Analysis of Deadlocks in Object Groups

Elena Giachino, Cosimo Laneve

Analysis of deadlocks in object groups ⋆

Elena Giachino Cosimo Laneve

Dipartimento di Scienze dell'Informazione, Università di Bologna

Abstract. Object groups are collections of objects that perform collective work. We study a calculus with object groups and develop a technique for the deadlock analysis of such systems based on abstract descriptions of method's behaviours.

1 Introduction

Object groups are collections of objects that perform collective work. The group abstraction is an encapsulation mechanism that is convenient in distributed programming in several circumstances. For example, in order to achieve continuous availability or load balancing through replication, or for retrieval services of distributed data. In these cases, in order to keep consistencies, the group abstraction must define suitable protocols to synchronize group members. As usual with synchronization protocols, it is possible that object groups may manifest deadlocks, which are particularly hard to discover in this context because of the two encapsulation levels of the systems (the object and the group levels).

Following the practice to define lightweight fragments of languages that are sufficiently small to ease proofs of basic properties, we define an object-oriented calculus with group operations and develop a technique for the analysis of deadlocks. Our object-oriented language, called `FJg`, is an imperative version of Featherweight Java [9] with method invocations that are asynchronous and group-oriented primitives that are taken from *Creol* [10] (*cf.* the JCoBoxes [19]).

In `FJg`, objects always belong to one group that is defined when they are created. Groups consist of multiple tasks, which are the running methods of the objects therein. Tasks are cooperatively scheduled, that is there is at most one task active at each time per group and the active task explicitly returns the control in order to let other tasks progress. Tasks are created by method invocation that are *asynchronous* in `FJg`: the caller activity continues after the invocation and the called code runs on a different task that may belong to a different group. The synchronization between the caller and the called methods is performed when the result is strictly necessary [10, 21, 3]. Technically, the decoupling of method invocation and the returned value is realized using *future variables* (see [6] and the references in there), which are pointers to values that may be not available yet. Clearly, the access to values of future variables may require waiting for the value to be returned.

In a model with object groups and cooperative scheduling, a typical deadlock situation occurs when two active tasks in different groups are waiting for each other to

⋆ The authors are member of the joint FOCUS Research Team INRIA/Università di Bologna. This research has been funded by the EU project FP7-231620 HATS.

return a value. This circular dependency may involve less or more than two tasks. For example, a case of circularity of size one is

```
Int fact(Int n){    if (n=0) then return 1 ;
                    else return n*(this!fact(n-1).get)   }
```

The above FJg code defines the factorial function (for the sake of the example we include primitive types Int and conditional into FJg syntax. See Section 2.1). The invocation of this!fact(n) deadlocks on the recursive call this!fact(n-1) because the caller does not explicitly release the group lock. The operation get is needed in order to synchronously retrieve the value returned by the invocation.

We develop a technique for the analysis of deadlocks in FJg programs based on *contracts*. Contracts are abstract descriptions of behaviours that retain the necessary informations to detect deadlocks [12, 11]. For example, the contract of fact (assuming it belongs to the class Ops) is $\texttt{G()\{ Ops.fact}^{\texttt{g}}\texttt{: G() \}}$. This contract declares that, when fact is invoked on an object of a group G, then it will call recursively fact on an object of the same group G without releasing the control – a *group dependency*. With this contract, any invocation of fact is fated to deadlock because of the circularity between G and itself (actually this.fact(0) never deadlocks, but the above contract is not expressive enough to handle such cases).

In particular, we define an inference system for associating a contract to every method of the program and to the expression to evaluate. Then we define a simple algorithm – the dla algorithm – returning informations about group dependencies. The presence of circularities in the result of dla reveals the possible presence of deadlocked computations. Overall, our results show the possibility and the benefit of applying techniques developed for process calculi to the area of object-oriented programming.

The paper is organized as follows. Section 2 defines FJg by introducing the main ideas and presenting its syntax and operational semantics. Section 3 discusses few sample programs in FJg and the deadlocks they manifest. Section 4 defines contracts and the inference algorithm for deriving contracts of expressions and methods. Section 5 considers the problem of extracting dependencies from contracts, presents the algorithm dla, and discusses its enhancements. Section 6 surveys related works, and we give conclusions and indications of further work in Section 7.

Due to space limitations, the technical details are omitted. We refer the interested reader to the full paper in the home-pages of the authors.

2 The calculus FJg

In FJg a program is a collection of class definitions plus an expression to evaluate. A simple definition in FJg is the class C in Table 1. This program defines a class C with a method m. When m is invoked, a new object of class C is created and returned. A distinctive feature of FJg is that an object belongs to a unique group. In the above case, the returned object belongs to a new group – created by the operator newg. If the new object had to belong to the group of the caller method, then we would have used the standard operation new.

```
class C {    C m() { return newg C() ;} }
class D extends C {    C n(D c) { return (c!m()).get ;} }
```

Table 1. Simple classes in FJg

Method invocations in FJg are asynchronous. For example, the class D in Table 1 defines an extension of C with method n. In order to emphasize that the semantics of method invocation is not as usual, we use the exclamation mark (instead of the dot notation). In FJg, when a method is invoked, the caller continues executing *in parallel with* the callee *without releasing its own group lock*; the callee gets the control by acquiring the lock of its group when it is free. This guarantees that, at each point in time, at most one task may be active per group. The get operation in the code of n constraints the method to wait for the return value of the callee before terminating (and therefore releasing the group lock).

In FJg, it is also possible to wait for a result without keeping the group lock. This is performed by the operation await that releases the group lock and leaves other tasks the chance to perform their activities until the value of the called method is produced. That is, x!m().await.get corresponds to a method invocation in standard object-oriented languages.

The decoupling of method invocation and the returned value is realized in FJg by using *future types*. In particular, if a method is declared to return a value of type C, then its invocations return values of type Fut(C), rather than values of type C. This means that the value is not available yet; when it will be, it is going to be of type C. The operation get takes an expression of type Fut(C) and returns C (as the reader may expect, await takes an expression of type Fut(C) and returns Fut(C)).

2.1 Syntax

The syntax of FJg uses four disjoint infinite sets of *class names*, ranged over by A, B, C, $\cdots$, *field names*, ranged over by f, g, $\cdots$, *method names*, ranged over by m, n, $\cdots$, and *parameter names*, ranged over by x, y, $\cdots$, that contains the special name this. We write $\bar{\mathtt{C}}$ as a shorthand for $\mathtt{C}_1, \cdots, \mathtt{C}_n$ and similarly for the other names. Sequences $\mathtt{C}_1\ \mathtt{f}_1, \cdots, \mathtt{C}_n\ \mathtt{f}_n$ are abbreviated as with $\bar{\mathtt{C}}\ \bar{\mathtt{f}}$.

The abstract syntax of *class declarations* CL, *method declarations* M, and *expressions* e of FJg is the following

```
CL ::=  class C extends C {C̄ f̄; M̄}
M  ::=  C m (C̄ x̄){return e ; }
e  ::=  x | this.f | this.f = e | e!m(ē) | new C(ē) | e; e
     |  newg C(ē) | e.get | e.await
```

Sequences of field declarations $\bar{\mathtt{C}}\ \bar{\mathtt{f}}$, method declarations $\bar{\mathtt{M}}$, and parameter declarations $\bar{\mathtt{C}}\ \bar{\mathtt{x}}$ are assumed to contain no duplicate names.

A program is a pair (CT, e), where the *class table* CT is a finite mapping from class names to class declarations CL and e is an expression. In what follows we always assume a fixed class table CT. According to the syntax, every class has a superclass declared with extends. To avoid circularities, we assume a distinguished class name

Field lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \mathit{fields}(\texttt{D}) = \bar{\texttt{C}}'\ \bar{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \bar{\texttt{C}}\ \bar{\texttt{f}},\ \bar{\texttt{C}}'\ \bar{\texttt{g}}}$$

Method type lookup:

$$\frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \texttt{C}'\ \texttt{m}\ (\bar{\texttt{C}}'\ \bar{\texttt{x}})\{\texttt{return e; }\} \in \bar{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \bar{\texttt{C}}' \to \texttt{C}'} \qquad \frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \texttt{m} \notin \bar{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \mathit{mtype}(\texttt{m}, \texttt{D})}$$

Method body lookup:

$$\frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \texttt{C}'\ \texttt{m}\ (\bar{\texttt{C}}'\ \bar{\texttt{x}})\{\texttt{return e; }\} \in \bar{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}.\texttt{e}} \qquad \frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \texttt{m} \notin \bar{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{D})}$$

Heap lookup functions:

$$\frac{\mathsf{H}(\mathsf{o}) = (\mathsf{C}, \mathsf{G}, [\bar{\mathsf{f}} : \bar{\mathsf{v}}])}{\mathit{class}(\mathsf{H}, \mathsf{o}) = \mathsf{C} \qquad \mathit{group}(\mathsf{H}, \mathsf{o}) = \mathsf{G} \qquad \mathit{field}(\mathsf{H}, \mathsf{o}, \mathsf{f}_i) = \mathsf{v}_i}$$

Table 2. Lookup auxiliary functions ($\bullet$ is the empty sequence)

`Object` with no field and method declarations whose definition does not appear in the class table. As usual, `class C` $\{\cdots\}$ abbreviates `class C extends Object` $\{\cdots\}$.

Let *types* T be either class names C or *futures* `Fut`(C). Let also *fields*(C), *mtype*(m, C), and *mbody*(m, C) [9] be the standard `FJ` lookup functions that are reported in Table 2. The class table satisfies the following well-formed conditions:

(*i*) `Object` $\notin$ *dom*(CT);
(*ii*) for every C $\in$ *dom*(CT), CT(C) = `class C` $\cdots$;
(*iii*) every class name occurring in CT belongs to *dom*(CT);
(*iv*) the least relation `<:` , called *subtyping relation*, over types `T`, closed by reflexivity and transitivity and containing

$$\frac{\texttt{C}_1 \texttt{ <: } \texttt{C}_2}{\texttt{Fut}(\texttt{C}_1) \texttt{ <: } \texttt{Fut}(\texttt{C}_2)} \qquad \frac{\textsc{ct}(\texttt{C}_1) = \texttt{class C}_1 \texttt{ extends C}_2 \{\cdots\}}{\texttt{C}_1 \texttt{ <: } \texttt{C}_2}$$

is antisymmetric;
(*v*) if CT(C) = `class C extends D` $\{\cdots\}$ and *mtype*(m, C) = $\bar{\texttt{C}}' \to \texttt{C}'$ and *mtype*(m, D) = $\bar{\texttt{D}}' \to \texttt{D}'$ then $\texttt{C}' \texttt{ <: } \texttt{D}'$ and $\bar{\texttt{D}}' \texttt{ <: } \bar{\texttt{C}}'$.

It is worth to remark that future types never appear in `FJg` programs, where types of fields and of methods are always classes. This restriction excludes either to store future values in fields or to invoke methods with future values (that later on may be

2.2 Semantics

Below we use an infinite set of *object names*, ranged over by o, o′, $\cdots$, an infinite set of *group names*, ranged over by G, G′, $\cdots$, and an infinite set of *task names*, ranged over

by $\mathtt{t}$, $\mathtt{t}'$, $\cdots$. We assume that the set of group names has a distinguished element $\Im$, associated to the expressions irrelevant for the deadlock analysis, such as `this.f`.

FJg has an operational semantics that is defined in terms of a transition relation $\longrightarrow$ between *configurations* $\mathsf{H} \Vdash \mathsf{S}$, where H is the *heap* and S is a set of *tasks*. The heap maps (i) objects names $\mathtt{o}$ to tuples $(\mathtt{C}, \mathtt{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}])$ that record their class, their group, and their fields' values; (ii) group names $\mathtt{G}$ to either $\bot$ or $\top$ that specify whether the group is unlocked or locked, respectively. We use the standard update operations on heaps $\mathsf{H}[\mathtt{o} \mapsto (\mathtt{C}, \mathtt{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}])]$ and $\mathsf{H}[\mathtt{G} \mapsto \mathtt{l}]$ and on fields $[\bar{\mathtt{f}} : \bar{\mathtt{v}}][\mathtt{f} : \mathtt{v}]$ with the usual meanings. Tasks are tuples $\mathtt{t} :^{\mathtt{l}}_{\mathtt{o}} \mathtt{e}$, where $\mathtt{t}$ is the task name, $\mathtt{o}$ is the object of the task, $\mathtt{l}$ is either $\top$ (if the task owns the group lock) or $\bot$ (if not), and $\mathtt{e}$ is the expression to evaluate. The superscript $\mathtt{l}$ is omitted when it is not relevant. In the semantic clauses, by abuse of the notation, the syntactic category $\mathtt{e}$ also addresses *values*, ranged over by $\mathtt{v}$, which are either object or task names. The set of object and task names in $\mathtt{e}$ is returned by the function *names*$(\mathtt{e})$. The same function, when applied to a set of tasks S returns the object, group, and task names in S. The operational semantics also uses

- the *heap lookup functions* *class*$(\mathsf{H}, \mathtt{o})$, *group*$(\mathsf{H}, \mathtt{o})$, and *field*$(\mathsf{H}, \mathtt{o}, \mathtt{f}_i)$ that respectively return the class, the group and the values of i-th field of $\mathtt{o}$ in H (see Table 2);
- *evaluation contexts* E whose syntax is:

$$\begin{array}{rcl} \mathsf{E} ::= & [\,] \mid \mathsf{E}!\mathtt{m}(\bar{\mathtt{e}}) \mid \mathtt{this.f} = \mathsf{E} \mid \mathtt{o}!\mathtt{m}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \\ \mid & \mathtt{newg}\ \mathtt{C}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \mid \mathsf{E}.\mathtt{get} \mid \mathsf{E}.\mathtt{await} \mid \mathsf{E}; \mathtt{e} \end{array}$$

The preliminary notions are all in place for the definition of the transition relation that is given in Table 3. It is worth to notice that, in every rule, a task moves if it owns the lock of its group. Apart this detail, the operations of update, object creation, and sequence are standard. We therefore focus on the operations about groups and futures. Rule (Invk) defines asynchronous method invocation, therefore the evaluation produces a future reference $\mathtt{t}'$ to the returned value, which may be retrieved by a `get` operation if needed. Rule (Newg) defines $\mathtt{newg}\ \mathtt{C}(\bar{\mathtt{v}})$, which creates a new group $\mathtt{G}'$ and a new object $\mathtt{o}'$ of that group with fields initialized to $\bar{\mathtt{v}}$. This object is returned and the group $\mathtt{G}'$ is unlocked – no task of its is running. It will be locked as soon as a method of $\mathtt{o}'$ is invoked – see (Lock). Rule (Release) models method termination that amounts to store the returned value in the configuration and releasing the group lock. Rule (Get) allows one to retrieve the value returned by a method. Rules (AwaitT) and (AwaitF) model the `await` operation: if the task $\mathtt{t}'$ is terminated – it is paired to a value in the configuration – then `await` is unblocking; otherwise the group lock is released and the task $\mathtt{t}$ is blocked. Rule (Config) has standard premises for avoiding unwanted name matches when lifting local reductions to complete configurations.

The initial configuration of a program $(\mathtt{CT}, \mathtt{e})$ is $\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{e}[^{\mathtt{o}}/_{\mathtt{this}}]$ where $\mathsf{H} = \mathtt{o} \mapsto (\mathtt{Object}, \mathtt{G}, [\,]), \mathtt{G} \mapsto \top$ (following our previous agreement, the class table is implicit).

Example 1. As an example, we detail the evaluation of the expression `(newg D())!n(newg D()).get`, where the class `D` is defined in Table 1.

$$\begin{array}{ll} \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{(newg\ D())!n(newg\ D()).get} & \\ \quad \longrightarrow \mathsf{H}_1 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{o1!n(newg\ D()).get} & (1) \\ \quad \longrightarrow \mathsf{H}_2 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{o1!n(o2).get} & (2) \\ \quad \longrightarrow \mathsf{H}_2 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\bot}_{\mathtt{o1}} \mathtt{o2!m().get} & (3) \end{array}$$

$$\textsc{(Update)}\ \frac{\mathsf{H}(\mathsf{o}') = (\mathsf{C}, \mathsf{G}, [\bar{\mathsf{f}} : \bar{\mathsf{v}}])}{\mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{o}'.\mathsf{f} = \mathsf{v}] \longrightarrow \mathsf{H}[\mathsf{o}' \mapsto (\mathsf{C}, \mathsf{G}, [\bar{\mathsf{f}} : \bar{\mathsf{v}}][\mathsf{f} : \mathsf{v}])] \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{v}]}$$

$$\textsc{(Invk)}\ \frac{\mathit{class}(\mathsf{H}, \mathsf{o}') = \mathsf{C} \quad \mathit{mbody}(\mathtt{m}, \mathsf{C}) = \bar{\mathtt{x}}.\mathsf{e} \quad \mathsf{t}' \neq \mathsf{t}}{\mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{o}'!\mathtt{m}(\bar{\mathsf{v}})] \longrightarrow \mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{t}'],\ \mathsf{t}' :^{\bot}_{\mathsf{o}'} \mathsf{e}[\mathsf{o}'/\mathtt{this}][\bar{\mathsf{v}}/\bar{\mathtt{x}}]}$$

$$\textsc{(New)}\ \frac{\mathit{group}(\mathsf{H}, \mathsf{o}) = \mathsf{G} \quad \mathit{fields}(\mathsf{C}) = \bar{\mathsf{T}}\ \bar{\mathsf{f}} \quad \mathsf{o}' \notin \mathrm{dom}(\mathsf{H}) \quad \mathsf{H}' = \mathsf{H}[\mathsf{o}' \mapsto (\mathsf{C}, \mathsf{G}, [\bar{\mathsf{f}} : \bar{\mathsf{v}}])]}{\mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{new}\ \mathsf{C}(\bar{\mathsf{v}})] \longrightarrow \mathsf{H}' \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{o}']}$$

$$\textsc{(Newg)}\ \frac{\mathit{fields}(\mathsf{C}) = \bar{\mathsf{T}}\ \bar{\mathsf{f}} \quad \mathsf{o}', \mathsf{G}' \notin \mathrm{dom}(\mathsf{H}) \quad \mathsf{H}' = \mathsf{H}[\mathsf{o}' \mapsto (\mathsf{C}, \mathsf{G}', [\bar{\mathsf{f}} : \bar{\mathsf{v}}])][\mathsf{G}' \mapsto \bot]}{\mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{newg}\ \mathsf{C}(\bar{\mathsf{v}})] \longrightarrow \mathsf{H}' \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{o}']}$$

$$\textsc{(Get)}\ \mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{t}'.\mathtt{get}],\ \mathsf{t}' :_{\mathsf{o}'} \mathsf{v} \longrightarrow \mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{v}],\ \mathsf{t}' :_{\mathsf{o}'} \mathsf{v}$$

$$\textsc{(AwaitT)}\ \mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{t}'.\mathtt{await}],\ \mathsf{t}' :_{\mathsf{o}'} \mathsf{v} \longrightarrow \mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{t}'],\ \mathsf{t}' :_{\mathsf{o}'} \mathsf{v}$$

$$\textsc{(AwaitF)}\ \frac{\mathit{group}(\mathsf{H}, \mathsf{o}) = \mathsf{G}}{\mathsf{H}[\mathsf{G} \mapsto \top] \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{t}'.\mathtt{await}] \longrightarrow \mathsf{H}[\mathsf{G} \mapsto \bot] \Vdash \mathsf{t} :^{\bot}_{\mathsf{o}} \mathsf{E}[\mathsf{t}'.\mathtt{await}]}$$

$$\textsc{(Lock)}\ \frac{\mathit{group}(\mathsf{H}, \mathsf{o}) = \mathsf{G} \quad \mathsf{e} \neq \mathsf{v}}{\mathsf{H}[\mathsf{G} \mapsto \bot] \Vdash \mathsf{t} :^{\bot}_{\mathsf{o}} \mathsf{e} \longrightarrow \mathsf{H}[\mathsf{G} \mapsto \top] \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{e}} \qquad \textsc{(Release)}\ \frac{\mathit{group}(\mathsf{H}, \mathsf{o}) = \mathsf{G}}{\mathsf{H}[\mathsf{G} \mapsto \top] \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{v} \longrightarrow \mathsf{H}[\mathsf{G} \mapsto \bot] \Vdash \mathsf{t} :^{\bot}_{\mathsf{o}} \mathsf{v}}$$

$$\textsc{(Seq)}\ \mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{v}; \mathsf{e} \longrightarrow \mathsf{H} \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathsf{e} \qquad \textsc{(Config)}\ \frac{\mathsf{H} \Vdash \mathsf{S} \longrightarrow \mathsf{H}' \Vdash \mathsf{S}' \quad (\mathit{names}(\mathsf{S}') \setminus \mathit{names}(\mathsf{S})) \cap \mathit{names}(\mathsf{S}'') = \emptyset}{\mathsf{H} \Vdash \mathsf{S}, \mathsf{S}'' \longrightarrow \mathsf{H}' \Vdash \mathsf{S}', \mathsf{S}''}$$

Table 3. The transition relation of FJg.

$$\longrightarrow \mathsf{H}_2[\mathtt{G1} \mapsto \top] \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{o2!m().get} \tag{4}$$

$$\longrightarrow \mathsf{H}_2[\mathtt{G1} \mapsto \top] \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{t2.get}, \mathtt{t2} :^{\bot}_{\mathtt{o2}} \mathtt{newg\ C()} \tag{5}$$

$$\longrightarrow \mathsf{H}_3 \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{t2.get}\ , \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{newg\ C()} \tag{6}$$

$$\longrightarrow \mathsf{H}_4 \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{t2.get}\ , \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{o3} \tag{7}$$

$$\longrightarrow \mathsf{H}_4 \Vdash \mathsf{t} :^{\top}_{\mathsf{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{o3}\ , \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{o3} \tag{8}$$

where $\mathsf{H} = \mathsf{o} \mapsto (\mathtt{Object}, \mathsf{G}, [\,]), \mathsf{G} \mapsto \top$ $\quad \mathsf{H}_1 = \mathsf{H}[\mathtt{o1} \mapsto (\mathtt{D}, \mathtt{G1}, [\,]), \mathtt{G1} \mapsto \bot]$
$\mathsf{H}_2 = \mathsf{H}_1[\mathtt{o2} \mapsto (\mathtt{D}, \mathtt{G2}, [\,]), \mathtt{G2} \mapsto \bot]$ $\quad \mathsf{H}_3 = \mathsf{H}_2[\mathtt{G1} \mapsto \top, \mathtt{G2} \mapsto \top]$
$\mathsf{H}_4 = \mathsf{H}_3[\mathtt{o3} \mapsto (\mathsf{C}, \mathtt{G3}, [\,]), \mathtt{G3} \mapsto \bot]$

The reader may notice that, in the final configuration, the tasks `t1` and `t2` will terminate one after the other by releasing all the group locks.

3 Deadlocks

We introduce our formal developments about deadlock analysis by discussing a couple of expressions that manifest deadlocks. Let `D` be the class of Table 1 and consider the expression `(newg D())!n(new D()).get`. This expression differs from the one of Example 1 for the argument of the method (now it is `new D()`, before it was `newg`

D()). The computation of (newg D())!n(new D()).get is the same of the one in Example 1 till step (5), replacing the value of H_2 with $\mathsf{H}_1[\mathtt{o2} \mapsto (\mathtt{D}, \mathtt{G1}, [\,])]$ (o1 and o2 belong to the same group G1). At step (5), the task t2 will indefinitely wait for getting the lock of G1 since t1 will never release it.

Deadlocks may be difficult to discover when they are caused by schedulers' choices. For example, let D′ be the following extension of the class C in Table 1:

```
class D′ extends C {  D′ n(D′ b, D′ c){ return b!p(c);c!p(b);this ;}
                      C p(D′ c){ return (c!m()).get ;} }
```

and consider the expression (newg D′())!n(newg D′(),new D′()).get. The evaluation of this expression yields the tasks $\mathtt{t} :^{\perp}_{\mathtt{o}} \mathtt{o1}, \mathtt{t1} :^{\perp}_{\mathtt{o1}} \mathtt{o1}, \mathtt{t2} :^{\perp}_{\mathtt{o2}} \mathtt{o3!m().get}, \mathtt{t3} :^{\perp}_{\mathtt{o3}}$ o2!m().get with $\mathtt{o} \in \mathtt{G}$, $\mathtt{o1}, \mathtt{o3} \in \mathtt{G1}$ and $\mathtt{o2} \in \mathtt{G2}$. If t2 is completed before t3 grabs the lock (or conversely) then no deadlock will be manifested. On the contrary, the above tasks may evolve into $\mathtt{t} :^{\perp}_{\mathtt{o}} \mathtt{o1}, \mathtt{t1} :^{\perp}_{\mathtt{o1}} \mathtt{o1}, \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{t4.get}, \mathtt{t3} :^{\top}_{\mathtt{o3}} \mathtt{t5.get}, \mathtt{t4} :^{\perp}_{\mathtt{o3}}$ newg C(), $\mathtt{t5} :^{\perp}_{\mathtt{o2}}$ newg C() that is a deadlock because neither t4 nor t5 will have any chance to progress.

4 Contracts in FJg

In the following we will consider *plain* FJg programs where methods never return fields nor it is possible to invoke methods with fields in the subject part or in the object part. For example, the expressions (this.f)!m() and x!n(this.f) are not admitted, as well as a method declaration like C p(){ return this.f ; }. (The contract system in Table 4 and 5 will ban not-plain programs.) This restriction simplifies the foregoing formal developments about deadlock analysis; the impact of the restriction on the analysis of deadlocks is discussed in Section 7.

The analysis of deadlocks in FJg uses abstract descriptions of behaviours, called *contracts*, and an inference system for associating contracts to expressions (and methods). (The algorithm taking contracts and returning details about deadlocks is postponed to the next section.) Formally, *contracts* $\gamma, \gamma', \cdots$ are terms defined by the rules:

$$\gamma ::= \varepsilon \mid \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}}); \gamma \mid \mathtt{C.m^g} : \mathtt{G}(\bar{\mathtt{G}}); \gamma \mid \mathtt{C.m^a} : \mathtt{G}(\bar{\mathtt{G}}); \gamma$$

As usual, $\gamma; \varepsilon = \gamma = \varepsilon; \gamma$. When $\bar{\gamma}$ is a tuple of contracts $(\gamma_1, \cdots, \gamma_n)$, $\mathtt{seq}(\bar{\gamma})$ is a shortening for the sequential composition $\gamma_1; \cdots; \gamma_n$. The sequence γ collects the method invocations inside expressions. In particular, the items of the sequence may be empty, noted ε; or $\mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})$, specifying that the method m of class C is going to be invoked on an object of group G and with arguments of group $\bar{\mathtt{G}}$; or $\mathtt{C.m^g} : \mathtt{G}(\bar{\mathtt{G}})$, a method invocation followed by a get operation; or $\mathtt{C.m^a} : \mathtt{G}(\bar{\mathtt{G}})$, a method invocation followed by an await operation. For example, the contract C.m:G(); D.n:G′() defines two method invocations on groups G and G′, respectively (methods carry no arguments). The contract $\mathtt{C.m:G();\ D.n^g:G'();\ E.p^a:G''()}$ defines three method invocations on different groups; the second invocation is followed by a get and the third one by an await.

Method contracts, ranged over by $\mathcal{G}, \mathcal{G}', \cdots$, are $\mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$, where $\mathtt{G}, \bar{\mathtt{G}}$ are pairwise different group names – $\mathtt{G}(\bar{\mathtt{G}})$ is the *header* –, $\mathtt{G}'$ is the *returned group*, and γ is a *contract*.

A contract $\mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$ binds the group of the object `this` and the group of the arguments of the method invocation in the sequence γ. The returned group $\mathtt{G}'$ may belong to $\mathtt{G}, \bar{\mathtt{G}}$ or not, that is it may be a new group created by the method. For example, let $\gamma = \mathtt{C.m{:}G();\ D.n^g{:}G'();\ E.p^a{:}G''()}$ in (i) $\mathtt{G}(\mathtt{G}', \mathtt{G}'')\{\gamma\}\ \mathtt{G}''$ and (ii) $\mathtt{G}(\mathtt{G}')\{\gamma\}\ \mathtt{G}''$. In case (i) every group name in γ is *bound* by names in the header. This means that method invocations are bound to either the group name of the caller or to group names of the arguments. This is not the case for (ii), where the third method invocation in its body and the returned group address a group name that is unbound by the header. This means that the method with contract (ii) is creating an object of class `E` belonging to a new group – called $\mathtt{G}''$ in the body – and is performing the third invocation to a method of this object.

Method contracts are quotiented by the least equivalence $=^\alpha$ identifying two contracts that are equivalent after an injective renaming of (free and bound) group names. For example $\mathtt{G}(\mathtt{G}')\{\mathtt{C.m{:}G();\ D.n^g{:}G'();\ E.p^a{:}G''()}\}\ \mathtt{G}' =^\alpha \mathtt{G}_1(\mathtt{G}_2)\{\mathtt{C.m{:}G_1();\ D.n^g{:}G_2();\ E.p^a{:}G''()}\}\ \mathtt{G}_2$. Additionally, since the occurrence of $\mathtt{G}''$ represents an unbound group, writing $\mathtt{G}''$ or any other free group name is the same. That is $\mathtt{G}(\mathtt{G}')\{\mathtt{C.m{:}G();\ D.n^g{:}G'();\ E.p^a{:}G''()}\}\ \mathtt{G}' =^\alpha \mathtt{G}_1(\mathtt{G}_2)\{\mathtt{C.m{:}G_1();\ D.n^g{:}G_2();\ E.p^a{:}G_3()}\}\ \mathtt{G}_2$.

Let Γ, called *environment*, be a map from either names to pairs $(\mathtt{T}, \mathtt{G})$ or class and method names, *i.e.* $\mathtt{C.m}$, to terms $\mathtt{G}(\bar{\mathtt{G}}) \to \mathtt{G}'$, called *group types*, where $\mathtt{G}$, $\bar{\mathtt{G}}$, $\mathtt{G}'$ are all different from $\circlearrowright$. The contract judgement for expressions has the following form and meaning: $\Gamma \vdash \mathtt{e} : (\mathtt{T}, \mathtt{G}), \gamma$ means that the expression `e` has type `T` and group `G` and has contract γ in the environment Γ.

Contract rules for expressions are presented in Tables 4 where,

- in rule (T-Invk) we use the operator $\mathit{fresh}(\bar{\mathtt{G}}, \mathtt{G})$ that returns $\mathtt{G}$ if $\mathtt{G} \in \bar{\mathtt{G}}$ or a fresh group name otherwise;
- in rules (T-Get) and (T-Await), we use the operator $\lozenge$ defined as follows:

$$\begin{array}{rl} \varepsilon \lozenge \mathtt{await} & = \varepsilon \\ (\gamma; \mathtt{C.m}\ \mathtt{G}(\bar{\mathtt{G}})) \lozenge \mathtt{await} & = \gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}}) \qquad (\gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}})) \lozenge \mathtt{await} = \gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}}) \\ (\gamma; \mathtt{C.m}\ \mathtt{G}(\bar{\mathtt{G}})) \lozenge \mathtt{get} & = \gamma; \mathtt{C.m^g}\ \mathtt{G}(\bar{\mathtt{G}}) \qquad (\gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}})) \lozenge \mathtt{get} = \gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}}) \end{array}$$

(the other combinations of `get` and `await` are forbidden by the contract system).

The rule (T-Field) associates the group $\circlearrowright$ to the expression `this.f`, provided the field `f` exists. This judgment, together with the premises of (T-Invk) and the assumption that $\circlearrowright$ does not appear in $\Gamma(\mathtt{C.m})$, imply that subjects and objects of method invocations cannot be expressions as `this.f`. Apart these constraint, the contract of $\mathtt{e!m}(\bar{\mathtt{e}})$ is as expected, *i.e.* the sequence of the contract of `e`, plus the one of $\bar{\mathtt{e}}$, with a tailing item $\mathtt{C.m}\ \mathtt{G}(\bar{\mathtt{G}})$. Rules (T-New) and (T-NewG) are almost the same, except the fact that the latter one returns a fresh group name while the former one return the group of `this`. The other rules are standard.

Let $\mathtt{G}(\bar{\mathtt{G}}) \to \mathtt{G}' =^\alpha \mathtt{H}(\bar{\mathtt{H}}) \to \mathtt{H}'$ if and only if $\mathtt{G}(\bar{\mathtt{G}})\{\varepsilon\}\mathtt{G}' =^\alpha \mathtt{H}(\bar{\mathtt{H}})\{\varepsilon\}\mathtt{H}'$. The contract judgements for method declarations, class declarations and programs have the following forms and meanings:

- $\Gamma; \mathtt{C} \vdash \mathtt{D'\ m}\ (\bar{\mathtt{D}}\ \bar{\mathtt{x}})\{\mathtt{return\ e;}\} : \mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$ means that the method $\mathtt{D'\ m}\ (\bar{\mathtt{D}}\ \bar{\mathtt{x}})\{\mathtt{return\ e;}\}$ has method contract $\mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$ in the class `C` and in the environment Γ;
- $\Gamma \vdash \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \ \bar{\mathtt{M}}\} : \{\bar{\mathtt{m}} \mapsto \bar{\mathcal{G}}\}$ means that the class declaration `C` has contract $\{\bar{\mathtt{m}} \mapsto \bar{\mathcal{G}}\}$ in the environment Γ;

$$\text{(T-Var)}\ \frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}), \varepsilon} \qquad \text{(T-Field)}\ \frac{\Gamma \vdash \mathtt{this} : (\mathtt{C}, \mathtt{G}), \varepsilon \quad \mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C})}{\Gamma \vdash \mathtt{this.f} : (\mathtt{D}, \eth), \varepsilon}$$

$$\text{(T-Invk)}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : (\mathtt{C}, \mathtt{G}), \gamma \quad \Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{D}}, \bar{\mathtt{G}}), \bar{\gamma} \quad \eth \notin \mathtt{G}\bar{\mathtt{G}} \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{C}} \to \mathtt{C}' \quad \bar{\mathtt{D}} <: \bar{\mathtt{C}} \\ \Gamma(\mathtt{C.m}) = \mathtt{G}'(\bar{\mathtt{G}}') \to \mathtt{G}'' \quad \mathtt{G}''' = \mathit{fresh}(\mathtt{G}\bar{\mathtt{G}}, \mathtt{G}''[{}^{\mathtt{G}\bar{\mathtt{G}}}/_{\mathtt{G}'\bar{\mathtt{G}}'}])\end{array}}{\Gamma \vdash \mathtt{e!m}(\bar{\mathtt{e}}) : (\mathtt{Fut}(\mathtt{C}'), \mathtt{G}'''), \gamma; (\mathtt{seq}(\bar{\gamma})); \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})}$$

$$\text{(T-New)}\ \frac{\Gamma \vdash \mathtt{this} : (\mathtt{C}, \mathtt{G}), \varepsilon \quad \Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{C}}, \bar{\mathtt{G}}), \bar{\gamma} \quad \mathit{fields}(\mathtt{C}') = \bar{\mathtt{C}}'\ \bar{\mathtt{f}}' \quad \bar{\mathtt{C}} <: \bar{\mathtt{C}}'}{\Gamma \vdash \mathtt{new}\ \mathtt{C}'(\bar{\mathtt{e}}) : (\mathtt{C}', \mathtt{G}), (\mathtt{seq}(\bar{\gamma}))} \qquad \text{(T-NewG)}\ \frac{\Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{C}}, \bar{\mathtt{G}}), \bar{\gamma} \quad \mathtt{G}\ \mathit{fresh} \quad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}'\ \bar{\mathtt{f}}' \quad \bar{\mathtt{C}} <: \bar{\mathtt{C}}'}{\Gamma \vdash \mathtt{newg}\ \mathtt{C}(\bar{\mathtt{e}}) : (\mathtt{C}, \mathtt{G}), (\mathtt{seq}(\bar{\gamma}))}$$

$$\text{(T-Get)}\ \frac{\Gamma \vdash \mathtt{e} : (\mathtt{Fut}(\mathtt{C}), \mathtt{G}), \gamma}{\Gamma \vdash \mathtt{e.get} : (\mathtt{C}, \mathtt{G}), \gamma \between \mathtt{get}} \qquad \text{(T-Await)}\ \frac{\Gamma \vdash \mathtt{e} : (\mathtt{Fut}(\mathtt{C}), \mathtt{G}), \gamma}{\Gamma \vdash \mathtt{e.await} : (\mathtt{Fut}(\mathtt{C}), \mathtt{G}), \gamma \between \mathtt{await}}$$

$$\text{(T-Update)}\ \frac{\Gamma \vdash \mathtt{this} : (\mathtt{C}, \mathtt{G}), \varepsilon \quad \mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C}) \quad \Gamma \vdash \mathtt{e} : (\mathtt{D}', \mathtt{G}'), \gamma \quad \mathtt{D}' <: \mathtt{D}}{\Gamma \vdash \mathtt{this.f} = \mathtt{e} : (\mathtt{D}', \mathtt{G}'), \gamma} \qquad \text{(T-Seq)}\ \frac{\Gamma \vdash \mathtt{e} : (\mathtt{T}, \mathtt{G}), \gamma \quad \Gamma \vdash \mathtt{e}' : (\mathtt{T}', \mathtt{G}'), \gamma'}{\Gamma \vdash \mathtt{e}; \mathtt{e}' : (\mathtt{T}', \mathtt{G}'), \gamma; \gamma'}$$

Table 4. Contract rules of FJg expressions

– $\vdash (\text{CT}, \mathtt{e}) : \text{CCT}, (\mathtt{T}, \mathtt{G}), \gamma$ means that the program $(\text{CT}, \mathtt{e})$ has *contract class table* CCT and type/group/contract $(\mathtt{T}, \mathtt{G}), \gamma$, where a contract class table maps class names to terms $\{\bar{\mathtt{m}} : \bar{\mathcal{G}}\}$.

Table 5 reports the typing judgments for method and class declarations and for programs. We use the auxiliary function $\mathit{mname}(\bar{\mathtt{M}})$ that returns the sequence of method names in $\bar{\mathtt{M}}$. We also write $m \in \text{CT}(\mathtt{C})$ if $\text{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \ \bar{\mathtt{M}}\}$ and $m \in \mathit{mname}(\bar{\mathtt{M}})$. Rule (T-Program) requires that if a subclass overrides a method of a superclass then the two methods must have equal contract. This constraint is expressed by the predicate CCT *is* CT *consistent* defined as follows:

$$\begin{array}{l}\textit{for every}\ \text{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\cdots\} : \\ \qquad m \in \text{CT}(\mathtt{C})\ \textit{and}\ m \in \text{CT}(\mathtt{D}) \quad \textit{implies} \quad \text{CCT}(\mathtt{C})(\mathtt{m}) =^{\alpha} \text{CCT}(\mathtt{D})(\mathtt{m})\end{array}$$

This consistency requirement may be definitely weakened: we defer to future works the issue of studying a sub-contract relation that is correct with respect to class inheritance.

The proof of correctness of the contract system in Tables 4 and 5 requires additional rules that define the contract correctness of (runtime) configurations. These rules are:

$$\text{(T-Task)}\ \frac{\Gamma \vdash \mathtt{this} : (\mathtt{C}, \mathtt{G}), \varepsilon \quad \Gamma \vdash \mathtt{t} : (\mathtt{Fut}(\mathtt{C}), \mathtt{G}'), \varepsilon}{\Gamma \vdash \mathtt{t.get} : (\mathtt{C}, \mathtt{G}'), (\mathtt{G}, \mathtt{G}')} \qquad \text{(T-GetR)}\ \frac{\Gamma \vdash \mathtt{e} : (\mathtt{Fut}(\mathtt{C}), \mathtt{G}), \varepsilon \quad \mathtt{e} \neq \mathtt{t}}{\Gamma \vdash \mathtt{e.get} : (\mathtt{C}, \mathtt{G}), \varepsilon}$$

Contractually correct method declaration and class declaration:

(T-METHOD)

$$\frac{\begin{array}{c} \mathit{mtype}(\mathtt{m},\mathtt{C}) = \bar{\mathtt{C}}' \to \mathtt{C}' \qquad \mathit{mbody}(\mathtt{m},\mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e} \\ \bar{\mathtt{G}},\mathtt{G}\ \text{fresh} \quad \Gamma + \bar{\mathtt{x}} : (\bar{\mathtt{C}}',\bar{\mathtt{G}}) + \mathtt{this} : (\mathtt{C},\mathtt{G}) \vdash \mathtt{e} \ : \ (\mathtt{T}',\mathtt{G}'),\gamma \quad \mathtt{T}' <: \mathtt{C}' \\ \Gamma(\mathtt{C}.\mathtt{m}) =^{\alpha} \mathtt{G}(\bar{\mathtt{G}}) \to \mathtt{G}' \end{array}}{\Gamma;\mathtt{C} \vdash \ \mathtt{C}'\ \mathtt{m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} : \mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'}$$

Contractually correct class and program:

(T-CLASS)

$$\frac{\Gamma;\mathtt{C} \vdash \bar{\mathtt{M}} : \bar{\mathcal{G}}}{\Gamma \vdash \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \ \bar{\mathtt{M}}\} : \{\mathit{mname}(\bar{\mathtt{M}}) \mapsto \bar{\mathcal{G}}\}}$$

(T-PROGRAM)

$$\frac{\begin{array}{c} \mathtt{C} \in \mathit{dom}(\mathrm{CT})\ \text{ implies }\ \Gamma \vdash \mathrm{CT}(\mathtt{C}) : \mathrm{CCT}(\mathtt{C}) \\ \mathrm{CCT}\ \textit{is}\ \mathrm{CT}\ \textit{consistent} \\ \mathtt{G}\ \textit{fresh} \quad \Gamma + \mathtt{this} : (\mathtt{Object},\mathtt{G}) \vdash \mathtt{e} : (\mathtt{T},\mathtt{G}),\gamma \end{array}}{\vdash (\mathrm{CT},\mathtt{e}) : \mathrm{CCT},(\mathtt{T},\mathtt{G}),\gamma}$$

Table 5. Contract rules for method declarations and class declarations

(T-CONFIGURATION)

$$\frac{\begin{array}{c} \mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}} \\ \mathsf{H}(\mathtt{o}) = (\mathtt{C},\mathtt{G},[\bar{\mathtt{f}} : \bar{\mathtt{v}}]) \quad \text{implies} \quad \Gamma \vdash \mathtt{o} : (\mathtt{C},\mathtt{G}),\varepsilon \text{ and } \Gamma \vdash \bar{\mathtt{v}} : \bar{\mathtt{C}}' \text{ and } \bar{\mathtt{C}}' <: \bar{\mathtt{C}} \\ \mathtt{t} :_{\mathtt{o}} \mathtt{e} \in \mathsf{S} \quad \text{implies} \quad \Gamma \vdash \mathtt{t} : (\mathtt{Fut}(\mathtt{D}),\mathtt{G}'),\varepsilon \text{ and } \Gamma \vdash \mathtt{e} : (\mathtt{D},\mathtt{G}'),\gamma \text{ and } \mathtt{o} \in \mathit{dom}(\mathsf{H}) \end{array}}{\Gamma \vdash (\mathsf{H} \Vdash \mathsf{S})}$$

Rule (T-TASK) define contract correctness of runtime expressions as `t.get`. The rule uses contracts extended with terms $(\mathtt{G},\mathtt{G}').\gamma$. While rule (T-GETR) deals with the expression `t.await.get`. It is worth to notice the absence of rules for the runtime expression `t.await`. In fact, the judgment of this expression follows by (T-AWAIT) and the definition of $\varepsilon \Downarrow \mathtt{await}$.

Theorem 1 (Subject reduction).

1. *If* $\vdash (\mathrm{CT},\mathtt{e}) : (\mathrm{CCT},\gamma)$ *then the initial configuration of* $(\mathrm{CT},\mathtt{e})$ *is contractually correct. Namely, there is* Γ *such that* $\Gamma \vdash (\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{e}[^{\mathtt{o}}/_{\mathtt{this}}])$*, where* $\Gamma = \mathtt{o} \mapsto (\mathtt{Object},\mathtt{G}), \mathtt{t} \mapsto (\mathtt{Fut}(\mathtt{C}),\mathtt{G})$ *and* $\mathsf{H} = \mathtt{o} \mapsto (\mathtt{Object},\mathtt{G},[\,]), \mathtt{G} \mapsto \top$.
2. *Let* $\Gamma \vdash (\mathsf{H} \Vdash \mathsf{S})$ *and* $\mathsf{H} \Vdash \mathsf{S} \longrightarrow \mathsf{H}' \Vdash \mathsf{S}'$. *Then there is* Γ' *such that* $\Gamma' \vdash (\mathsf{H}' \Vdash \mathsf{S}')$.

5 Deadlock analysis in FJg

The contract system in Tables 4 and 5 does not convey any information about deadlocks: it only associates contracts to expressions (and methods). The point is that contracts retain the necessary informations about deadlocks and the analysis may be safely reduced to them, overlooking all the other details. We begin with the formal definition of a deadlock.

Definition 1. *A configuration* $\mathsf{H} \Vdash \mathsf{S}$ *is* deadlocked *if there are* $\mathtt{t}_i$, $\mathtt{o}_i$, E_i, *and* e_i, *with* $1 \le i \le n+k$, *such that* $n \ge 1$ *and*

- *every* $1 \le i \le n$ *is* $\mathtt{t}_i :^{\top}_{\mathtt{o}_i} \mathsf{E}_i[\mathtt{t}_{\ell_i}.\mathtt{get}]$ *with* $\ell_i \in 1..n+k$ *and*
- *every* $n{+}1 \le j \le n{+}k$ *is* $\mathtt{t}_j :^{\bot}_{\mathtt{o}_j} \mathsf{e}_j$ *with* $\mathit{group}(\mathsf{H}, \mathtt{o}_j) \in \{\mathit{group}(\mathsf{H}, \mathtt{o}_1), \cdots, \mathit{group}(\mathsf{H}, \mathtt{o}_n)\}$.

A configuration $\mathsf{H} \Vdash \mathsf{S}$ *is* deadlock-free *if, for every* $\mathsf{H} \Vdash \mathsf{S} \longrightarrow^* \mathsf{H}' \Vdash \mathsf{S}'$, $\mathsf{H}' \Vdash \mathsf{S}'$ *is not deadlocked. A program* (CT, e) *is deadlock-free if its initial configuration is deadlock-free.*

It is easy to verify that the programs discussed in Section 3 are not deadlock-free. We observe that a configuration may have a blocked task without retaining any deadlock. This is the case of the process $\mathtt{C.m}^{\mathtt{g}}\ \mathtt{G}()$, where $\mathtt{C.m}\ :\ \mathtt{G}()\{\mathtt{C.m}^{\mathtt{g}}(\mathtt{G}')\}\ \mathtt{G}$, that produces an infinite chain of tasks $\mathtt{t}_i :^{\top}_{\mathtt{o}_i} \mathtt{t}_{i+1}.\mathtt{get}$. (The following `dla` algorithm will reject this contract.)

We say that a configuration $\mathsf{H} \Vdash \mathsf{S}$ has a *group-dependency* $(\mathtt{G}, \mathtt{G}')$ if S contains either the tasks $\mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{t}'.\mathtt{get}], \mathtt{t}' :_{\mathtt{o}'} \mathsf{e}$, with $\mathtt{t}'$ retaining or not its group lock, or the tasks $\mathtt{t} :^{\bot}_{\mathtt{o}} \mathsf{e}, \mathtt{t}' :^{\top}_{\mathtt{o}'} \mathsf{E}[\mathtt{t}''.\mathtt{get}]$ (in both cases e is not a value) and $\mathtt{G} = \mathit{group}(\mathsf{H}, \mathtt{o})$ and $\mathtt{G}' = \mathit{group}(\mathsf{H}, \mathtt{o}')$. A configuration contains a *group-circularity* if the transitive closure of its group-dependencies has a pair $(\mathtt{G}, \mathtt{G})$. The following statement asserts that a group-circularity signals the presence of a sequence of tasks mutually waiting for the release of the group lock of the other.

Proposition 1. *A configuration is deadlocked if and only if it has a* group-circularity.

In the following, sets of dependencies will be noted $\boldsymbol{G}, \boldsymbol{G}', \cdots$. Sequences $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$ are also used and shortened into $\overline{\boldsymbol{G}}$. Let $\boldsymbol{G} \cup (\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n)$ be $\boldsymbol{G} \cup \boldsymbol{G}_1; \cdots; \boldsymbol{G} \cup \boldsymbol{G}_n$. A set $\boldsymbol{G}$ is *not circular*, written $\boldsymbol{G}$: `not-circular`, if the transitive closure of $\boldsymbol{G}$ does not contain any pair $(\mathtt{G}, \mathtt{G})$. The definition of being not circular is extended to sequences $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$, written $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$: `not-circular`, by constraining every $\boldsymbol{G}_i$ to be not circular.

Dependencies between group names are extracted from contracts by the algorithm `dla` defined in Table 6. This algorithm takes an *abstract class contract table* Δ_{CCT}, a group name G and a contract γ and returns a sequence $\overline{\boldsymbol{G}}$. The abstract class contract table Δ_{CCT} takes a pair class name C/method name m, written C.m, and returns an abstract method contract $(\mathtt{G}, \bar{\mathtt{G}})\boldsymbol{G}$. The map Δ_{CCT} is *the least one* such that

$$\Delta_{\mathrm{CCT}}(\mathtt{C.m}) = (\mathtt{G}, \bar{\mathtt{G}}) \bigcup_{i \in 1..n} \boldsymbol{G}_i \quad \text{if and only if} \quad \begin{array}{l} \mathrm{CCT}(\mathtt{C})(\mathtt{m}) = \mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}' \\ \text{and } \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma) = \boldsymbol{G}_1; \cdots; \boldsymbol{G}_n \end{array}$$

We notice that Δ_{CCT} is well-defined because: (i) group names in CCT are finitely many; (ii) `dla` never introduces new group names; (iii) for every C.m, the element $\Delta_{\mathrm{CCT}}(\mathtt{C.m})$ is a finite lattice where elements have shape $(\mathtt{G}, \bar{\mathtt{G}})\boldsymbol{G}$ and where the greatest set $\boldsymbol{G}$ is the cartesian product of group names in CCT. Additionally, in order to augment the precision of Δ_{CCT}, we assume that CCT satisfies the constraint that, for every C.m and D.n such that $\mathtt{C.m} \neq \mathtt{D.n}$, $\mathrm{CCT}(\mathtt{C})(\mathtt{m})$ and $\mathrm{CCT}(\mathtt{D})(\mathtt{n})$ have no group name in common (both bound and free). (When this is not the case, sets in the codomain of Δ_{CCT} are smaller, thus manifesting more circularities.)

$$\texttt{dla}(\Delta_{\text{CCT}}, \texttt{G}, \varepsilon) = \emptyset \qquad \frac{\Delta_{\text{CCT}}(\texttt{C.m}) = (\texttt{G}'';\bar{\texttt{G}}'')\boldsymbol{G} \quad \texttt{dla}(\Delta_{\text{CCT}}, \texttt{G}, \gamma') = \overline{\boldsymbol{G}}}{\texttt{dla}(\Delta_{\text{CCT}}, \texttt{G}, \texttt{C.m}\ \texttt{G}'(\bar{\texttt{G}}');\gamma') = \boldsymbol{G}[^{\texttt{G}';\bar{\texttt{G}}'}/_{\texttt{G}'';\bar{\texttt{G}}''}] \cup \overline{\boldsymbol{G}}}$$

$$\frac{\Delta_{\text{CCT}}(\texttt{C.m}) = (\texttt{G}'';\bar{\texttt{G}}'')\boldsymbol{G} \quad \texttt{dla}(\Delta_{\text{CCT}}, \texttt{G}, \gamma') = \overline{\boldsymbol{G}}}{\texttt{dla}(\Delta_{\text{CCT}}, \texttt{G}, \texttt{C.m}^{\texttt{g}}\ \texttt{G}'(\bar{\texttt{G}}');\gamma') = (\{(\texttt{G},\texttt{G}')\} \cup \boldsymbol{G}[^{\texttt{G}';\bar{\texttt{G}}'}/_{\texttt{G}'';\bar{\texttt{G}}''}]);\overline{\boldsymbol{G}}}$$

$$\frac{\Delta_{\text{CCT}}(\texttt{C.m}) = (\texttt{G}'';\bar{\texttt{G}}'')\boldsymbol{G} \quad \texttt{dla}(\Delta_{\text{CCT}}, \texttt{G}, \gamma') = \overline{\boldsymbol{G}}}{\texttt{dla}(\Delta_{\text{CCT}}, \texttt{G}, \texttt{C.m}^{\texttt{a}}\ \texttt{G}'(\bar{\texttt{G}}');\gamma') = \boldsymbol{G}[^{\texttt{G}';\bar{\texttt{G}}'}/_{\texttt{G}'';\bar{\texttt{G}}''}];\overline{\boldsymbol{G}}}$$

Table 6. The algorithm `dla`

Let us comment the rules of Table 6. The second rule of `dla` accounts for method invocations $\texttt{C.m}\ \texttt{G}'(\bar{\texttt{G}}');\gamma'$. Since the code of `C.m` will run asynchronously with respect to the continuation γ', *i.e.* it may be executed at any stage of γ', the rule adds the pairs of `C.m` (stored in $\Delta_{\text{CCT}}(\texttt{C.m})$) to every set of the sequence corresponding to γ'. The third rule of `dla` accounts for method invocations followed by `get` $\texttt{C.m}^{\texttt{g}}\ \texttt{G}'(\bar{\texttt{G}}');\gamma'$. Since the code of `C.m` will run *before* the continuation γ', the rule prefixes the sequence corresponding to γ' with the pairs of `C.m` extended with $(\texttt{G},\texttt{G}')$, where `G` is the group of the caller and `G'` is the group of the called method. The rule for method invocations followed by `await` is similar to the previous one, except that no pair is added because the `await` operation releases the group lock of the caller.

A program $(\text{CT}, \texttt{e})$ is deadlock free if $\vdash (\text{CT}, \texttt{e}) : \text{CCT}, (\texttt{T}, \texttt{G}), \gamma$ and $\texttt{dla}(\Delta_{\text{CCT}}, \texttt{G}, \gamma)$: `not-circular`, where `G` is a fresh group name, that is `G` does not clash with group names in CCT (group names in γ are either `G` or fresh as well – see Table 4).

Example 2. Let `C` and `D` be the classes of Table 1 and `D'` be the class in Section 3. We derive the following contract class table CCT and abstract contract class table Δ_{CCT}:

`C.m` $\mapsto$ `G(){ ε } G'`	`C.m` $\mapsto$ `(G)∅`
`D.n` $\mapsto$ `E(E'){D.m`$^{\texttt{g}}$`:E'()} E''`	`D.n` $\mapsto$ `(E,E'){ (E,E') }`
`D.m` $\mapsto$ `F(){ ε } F'`	`D.m` $\mapsto$ `(F)∅`
`D'.n` $\mapsto$ `H(H',H''){D'.p:H'(H''); D'.p:H''(H')} H`	`D'.n` $\mapsto$ `(H,H',H''){(H',H''),(H'',H')}`
`D'.p` $\mapsto$ `I(I'){ D'.m`$^{\texttt{g}}$`: I'() } I''`	`D'.p` $\mapsto$ `(I,I'){ (I,I') }`
`D'.m` $\mapsto$ `L() { ε } L'`	`D'.m` $\mapsto$ `(L)∅`

Now consider the expressions `(newg D())!n(newg D()).get` and `(newg D())!n(new D()).get` of Section 2, which have contracts `D.n`$^{\texttt{g}}$`:`$\text{L}_2(\text{L}_3)$ and `D.n`$^{\texttt{g}}$`:`$\text{L}_2(\text{L}_1)$, respectively, with L_1 being the group of `this`. We obtain $\texttt{dla}(\Delta_{\text{CCT}}, \text{L}_1, \texttt{D.n}^{\texttt{g}} : \text{L}_2(\text{L}_3)) = \{(\text{L}_2,\text{L}_3),(\text{L}_1,\text{L}_2)\}$ and $\texttt{dla}(\Delta_{\text{CCT}}, \text{L}_1, \texttt{D.n}^{\texttt{g}} : \text{L}_2(\text{L}')) = \{(\text{L}_2,\text{L}_1),(\text{L}_1,\text{L}_2)\}$ where the first set of dependencies has no group-circularity – therefore `(newg D())!n(newg D()).get` is deadlock-free – while the second has a group-circularity – `(newg D())!n(new D()).get` – may manifest a deadlock, and indeed it does.

Next consider the expression `(newg D'())!n(newg D'(),new D'()).get` of Section 3, which has contract `D'.n`$^{\texttt{g}}$`:L''(L''',L')`, being `L'` the group of `this`. We obtain

$$\texttt{dla}(\Delta_{\text{CCT}}, \text{L}', \texttt{(newg D'())!n(newg D'(),new D'()).get}) = \{(\text{L}''',\text{L}'),(\text{L}',\text{L}'''),(\text{L}',\text{L}'')\}$$

where the set of dependencies manifests circularities. In fact, in Section 3, we observed that the above expression may manifest a deadlock.

The `dla` algorithm is correct, that is, if its result contains a group-circularity, then the evaluation of the analyzed expression may manifest a deadlock (*vice versa*, if there is no group-circularity then no deadlock will be ever manifested).

Theorem 2. *If* $\vdash$ (CT, e) : (CCT, γ) *and* dla(Δ_{CCT}, G, γ) *is not circular then* (CT, e) *is deadlock-free.*

The algorithm `dla` may be strengthened in several ways. Let us discuss this issue with a couple of examples. Let C′ be the class

```
class C′ { C′ m(C′ b, C′ c){ return b!n(c).get ; c!n(b).get ;}
           C′ n(D′ c){ return (c!p).get ;}
           C′ p() { return new C′() ;} }
```

and let CCT be its contract class table:

$$\begin{aligned}
&\mathtt{C'.m} \mapsto \mathtt{G(G',G'')\{C'.n^g{:}G'(G'')\ ;\ C'.n^g{:}G''(G')\}\ G'}\\
&\mathtt{C'.n} \mapsto \mathtt{F(F')\{C'.p^g{:}F'()\}\ F'}\\
&\mathtt{C'.p} \mapsto \mathtt{E()\{}\epsilon\mathtt{\}\ E}
\end{aligned}$$

The reader may verify that the expression `(new C′()).m1(newg C′(), newg C′())` never deadlocks. However, since Δ_{CCT}(C′.m) = (G, G′, G″){(G, G′), (G′, G″), (G, G″), (G″, G′)}, the algorithm `dla` wrongly returns a circular set of dependencies. This problem is due to the fact that Δ_{CCT} melds the group dependencies of different time points into a single set. Had we preserved the temporal separation, that is {(G, G′), (G′, G″)}; {(G, G″), (G″, G′)}, no group-circularity should have been manifested.

The second problem is due to the fact that free group names in method contracts should be renamed each time the method is invoked (with fresh group names) because two invocations of a same method create different group names. On the contrary, the algorithm `dla` always uses the same (free) name. This oversimplification gives a wrong result in this case. Let C″ be `class C″ { C″ m(){ return (newg C″()).m().get ;} }` (with CCT(C″.m) = G(){C′.m^g : G′()}G″) and consider the expression `(newg C″()).m().get`. The evaluation of this expression never manifests a deadlock, however its contract is C″.m^g : F() and the algorithm `dla` will return the set {(G, F), (F, G′), (G′, G′), }, which has a group-circularity. In the conclusions we will discuss the techniques for reducing these errors.

6 Related works

The notion of grouping objects dates back at least to the mid 80'es with the works of Yonezawa on the language ABCL/1 [8, 21]. Since then, several languages have a notion of group for structuring systems, such as Eiffel// [3], Hybrid [17], and ASP [4]. A library for object groups has also been defined for CORBA [7]. In these proposals, a single task is responsible for executing the code inside a group. Therefore it is difficult to model behaviours such as waiting for messages without blocking the group for other activities.

Our FJg calculus is inspired to the language Creol that proposes object groups, called *JCoBoxes*, with multiple cooperatively scheduled tasks [10]. In particular FJg is a subcalculus of JCoBox^c in [19], where the emphasis was the definition of the semantics and the type system of the calculus and the implementation in Java.

The proposals for static analyses of deadlocks are largely based on types (see for instance [11, 20] and, for object-oriented programs [1]). In these papers, a type system is defined that computes a partial order of the locks in a program and a subject reduction theorem demonstrates that tasks follow this order. On the contrary, our technique does not computes any ordering of locks, thus being more flexible: a computation may acquire two locks in different order at different stages, thus being correct in our case, but incorrect with the other techniques. A further difference with the above works is that we use contracts that are terms in simple (= with finite states) process algebras [12]. The use of simple process algebras to describe (communication or synchronization) protocols is not new. This is the case of the exchange patterns in SSDL [18], which are based on CSP [2] and the pi-calculus [14], or of the behavioral types in [16] and [5], which use CCS [13]. We expect that finite state abstract descriptions of behaviors can support techniques that are more powerful than the one used in this contribution.

7 Conclusions

We have developed a technique for the deadlock analysis of object groups that is based on abstract descriptions of methods behaviours.

This study can be extended in several directions. One direction is the total coverage of the full language FJg. This is possible by using *group records* $\Theta, \Theta' = \mathtt{G}[\mathtt{f}_1 : \Theta_1, \cdots, \mathtt{f}_k : \Theta_k]$ instead of simple group names. Then contracts such as $\mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})$ become $\mathtt{C.m} : \Theta(\Theta_1, \cdots, \Theta_n)$ and the rule (T-FIELD) is refined into

$$\frac{\Gamma \vdash \mathtt{this} : (\mathtt{C}, \mathtt{G}[\bar{\mathtt{f}} : \overline{\Theta}]), \varepsilon \quad \mathtt{D\ f} \in \mathit{fields}(\mathtt{C}) \quad \mathtt{f} : \Theta' \in \bar{\mathtt{f}} : \overline{\Theta}}{\Gamma \vdash \mathtt{this.f} : (\mathtt{D}, \Theta'), \varepsilon} \text{(T-FIELD-REF)}$$

The overall effect of this extension is to hinder the notation used in the paper, without conveying any interesting difficulty (for this reason we have restricted our analysis to a sublanguage). We claim that every statement for plain FJg in this paper also hold for full FJg.

A different direction of research is the study of techniques for augmenting the accuracy of the algorithm dla, which is imprecise at the moment. The intent is to use finite state automata with name creation, such as those in [15], and modeling method contracts in terms of finite automata and study deadlocks in sets of these automata.

Other directions address extensions of the language FJg. One of these extensions is the removal of the constraint that future types cannot be used by programmers in FJg. Future types augment the expressivity of the language. For example it is possible to synchroniza several tasks and define *livelocks*:

```
class C { f: Fut(C) ; C m() { return this.f = this!n() ; new C() ;}
                      C n() { return this.f.get ; new C() ;}   }
```

Another extension is about re-entrant method invocations (usually used for tail recursions), which are synchronous invocations. Such extension requires revisions of semantics rules, of the contract rules in Table 4, and of the `dla` algorithm.

References

1. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28, 2006.
2. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31:560–599, 1984.
3. D. Caromel. Towards a method of object-oriented concurrent programming. *Commun. ACM*, 36(9):90–102, 1993.
4. D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *Proc. POPL'04*, pages 123–134. ACM, 2004.
5. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. *SIGPLAN Not.*, 37(1):45–57, 2002.
6. F. de Boer, D. Clarke, and E. Johnsen. A complete guide to the future. In *Progr. Lang. and Systems*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
7. P. Felber and R. Guerraoui. Programming with object groups in corba. *IEEE Concurrency*, 8:48–58, 2000.
8. Y. Honda and A. Yonezawa. Debugging concurrent systems based on object groups. In *Proc. ECOOP'88*, volume 322 of *LNCS*, pages 267–282. Springer, 1988.
9. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23:396–450, 2001.
10. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58, 2007.
11. N. Kobayashi. A new type system for deadlock-free processes. In *Proc. CONCUR 2006*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.
12. C. Laneve and L. Padovani. The *must* preorder revisited. In *Proc. CONCUR 2007*, volume 4703 of *LNCS*, pages 212–225. Springer, 2007.
13. R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.
14. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Inf. and Comput.*, 100:41–77, 1992.
15. U. Montanari and M. Pistore. History-dependent automata: An introduction. In *Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 3465 of *LNCS*, pages 1–28. Springer, 2005.
16. H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proc. POPL '94*, pages 84–97. ACM, 1994.
17. O. Nierstrasz. Active objects in Hybrid. In *Proc. OOPSLA'87*, pages 243–253, 1987.
18. S. Parastatidis and J. Webber. *MEP SSDL Protocol Framework*, Apr. 2005. `http://ssdl.org`.
19. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. ECOOP'10*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.
20. V. T. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *Proc. PLACES'09*, volume 17 of *EPTCS*, pages 95–109, 2009.
21. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proc. OOPSLA'86*, pages 258–268, 1986.