ON STATE CLASSES AND THEIR DYNAMIC SEMANTICS

Ferruccio Damiani, Elena Giachino
Dipartimento di Informatica, Università degli Studi di Torino
Corso Svizzera 185, 10149 Torino, Italy

Paola Giannini, Emanuele Cazzola
Dipartimento di Informatica, Università del Piemonte Orientale
Via Bellini 25/G, 15100 Alessandria, Italy

Keywords: Java, concurrent object-oriented language, small-step semantics, core calculus, implementation by translation.

Abstract: We introduce *state classes*, a construct to program objects that can be safely concurrently accessed. State classes model the notion of object's *state* (intended as some abstraction over the value of fields) that plays a key role in concurrent object-oriented programming (as the *state* of an object changes, so does its coordination behavior). We show how state classes can be added to Java-like languages by presenting STATEJ, an extension of JAVA with state classes. The operational semantics of the state class construct is illustrated both at an abstract level, by means of a core calculus for STATEJ, and at a concrete level, by defining a translation from STATEJ into JAVA.

1 INTRODUCTION

The notion of object's state, intended as some abstraction on the values of fields, plays a key role in concurrent object-oriented programming. Various language constructs for expressing object's state abstractions have been proposed in the literature (see, e.g., (Philippsen, 2000) for a survey). We propose *state classes*, a programming feature that could be added to JAVA-like programming languages. The main novelties in our proposal are: (1) The ability of states to carry values, thanks to the fact that states may be parameterized by special fields, that we call *attributes*; and (2) The presence of a static type and effect system guaranteeing that, even though the state of the objects may vary through states with different attributes, no attempt will be made to access non-existing attributes (this is, for state attributes, the standard requirement that well typed programs cannot cause a *field not found error*).

This paper focuses on the dynamic semantics of state classes. Typing issues are addressed in (Damiani et al., 2006). The paper is organized as follows: Section 2 introduces STATEJ, an extension of JAVA with state classes, through an example. Section 3 gives the FSJ calculus (a core calculus for STATEJ). Section 4 outlines how STATEJ can be implemented by translation into plain JAVA. Sections 5 and 6 conclude by discussing related and further work, respectively.

2 AN EXAMPLE

In this section we motivate STATEJ through an example. The *state class construct* is designed to program objects that can be safely concurrently accessed. Therefore, in a state class, all the fields are private and all the methods are synchronized (that is, they are executed in mutual exclusion on the receiver object). A state class may extend an *ordinary* (i.e., non-state) class, but only state classes may extend state classes. Each state class specifies a collection of states. Each state is parameterized by some special fields, called *attributes*, and declares some methods. The state of an object `o` can be changed only inside methods of `o`, by means of a state *transition statement*, $\mathtt{this!!S}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$, where "`S`" is the name of the target state and "$\mathtt{e}_1, \ldots, \mathtt{e}_n$" $(n \geq 0)$ supply the values for all the attributes of `S`. An object belonging to a state class is always in one of the states specified in its class. Each state class constructor must set the state of the created object. The default constructor of the root of a hierarchy of state classes sets the state to the first state defined in the class.

The class `ReaderWriter` (in Fig. 1) implements a *multiple reader, single-writer lock* — see (Birrel, 1989), for an implementation using traditional concurrency primitives in a dialect of MODULA 2, and (Benton et al., 2004), for an implementation using chords in POLYPHONIC C♯.

```
public state class ReaderWriter {
 state FREE {
  public void shared() {this!!SHARED(1);}
  public void exclusive() {this!!EXCLUSIVE;}
 }
 state SHARED(int n) {
  public void shared() {n++;}
  public void releaseShared()
   {n--; if (n==0) this!!FREE;}
 }
 state EXCLUSIVE {
  public void releaseExclusive()
   {this!!FREE;}
 }
}
```

Figure 1: A multiple-reader, single-writer lock.

```
public state class ReaderWriterFair
             extends ReaderWriter {
 state SHARED(int n) {
  public void exclusive()
   {this!!PENDING_WRITER(n);
    pre_exclusive();
    this!!EXCLUSIVE;}
 }
 state PENDING_WRITER(int n) {
  public void releaseShared()
   {n--; if (n==0) this!!PRE_EXCLUSIVE;}
 }
 state PRE_EXCLUSIVE {
  private void pre_exclusive() { }
 }
}
```

Figure 2: A fair multiple-reader, single-writer lock.

When a thread e invokes a method `m` on an object `o` belonging to a state class (e.g., to the class `ReaderWriter` in Fig. 1), if either `o` is in a state that does not support the invoked method (e.g., `shared` invoked on an `EXCLUSIVE` `ReaderWriter`) or some other thread is executing a method on `o`, then the execution of e is blocked until `o` reaches (because of the action of some other thread) a state where the invoked method is available and no other thread is executing a method on `o`.

The policy implemented by the `ReaderWriter` class above is prone to writers' starvation. The class `ReaderWriterFair` (in Fig. 2) extends the class `ReaderWriter` to implement a writer starvation free policy.

An extending class inherits all the states of the extended class, and may add/override methods and introduce new states. Thus, class `ReaderWriteFair` has states `FREE`, `SHARED`, `EXCLUSIVE`, `PENDING_WRITER` and `PRE_EXCLUSIVE`. When the request `exclusive` is received by an object `o` in state `SHARED`(n), then the state of `o` is set to `PENDING_WRITER`(n) and the method body suspends; in this state `o` can only execute up to *n* requests of `releaseShared`; after the *n*-th such request, the state of `o` is set to `PRE_EXCLUSIVE`; in state `PRE_EXCLUSIVE` the method body for `exclusive` can continue, and will set the state of `o` to `EXCLUSIVE`.

The `ReaderWriterFair` class illustrates a common pattern in state class programming: the private method `pre_exclusive` has an empty body, and acts as a test that the receiver has reached the state `PRE_EXCLUSIVE`.

3 A CALCULUS FOR STATEJ

This section gives syntax and operational semantics of FSJ, a minimal imperative core calculus for STATEJ. FSJ models the innovative features of the state construct (namely state classes, state attributes and methods, and state transitions) and multi-threaded computations.

A FSJ program consists of a set of class definitions plus an expression to be evaluated, that we will call the *main expression* of the program.

3.1 Syntax

The abstract syntax of FSJ class declarations (`L`), class constructor declarations (`K`), state declarations (`N`), method declarations (`M`), and expressions (`e`) is given in Fig. 3. The metavariables `A`, `B`, `C`, and `D` range over class names; `S` ranges over state names; `f` and `g` range over attribute names; `m` ranges over method names; `x` ranges over method parameter names; and `a`, `b`, `c`, `d`, and `e` range over expressions.

We write "$\bar{\texttt{e}}$" as a shorthand for a possibly empty sequence "$\texttt{e}_1, \cdots, \texttt{e}_n$" (and similarly for `C`, `f`, `S`, `x`) and write "$\bar{\texttt{N}}$" as a shorthand for "$\texttt{N}_1 \cdots \texttt{N}_n$" with no commas (and similarly for $\bar{\texttt{M}}$). We write the empty sequence as "$\bullet$" and denote the concatenation of sequences using either comma or juxtaposition, as appropriate. We abbreviate operations on pair of sequences by writing "$\bar{\texttt{C}}\ \bar{\texttt{f}}$" for "$\texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n$", where n is the length of $\bar{\texttt{C}}$ and $\bar{\texttt{f}}$. We assume that sequences of state declarations or names, attribute declarations or names, method parameter declarations or names, method declarations do not contain duplicate names.

The class declaration

$$\texttt{state class C extends D}\ \{\texttt{K}\ \bar{\texttt{N}}\}$$

defines a state class of name `C` with superclass `D`. The new class has a single constructor `K` and a set of states $\bar{\texttt{N}}$. The state declarations $\bar{\texttt{N}}$ may either refine (by adding/overrinding methods) states that are already present in `D` or add new states.

Syntax:

$$\begin{array}{lcl}
\mathtt{L} & ::= & \mathtt{state\ class\ C\ extends\ C}\ \{\mathtt{K}\ \bar{\mathtt{N}}\} \\
\mathtt{K} & ::= & \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{f}})\{\mathtt{this!!S}(\bar{\mathtt{f}})\} \\
\mathtt{N} & ::= & \mathtt{state\ S}\ (\bar{\mathtt{C}}\ \bar{\mathtt{f}})\{\bar{\mathtt{M}}\} \\
\mathtt{M} & ::= & \mathtt{C\ m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\mathtt{e}\} \\
\mathtt{e} & ::= & \mathtt{x}\ \mid\ \mathtt{this}\ \mid\ \mathtt{this.f}\ \mid\ \mathtt{e;\ e}\ \mid\ \mathtt{new\ C}(\bar{\mathtt{e}}) \\
 & \mid & \mathtt{this!!S}(\bar{\mathtt{e}})\ \mid\ \mathtt{spawn}(\mathtt{e})\ \mid\ \mathtt{e.m}(\bar{\mathtt{e}})
\end{array}$$

Subtyping:

$$\mathtt{C} <: \mathtt{C} \qquad \frac{\mathtt{C}_1 <: \mathtt{C}_2 \qquad \mathtt{C}_2 <: \mathtt{C}_3}{\mathtt{C}_1 <: \mathtt{C}_3}$$

$$\frac{\mathtt{state\ class\ C}_1\ \mathtt{extends\ C}_2\ \cdots\ \{\cdots\}}{\mathtt{C}_1 <: \mathtt{C}_2}$$

State attributes lookup:

$$\frac{\mathtt{state\ class\ C}\ \cdots\ \{\cdots\mathtt{state\ S}(\bar{\mathtt{C}}\ \bar{\mathtt{f}})\{\cdots\}\cdots\}}{\mathit{attributes}(\mathtt{C},\mathtt{S}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}}}$$

$$\frac{\mathtt{state\ class\ C\ extends\ D}\ \{\mathtt{K}\ \bar{\mathtt{N}}\} \qquad \mathtt{S} \not\in \bar{\mathtt{N}}}{\mathit{attributes}(\mathtt{C},\mathtt{S}) = \mathit{attributes}(\mathtt{D},\mathtt{S})}$$

Method definition lookup:

$$\frac{\begin{array}{l}\mathtt{state\ class\ C}\ \cdots\ \{\mathtt{K}\ \bar{\mathtt{N}}\} \qquad \mathtt{state\ S}\{\bar{\mathtt{M}}\} \in \bar{\mathtt{N}} \\ \mathtt{A\ m}(\bar{\mathtt{A}}\ \bar{\mathtt{x}})\ \{\mathtt{e}\} \in \bar{\mathtt{M}}\end{array}}{\mathit{mDef}(\mathtt{m},\mathtt{C},\mathtt{S}) = \mathtt{A\ m}(\bar{\mathtt{A}}\ \bar{\mathtt{x}})\ \{\mathtt{e}\}}$$

$$\frac{\begin{array}{l}\mathtt{state\ class\ C\ extends\ D}\ \{\mathtt{K}\ \bar{\mathtt{N}}\} \\ (\mathtt{S} \not\in \bar{\mathtt{N}}\ \ \text{or}\ \ (\mathtt{state\ S}\{\bar{\mathtt{M}}\} \in \bar{\mathtt{N}}\ \ \text{and}\ \ \mathtt{m} \not\in \bar{\mathtt{M}}))\end{array}}{\mathit{mDef}(\mathtt{m},\mathtt{C},\mathtt{S}) = \mathit{mDef}(\mathtt{m},\mathtt{D},\mathtt{S})}$$

Figure 3: FSJ syntax, subtyping rules, and lookup functions.

The constructor declaration $\mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{f}})\ \{\mathtt{this!!S}(\bar{\mathtt{f}})\}$ specifies how to initialize the state and the state attributes of an instance of C. It takes exactly as many parameters as there are attributes of the state S and its body consists of a state transition statement.

The state declaration $\mathtt{state\ S}(\bar{\mathtt{C}}\ \bar{\mathtt{f}})\ \{\bar{\mathtt{M}}\}$ introduces a state with name S and attributes of names $\bar{\mathtt{f}}$ and types $\bar{\mathtt{C}}$. The declaration provides a suite of methods $\bar{\mathtt{M}}$ that are available in the state S of the class C containing the state declaration. A state S declared in a class C inherits all the (not overridden) methods that are defined in the (possible) declarations of S contained in the superclasses of C.

The method declaration $\mathtt{C\ m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\mathtt{e}\}$ introduces a method named m with result type C, parameters $\bar{\mathtt{x}}$ of types $\bar{\mathtt{C}}$, and body e. The variables $\bar{\mathtt{x}}$ and the pseudo-variable this are bound in e.

The class declarations in a program must satisfy the following conditions: (1) Object is a distinguished class name whose declaration does *not* appear in the program; (2) For every class name C (except Object) appearing anywhere in the program, one and only one class with name C is declared in the program; and (3) The subtype relation induced by the class declarations in the program (denoted by $<:$ and formally defined in the middle of Fig. 3) is acyclic. To simplify the notation in what follows (as in (Igarashi et al., 2001)), we always assume a *fixed* program.

The lookup functions are given at the bottom of Fig. 3. We write $\mathtt{S} \not\in \bar{\mathtt{N}}$ to mean that no declaration of the state S is included in $\bar{\mathtt{N}}$, and $\mathtt{m} \not\in \bar{\mathtt{M}}$ to mean that no declaration of the method m is included in $\bar{\mathtt{M}}$. Lookup of the attributes of a state S of a class C, written $\mathit{attributes}(\mathtt{C},\mathtt{S})$, returns a sequence $\bar{\mathtt{C}}\,\bar{\mathtt{f}}$ pairing the type of each attribute declared in the state with its name. Lookup of the definition of the method m in the state S of a state class C is denoted by $\mathit{mDef}(\mathtt{m},\mathtt{C},\mathtt{S})$.[1] Note that $\mathit{attributes}(\mathtt{C},\mathtt{S})$ and $\mathit{mDef}(\mathtt{m},\mathtt{C},\mathtt{S})$ are undefined when $\mathtt{C} = \mathtt{Object}$.[2]

3.2 Operational Semantics

In this section we introduce the operational semantics of FSJ, by defining the reduction rules that transform *configurations* representing multi-threaded computation. A configuration is a pair "$\bar{e}, \mathcal{H}$", where $\bar{e}$ is a sequence of $n \geq 1$ *runtime expressions* and $\mathcal{H}$ is a *heap* mapping *addresses* to *objects*. *Addresses*, ranged over by the metavariable ι, are the elements of the denumerable set $\mathbf{I}$. *Objects* are finite mappings associating: (1) the distinguished name "class" to a class name indicating the class of the object; (2) the distinguished name "state" to a state name indicating the state of the object; and (3) a mapping associating a finite number (possibly zero) of state attribute names to addresses. Objects will be denoted by $[\![\mathsf{class} : \mathtt{C},\ \mathsf{state} : \mathtt{S},\ \bar{\mathtt{f}} : \bar{\iota}]\!]$.

The first component of a configuration, $\bar{e}$, will be called "sequence of threads". A thread of computation is represented by the evaluation of a runtime expression e_i in the heap $\mathcal{H}$. The different threads share the same heap $\mathcal{H}$. Threads do not have, as in full STATEJ and JAVA, an associated stack, keeping the association between parameters and values. In fact, since FSJ does not include assignment, method calls are evaluated by directly substituting the formal parameters and the metavariable this with the corresponding values (in FSJ the only values are addresses). We call the result of this substitution, which is no longer an expression of the source language, a *simple runtime expression*. Simple runtime expressions, ranged over by s, are obtained from the pseudo grammar defining expressions (in Fig. 3) by replacing the clauses "$\mathtt{x}\ \mid\ \mathtt{this}\ \mid\ \mathtt{this.f}\ \mid$" with the clauses "$\iota\ \mid\ \iota.\mathtt{f}\ \mid$" (see the top of Fig. 4).

Runtime expressions, ranged over by e, are defined by the grammar at top of Fig. 4. In FSJ every method is synchronized, therefore on method call the lock of the object receiving the call must be ac-

[1] In full STATEJ, like in JAVA, the lookup functions take into account method overloading, that (for simplicity) is not included in FSJ.

[2] In full STATEJ the class Object has several methods.

Simple runtime expressions, runtime expressions, evaluation contexts, redexes, and auxiliary functions:

$$
\begin{array}{rcl}
s & ::= & \iota \mid \iota.\texttt{f} \mid s;s \mid \texttt{new C}(\bar{s}) \mid \iota\texttt{!!S}(\bar{s}) \mid \texttt{spawn}(s) \mid s.\texttt{m}(\bar{s}) \\
e & ::= & \iota \mid \iota.\texttt{f} \mid e;s \mid \texttt{new C}(\bar{\iota},\dot{e},\bar{s}) \mid \iota\texttt{!!S}(\bar{\iota},\dot{e},\bar{s}) \mid \texttt{spawn}(e) \mid e.\texttt{m}(\bar{s}) \mid \iota.\texttt{m}(\bar{\iota},\dot{e},\bar{s}) \\
 & \mid & \texttt{ret}(\iota,\texttt{m},e) \mid \texttt{unlock}(\iota.\texttt{m}(\bar{\iota})) \\
\mathcal{E} & ::= & [\,] \mid \mathcal{E};s \mid \texttt{new C}(\bar{\iota},\mathcal{E},\bar{s}) \mid \iota\texttt{!!S}(\bar{\iota},\mathcal{E},\bar{s}) \mid \texttt{spawn}(\mathcal{E}) \mid \mathcal{E}.\texttt{m}(\bar{s}) \mid \iota.\texttt{m}(\bar{\iota},\mathcal{E},\bar{s}) \mid \texttt{ret}(\iota,\texttt{m},\mathcal{E}) \\
r & ::= & \iota.\texttt{f} \mid \iota;s \mid \texttt{new C}(\bar{\iota}) \mid \iota\texttt{!!S}(\bar{\iota}) \mid \texttt{spawn}(\iota) \mid \iota.\texttt{m}(\bar{\iota}) \mid \texttt{ret}(\iota,\texttt{m},\iota) \mid \texttt{unlock}(\iota.\texttt{m}(\bar{\iota}))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{lockedBy}(e) & = & \{\iota \mid \texttt{ret}(\iota,\cdots,\cdots) \text{ is a subexpression of } e \text{ and } \texttt{unlock}(\iota.\cdots(\cdots)) \text{ is not a subexpression of } e\} \\
\mathit{lockedBy}(e_1\cdots e_n) & = & \bigcup_{1\le i\le n} \mathit{lockedBy}(e_i)
\end{array}
$$

Reduction rules:

$$\frac{\mathcal{H}(\iota)=\texttt{o} \qquad \texttt{o}(\texttt{f})=\iota'}{\bar{a}\,\mathcal{E}[\iota.\texttt{f}]\,\bar{c},\ \mathcal{H} \longrightarrow \bar{a}\,\mathcal{E}[\iota']\,\bar{c},\ \mathcal{H}} \qquad (\text{R-ATTR})$$

$$\bar{a}\,\mathcal{E}[\iota;\ s]\,\bar{c},\ \mathcal{H} \longrightarrow \bar{a}\,\mathcal{E}[s]\,\bar{c},\ \mathcal{H} \qquad (\text{R-SEQ})$$

$$\frac{\texttt{state class C} \ \cdots\ \{\texttt{C}(\bar{\texttt{C}}\ \bar{\texttt{f}})\{\texttt{this!!S}(\bar{\texttt{f}})\}\ \cdots\} \qquad \texttt{o}=\llbracket\mathsf{class}:\texttt{C},\ \mathsf{state}:\texttt{S},\ \bar{\texttt{f}}:\bar{\iota}\rrbracket \qquad \iota\notin\mathrm{Dom}(\mathcal{H})}{\bar{a}\,\mathcal{E}[\texttt{new C}(\bar{\iota})]\,\bar{c},\ \mathcal{H} \longrightarrow \bar{a}\,\mathcal{E}[\iota]\,\bar{c},\ \mathcal{H}[\iota:\texttt{o}]} \qquad (\text{R-NEW})$$

$$\frac{\mathcal{H}(\iota)(\texttt{class})=\texttt{C} \qquad \mathit{attributes}(\texttt{C},\texttt{S})=\bar{\texttt{C}}\,\bar{\texttt{f}} \qquad \texttt{o}=\llbracket\mathsf{class}:\texttt{C},\ \mathsf{state}:\texttt{S},\ \bar{\texttt{f}}:\bar{\iota}\rrbracket}{\bar{a}\,\mathcal{E}[\iota\texttt{!!S}(\bar{\iota})]\,\bar{c},\ \mathcal{H} \longrightarrow \bar{a}\,\mathcal{E}[\iota]\,\bar{c},\ \mathcal{H}[\iota:\texttt{o}]} \qquad (\text{R-TRANS})$$

$$\bar{a}\,\mathcal{E}[\texttt{spawn}(\iota)]\,\bar{c},\ \mathcal{H} \longrightarrow \bar{a}\,\mathcal{E}[\iota]\,\bar{c}\ \iota.\texttt{run}(),\ \mathcal{H} \qquad (\text{R-SPAWN})$$

$$\frac{\iota\notin\mathit{lockedBy}(\bar{a}\bar{c}) \qquad \mathcal{H}(\iota)=\llbracket\mathsf{class}:\texttt{D},\ \mathsf{state}:\texttt{S},\ \cdots\rrbracket \qquad \mathit{mDef}(\texttt{m},\texttt{D},\texttt{S})=\texttt{C m }(\bar{\texttt{C}}\ \bar{\texttt{x}})\ \{\texttt{e}\}}{\bar{a}\,\mathcal{E}[\iota.\texttt{m}(\bar{\iota})]\,\bar{c},\ \mathcal{H} \longrightarrow \bar{a}\,\mathcal{E}[\texttt{ret}(\iota,\texttt{m},\texttt{e}[\texttt{this}:=\iota,\bar{\texttt{x}}:=\bar{\iota}])]\,\bar{c},\ \mathcal{H}} \qquad (\text{R-INVK-1})$$

$$\frac{\iota\in\mathit{lockedBy}(\mathcal{E}[\iota.\texttt{m}(\bar{\iota})]) \qquad \mathcal{H}(\iota)=\llbracket\mathsf{class}:\texttt{D},\ \mathsf{state}:\texttt{S},\ \cdots\rrbracket \qquad \mathit{mDef}(\texttt{m},\texttt{D},\texttt{S}) \text{ undefined}}{\bar{a}\,\mathcal{E}[\iota.\texttt{m}(\bar{\iota})]\,\bar{c},\ \mathcal{H} \longrightarrow \bar{a}\,\mathcal{E}[\texttt{unlock}(\iota.\texttt{m}(\bar{\iota}))]\,\bar{c},\ \mathcal{H}} \qquad (\text{R-INVK-2})$$

$$\frac{\iota\notin\mathit{lockedBy}(\bar{a}\bar{c}) \qquad \mathcal{H}(\iota)=\llbracket\mathsf{class}:\texttt{D},\ \mathsf{state}:\texttt{S},\ \cdots\rrbracket \qquad \mathit{mDef}(\texttt{m},\texttt{D},\texttt{S})=\texttt{C m }(\bar{\texttt{C}}\ \bar{\texttt{x}})\ \{\texttt{e}\}}{\bar{a}\,\mathcal{E}[\texttt{unlock}(\iota.\texttt{m}(\bar{\iota}))]\,\bar{c},\ \mathcal{H} \longrightarrow \bar{a}\,\mathcal{E}[\texttt{ret}(\iota,\texttt{m},\texttt{e}[\texttt{this}:=\iota,\bar{\texttt{x}}:=\bar{\iota}])]\,\bar{c},\ \mathcal{H}} \qquad (\text{R-UNLOCK})$$

$$\bar{a}\,\mathcal{E}[\texttt{ret}(\iota,\texttt{m},\iota_0)]\,\bar{c},\ \mathcal{H} \longrightarrow \bar{a}\,\mathcal{E}[\iota_0]\,\bar{c},\ \mathcal{H} \qquad (\text{R-RET})$$

Figure 4: FSJ (simple) runtime expressions, evaluation contexts, redexes, auxiliary functions, and reduction rules.

quired, unless the call is inside a method of the object itself, in which case the call can proceed (the lock is *reentrant*). Moreover, when the method call is on a method not defined in the current state, the lock of the object must be released. This gives to other threads a chance to change the state of the object to a state in which the method is defined. Both these situations are modelled by particular *runtime expressions*: (1) $\texttt{ret}(\iota,\texttt{m},e)$, where e does not contain occurrences of $\texttt{unlock}(\iota.\cdots(\cdots))$, specifies that a thread is currently holding the lock of the receiver ι, in order to evaluate the expression e, which represents the body of the method `m`, and (2) $\texttt{unlock}(\iota.\texttt{m}(\bar{\iota}))$ specifies that the lock of ι has been released in order to give a chance to another thread to change the state of ι to a state in which `m` is defined. Note that, the definition of the syntax for runtime expressions implies that there can be nested `ret` expressions but only one `unlock`. The metavariables a, b, c, d, and e range over runtime expressions. We write $\bar{a}$ as a shorthand for a possibly empty sequence $a_1 \cdots a_n$ and $\dot{a}$ as a shorthand for a possibly empty sequence of length almost one. The function $\mathit{lockedBy}(\bar{e})$, defined in Fig. 4, returns the set of addresses that are locked by the thread sequence $\bar{e}$.

The reduction relation has the form "$\bar{a}\,b_1\,\bar{c},\mathcal{H}_1 \longrightarrow \bar{a}\,b_2\,\bar{c}\,\dot{d},\mathcal{H}_2$", read "configuration $\bar{a}\,b_1\,\bar{c},\mathcal{H}_1$ reduces to configuration $\bar{a}\,b_2\,\bar{c}\,\dot{d},\mathcal{H}_2$ in one step". The (empty or singleton) sequence $\dot{d}$ indicates that a new thread might have been spawned because of the reduction of a `spawn` expression. We write $\longrightarrow^\star$ for the reflexive and transitive closure of $\longrightarrow$.

By using the definition of *evaluation context* and *redex* (see $\mathcal{E}$ and r in Fig. 4), the reduction rules ensure that inside each thread the computation follows a call-by-value left-to-right reduction strategy. This implies that expressions such as `ret` and `unlock` can only be preceded by values and followed by simple runtime expressions, which do not contain `ret` and `unlock` (see the definition of s and e in Fig. 4).

The following property asserts that a context can be decomposed in a unique way in sub-contexts showing the activation stack of method calls.

Property 1 (Unique Decomposition) *Every evalua-*

tion context $\mathcal{E}$ can be written as

$$\underbrace{\mathcal{E}_{1,1}[\mathtt{ret}(\iota_1, \mathtt{m}_{1,1}, \cdots \mathcal{E}_{1,q_1}[\mathtt{ret}(\iota_1, \mathtt{m}_{1,q_1}}_{q_1} \cdots$$

$$\underbrace{\mathcal{E}_{p,1}[\mathtt{ret}(\iota_p, \mathtt{m}_{p,1}, \cdots \mathcal{E}_{p,q_p}[\mathtt{ret}(\iota_p, \mathtt{m}_{p,q_p}, \mathcal{E}_0)] \cdots)]}_{q_p}$$

$$\cdots)]\cdots)],$$

where $\mathcal{E}_{1,1}$, ..., $\mathcal{E}_{1,q_1}$, ..., $\mathcal{E}_{p,1}$, ..., $\mathcal{E}_{p,q_p}$ ($p \geq 0$, $q_1 \geq 1$, ..., $q_p \geq 1$) and $\mathcal{E}_0$ do not contain $\mathtt{ret}(\cdots)$ *subexpressions.*

The reduction rules are given at the bottom of Fig. 4. Each reduction rule rewrites a configuration of the form "$\bar{a}\,\mathcal{E}[r]\,\bar{c}, \mathcal{H}_1$", where $\mathcal{E}$ is an evaluation context and r is a redex, into a configuration of the form "$\bar{a}\,\mathcal{E}[e]\,\bar{c}\,\dot{d}, \mathcal{H}_2$". The metavariable o ranges over objects. We use $\mathcal{H}[\iota : \mathsf{o}]$ to denote the heap such that $\mathcal{H}[\iota : \mathsf{o}](\iota) = \mathsf{o}$ and $\mathcal{H}[\iota : \mathsf{o}](\iota') = \mathcal{H}(\iota')$, for $\iota' \neq \iota$.

The reduction rules for attribute selection, (R-ATTR), and sequential composition, (R-SEQ), are standard. The rule for object creation, (R-NEW), stores the newly created object at a fresh address of the heap and returns the address. The pseudo fields class and state, and the parameters of the initial state are initialized as specified by the class constructor. The rule for state transition, (R-TRANS), changes the current state of the object and returns its address. Rule (R-SPAWN) replaces the spawn expression with the address ι and adds a new thread evaluating the call of the method run on the object at ι. Rule (R-INVK-1) is applied if the method m is defined in the current state of the receiver, ι, and no other thread holds the lock of ι. The expression produced replaces the call with $\mathtt{ret}(\iota, \mathtt{m}, e')$, indicating that the current thread holds the lock of ι. The expression e' is the body of the method m in which this and the formal parameters are replaced with the address ι and the actual parameters. Rule (R-INVK-2) is applied if the method m is not defined in the current state of the receiver and the current thread holds the lock of ι. In this case, the lock of ι must be released and the thread must wait that some other thread changes the state of ι to a state in which m is defined. This is achieved by replacing the method call redex with the expression $\mathtt{unlock}(\iota.\mathtt{m}(\bar{\iota}))$. Note that, since the current thread had the lock of ι, the newly introduced unlock expression is a subexpression of an expression $\mathtt{ret}(\iota, \mathtt{m}', e')$ for some m′ and e'. Rule (R-UNLOCK) replaces the expression $\mathtt{unlock}(\iota.\mathtt{m}(\bar{\iota}))$, if ι is not locked and the method m is defined in its state, with $\mathtt{ret}(\iota, \mathtt{m}, e')$, where e' is the body of the method m in which this and the formal parameters are replaced with the address ι and the actual parameters. Rule(R-RET), that applies when the body of the method m on object ι has been evaluated completely, producing a value, releases the lock of ι by removing the $\mathtt{ret}(\iota, \mathtt{m}, \iota_0)$ subexpression.

Example 2 (Application of the reduction rules) First we define the following classes CR and CW representing the class of threads that have a shared access to a ReaderWriter object rw and the class of threads that have an exclusive access to it, respectively.

```
state class CR extends Object {
  CR(ReaderWriter rw) { this!!S(rw) }
  state S (ReaderWriter rw) {
    Object run () {
      rw.shared();
      ...
      rw.releaseShared();
      this.run() } }
}

state class CW extends Object {
  CW(ReaderWriter rw) { this!!S(rw) }
  state S (ReaderWriter rw) {
    Object run () {
      rw.exclusive();
      ...
      rw.releaseExclusive();
      this.run() } }
}
```

We consider as the *main expression* of the program, that is the expression to be evaluated,

$$\mathtt{spawn}(\mathtt{new}\,\mathsf{CR}(\iota)); (\mathtt{new}\,\mathsf{CW}(\iota)).\mathtt{run}(),$$

where ι is a ReaderWriter object, so the computation starts from the following *configuration*:

$$\mathtt{spawn}(\mathtt{new}\,\mathsf{CR}(\iota)); (\mathtt{new}\,\mathsf{CW}(\iota)).\mathtt{run}(),\ \mathcal{H}$$

where $\mathcal{H} = \iota : [\![\mathsf{class} : \mathtt{ReaderWriterFair},\ \mathsf{state} : \mathtt{FREE}]\!]$.

A possible computation is as in Fig. 5, where SH stands for SHARED, PW stands for PENDING_WRITER, EX stands for EXCLUSIVE, and PE stands for PRE_EXCLUSIVE. We adopt the following notations: (1) Threads e_1, e_2 being part of the *configuration* are written $\begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$; (2) Redexes are underlined; (3) Redexes of suspended threads are underlined and written in grey; (4) the arrow $\Longrightarrow$ indicates one step of reduction for each thread of the sequence; (5) In ret expressions we omit method names. As we see in Fig. 5, in the example we assumed to have integers, decrement and if-statement. These are assumed, in line (#), to be reduced following the standard semantics.

4 FROM STATEJ TO JAVA

This section briefly illustrates a translation from STATEJ to plain JAVA. The basic idea of the translation is to map a state class into a JAVA class using synchronized methods and the primitives wait() and notify(). A class contains a field indicating the

$$\texttt{spawn}(\underline{\texttt{new CR}(\iota)});(\texttt{new CW}(\iota)).\texttt{run}(),\ \mathcal{H} \longrightarrow \underline{\texttt{spawn}(\iota')};(\texttt{new CW}(\iota)).\texttt{run}(),\ \mathcal{H}_1 \longrightarrow \quad \text{first by (R-}\textsc{New}\text{) and second by (R-}\textsc{Spawn}\text{)}$$

$$\begin{pmatrix}\underline{\iota';(\texttt{new CW}(\iota)).\texttt{run}()}\\ \underline{\iota'.\texttt{run}()}\end{pmatrix},\ \mathcal{H}_1 \Longrightarrow \quad \text{by (R-}\textsc{Seq}\text{) and (R-}\textsc{Invk}\text{-1)}$$

$$\begin{pmatrix}\underline{(\texttt{new CW}(\iota))}.\texttt{run}()\\ \texttt{ret}(\iota',\ \underline{\iota.\texttt{shared}()};\ \ldots;\iota.\texttt{releaseShared}();\iota'.\texttt{run}())\end{pmatrix},\ \mathcal{H}_1 \Longrightarrow \quad \text{by (R-}\textsc{New}\text{) and (R-}\textsc{Invk}\text{-1)}$$

$$\begin{pmatrix}\underline{\iota''.\texttt{run}()}\\ \texttt{ret}(\iota',\ \texttt{ret}(\iota,\ \underline{\iota!!\texttt{SH}(1)});\ \ldots;\iota.\texttt{releaseShared}();\iota'.\texttt{run}())\end{pmatrix},\ \mathcal{H}_2 \Longrightarrow \quad \text{by (R-}\textsc{Invk}\text{-1) and (R-}\textsc{Trans}\text{)}$$

$$\begin{pmatrix}\texttt{ret}(\iota'',\ \underline{\iota.\texttt{exclusive}();}\ldots;\iota.\texttt{releaseExclusive}();\iota''.\texttt{run}())\\ \texttt{ret}(\iota',\ \underline{\texttt{ret}(\iota,\ \iota)};\ \ldots;\iota.\texttt{releaseShared}();\iota'.\texttt{run}())\end{pmatrix},\ \mathcal{H}_3 \longrightarrow \quad \text{by (R-}\textsc{Ret}\text{)}$$

$$\begin{pmatrix}\texttt{ret}(\iota'',\ \underline{\iota.\texttt{exclusive}()};\ \ldots;\iota.\texttt{releaseExclusive}();\iota''.\texttt{run}())\\ \texttt{ret}(\iota',\ \underline{\iota;\ \ldots};\iota.\texttt{releaseShared}();\iota'.\texttt{run}())\end{pmatrix},\ \mathcal{H}_3 \longrightarrow^{\star} \quad \text{by first applying (R-}\textsc{Invk}\text{-1) and (R-}\textsc{Seq}\text{)}$$

$$\begin{pmatrix}\texttt{ret}(\iota'',\ \texttt{ret}(\iota,\ \underline{\iota!!\texttt{PW}(1)};\iota.\texttt{pre\_exclusive}();\iota!!\texttt{EX});\ \ldots;\iota.\texttt{releaseExclusive}();\iota''.\texttt{run}())\\ \texttt{ret}(\iota',\ \underline{\iota.\texttt{releaseShared}()};\iota'.\texttt{run}())\end{pmatrix},\ \mathcal{H}_3 \longrightarrow \quad \text{by (R-}\textsc{Trans}\text{)}$$

$$\begin{pmatrix}\texttt{ret}(\iota'',\ \texttt{ret}(\iota,\ \underline{\iota;\iota.\texttt{pre\_exclusive}()};\iota!!\texttt{EX});\ \ldots;\iota.\texttt{releaseExclusive}();\iota''.\texttt{run}())\\ \texttt{ret}(\iota',\ \underline{\iota.\texttt{releaseShared}()};\iota'.\texttt{run}())\end{pmatrix},\ \mathcal{H}_4 \longrightarrow \quad \text{by (R-}\textsc{Seq}\text{)}$$

$$\begin{pmatrix}\texttt{ret}(\iota'',\ \texttt{ret}(\iota,\ \underline{\iota.\texttt{pre\_exclusive}()};\iota!!\texttt{EX});\ \ldots;\iota.\texttt{releaseExclusive}();\iota''.\texttt{run}())\\ \texttt{ret}(\iota',\ \underline{\iota.\texttt{releaseShared}()};\iota'.\texttt{run}())\end{pmatrix},\ \mathcal{H}_4 \longrightarrow \quad \text{by (R-}\textsc{Invk}\text{-2)}$$

$$\begin{pmatrix}\texttt{ret}(\iota'',\ \texttt{ret}(\iota,\ \underline{\texttt{unlock}(\iota.\texttt{pre\_exclusive}())};\iota!!\texttt{EX});\ \ldots;\iota.\texttt{releaseExclusive}();\iota''.\texttt{run}())\\ \texttt{ret}(\iota',\ \underline{\iota.\texttt{releaseShared}()};\iota'.\texttt{run}())\end{pmatrix},\ \mathcal{H}_4 \longrightarrow \quad \text{by (R-}\textsc{Invk}\text{-1)}$$

$$(\#) \quad \begin{pmatrix}\texttt{ret}(\iota'',\ \texttt{ret}(\iota,\ \underline{\texttt{unlock}(\iota.\texttt{pre\_exclusive}())};\iota!!\texttt{EX});\ \ldots;\iota.\texttt{releaseExclusive}();\iota''.\texttt{run}())\\ \texttt{ret}(\iota',\ \texttt{ret}(\iota,\ \texttt{n}--;\texttt{if}\ (\texttt{n}=0)\ \iota!!\texttt{PE});\iota'.\texttt{run}())\end{pmatrix},\ \mathcal{H}_4 \longrightarrow^{\star}$$

$$\begin{pmatrix}\texttt{ret}(\iota'',\ \texttt{ret}(\iota,\ \underline{\texttt{unlock}(\iota.\texttt{pre\_exclusive}())};\iota!!\texttt{EX});\ \ldots;\iota.\texttt{releaseExclusive}();\iota''.\texttt{run}())\\ \texttt{ret}(\iota',\ \underline{\iota;\iota'.\texttt{run}()})\end{pmatrix},\ \mathcal{H}_5 \Longrightarrow \quad \text{by (R-}\textsc{Unlock}\text{) and (R-}\textsc{Seq}\text{)}$$

$$\begin{pmatrix}\texttt{ret}(\iota'',\ \texttt{ret}(\iota,\ \underline{\texttt{ret}(\iota,\ \_)};\iota!!\texttt{EX});\ \ldots;\iota.\texttt{releaseExclusive}();\iota''.\texttt{run}())\\ \texttt{ret}(\iota',\ \underline{\iota'.\texttt{run}()})\end{pmatrix},\ \mathcal{H}_5 \longrightarrow^{\star} \quad \text{by (R-}\textsc{Ret}\text{) and (R-}\textsc{Invk}\text{-1)}$$

$$\begin{pmatrix}\texttt{ret}(\iota'',\ \texttt{ret}(\iota,\ \underline{\iota.\texttt{releaseExclusive}()};\iota''.\texttt{run}()))\\ \texttt{ret}(\iota',\ \texttt{ret}(\iota',\ \ldots))\end{pmatrix},\ \mathcal{H}_6 \longrightarrow^{\star} \cdots$$

where

$\mathcal{H} = \iota : [\![\text{class} : \texttt{ReaderWriterFair},\ \text{state} : \texttt{FREE}]\!]$

$\mathcal{H}_1 = \mathcal{H}[\iota' : [\![\text{class} : \texttt{CR},\ \text{state} : \texttt{S},\ \texttt{rw} : \iota]\!]]$ $\qquad$ $\mathcal{H}_2 = \mathcal{H}_1[\iota'' : [\![\text{class} : \texttt{CW},\ \text{state} : \texttt{S},\ \texttt{rw} : \iota]\!]]$

$\mathcal{H}_3 = \mathcal{H}_2[\iota : [\![\text{class} : \texttt{ReaderWriterFair},\ \text{state} : \texttt{SH},\ \texttt{n} : 1]\!]]$ $\qquad$ $\mathcal{H}_4 = \mathcal{H}_3[\iota : [\![\text{class} : \texttt{ReaderWriterFair},\ \text{state} : \texttt{PW},\ \texttt{n} : 1]\!]]$

$\mathcal{H}_5 = \mathcal{H}_4[\iota : [\![\text{class} : \texttt{ReaderWriterFair},\ \text{state} : \texttt{PE}]\!]]$ $\qquad$ $\mathcal{H}_6 = \mathcal{H}_5[\iota : [\![\text{class} : \texttt{ReaderWriterFair},\ \text{state} : \texttt{EX}]\!]]$

Figure 5: An example of reduction.

current state of the object, and methods corresponding to the methods of the original STATEJ class. The translation can be briefly described as follows.

Method. Methods defined in more than one state have more than one body. To be able to execute different bodies in different states our translation creates a unique `synchronized` method containing all the different bodies. At run-time, when the method is called, we have to check the current state of the object, and see whether the method was defined in this state or not. In case it is defined, then the corresponding body is executed, otherwise the thread calls a `wait()` putting it in hold. To keep the information on the methods defined in a certain state we use a hash table. Due to the limitation of the switch statement of JAVA, states are codified by the primitive type `int`. For example the following class

```
state class Ex extends Object {
  Ex() { this!!A(); }
  state A () {
    Object m() { /* body of m in A */ } }
  state B () {
    Object m() { /* body of m in B */ } } }
```

is translated into

```
class Ex extends Object {
  Ex() { ... }
  final static int A = 1;
  final static int B = 2;
  Hashtable stateMethods;
  int currentState;
  synchronized Object m() {
    while (!existsInCurrentState) wait();
    switch (currentState) {
      case A : /* body of m in A */ break;
      case B : /* body of m in B */ break;
} } }
```

where the existence of a method in a given state and its selection are done using the hash table of methods.

State Transition. The state transition expression `this!!A()` is translated into

```
   currentState = A;   notifyAll();
```

so in addition to change the state of the objects it notifies all the threads waiting for the lock of the current object. When the current thread will release the lock the notified threads will compete to get it to have a chance to see whether the method that caused the waiting is defined in the current state. If the method is defined, then the thread can proceed, otherwise it calls a `wait()`. Due to the non deterministic nature of JAVA scheduling we cannot insure the order in which notified threads will be waken up.

Constructor. The constructor of the translated class should initialize the hash table and then include the translation of the constructor of the original class.

Inheritance. A state class may extend another class (either state or not). In the subclass we inherit all the states and may add others. Therefore, we have to be careful to clashes of constants of state. Moreover, methods may be added/redefined. For instance method `exclusive()` of the example in Sect. 2, is defined in state `FREE` of `ReaderWriter`, and redefined in state `SHARED` of `ReaderWriterFair`. When a method is redefined in its translation we use the `default` clause as follows.

```
class ReaderWriterFair
      extends ReaderWriter {
  ...
  synchronized void exclusive () {
    while (!existsInCurrentState) wait();
    switch (currentState) {
    case SHARED:
      currentState =PENDING_WRITER;
      notifyAll();
      pre_exclusive();
      currentState =EXCLUSIVE;
      notifyAll();
      break;
    default :
      super.exclusive;
      break; } } }
```

The current implementation of the translator (www.di.unito.it/~giannini/stateJimpl/) takes as input a program written in JAVA 1.4 extended with state classes *with attribute-free states* (attributes can be straightforwardly codified by class fields; however, their implementation would require to implement the type and effect analysis). The translation uses the tool for Language Recognition ANTLR, see (Parr and project group, 2005), and the StringTemplate tecnology, see (Parr, 2005). We first made a JAVA 1.4 to JAVA 1.4 translation taking advantage of the grammar defined by Parr and then modified the grammar to include our state related constructs. The use of ANTLR and StringTemplate makes the translator easily adaptable to different translation schemes and also to addition to the input language.

5 RELATED WORK

According to (Philippsen, 2000) `states` provide a *boundary coordination* mechanism (we refer to Sect. 4.2 of (Philippsen, 2000) for a survey of several COOLs with boundary coordination). In particular, the state class construct is related to the *actor model* (Agha, 1986) and to the *behaviour abstraction* and *behaviour/enable sets* proposals (Kafura and Lavender, 1996; Tomlinson and Singh, 1989).

At the best of our knowledge, the main novelties in our proposal are: the ability of states to carry values (thanks to the presence of attributes); the formalization of an abstract operational semantics of a notion of state for expressing coordination in JAVA-like languages; and the presence of a static type and effect system (presented in (Damiani et al., 2006)) guaranteeing that during the execution there cannot be any access to undefined attributes of objects. Type systems for concurrent objects have been investigated in the literature, see, e.g., “regular object types” (Nierstrasz, 1993), the TYCO object calculus (Ravara and Vasconcelos, 2000), and the F*ickle*$_{\mathrm{MT}}$ proposal (Damiani et al., 2004).

Various improvements of the concurrency model of JAVA-like languages have been proposed. In JOIN JAVA (Itzstein and Kearney, 2001) and POLYPHONIC C♯ (Benton et al., 2004) the synchronization mechanism relies on the *join pattern*, called *chord* in POLYPHONIC C♯, construct. Chords can be used to codify the state of an object through the pattern (illustrated, for instance, in (Benton et al., 2004)) of using private asynchronous method to carry object state. However, this pattern could be misused leading to deadlock or errors. In STATEJ the notion of object state is in the language definition, thus eliminating the possibility of many of such errors. In JEEG (Milicia and Sassone, 2005) the synchronization conditions on an object `o` are expressed with *linear temporal logic constraints* involving the value of fields and the method invocation history of `o`. These constraints could be used to codify the state of an object `o`. However, state attributes have to be mapped on object fields and there is no way to express the fact that some fields should be accessible only in some states.

STATEJ (as JOIN JAVA, POLYPHONIC C♯, and JEEG) focuses on a specific coordination mechanishm. The JR programming language (Keen et al., 2004) takes a different approach: it extends JAVA providing a rich concurrency model with a variety of mechanisms. None of this mechanisms directly models the notion of object state.

6 FUTURE WORK

The current prototypical implementation of STATEJ (www.di.unito.it/~giannini/stateJimpl/) is based on the translation scheme outlined in Sect. 4. It consists of a preprocessor that maps code written in JAVA 1.4 extended with state classes into plain JAVA. The current approach favors simplicity over efficiency. Its major drawback is that each state transition of an object `o` notifies *all* the threads waiting for *any* state of `o`. Note that, notifying just the threads waiting for the target state of the transition would not represent a significative improvement, since multiple state transitions may occur before the lock on `o` is released. A more significative improvement would be moving notification from state transition on `o` to lock release on `o`: this would allow notifying just the threads waiting for the current state of `o`. Note that, however, all but the first (according to the scheduling mechanism of JAVA) of such threads have to sleep again. We are currently investigating a quite different approach that support selective wakeups. It can be roughly described as follows:

- Each object `o` is equipped with a set of FIFO queues (one for each state).
- Whenever a thread invokes a method `m` on `o`, IF `o` is locked by some other thread OR `m` is not available in the current state of `o`
  - THEN the thread is suspended and enqueued in all the queues associated to the `states` of `o` where `m` is available, and the lock on `o` (if held by the suspended thread) is released
  - ELSE the method executed and the lock on `o` (if not already held by the invoking thread) is taken.
- Whenever the lock on `o` is released, IF the queue associated to the current state of `o` is not empty, THEN a thread `e` is extracted from the queue, removed from all the other queues, resumed, and it takes the lock on `o`.

Other future work includes: Refinement of the type and effect system given in (Damiani et al., 2006); Further investigations on the expressivity of the state class construct and on its integration in JAVA-like languages (by analyzing the interaction of state classes and their types with the advanced features of JAVA-like languages); Development of a new prototype (based on the translation scheme outlined above) including state attributes and the related type and effect analysis; and Development of benchmarks.

REFERENCES

Agha, G. A. (1986). *ACTORS: A Model of Concurrency Computation in Distribuited Systems*. MIT Press.

Benton, N., Cardelli, L., and Fournet, C. (2004). Modern Concurrency Abstractions for C♯. *ACM TOPLAS*, 26(5):769–804.

Birrel, A. D. (1989). An introduction to programming with threads. Technical Report 35, DEC SRC.

Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. (2004). On re-classification and multithreading. *JOT (*www.jot.fm*)*, 3(11):5–30. Special issue: OOPS track at SAC 2004.

Damiani, F., Giachino, E., Giannini, P., Cameron, N., and Drossopoulou, S. (2006). A state abstraction for coordination in java-like languages. In *Electronic proceedings of FTfJP'06 (*www.cs.ru.nl/ftfjp/*)*.

Igarashi, A., Pierce, B., and Wadler, P. (2001). Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450.

Itzstein, G. S. and Kearney, D. (2001). Join Java: an alternative concurrency semantics for Java. Technical Report ACRC-01-001, Univ. of South Australia.

Kafura, D. G. and Lavender, R. G. (1996). Concurrent object-oriented languages and the inheritance anomaly. In Casavant, T., Tvrdil, P., and Plásil, F., editors, *Parallel Computers: Theory and Practice*, pages 221–264. IEEE Press.

Keen, A. W., Ge, T., Maris, J. T., and Olsson, R. A. (2004). JR: Flexible distributed programming in an extended java. *TOPLAS*, 26(3):578–608.

Milicia, G. and Sassone, V. (2005). Jeeg: Temporal Constraints for the Synchronization of Concurrent Objects. *Concurrency Computat.: Pract. Exper.*, 17(5-6):539–572.

Nierstrasz, O. (1993). Regular Types for Active Objects. In *OOPSLA'93*, volume 28 of *ACM SIGPLAN Notices*, pages 1–15.

Parr, T. (2003-2005). StringTemplate Documentation. Available at www.stringtemplate.org./doc/doc.html.

Parr, T. and project group (2005). ANTLR Reference Manual, Version 2.7.5. Available at www.antlr.org./doc/index.html.

Philippsen, M. (2000). A Survey of Concurrent Object-Oriented Languages. *Concurrency Computat.: Pract. Exper.*, 12(10):917–980.

Ravara, A. and Vasconcelos, V. T. (2000). Typing Non-uniform Concurrent Objects. In *CONCUR'00*, volume 1877 of *LNCS*, pages 474–488, Berlin. Springer.

Tomlinson, C. and Singh, V. (1989). Inheritance and synchronization with enabled-sets. In *OOPSLA'89*, pages 103–112. ACM.