Fundamenta Informaticae LX (2004) 1–14
IOS Press

Abstract Interpretation against Races

Roberto Barbuti

Stefano Cataudella

Luca Tesei

Dipartimento di Informatica – Università di Pisa
Via F. Buonarroti, 2 56127 Pisa - Italy
email: {barbuti,cataudel,tesei}@di.unipi.it

Abstract. In this paper we investigate the use of abstract interpretation techniques for statically preventing race conditions. To this purpose we enrich the concurrent object calculus **conc**ς by annotating terms with the set of "locks" owned at any time. We use an abstract form of the object calculus to check the absence of race conditions. We show that abstract interpretation is more flexible than type analyses, and it allows to certify as "race free" a larger class of programs.

1. Introduction

When programming with multithread languages, insidious errors, usually denoted as *race conditions*, can arise [2]. A race condition occurs when two processes access a shared resource simultaneously, often provoking an incorrect and unexpected behavior.

A usual method to avoid such conditions is to provide each resource with a *lock*. A process must acquire the lock on a resource before using it, and a locked resource cannot be used by other processes. Concurrent object oriented languages are often based on this approach: resources are embedded in an object and a lock is attached to the object. Java methods adopt this strategy: a method or a block can be declared `synchronized`. A lock is associated to every object which has a synchronized code [12].

Despite this synchronization method, it is not unusual to write multithreaded programs which access objects without acquiring locks on them, thus creating error conditions. The non-acquisition can be originated by different reasons, the most common being mistakes or the conviction that an object is accessed by a single thread.

Many works have been devoted to the static analysis of programs to find possible race conditions. Such methods are essentially based on type analysis [10, 11, 4, 5]: a program is well-typed iff an object is accessed only when a suitable set of locks, corresponding to a policy of synchronization, is acquired. Obviously, the type correctness can be checked statically by applying a set of typing rules.

The properties of concurrent object languages, including race-free conditions, can be suitably studied on languages with a few basic primitives which serve as a foundation for object oriented languages. To this purpose many calculi were introduced in the past [3, 8, 9, 14]. In this paper we refer to the imperative *object calculus*, **imp**ς, which was introduced in [1], and extended to a concurrent one, **conc**ς, in[11, 13].

In [11] a type analysis checks that an object, in a **conc**ς term, is accessed by a process only if a lock on that object is owned by the process itself.

All the mentioned type analyses check a program under correctness assumptions which are somewhat rigid. For example the above rule could be relaxed when no concurrent accesses to the same object can be done during the execution of processes. A method for a less rigid analysis can be based on *abstract interpretation* [6, 7]. Abstract interpretation executes the program in an abstract (approximated) way to statically check dynamic properties, and, in many cases, it can be more precise than type analyses.

To apply abstract interpretation techniques we define a suitable untyped object calculus, based on the one defined in [11, 13], which we call **aconc**ς. It is imperative and concurrent, and it embodies, in its terms, the knowledge on the locks owned at any time. On the basis of this information the semantic definition can be aware, at the time of an access to an object, whether the lock to that object is owned. Thus an analysis can be performed to check that processes accessing an object own the right locks, or that no concurrent accesses to an object are performed at the same time.

The plan of the paper is the following: Section 2 introduces the imperative concurrent object calculus **aconc**ς. Section 3 defines the abstract interpretation of the calculus and shows how the abstract semantics can be used to discover possible race conditions. Finally, Section 4 concludes.

2. The object calculus aconcς

This section describes a concurrent object calculus which is based on the calculi in [11, 13].

2.1. Syntax

The Table 1 defines the following syntactic categories: results, denotations, terms and statements.

Results are defined as *variables*, *numbers* or *references* to objects.

A *denotation* $[\ell_i = \varsigma(x_i)t_i\,^{i\in 1..n}]^l$ describes an object with a collections of methods with names ℓ_i. Note that also *instance variables* are considered methods (like in [1]) with a constant definition. The self parameter x_i of each method corresponds to the reference

u	$::=$		<u>results</u>
		x	variable
		p	location
		n	integer number
d	$::=$		<u>denotations</u>
		$[\ell_i = \varsigma(x_i)t_i{}^{i\in 1\ldots n}]^l$	object
l	$::=$		<u>lock states</u>
		$\circ$	unlocked
		$\bullet$	locked
s, t	$::=$		<u>terms</u>
		u	result
		$\nu p.t$	restriction
		$p \mapsto d$	reference
		$p.\ell$	method invocation
		$p.\ell \Leftarrow \varsigma(x)t$	method update
		lock p in t	lock acquisition
		let $x = s$ in t	let
		$s \upharpoonright t$	parallel composition
		e	integer expression
		if e then s else t	if
a, b	$::=$		<u>statements</u>
		u	result
		$\nu p.a$	restriction
		$p \mapsto d$	reference
		locked p in a	lock acquired
		let $x = a$ in b	let
		$[\![t]\!]^L$	lock environment
		$a \upharpoonright b$	parallel composition
		if e then a else b	if

Table 1. Syntax of **aconc**ς

`this` used inside the object definitions in object oriented programming. In addition the object has a *lock state* which can be either $\circ$, meaning that the object is not locked by any process, or $\bullet$, meaning that a process owns a lock on it.

A *term* is a *result*, a *restriction*, a *reference*, a *method invocation*, a *method update*, a *lock acquisition*, a *let expression*, a *parallel composition* of terms, an *integer expression* or an *if*. The reference term says that an object is identified by the reference, p, to it. The reference p is introduced by a restriction, $\nu p.t$, which binds the reference p with scope t. The method invocation and method update are the usual ones. A lock acquisition is a term which describes an execution after having acquired a lock to an object. The parallel composition of terms, $s \upharpoonright t$, indicates the parallel execution of s and t. The result of the construct is the result of t; s is evaluated only for effect. Integer expressions, let terms and if terms are usual.

Finally, the syntax of statements, based on terms, gives the structure of programs. A result is a statement the meaning of which is the result itself. Analogously to terms, the *reference statement*, $p \mapsto d$, states that the object denoted by d is pointed by p. A *lock acquired statement*, locked p in a, says that a lock on p is owned and that the statement a can be executed under this lock. A *lock environment*, $[\![t]\!]^L$, where L is a set of references, indicates that the term t is executed while owning a lock on all the objects the references of which belong to the set L. The *let statement* can be used to implement sequence of statements. let $x = a$ in b corresponds to $a; b$ if x does not occur in b. Finally, the parallel composition of statements and the if statement are analogous to the ones of terms.

2.2. Semantics of aconcς

The semantics of **aconc**ς is given in terms of a structural congruence and a set of reduction rules. Structural congruence allows to syntactically transform statements in order to apply the reduction rules. The application of a reduction rule corresponds to a computation step.

Both the structural congruence and reduction rules are given in terms of *evaluations contexts*, like in [11]. An evaluation context $\mathcal{E}[]$ is a statement with the hole $[]$. The hole can be filled by a statement, thus $\mathcal{E}[a]$ means the evaluation context $\mathcal{E}[]$ with the hole filled by the statement a.

The possible holes in a statement are given in Table 2, where the syntax of contexts is given. $[\cdot]$ means that the context can be the whole statement.

The structural congruence rules are given in Table 3.

The first rule says that the left statement in a parallel composition can be inserted or extracted from an evaluation context. The reason of such a congruence is that the left statement is only evaluated for effect. For the same reason the order of statements on the left of the rightmost one in a parallel composition is irrelevant (second rule). The third rule is slightly more complex. When a parallel composition of terms is executed under the locks in the set L, it is congruent with the execution of the left term under the empty set of locks and the execution of the right one with the whole set of locks L. This because the parallel composition corresponds to the `fork` statement in object oriented languages: the process started in parallel by a `fork` inherits no locks from the father process (like the term on the

$$
\begin{array}{lll}
\mathcal{E} & ::= & \\
 & & [\cdot] \\
 & & \mathcal{E} \Rsh b \\
 & & a \Rsh \mathcal{E} \\
 & & \text{locked } p \text{ in } \mathcal{E} \\
 & & \text{let } x = \mathcal{E} \text{ in } b \\
 & & \text{if } \mathcal{E} \text{ then } a \text{ else } b \\
 & & \nu p.\mathcal{E}
\end{array}
$$

Table 2. Reduction contexts

$$
\begin{array}{l}
a \Rsh \mathcal{E}[b] \;\equiv\; \mathcal{E}[a \Rsh b] \\
a \Rsh b \Rsh c \;\equiv\; b \Rsh a \Rsh c \\
[\![s \Rsh t]\!]^L \;\equiv\; [\![s]\!]^{\emptyset} \Rsh [\![t]\!]^L \\
[\![\text{let } x = s \text{ in } t]\!]^L \equiv \text{let } x = [\![s]\!]^L \text{ in } [\![t]\!]^L \\
[\![\text{if } e \text{ then } s \text{ else } t]\!]^L \equiv \text{if } [\![e]\!]^L \text{ then } [\![s]\!]^L \text{ else } [\![t]\!]^L \\
[\![\nu p.t]\!]^L \equiv \nu p.[\![t]\!]^L \\
[\![u]\!]^L \equiv u \\
[\![p \mapsto d]\!]^L \equiv p \mapsto d
\end{array}
$$

Table 3. Structural congruence rules

left of $\Rsh$ in our case). Executing the sequencing statement, $[\![\text{let } x = a \text{ in } b]\!]^L$, under the set of locks L corresponds to execute a followed by b under the same set. All the components of an if statement are executed under the lock set of the whole statement. The set of locks on a restriction can be transferred inside the body of the restriction. Finally, either a result or a reference statement are not affected by a set of locks.

The reduction rules are given in Table 4.

(Red invoke) requires a statement, in parallel, which defines the reference to the object the method of which is called; then the method is invoked with the instantiation of the self parameter to the reference to the object itself. (Red update) updates a method in an object; the result of the statement is the reference to the modified object (as in [11, 13]). Note that, both in method invocation and update, we can know, by inspecting the set L, whether the statement owns a lock on the object. This information cannot be inferred in **conc**ς. (Red lock) acquires a lock to an object and (Red unlock) unlocks the object when a result is computed. (Red let) performs a substitution for the variable x when a result u is reached. (Red if0) and (red ifn) reduce the if statement in the standard way. Finally (Red context) says that the reduction rules can be applied to any evaluation context. Note that no rule is given for integer expressions. We assume for them the standard reductions.

3. Abstract interpretation of aconcς

In this section we define an abstract interpretation of **aconc**ς. Such an interpretation is given with respect to an abstract calculus which approximates the concrete one. In particular, given a statement in the abstract calculus, the set of possible statements which can be generated by reduction and structural congruence, starting from it, is finite. This allows to construct a finite transition system, the states of which are statements, which can be finitely analyzed to establish properties of it.

$$\frac{d = [\ell_i = \varsigma(x_i)t_i{}^{i\in(1\ldots n)}]^l \quad j \in (1,\ldots,n)}{p \mapsto d \Rsh [\![p.\ell_j]\!]^L \longrightarrow p \mapsto d \Rsh [\![t_{j\{\{x_j \leftarrow p\}\}}]\!]^L}\text{(Red invoke)}$$

$$\frac{d = [\ell_i = \varsigma(x_i)t_i{}^{i\in(1\ldots n)}]^l \quad d' = [\ell_j = \varsigma(x)t, \ell_i = \varsigma(x_i)t_i{}^{i\in(1\ldots n)-\{j\}}]^l}{p \mapsto d \Rsh [\![p.\ell_j \Leftarrow \varsigma(x)t]\!]^L \longrightarrow p \mapsto d' \Rsh p}\text{(Red update)}$$

$$\frac{d = [\ell_i = \varsigma(x_i)t_i{}^{i\in(1\ldots n)}]^\circ \quad d' = [\ell_i = \varsigma(x_i)t_i{}^{i\in(1\ldots n)}]^\bullet}{p \mapsto d \Rsh [\![\text{lock } p \text{ in } t]\!]^L \longrightarrow p \mapsto d' \Rsh \text{locked } p \text{ in } [\![t]\!]^{L\cup\{p\}}}\text{(Red lock)}$$

$$\frac{d = [\ell_i = \varsigma(x_i)t_i{}^{i\in(1\ldots n)}]^\bullet \quad d' = [\ell_i = \varsigma(x_i)t_i{}^{i\in(1\ldots n)}]^\circ}{p \mapsto d \Rsh \text{locked } p \text{ in } u \longrightarrow p \mapsto d' \Rsh u}\text{(Red unlock)}$$

$$\frac{}{[\![\text{let } x = u \text{ in } t]\!]^L \longrightarrow [\![t_{\{\{x\leftarrow u\}\}}]\!]^L}\text{(Red let)}$$

$$\frac{}{\text{if } 0 \text{ then } a \text{ else } b \longrightarrow a}\text{(Red if0)}$$

$$\frac{n \neq 0}{\text{if } n \text{ then } a \text{ else } b \longrightarrow b}\text{(Red ifn)}$$

$$\frac{a \longrightarrow a'}{\mathcal{E}[a] \longrightarrow \mathcal{E}[a']}\text{(Red context)}$$

Table 4. Reduction rules

We call the abstract object calculus **aconc**$\varsigma^\sharp$. It syntax is given in Table 5.

$$
\begin{array}{lll}
u^\sharp & ::= & x \mid p \mid \odot \\
\\
d^\sharp & ::= & [\ell_i = \varsigma(x_i)t_i^\sharp \ {}^{i\in 1\ldots n}]^l \\
\\
l & ::= & \circ \mid \bullet \\
\\
s^\sharp, t^\sharp & ::= & u \mid \nu p.t^\sharp \mid p \mapsto d^\sharp \mid p.\ell \mid p.\ell \Leftarrow \varsigma(x)t^\sharp \mid \text{lock } p \text{ in } t^\sharp \\
 & & \text{let } x = s^\sharp \text{ in } t^\sharp \mid s^\sharp \upharpoonright t^\sharp \mid \ \odot \ \mid \text{if } \odot \text{ then } s^\sharp \text{ else } t^\sharp \\
\\
a^\sharp, b^\sharp & ::= & u \mid \nu p.a^\sharp \mid p \mapsto d^\sharp \mid \text{locked } p \text{ in } a^\sharp \\
 & & \text{let } x = a^\sharp \text{ in } b^\sharp \mid [\![t^\sharp]\!]^L \mid a^\sharp \upharpoonright b^\sharp \mid \text{if } \odot \text{ then } a^\sharp \text{ else } b^\sharp
\end{array}
$$

Table 5. Syntax of **aconc**$\varsigma^\sharp$

There are a few differences between the syntax of the concrete and the abstract calculus. In the abstract calculus, for the sake of finiteness, all the integer values and the integer expressions are collapsed to a unique value, denote by $\odot$.

Recall that we deal with a calculus, thus the concrete and abstract domains are the concrete and abstract syntax, respectively. The set of statements (either concrete or abstract) can be defined as lattices by adding to the flat set of them a top and a bottom element.

To formalize the abstract interpretation we define *abstraction functions*, α, and *concretization functions*, γ, between the concrete and the abstract domains. In particular we define an abstract function for each syntactic category, thus we define $\alpha_r : u \to u^\sharp$, $\alpha_d : d \to d^\sharp$, and so on.

The definition of α functions is given in Table 6 where, for the sake of readability, the argument of each function is an element of the concrete syntactic categories rather than a set. The abstraction of a set of concrete elements is defined, as usual, as the least upper bound of the abstractions of the single elements of the set. Note that our abstract domains are such that the least upper bound of any two syntactically different abstract elements is the top element.

The concretization functions, γ, are defined in Table 7. Note that γ functions produce sets of concrete syntactic objects. The concretization of the top element is the set of all concrete syntactic objects. A concretization of an abstract syntactic object $o^\sharp$ results in the set of concrete syntactic objects the abstraction of which is $o^\sharp$ itself.

The abstract semantics is given, analogously to the concrete one, by means of structural congruence and reduction rules. Because of their similarity to the concrete rules we redefine only the ones which differ. The other ones are identical, apart from the fact that the concrete syntactic categories should be substituted by the abstract ones.

$$
\begin{array}{lcl}
\alpha_r(x) & = & x \\
\alpha_r(p) & = & p \\
\alpha_r(n) & = & \odot \\
\\
\alpha_d([\ell_i = \varsigma(x_i)t_i{}^{i \in 1 \ldots n}]^l) & = & [\ell_i = \varsigma(x_i)\alpha_t(t_i)\ {}^{i \in 1 \ldots n}]^l \\
\\
\alpha_t(u) & = & \alpha_r(u) \\
\alpha_t(\nu p.t) & = & \nu p.\alpha_t(t) \\
\alpha_t(p \mapsto d) & = & p \mapsto \alpha_d(d) \\
\alpha_t(p.\ell) & = & p.\ell \\
\alpha_t(p.\ell \Leftarrow \varsigma(x)t) & = & p.\ell \Leftarrow \varsigma(x)\alpha_t(t) \\
\alpha_t(\text{let } x = s \text{ in } t) & = & \text{let } x = \alpha_t(s) \text{ in } \alpha_t(t) \\
\alpha_t(s \upharpoonright t) & = & \alpha_t(t) \upharpoonright \alpha_t(s) \\
\alpha_t(e) & = & \odot \\
\alpha_t(\text{if } e \text{ then } s \text{ else } t) & = & \text{if } \odot \text{ then } \alpha_t(s) \text{ else } \alpha_t(t) \\
\\
\alpha_s(u) & = & \alpha_r(u) \\
\alpha_s(\nu p.a) & = & \nu p.\alpha_s(a) \\
\alpha_s(p \mapsto d) & = & p \mapsto \alpha_d(d) \\
\alpha_s(\text{locked } p \text{ in } a) & = & \text{locked } p \text{ in } \alpha_s(a) \\
\alpha_s(\text{let } x = a \text{ in } b) & = & \text{let } x = \alpha_s(a) \text{ in } \alpha_s(b) \\
\alpha_s([\![t]\!]^L) & = & [\![\alpha_t(t)]\!]^L \\
\alpha_s(a \upharpoonright b) & = & \alpha_s(a) \upharpoonright \alpha_s(b) \\
\alpha_s(\text{if } e \text{ then } a \text{ else } b) & = & \text{if } \odot \text{ then } \alpha_s(a) \text{ else } \alpha_s(b)
\end{array}
$$

Table 6. Abstraction functions

$$
\begin{array}{lcl}
\gamma_r(x) & = & \{x\} \\
\gamma_r(p) & = & \{p\} \\
\gamma_r(\odot) & = & \{n \mid n \text{ is an integer number}\} \\
 & & \\
\gamma_d([\ell_i = \varsigma(x_i)t_i^\sharp\ ^{i \in 1 \ldots n}]^l) & = & \{[\ell_i = \varsigma(x_i)t_i\ ^{i \in 1 \ldots n}]^l \mid t_i \in \gamma_t(t_i^\sharp)\} \\
 & & \\
\gamma_t(u) & = & \gamma_r(u) \\
\gamma_t(\nu p.t^\sharp) & = & \{\nu p.t \mid t \in \gamma_t(t^\sharp)\} \\
\gamma_t(p \mapsto d^\sharp) & = & \{p \mapsto d \mid d \in \gamma_d(d^\sharp)\} \\
\gamma_t(p.\ell) & = & \{p.\ell\} \\
\gamma_t(p.\ell \Leftarrow \varsigma(x)t^\sharp) & = & \{p.\ell \Leftarrow \varsigma(x)t \mid t \in \gamma_t(t^\sharp)\} \\
\gamma_t(\text{let } x = s^\sharp \text{ in } t^\sharp) & = & \{\text{let } x = s \text{ in } t \mid s \in \gamma_t(s^\sharp), t \in \gamma_t(t^\sharp)\} \\
\gamma_t(s^\sharp \Rsh t^\sharp) & = & \{s \Rsh t \mid s \in \gamma_t(s^\sharp), t \in \gamma_t(t^\sharp)\} \\
\gamma_t(\odot) & = & \{e \mid e \text{ integer expression }\} \\
\gamma_t(\text{if } \odot \text{ then } s^\sharp \text{ else } t^\sharp) & = & \{\text{if } e \text{ then } s \text{ else } t \mid s \in \gamma_t(s^\sharp), t \in \gamma_t(t^\sharp), \\
 & & e \text{ integer expression }\} \\
 & & \\
\gamma_s(u) & = & \gamma_r(u) \\
\gamma_s(\nu p.a^\sharp) & = & \{\nu p.a \mid a \in \gamma_s(a^\sharp)\} \\
\gamma_s(p \mapsto d^\sharp) & = & \{p \mapsto d \mid d \in \gamma_d(d^\sharp)\} \\
\gamma_s(\text{locked } p \text{ in } a^\sharp) & = & \{\text{locked } p \text{ in } a \mid a \in \gamma_s(a^\sharp)\} \\
\gamma_s(\text{let } x = a^\sharp \text{ in } b^\sharp) & = & \{\text{let } x = a \text{ in } b \mid a \in \gamma_s(a^\sharp), b \in \gamma_s(b^\sharp)\} \\
\gamma_s(\llbracket t^\sharp \rrbracket^L) & = & \{\llbracket t \rrbracket^L \mid t \in \gamma_t(t^\sharp)\} \\
\gamma_s(a^\sharp \Rsh b^\sharp) & = & \{a \Rsh b \mid a \in \gamma_s(a^\sharp), b \in \gamma_s(b^\sharp)\} \\
\gamma_s(\text{if } \odot \text{ then } a^\sharp \text{ else } b^\sharp) & = & \{\text{if } e \text{ then } a \text{ else } b \mid a \in \gamma_s(a^\sharp), b \in \gamma_s(b^\sharp), \\
 & & e \text{ integer expression }\}
\end{array}
$$

Table 7. Concretization functions

Thus, structural congruence rules are still the ones in Table 3 (applied to the abstract syntax).

Abstract reduction rules are analogous to the ones defined in Table 4, the only differences being the definitions of the rules for the if statement. The abstract rules for this statement are given in Table 8.

$$\frac{}{\text{if } \odot \text{ then } a^\sharp \text{ else } b^\sharp \longrightarrow^\sharp a^\sharp}(\text{Red if}^{\sharp 1})$$

$$\frac{}{\text{if } \odot \text{ then } a^\sharp \text{ else } b^\sharp \longrightarrow^\sharp b^\sharp}(\text{Red if}^{\sharp 2})$$

Table 8. Abstract reduction rules

Let us note that, differently from the concrete rules, the abstract reduction of an if statement produces two different results.

We can state the correctness of the abstract interpretation by the following results.

First we have to show that α and γ form a Galois insertion between the concrete and abstract domains.

Proposition 3.1. Let $a^\sharp$ be an abstract statement, and $S \in \wp(a)$ be a set of concrete statements. α_s and γ_s forms a Galois insertion.

That is:

α_s and γ_s are monotonic,

$S \subseteq \gamma_s(\alpha_s(S))$, where α_s and γ_s are applied pointwise,

$\alpha_s(\gamma_s(a^\sharp)) = a^\sharp$.

Proof:
The concrete and abstract set of statements are flat, thus α_s is monotonic on these sets. Adding a top and a bottom element does not change the property.

For each concrete statement a, we have, by the definition of α_s and γ_s, that $a \in \gamma_s(\alpha_s(a))$. Thus $S \subseteq \gamma_s(\alpha_s(S))$ for each set S of concrete statements.

Given a concrete statement s, the cases of the definitions of α_s and γ_s are such that $s \in \gamma_s(a^\sharp) \Rightarrow \alpha_s(s) = a^\sharp$. Thus, for any abstract statement $a^\sharp$, $\alpha_s(\gamma_s(a^\sharp)) = a^\sharp$. □

Proposition 3.2. Let a and b be concrete statements, the following condition holds.

If $a \longrightarrow b$ then there exists an abstract statement $b^\sharp$ such that $\alpha_s(a) \longrightarrow^\sharp b^\sharp$ and $b \in \gamma_s(b^\sharp)$. $\longrightarrow$ and $\longrightarrow^\sharp$ include the congruence rule applications which make possible the reduction.

Proof:
The proposition is trivially true for all the statements different from the if one. The abstract rules for the if statement, in Table 8, make the proposition true for it as well. □

This proposition states that the abstract reduction correctly approximates the concrete one. That is every concrete computation has a corresponding abstract one.

Thus, if a property is verified for all the reductions of an abstract statement $\alpha_s(a)$ then it is verified also for a.

As a consequence we can check, by abstract reduction, that every access to an object is done while owning the lock to that object. This analysis corresponds to the one in [11].

By using abstract interpretation we can apply a less rigid analysis to detect races. In particular we check that, during the reduction of a statement a, no *parallel* accesses to an object, referred by p, can be performed. That is a (sub)statement of the form $a' \upharpoonright a''$, where $a', a'' \in \{[\![p.\ell_i]\!]^{L1}\} \cup \{[\![p.\ell_j \Leftarrow \varsigma(x)t]\!]^{L2}\}$, is never reached.

This can be done by analyzing the abstraction of a. Recall that we can construct a finite transition system representing all possible computations of $\alpha_s(a)$. Thus, we can statically check that the above kind of statement is never introduced in any abstract computation. If this is true, we can conclude that it cannot be introduced in the concrete computation of a as well. Of course, given the approximation introduced by abstract interpretation, the vice-versa does not hold in general.

Let us state the result formally.

Definition 3.1. A statement (either concrete or abstract) is *race free* iff, for all statements b reached during the reductions, $a \xrightarrow{*} b$, b does *not* contain a (sub)statement of the form $p \mapsto d \upharpoonright b' \upharpoonright b''$, where $d = [\ell_i = \varsigma(x_i)t_i{}^{i \in (1 \ldots n)}]^l$ and b', b'' are statements in the set
$\{[\![p.\ell_i]\!]^{L1} \mid i \in \{1, \ldots, n\},\ L1 \text{ is a set of references}\} \cup$
$\{[\![p.\ell_j \Leftarrow \varsigma(x)t]\!]^{L2} \mid j \in \{1, \ldots, n\},\ L2 \text{ is a set of references}\}$

Note that in the previous definition $\xrightarrow{*}$ means the application of an arbitrary number of reduction rules and structural congruence rules.

Proposition 3.3. Given a statement a, if $\alpha_s(a)$ is race free, then a is race free.

Proof:
By correctness of abstract interpretation [6, 7]. □

Let us give a simple example. Consider the following statement a_1:

$a_1 =$
$[\![\nu p.p \mapsto [\ell = \varsigma(x)5]^\circ \upharpoonright \text{let } x = p.\ell \text{ in if } n \text{ then } p.\ell \text{ else (lock } p \text{ in } p.\ell \upharpoonright \text{ lock } p \text{ in } p.\ell \Leftarrow \varsigma(x)6)]\!]^\emptyset$

$\alpha_s(a_1)$ is analogous to a_1 apart from the substitution of $\odot$ for every integer value. The graph of all possible abstract reductions for $\alpha(a_1)$ is shown in Figure 1. For the sake of readability we removed the restriction νp and the reference statement $p \mapsto [\ell = \varsigma(x)5]^\circ$ from the

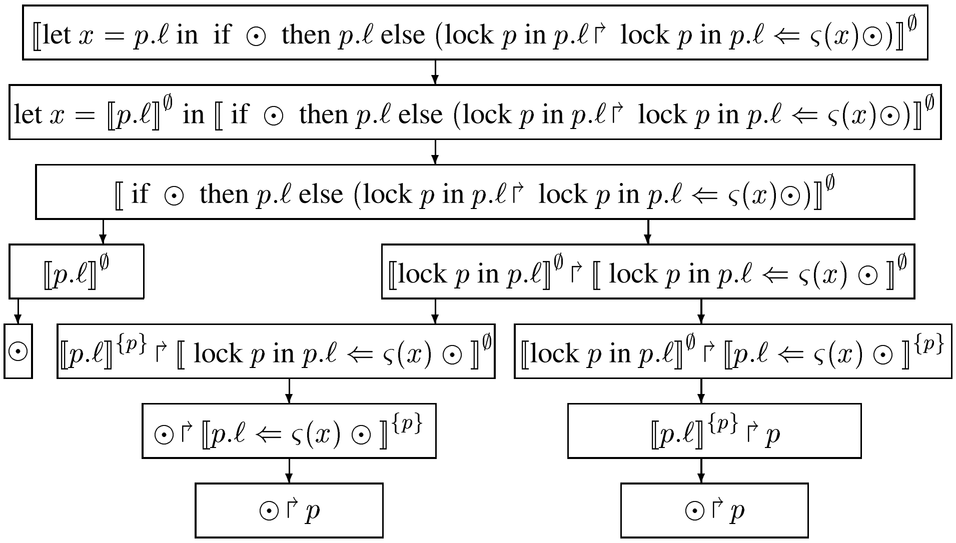

Figure 1. Abstract reduction graph for $\alpha_s(a_1)$

figure. Moreover, because it is clear from the context, we removed the word "locked" from the terms when a lock is acquired. The structural congruence rules are applied implicitly.

It is easy to check that, in the abstract computation graph, all the statements are race free. Thus we can conclude that the concrete computation is race free as well.

Let us remark that the statement a_1 is not certified by current type-based analyses, because the object referred by p is accessed, in two cases, without locking it. However such accesses are safe because they are performed without any other access composed in parallel.

4. Conclusions

In this paper we have applied abstract interpretation techniques to a concurrent object calculus for checking the absence of race conditions.

We have shown that the use of such techniques is more flexible of type-based ones. In particular abstract interpretation allows to certify, as race free, programs which are not certified by current type techniques.

The use of abstract interpretation is based on an extension of a concurrent object calculus which consists in annotating terms with sets of locks. This extension is simple and it does not complicate the semantics of the calculus itself.

The analysis we defined can be the base for the certification of real object oriented programs.

References

[1] Martín Abadi, and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan J. Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. Static Analysis Symposium (SAS) 1999. 19-38

[3] Juan Bicarregui, Kevin Lano, and T. S. E. Maibaum. Formalizing Object-Oriented Models in the Object Calculus. ECOOP Workshops 1997, Lecture Notes in Computer Science 1357, Springer, 1998. 155-160

[4] Chandrasekhar Boyapati, and Martin C. Rinard: A Parameterized Type System for Race-Free Java Programs. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), SIGPLAN Notices 36(11), 2001. 56-69

[5] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), SIGPLAN Notices 37(11), 2002. 211-230

[6] Patrick Cousot, and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2, 1992. 511-547

[7] Patrick Cousot. Abstract Interpretation. *ACM Computing Surveys*, 28, 1996. 324-328

[8] Jos Luiz Fiadeiro, and T. S. E. Maibaum. Describing, Structuring and Implementing Objects. Foundations of Object Oriented Languages. Lecture Notes in Computer Science 489, Springer, 1991.

[9] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing* 1, 1994. 3-37

[10] Cormac Flanagan, and Martín Abadi. Types for Safe Locking. 8th European Symposium on Programming (ESOP), Lecture Notes in Computer Science 1576, Springer, 1999. 91-108

[11] Cormac Flanagan, and Martín Abadi. Object Types against Races. 10th International Conference on Concurrency Theory (CONCUR), Lecture Notes in Computer Science 1664, Springer, 1999. 288-303

[12] James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[13] Andrew D. Gordon, and Paul D. Hankin. A Concurrent Object Calculus: Reduction and Typing. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 16, 1998.

[14] Kohei Honda,and Mario Tokoro. An Object Calculus for Asynchronous Communication. ECOOP 1991. Lecture Notes in Computer Science 512, Springer, 1991. 133-147

[15] Robert H. B. Netzer, and Barton P. Miller. What Are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1, 1992. 74-88