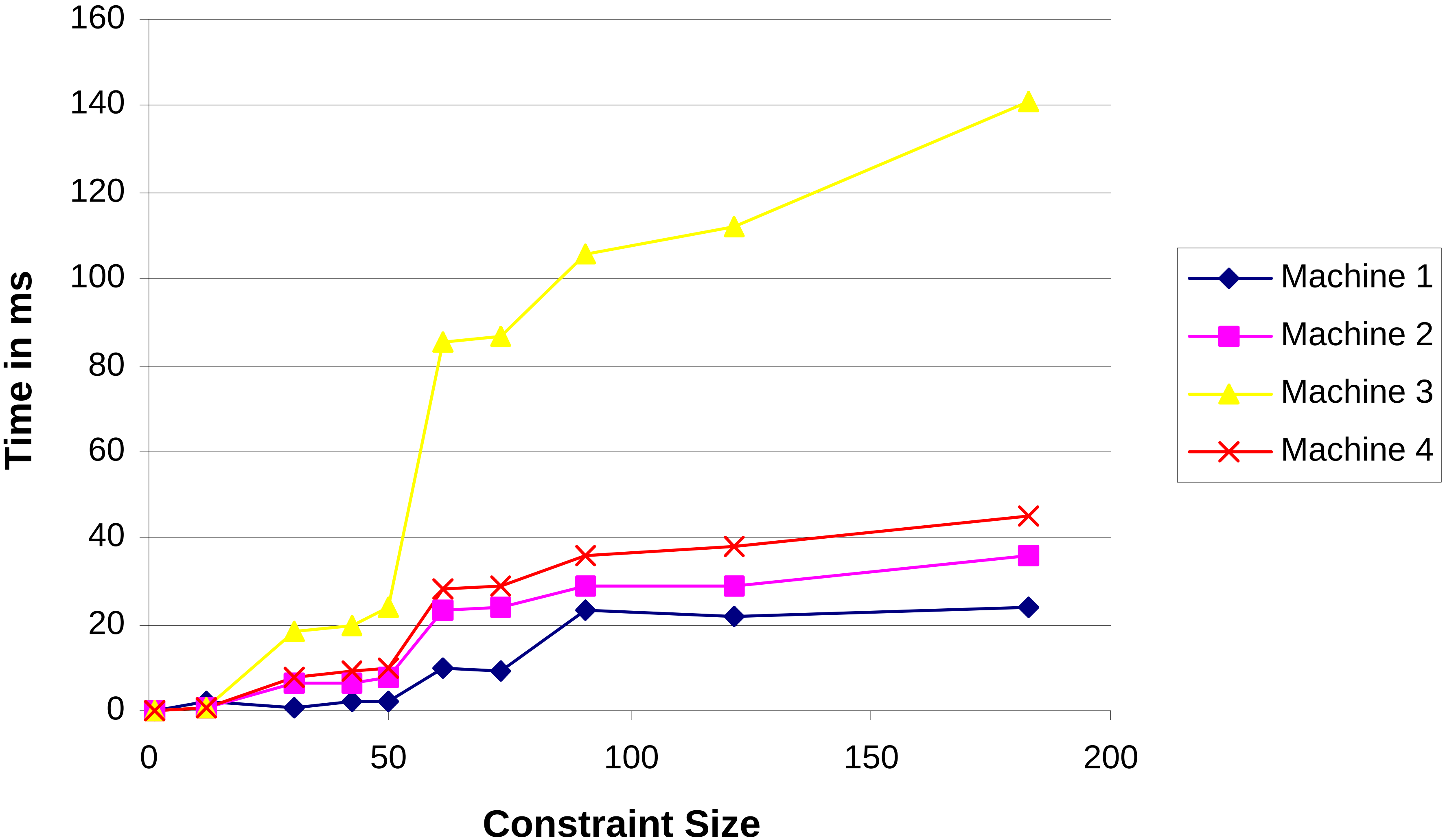
To have access to private/protected fields, the synchronization manager an inner class of the object it manages.

Benchmarks: Object Creation



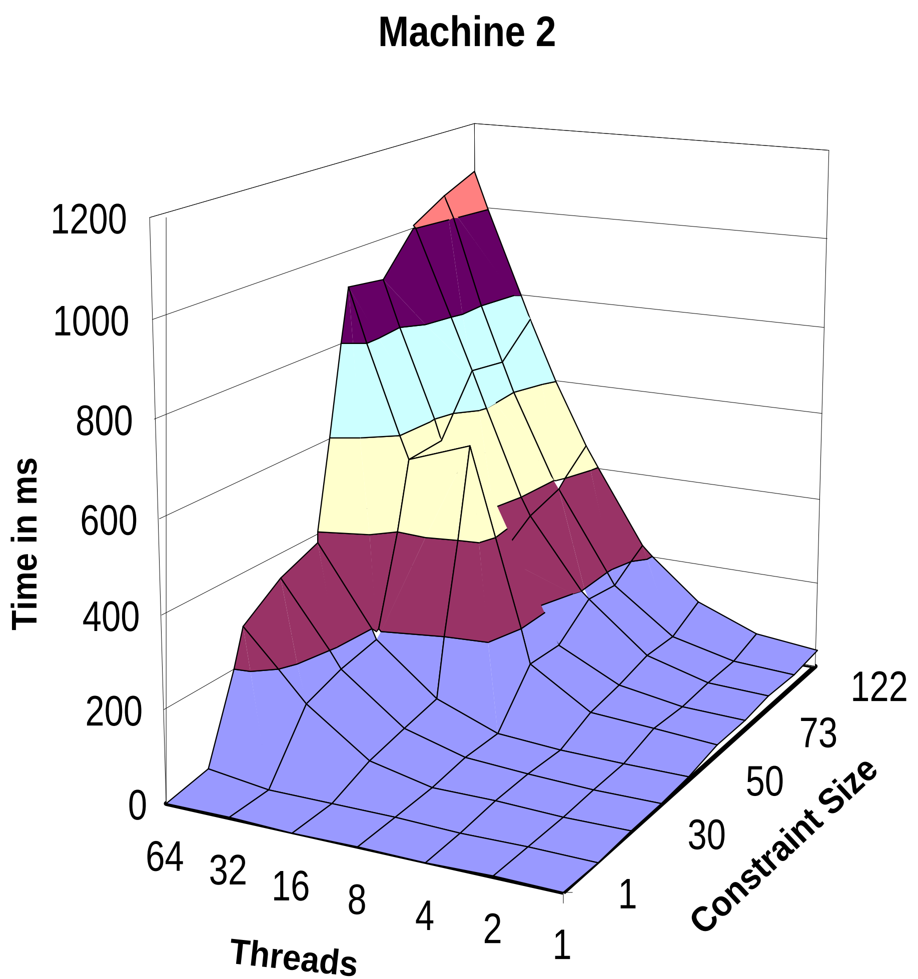
Object creation triggers the creation of data structures for formulas

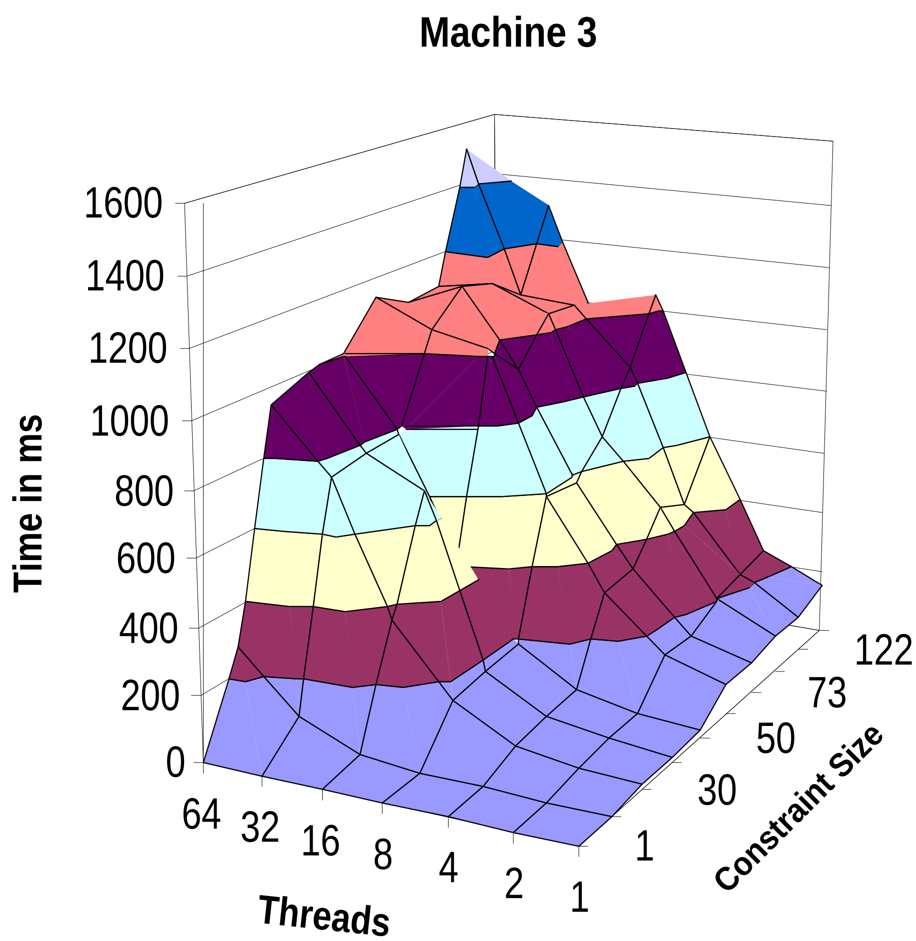
Benchmarks: Method Call



Method calls trigger the evaluation of formulas

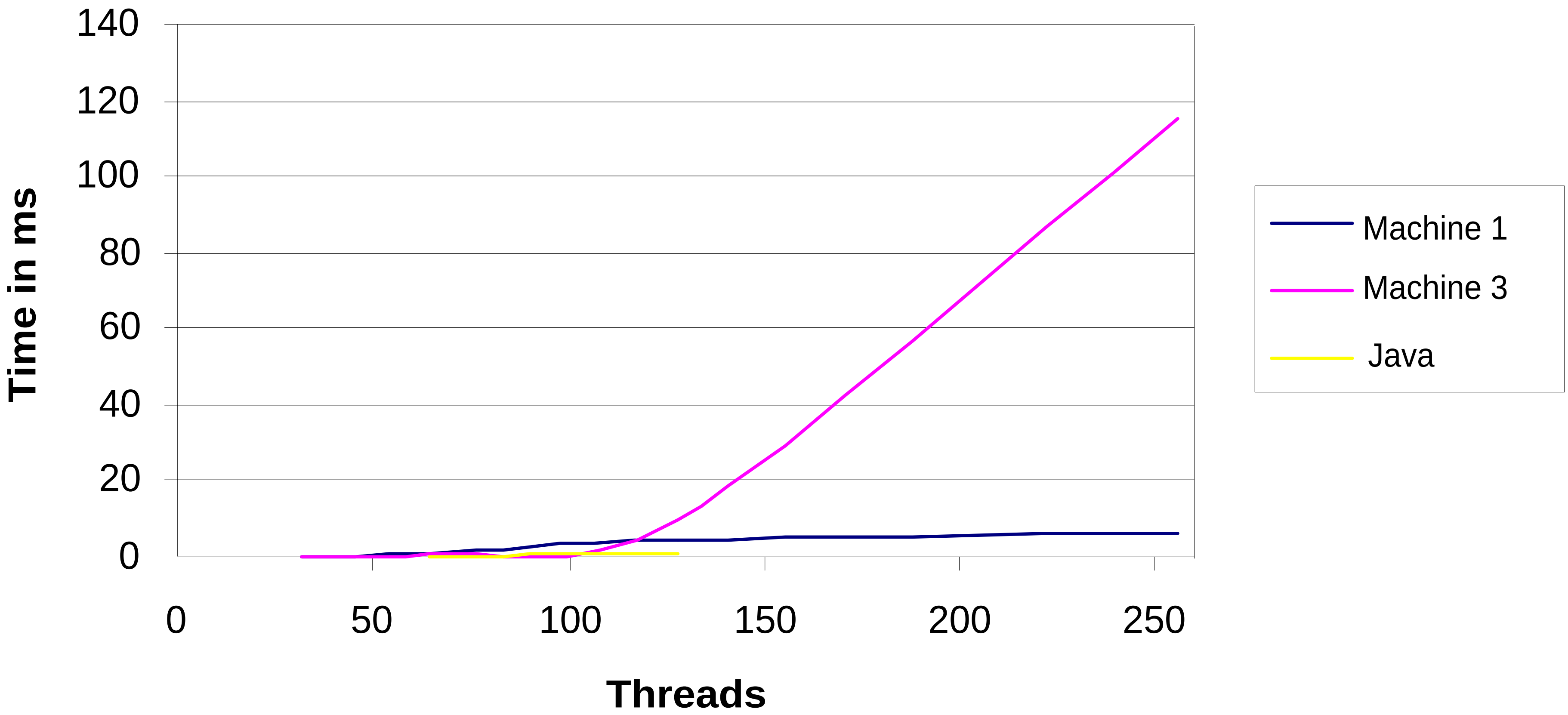
Benchmarks: Details of Method Call





Formulae evaluation triggers mutual exclusion protocols

Benchmarks: Comparison



However, synchronisation must be performed also in Java!

Performance Evaluation

Testing shows that:

- Under low-load (below 70 threads) even complex synchronization constraints yield little performance overhead.
- Low-end machines face worse scalability problems due object locking: The slower the evaluation algorithm, the longer a large number of threads are kept waiting.

≪ ⋘ ⋙ ≫

Conclusion

Jeeg

- Synchronization constraints written in LTL and specified in a aspect-oriented, declarative manner.
- CL is helpful in treating the inheritance anomaly.
- Characterisation of CL in terms of regular languages
- Efficiently implementable (available at `http://www.brics.dk/~milicia/Jeeg`).

Conclusion

Jeeg

- Synchronization constraints written in LTL and specified in a aspect-oriented, declarative manner.
- CL is helpful in treating the inheritance anomaly.
- Characterisation of CL in terms of regular languages
- Efficiently implementable (available at `http://www.brics.dk/~milicia/Jeeg`).

Future Work:

- Quantified linear temporal logic (QLTL) or monadic second order logic (MSOL), 'second order' variations of LTL of greater expressiveness.
- optimizing the LTL evaluation procedure by using ad-hoc static-analysis techniques.