Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 241 (2009) 113–133

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Synchronization as a Special Case of Access Control

Franz Puntigam[1]

Institute of Computer Languages
Vienna University of Technology
Vienna, Austria

Abstract

Synchronization ensures exclusive shared-variable access at runtime, and static access control mechanisms give similar guarantees at compilation time. Usually we treat these language concepts as separate. In this work we propose to integrate synchronization into access control in a Java-like language: Shared-variable access depends on the availability of tokens (as a form of access control), and the compiler generates code for locking to gain the needed tokens (synchronization). We get more freedom in expressing synchronization at appropriate points in a program and weaker influence of concurrency on the program structure.

Keywords: Synchronization, access control, concurrent programming, object-oriented programming.

1 Introduction

Concurrency is an important aspect of programming where we urgently need more advanced support. Independently, there is work on language concepts for describing software architectures at a higher level [3] where techniques allowing us to constrain access to objects in certain ways play an important role. We explore a possibility to integrate synchronization of concurrent threads into an access control mechanism.

Synchronization is a technique to ensure unique access. Language concepts for access control (like private visibility) are static while synchronization is inherently dynamic. To overcome this discrepancy we add a dynamic quality to access control.

We demonstrate basics of our approach with an example in a Java-like language:

```
class SimpleBuffer {
   void put(int i)[empty:true->empty:false] {el=i; empty=false;}
   int get()[empty:false->empty:true] {empty=true; return el;}
   SimpleBuffer()[->empty:true] {empty=true;}
```

[1] Email: franz@complang.tuwien.ac.at

doi:10.1016/j.entcs.2009.06.007

```
    int el;  boolean empty;
}
```

Square brackets on methods represent access constraints. They contain required tokens (that must be available on method invocation) to the left of `->` and ensured tokens (available after return) to the right. A token of the form $x{:}v$ guarantees that the owner of the token has exclusive access to variable x which currently holds value v.[2] We can invoke `y.put(...)` only where we have exclusive access to `y.empty` while this variable holds `true`. After return we again have exclusive access, but then `empty` holds `false`. Hence, `put` and `get` are invokable only in alternation. We need no dynamic synchronization within buffers: There can always exist only one client owning an exclusive token needed to access a buffer.

Buffers are useful only if several clients access them concurrently. Clients compete for exclusive access. We use non-exclusive tokens of the form $x{:}v?w$ promising that no client permanently has a token $x{:}u$ for any u. However, each client owning $x{:}v?w$ can temporarily get $x{:}v$ together with a write-lock on x when invoking a method with access constraint `[`$x{:}v$ `->` $x{:}w$`]`. The client's thread must wait until x holds value v and then acquire and hold a lock on x while executing the method. Within the method we use the exclusive token $x{:}v$ (or $x{:}w$ after replacement) without further locking, for instance, in recursive invocations.

The following code snippet shows the use of non-exclusive tokens:

```
void produce_consume()
  {SimpleBuffer b=new SimpleBuffer(); produce(b); consume(b);}
async produce(SimpleBuffer [empty:true?false ->] p)
  {while(true) {...; p.put(...); ...}
async consume(SimpleBuffer [empty:false?true ->] c)
  {while(true) {...; i=c.get(); ...}
```

Methods declared as `async` are executed in new threads as in Polyphonic C# [5]. An invocation of `produce_consume` creates a buffer and two threads concurrently accessing the buffer. The compiler associates `b` first with `empty:true` specified in the constructor of `SimpleBuffer` and then replaces this token with the less informative tokens `empty:true?false` and `empty:false?true` needed in `produce` and `consume` (see Section 3). Arrows in the annotations of the formal parameters `p` and `c` indicate that the tokens move from the actual parameter `b` to `p` and `c` and never come back to `b`. For invocations of `put` in `produce` (and `get` in `consume`) the compiler generates code acquiring a lock after waiting until `empty` is `true` (`false` in `consume`).

In our example, `put` and `get` can be executed only in alternation. If a client repeatedly uses `empty:true?false` to invoke `put`, there must be a concurrent client repeatedly using `empty:false?true` to invoke `get`. In general, for continuous operation we need concurrent clients owning tokens $x{:}v_1?v_2, \ldots, x{:}v_{n-1}?v_n$ as well as $x{:}v_n?v_1$ (that is, there must be a closed loop on successive values of x), and the clients must repeatedly use these tokens in invocations. Non-exclusive tokens

[2] In general, we can use a type τ instead of a value v. In this case x holds an instance of τ. Furthermore, $x{:}v$ guarantees exclusive access not only to x, but also to a set of variables protected by x.

- highlight the need of synchronization in invocations,
- ensure the absence of corresponding permanent exclusive tokens (without a lock),
- and help the compiler to ensure continuous operation of the system.

Our approach promises to give us a number of advantages:

- Programmers think in terms of accessibility at a high level instead of low-level synchronization. Compilers ensure synchronization with properties like race-freeness and (to some extent) continuity. Concurrency need not dominate the program structure, and we can concentrate on other important programming principles.
- Interfaces specify accessibility. Clients need this information to avoid access and synchronization conflicts. Since we specify in interfaces only necessary information we keep the black-box view of objects and can take advantage of data hiding: Changes of implementation details do not affect clients.
- There is more freedom in ensuring uniqueness at appropriate points in the program. By means of access control we safely move synchronization from servers to clients. The difference between access control and synchronization vanishes.
- We consider unique access in subtyping and ensure uniqueness in a modular way.

In one aspect the proposed concept resembles the SCOOP model of Eiffel [22] where preconditions represent synchronization conditions: Synchronization depends only on current values of variables in objects – a concept familiar to every programmer.

We introduce the proposed language concepts in more detail in Section 2 and look at static guarantees in Section 3. A discussion of related work follows in Section 4 and concluding remarks in Section 5.

2 Accessibility and Synchronization

2.1 Access Constraints and Annotations

In most language concepts ensuring unique access, programmers specify that some object is accessible only through a specific reference, that is, there are no aliases [12]. Instead, to support aliasing we express accessibility of specific methods in objects depending on the availability of tokens:

- Programmers annotate methods and constructors with access constraints of the form `[`$t_1, \ldots, t_m$ `->` $t'_1, \ldots, t'_n$`]`. Clients must deliver the required tokens $t_1, \ldots, t_m$ on invocation and get the ensured tokens $t'_1, \ldots, t'_n$ on return. Variable names in these tokens belong to the same scope as the annotated methods and constructors.
- Object references can carry annotations with tokens. Programmers annotate
  - · formal parameters with required tokens to be delivered by clients on invocation and ensured tokens delivered to clients on return using the same syntax as for access constraints;
  - · method results with tokens $t_1, \ldots, t_n$ to be returned (by writing `[`$t_1, \ldots, t_n$`]` immediately after the result type).

  Variable names in these tokens belong to the scopes of the references' declared

$x{:}\tau$	(Exclusive access to variables protected by x; value in x is of type τ)
$*x{:}\tau$	(Shared read-access to variables protected by x; value in x is of type τ)
$x{:}\sigma?\tau$	(Usable in `[`$x{:}\sigma$ `->` $x{:}\tau$`]` and `[`$*x{:}\sigma$ `->` $*x{:}\sigma$`]` while holding a lock on x)

Table 1
Kinds of tokens representing knowledge about instance variable x

types. The compiler infers annotations of all other occurrences of references (including instance variables, class variables, local variables, and local uses of formal parameters).

We distinguish between the three kinds of tokens shown in Table 1. A token $x{:}\tau$ ensures exclusive access to all variables protected by x (see Section 3.2). The second kind $*x{:}\tau$ supports shared read-access to variables protected by x and prohibits concurrent write-accesses. Tokens $x{:}\tau$ and $*x{:}\tau$ guarantee the value of x to be of type τ. Using the last kind $x{:}\sigma?\tau$ in method invocations the compiler automatically generates code acquiring write-locks for exclusive access and read-locks for shared read-access; threads will have to wait until x holds a value of type σ, and on return x will hold a value of type τ.

Although the syntax requires types in tokens we often use literals instead; that is, we consider literals also to be types. Furthermore, we use range types like `0..7` and `1..` where bounds must be constant, but not necessarily static. If a type in a token does not matter, we use the declared type of the variable.

Another buffer example shows the use of such tokens:

```
class Buffer {
   void put(int i)[size:0..(max-1)->size:1..max] {el[size++]=i;}
   int get()[size:1..max->size:0..(max-1)] {return el[--size];}
   Buffer((1..) m)[->size:0] {max=m; el=new int[max];}
   final (1..) max;
   (0..max) size = 0;
   int el[];
}
```

When creating a new instance of `Buffer` we get a token `size:0`. This token can be used in an invocation of `put` because `0` is in the range `0..(max-1)`, and the resulting token `size:1..max` can be used in an invocation of `get`. In most cases we will repeatedly invoke `put` with a token `size:0..(max-1)?1..max` and `get` with `size:1..max?0..(max-1)` using synchronization. We can get these tokens from `size:0` as shown in Section 3.1.

For synchronization using a token of the form $x{:}\sigma?\tau$ the corresponding access constraint or parameter annotation `[`$t_1, \ldots, t_m$ `->` $t'_1, \ldots, t'_n$`]` of the invoked method must satisfy one of these conditions:

- If we need exclusive access to variables protected by x (and a write-lock on x), then there are indices i and j with $t_i = x{:}\sigma$ and $t'_j = x{:}\tau$, and there is no other token in $t_1, \ldots, t_m, t'_1, \ldots, t'_n$ where x occurs to the left of ":". The value of x can (or must if σ and τ do not overlap) change from one of type σ to one of type τ. In the buffer example, the access constraints of `put` and `get` are of this form.

- If we need shared read-access to variables protected by x (and a read-lock on x), then we have $\sigma = \tau$, there is a number k ($1 \leq k \leq m$) of occurrences of $*x{:}\tau$ in $t_1, \ldots, t_m$, there is the same number k of occurrences of $*x{:}\tau$ in $t'_1, \ldots, t'_n$, and there is no other token in $t_1, \ldots, t_m, t'_1, \ldots, t'_n$ where x occurs to the left of ":". The value of x must not change while executing the method.

If an access constraint or parameter annotation is not of one of these forms for a variable x, then no synchronization on x is possible. Such access constraints and annotations can still be useful where we need only access control.

Often we need several synchronization steps in a single invocation, that is, we use several tokens of the form $x{:}\sigma?\tau$ on different parameters or on different variables x. The execution of the invoked method must be delayed until all needed locks have been acquired, no matter if the variables to be locked belong to parameters or the object where the method is invoked within.

2.2 Token Movement

Tokens can move from one reference to another as a side-effect of parameter passing and assignment:

```
void foo(Buffer[size:0..(max-1)->size:1..max] b)
   { b.put(1); }
void bar(Buffer[size:0..(max-1)->size:0..max] b)
   { if(unknown) {b.put(1);} }
Buffer[size:1..max] baz(Buffer[size:0..(max-1)->] b)
   { if(unknown) {b.put(1); return b;} else {return null;} }
```

On invocation of `foo` the token `size:0..(max-1)` in the annotation of the formal parameter `b` moves from the actual parameter to `b`, and on return `size:1..max` moves from `b` back to the actual parameter. An invocation of `put` in the method body changes the token accordingly. In `bar` a statically unpredictable computation causes a loss of type information in a token. By invoking `baz` we lose the token if the method returns `null`. We assume `null` to be associated with every token, but such tokens are useless since no method is invokable through `null`.

When executing `b.put(1)` the token `size:0..(max-1)` moves from `b` to `this` of the buffer referenced by `b`, and on return `size:1..max` moves from `this` to `b`. The value of `size` can be modified only if `this` has an exclusive token `size:`τ, and as a side-effect this token is modified, too.

The following program fragment shows the use of tokens when creating threads:

```
void produce_consume() {
   Buffer b = new Buffer(8);
   Counter c = new Counter();
   produce(b,c);
   produce(b,c);
```

```
        while(true) { b.get(); }
    }
    async produce(Buffer[size:0..(max-1)?1..max->] b, Counter c)
        { while(true) { b.put(c.next()); }
```

Asynchronous methods (these are methods where `async` occurs instead of the result type as in Polyphonic C# [5]) are executed in new threads concurrently with the threads invoking the methods. Since there is no return from asynchronous methods, there must not be any ensured token in access constraints of such methods and in annotations of their formal parameters. There is no synchronization when invoking asynchronous methods because ensured tokens as in [x:σ -> x:τ] are a prerequisite of synchronization. In `produce_consume`, variable `b` is initially annotated with `size:0` as specified in the constructor of `Buffer`. As shown in Section 3.1, this token can be replaced by two tokens `size:0..(max-1)` and a token `size:1..max`. On each invocation of `produce` a token moves to the new thread, and one token remains in `produce_consume`. These tokens are repeatedly used in method invocations to ensure continuous operation. An instance of `Counter` produces new integer values:

```
class Counter {
    int next()[i:int -> i:int] { return i++; }
    Counter() { i=0; }
    int i=0;
}
```

As discussed in Section 2.3 each instance of `Counter` has an implicit token `i:int?int`, and in `produce_consume` we need not take care for such tokens although two producers concurrently access the same object.

Tokens move between references also on assignment. When executing `x = y;` the tokens associated with `x` before assignment get lost. The tokens associated with `y` are divided into two parts as needed in further computations; `x` becomes associated with one part, and `y` remains to be associated with the other part. The compiler can always determine at compile time (for each class separately) how to divide the tokens using techniques similar to those proposed in [29].

2.3 *Information Exposed by Servers*

Clients invoking `put` in an instance of `Buffer` usually own only a non-exclusive token `size:0..(max-1)?1..max`, and those invoking `get` own `size:1..max?0..(max-1)`. In this case clients have no static knowledge of `size`, and when acquiring locks the threads must wait for appropriate values of `size`.

Static type information as in `size:0` is useful for purposes like putting initial data into a buffer. Even without static type information we can use access control instead of synchronization using the exclusive token returned by the constructor:

```
Buffer b = new Buffer(unknown_size);
do { b.put(1); }
while (b.size instanceof (0..(b.max-1)))
```

Because of exclusive access to `b` (shared read access would be sufficient) we can check the dynamic type of `size` and thereby implicitly change the type in the token from `1..max` to `0..(max-1)` as needed when invoking `put`. This way we can change every token $x{:}\tau$ and $*x{:}\tau$ to $x{:}\sigma$ and $*x{:}\sigma$, respectively, when the value of x is of type σ. Type information in ensured tokens (valid on return) of the forms $x{:}\tau$ and $*x{:}\tau$ does not influence whether methods are invokable, but such information helps us to avoid dynamic checks and causes programs to be more readable and reliable.

Type information in ensured tokens is part of a mechanism to guarantee continuous operation by helping us to show the existence of loops in successive values that we need in tokens of the form $x{:}\sigma?\tau$ (see the introduction). If the ensured token of `put` was just `size:0..max` (instead of `size:1..max`), then after changing implementation details of `Buffer` the value of `size` could be zero, causing invocations of `get` to never become executable. Such situations cannot happen if for each required exclusive token in an access constraint there is at least one equal or more restrictive ensured token in the same or another access constraint.

We distinguish between two kinds of synchronization:

Dependent synchronization on x occurs when invoking a method (that requires $x{:}\tau$ or $*x{:}\tau$ in the access constraint) using $x{:}\tau?\sigma$ for synchronization where

- τ differs from (and is a subtype of) the declared type of x
- or the invoked method modifies x and there is a method that requires $x{:}\tau$ or $*x{:}\tau$ with τ being different from the declared type of x.

The active thread must wait until x holds a value of type τ (that is, the execution possibly depends on other threads changing the value of x) or the execution of the method changes the value of x and thereby possibly wakes up another thread. For example, we have dependent synchronization on `size` when using an instance of `Buffer`.

Simple mutex synchronization on x occurs in all other cases of synchronization on x. The execution neither depends on variable assignments in other threads nor wakes up other threads because of variable assignments. For example, we have simple mutex synchronization on `i` when using an instance of `Counter`. Simple mutex synchronization is a property of the access constraints of all methods in a class and does not depend on the context of an invocation.

Differences between `Counter` and `Buffer` are obvious: For continuous operation it is necessary to concurrently and repeatedly invoke `put` and `get` in a buffer as discussed above; otherwise an invoked method may never become executable. However, no thread has to wait forever just because we do not repeatedly invoke `next` in an instance of `Counter`. Clients have to know about dependent synchronization (expressed by tokens of the form $x{:}\sigma?\tau$) to avoid synchronization conflicts. Clients need not know about simple mutex synchronization that affects other threads only by delaying them for a finite amount of time (except in a deadlock[3]).

[3] Both dependent synchronization and simple mutex synchronization can suffer from deadlocks. We could argue that clients must know about simple mutex synchronization to avoid deadlocks. However, a deadlock is a cycle of threads waiting for each other, and it is not clear which thread is mainly responsible for the situation. It is unfair to make clients responsible although the server is always at least equally involved. On

To take advantage of this difference we allow a client to invoke a method that requires a token $x{:}\tau$ or $*x{:}\tau$ even without availability of the required token or a token $x{:}\tau?\sigma$ if

- we have just simple mutex synchronization (no dependent synchronization) on x,
- and to avoid conflicts with access control no constructor in the corresponding class ensures a token $x{:}\tau$ or $*x{:}\tau$ (for any τ; hence, there is no access constraint on the constructor of `Counter`).

In other words, we use an *implicit token* $x{:}\tau?\tau$ in this case where τ is the declared type of x. It is still necessary to acquire a lock on invocation, but clients need not take care of implicit tokens. Implicit tokens

- need not be considered in subtyping (see Section 3.3)
- and do not pollute the whole program code with tokens (which easily can happen with dependent synchronization in badly organized programs).

2.4 Inference of Tokens and Locks

Annotations of instance variables, class variables, and local variables are inferred by a compiler. A reason for doing so can be seen by looking at the annotation of the instance variable `b` in class `Wrapper`:

```
class Wrapper {
   void set(int i)[e:true->e:false] { b.put(i); e=false; }
   int take()[e:false->e:true] { b.get(i); e=true; }
   Wrapper()[->e:true] { b=new Buffer(1); e=true; }
   Buffer b;  boolean e;
}
```

Sometimes `b` has to be annotated with `size:0..(max-1)`, and sometimes with `size:1..max` (where `max` = 1) depending on the value of `e`. In larger examples there can be many more conditional annotations of the same variable, and explicit annotations with all possibilities would be a large burden for programmers.

Fortunately, signatures of methods and constructors hold all the information needed to infer annotations: In a walk through the program code of each method we determine all tokens used in the method body depending on the required tokens in the method's access constraint as well as tokens available on termination depending on the method's ensured tokens. While walking through `set` we determine from the signature of `put` that `b` has to be annotated with `size:0..(max-1)` depending on `e:true` and can be annotated with `size:1..max` depending on `e:false`. It would also be possible to annotate `b` with `size:0..(max-1)?1..max` independent of any token, but that is undesirable as discussed in Section 3.4. Similarly (according to `take`) `b` must be annotated with `size:1..max` depending on `e:false` and can be annotated with `size:0..(max-1)` depending on `e:true`, or `b` has to be anno-

the other side, most conflicts of dependent synchronization result from insufficient concurrency in clients outside of the server's responsibility. We discuss deadlock prevention in Section 3.4.

$$M \vdash A \xrightarrow{\epsilon} A/\epsilon \qquad \frac{M \vdash A \xrightarrow{B} C/L}{M \vdash A \xrightarrow{B,x[\epsilon \to \epsilon]} C/L}$$

$$\frac{M, x \mapsto y \vdash A, y[a] \xrightarrow{B,x[b \to c]} C/L}{M, x \mapsto y \vdash A, y[a,d] \xrightarrow{B,x[b,d \to c]} C/L} \qquad \frac{M, x \mapsto y \vdash A \xrightarrow{B,x[a \to b]} C, y[c]/L}{M, x \mapsto y \vdash A \xrightarrow{B,x[a \to b,d]} C, y[c,d]/L}$$

$$\frac{M, x \mapsto y \vdash A, y[a] \xrightarrow{B,x[b \to d]} C, y[c]/L}{M, x \mapsto y \vdash A, y[a, z{:}\sigma?\tau] \xrightarrow{B,x[b,z:\sigma \to d,z:\tau]} C, y[c, z{:}\sigma?\tau]/L, \mathrm{wlock}(y.z, \sigma)}$$

$$\frac{M, x \mapsto y \vdash A, y[a] \xrightarrow{B,x[b \to d]} C, y[c]/L}{M, x \mapsto y \vdash A, y[a, z{:}\tau?\tau] \xrightarrow{B,x[b,*z:\tau \to d,*z:\tau]} C, y[c, z{:}\tau?\tau]/L, \mathrm{rlock}(y.z, \tau)}$$

Table 2
Token checking and lock inference for method invocation (simplified)

tated with `size:1..max?0..(max-1)` independent of any token. According to the constructor, `b` can be annotated with `size:0` depending on `e:true`. In a final step we combine this information: We avoid annotations with tokens of the form $x{:}\sigma?\tau$ whenever possible, and the compiler issues a warning if there is no alternative to using such tokens. There remains only one possibility: The variable `b` is annotated with `size:0..(max-1)` depending on `e:true` and with `size:1..max` depending on `e:false`. Since `size:0` can be used where `size:0..(max-1)` is expected (see Section 3.1), the information inferred from the constructor is compatible with that inferred from the methods. As in this example there can always be only one most general solution when avoiding tokens of the form $x{:}\sigma?\tau$.

Table 2 shows essential parts of type checking rules for method invocations. By A, B, C we denote comma-separated lists of annotated arguments or formal parameters of the form $x_1[a_1], \ldots, x_n[a_n]$ or $x_1[b_1 \to c_1], \ldots, x_n[b_n \to c_n]$ where the x_i ($i = 1..n$) are pairwise different arguments or parameter names (including the implicit parameter `this`) and the a_i, b_i, c_i comma-separated lists of tokens. We denote empty lists of all kinds by ϵ. A mapping M associates formal parameters with arguments. In $M \vdash A \xrightarrow{B} C/L$ the list B associates the access constraint of an invoked method with `this` and specifies parameter annotations, A specifies annotations (of all variables and parameters usable as arguments) with tokens regarded as available before invocation, C does so for tokens that will be available after return, and L is a list of locks to be acquired before method execution. By applying the rules in Table 2 we check if all tokens needed in an invocation are available, and as side-effects we infer assumed annotations of local variables, instance variables, and class variables (avoiding the last two rules if possible, except for implicit tokens always assumed to be available), and derive locks to be acquired.

If several locks are needed for a single invocation, they have to be acquired in a specific order depending on the variable names in the tokens. For this purpose we assume a global total ordering of variable names as given. Variables declared in different classes are considered to have different names. We use this order also in deadlock prevention (see Section 3.4). Unfortunately, we can pinpoint the order only when we know all dependencies – at link time for languages like C++ and only at runtime for Java-like languages where classes are dynamically loaded. The compiler issues a warning if several locks on the same variable name are needed for a single invocation; in this case there is a high probability of a deadlock.

The compiler has detailed information about available tokens associated with references. However, it may be difficult to condense this information so that a programmer can easily see from error messages what is wrong. A missing token t for an instance variable v can be reported as "v needs t" (appropriate if a local invocation is wrong), as "v needs t which may not be available after executing $m_1, \ldots, m_n$" (appropriate if another method may be wrong), or as "v needs t which is available only if w has a token t'" (appropriate if the object may be in a wrong state). The message can depend on the tokens inferred for v from each method of the class: If no t was inferred at all, the first message may be appropriate. Otherwise, if all methods that inferred t have a common required token in their access constraints, the third message may be preferable over the second one.

3 Static Guarantees

3.1 Token Equivalence, Token Subsumption, and Exclusivity of Tokens

Table 3 shows an equivalence and a subsumption relation on comma-separated token lists. Tokens a can be used where tokens b are expected if $a \leqq b$ holds. For example, we can use $x{:}\sigma, y{:}\tau$ where $*x{:}\sigma, y{:}\tau?\tau$ is expected.

Non-exclusive tokens of the form $x{:}\sigma?\tau$ can be duplicated as often as needed while the compiler must prevent duplication of exclusive tokens $x{:}\tau$ as well as tokens of the form $*x{:}\tau$.[4] Tokens of the forms $x{:}\tau$ and $*x{:}\tau$ can get lost, that is, they can be used where an empty token sequence ϵ is expected. In general, tokens of the form $x{:}\sigma?\tau$ must not get lost to ensure continuous operation; their owners have to repeatedly invoke corresponding methods. However, two equal tokens $x{:}\sigma?\tau$ can be combined to a single one because a single token is sufficient to guarantee repeated invocations. It is possible to add or delete a token $x{:}\sigma?\sigma$ (not implying a state change when used in synchronization) at the presence of another token $x{:}\sigma?\tau$.

Type information in tokens can safely be lost. For example, we can use $x{:}\sigma?\tau$ where $x{:}\sigma'?\tau'$ is expected if σ is a subtype of σ' and τ' of τ because

- a thread can wait for a value of x in a broader range than the value will be in,
- and after return the value of x can be in a smaller range than it has to be.

[4] It may be surprising that we must not duplicate tokens of the form $*x{:}\tau$ since any number of them can exist simultaneously. The reason is the necessary control of the number of such tokens while there is a read-lock on x. If these tokens were duplicated in an uncontrolled way, it would be impossible to ensure that all tokens introduced at the begin of method execution will disappear on termination. An extension of the proposed concept with a mechanism to control the number of such tokens or with another kind of tokens supporting read-only access may be useful. We refrain from such extension to keep the model simple.

$$a \equiv a \qquad \frac{a \equiv b \quad b \equiv c}{a \equiv c} \qquad \frac{a \equiv b}{b \equiv a} \qquad \frac{a \equiv a' \quad b \equiv b'}{a, b \equiv a', b'}$$

$$a, b \equiv b, a \qquad a, (b, c) \equiv (a, b), c \qquad a \equiv a, \epsilon$$

$$x{:}\sigma?\tau \equiv x{:}\sigma?\tau, x{:}\sigma?\tau \qquad x{:}\sigma?\tau \equiv x{:}\sigma?\sigma, x{:}\sigma?\tau$$

$$\frac{a \equiv b}{a \leqq b} \qquad \frac{a \leqq b \quad b \leqq c}{a \leqq c} \qquad \frac{a \leqq a' \quad b \leqq b'}{a, b \leqq a', b'} \qquad x{:}\tau \leqq \epsilon \qquad *x{:}\tau \leqq \epsilon$$

$$\frac{\sigma \leq \tau}{x{:}\sigma \leqq x{:}\tau} \qquad \frac{\sigma \leq \tau}{*x{:}\sigma \leqq *x{:}\tau} \qquad \frac{\sigma \leq \sigma' \quad \tau' \leq \tau}{x{:}\sigma?\tau \leqq x{:}\sigma'?\tau'}$$

$$\frac{\sigma \leq \tau \quad x \text{ instanceof } \sigma}{x{:}\tau \leqq x{:}\sigma} \qquad \frac{\sigma \leq \tau \quad x \text{ instanceof } \sigma}{*x{:}\tau \leqq *x{:}\sigma}$$

$$x{:}\tau \leqq *x{:}\tau \qquad x{:}\tau \leqq x{:}\tau?\tau \qquad x{:}\sigma?\tau \leqq x{:}\sigma?\rho, x{:}\rho?\tau$$

Table 3
Equivalence $\equiv$ and subsumption $\leqq$ of token sequences

`size:0` $\leqq$	(loss of type information)
`size:0..(max-1)` $\leqq$	(to non-exclusive token)
`size:0..(max-1)?0..(max-1)` $\leqq$	(last rule in Table 3)
`size:0..(max-1)?1..max`, `size:1..max?0..(max-1)` $\equiv$	(token duplication)
`size:0..(max-1)?1..max`, `size:0..(max-1)?1..max`, `size:1..max?0..(max-1)`	

Table 4
Example of deriving needed tokens using subsumption

Where we have additional information about the value of x (gained by an application of `instanceof`) we can use this information in tokens of the form $x{:}\tau$ and $*x{:}\tau$.

The last rule in Table 3 allows us to divide a token $x{:}\sigma?\tau$ into $x{:}\sigma?\rho$ and $x{:}\rho?\tau$. Two consecutive method executions according to the divided tokens change the value of x in the same way as a single method execution does according to $x{:}\sigma?\tau$, except that an intermediate value of x can become visible. Once a token has been divided, repeated invocations may depend on this intermediate value, and we cannot combine the tokens again in the reverse way.

In the `producer_consumer` example (see Section 2.2), the compiler uses subsumption to generate the needed tokens as shown in Table 4.

Let us summarize which tokens simultaneously exist in a programming system:

- Object creation can introduce new tokens as ensured in a constructor. For each instance variable x in the new object we demand that there can be either at most one token of the form $x{:}\tau$ or any number of equal tokens of the form $*x{:}\tau$ or $x{:}\tau?\tau$. If a constructor does not ensure any token for x although there are methods with access constraints requiring such tokens, then we assume an implicit token $x{:}\tau?\tau$ (with τ being the declared type of x for simple mutex synchronization) as globally available. Because there is no constructor for classes, all initial tokens on class variables are implicit in this sense. In each case the value of x is of type τ. Since the created object is new, there cannot exist other tokens for x anywhere else.

- Tokens can move as side-effect of parameter passing and assignment. Thereby, token sequences can be modified only according to $\leqq$ (from left to right).
- As side-effect of assigning a new value to x a token $x{:}\sigma$ can change to $x{:}\tau$ (for some types σ and τ). The assigned value is of type τ. If x occurs in any token, then the assignment is possible only at the presence of a token $x{:}\sigma$.
- When using a token $x{:}\sigma?\tau$ in an invocation of a method with a required token $x{:}\sigma$ (or k required tokens $*x{:}\sigma$ provided that σ equals τ) and an ensured token $x{:}\tau$ (or k ensured tokens $*x{:}\sigma$) we introduce a new token $x{:}\sigma$ (or k tokens $*x{:}\sigma$). This token (or these tokens) can come into existence only after acquiring a write-lock (or read-lock) on x when the value of x is of type σ, and the corresponding token(s) will disappear when returning the lock.

From these possible changes of the set of tokens available in a system we can (without proof) derive the following important properties:

- For each variable x in an object there can always exist at most one token $x{:}\tau$ (for any τ). Further tokens of the form $x{:}\sigma?\rho$ can exist simultaneously.
- If for some variable x in an object there exists a token $*x{:}\sigma$, then there cannot at the same time exist a token $x{:}\tau$ (for any τ). Only further tokens of the forms $x{:}\sigma'?\tau'$ and $*x{:}\sigma'$ can exist at the same time.
- If for a variable x there is a token $x{:}\tau$ or $*x{:}\tau$, then the value of x is of type τ.
- Let us assume that all methods invoked using synchronization terminate in finite time. Then, if there is a token $x{:}\sigma?\tau$ and simultaneously $x{:}\rho$ or $*x{:}\rho$ (for any ρ), there is a future program state where no token $x{:}\rho'$ and $*x{:}\rho'$ exists (for any ρ').

3.2 Race-Free Programs

We use access control and synchronization mainly to avoid races in variable accesses: Well-defined program segments shall be executed atomically so that it is impossible to see intermediate program states within other program parts. During execution of such a program segment no variable possibly modified shall be accessible outside, and no variable possibly read within shall be modifiable outside.[5]

In our approach we use methods as program segments to be executed atomically. Required tokens of the form $x{:}\tau$ and $*x{:}\tau$ in the methods' access constraints determine the shared variables to be accessed within the method exclusively and read-only, respectively. The variable x occurring in a token $x{:}\tau$ or $*x{:}\tau$ is a representative of a set of accessible shared variables. We say that x *protects* all variables in this set (from inconsistent concurrent accesses).

[5] In this article we take a rather conventional view of synchronization based on locking. When using techniques like memory transactions [17] we can relax this condition to get more concurrency. The use of such techniques in our approach especially for simple mutex synchronization is important future work. We do not discuss this topic here because the focus is on dependent synchronization.

The set of shared variables protected by a variable x (which must be an instance variable or class variable not declared as final) is constructed from a class definition: An instance variable or class variable y not declared as final is in the set if

- each method (and if y is a class variable, then also each constructor) possibly directly writing to y has a required token $x{:}\tau$ (for some τ) in its access constraint,
- and each method (and constructor if y is a class variable) possibly directly reading from y has a required token $x{:}\tau$ or $*x{:}\tau$ (for some τ) in its access constraint,
- and if y is a class variable, then also x is a class variable.

Direct accesses literally occur in the code of the method. Accesses within other methods invoked by the method in question do not count. Formal parameters, local variables, and constants (these are variables declared as final) need not be considered because they are not shared or not modifiable. Instance variables in constructors are not considered because we preclude concurrency during object creation.

A program is race-free if each instance variable (except if occurring only in constructors) and class variable used in the program is protected by some variable. This property follows from the definition of protected variables and the facts that

- there cannot exist several tokens of the form $x{:}\tau$ for the same x (ensuring exclusive access to variables protected by x),
- and at the presence of a token $*x{:}\tau$ there cannot exist a token $x{:}\sigma$ (for any σ, ensuring that no concurrent write-access to variables protected by x can occur).

To ensure race-free programs this way we use only access control, no synchronization. We expect programmers to think in terms of accessibility instead of synchronization although each method shall be invokable in concurrent environments.

3.3 Subtyping

Programmers must provide access constraints on methods as well as annotations of formal parameters and results. This information must be considered in (behavioral) subtyping because it has major influence on object behavior. We consider annotations of formal parameters and method results to belong to corresponding types. On such annotated types we use the following subtyping rules:

$$\frac{\sigma \leq \tau \quad a \leqq b}{\sigma\,\texttt{[}a\texttt{]} \leq \tau\,\texttt{[}b\texttt{]}} \qquad \frac{\sigma \leq \tau \quad a \leqq c \quad d \leqq b}{\sigma\,\texttt{[}a\,\texttt{->}\,b\texttt{]} \leq \tau\,\texttt{[}c\,\texttt{->}\,d\texttt{]}}$$

As usual, formal parameter types can be contra-variant while result types can be co-variant. Subtyping of annotated types resembles subtyping with assertions where a subtype can have weaker pre-conditions (requiring less restrictive tokens) and stronger post-conditions (ensuring more restrictive tokens) than supertypes [21].

Because of implicit tokens for simple mutex synchronization, subtyping on method signatures (especially access constraints) is slightly more difficult:

$$\frac{(\sigma_i \leq \tau_i)^{i=1..n} \quad c \leqq a \quad b \leqq d \quad A \rhd e \quad B \rhd f}{A.m(\tau_1, \ldots, \tau_n)\,\texttt{[}a, e\,\texttt{->}\,b, e\texttt{]} \leq B.m(\sigma_1, \ldots, \sigma_n)\,\texttt{[}c, f\,\texttt{->}\,d, f\texttt{]}}$$

where A, B, C denote classes, m a method name, and σ, τ annotated types. The following rules define $C \rhd a$ (a is a token sequence compliant with implicit tokens introduced by class C):

$$\frac{C \rhd a \quad C \rhd b}{C \rhd a, b} \qquad \frac{\text{implicit } x{:}\tau?\tau \text{ in } C}{C \rhd x{:}\tau} \qquad \frac{\text{implicit } x{:}\tau?\tau \text{ in } C}{C \rhd *x{:}\tau} \qquad C \rhd \epsilon$$

That is, access constraints can contain the same sequences of tokens to the left and to the right of `->` which are not considered in subtyping. Such token sequences can contain only tokens of the form $x{:}\tau$ and $*x{:}\tau$, and there must be a global implicit token $x{:}\tau?\tau$ for instances of the class. Although not formally required we expect the token sequences to be of a form supporting synchronization (see Section 2.1). Otherwise it would not be possible to invoke the methods.

Access constraints constrain possible sequences of method executions on the object parts consisting of all variables protected by variables occurring in the tokens. Since accesses to these variable sets are always serializable we have actually serial sequences of method executions and can use simple interleaving semantics. Formally we can specify the semantics of annotated types as prefix-closed trace sets. If two annotated types are equivalent, then the corresponding trace sets are equal.

Subtyping conforms to the principle of substitutability: An instance of a subtype can be used where an instance of a supertype is expected. Semantically, the principle of substitutability implies essentially that the trace set of a supertype is a subset of the trace set of a subtype [28]. If a client can invoke a method according to access constraints of a supertype, then it can do so also according to access constraints of the subtype. Similarly for synchronization, if a thread gets all needed locks to execute a method according to a supertype, then it does so also according to the subtype. This property is obvious for dependent synchronization. For synchronization using implicit tokens not considered in the subtyping relation this property follows from the fact that we always have simple mutex synchronization in this case which precludes simultaneous execution, but has no influence on the sequential ordering of executions.

Unfortunately, access constraints and synchronization in subtypes cannot be more restrictive than in supertypes (except for simple mutex synchronization) according to the principle of substitutability. Programmers usually want to have it the other way around. Therefore, the more flexible solution for simple mutex synchronization (which probably occurs more often than dependent synchronization) can be quite helpful in many situations. It allows programmers to introduce synchronization in derived classes even if there is no synchronization in base classes.

3.4 Continuity

Continuity is a very powerful and usually desirable property. It statically ensures useful computations to go on and produce results forever. Taken to its full extent, continuity implies amongst others that all the required resources (memory and time) are available and the computation does not suffer from deadlocks, starvation, and livelocks. Where continuity is extremely important (for example, in safety-critical systems) it is actually possible to get static guarantees. However, they come at very high costs. We must accept restrictive design rules and inflexible tools, need experienced and expensive experts who put much effort into low-level programming and program analysis, and get long development times.

Fortunately, strong guarantees are rarely necessary. For example, accidental infinite loops occur seldom because programmers have learned to avoid them. Instead of guarantees of everything we often want to have a combination of

- static guarantees where it is possible to give them without restricting flexibility (or with minor restrictions for properties that are essential in most applications),
- warnings where programmers shall have a closer look to non-local code,
- and design rules (that can easily be obeyed and deliberately be ignored).

We propose to deal with properties related to continuity as follows:

- No token of the form $x{:}\sigma?\tau$ ever gets lost. While static type checking can easily ensure this property for tokens associated with parameters and local variables in the normal case, it is very difficult to give static guarantees in exceptional cases and for tokens associated with instance variables. For example, tokens get lost when an out-of-memory exception occurs or garbage collection deletes an object. In the former case we use an escape strategy as shown below. To deal with the latter case we propose that the compiler gives warnings whenever non-implicit tokens of the form $x{:}\sigma?\tau$ get associated with instance variables or class variables.
- The owner of a token $x{:}\sigma?\tau$ repeatedly invokes methods using this token or moves the token to another method that does so. The compiler can check if a method contains corresponding code and give a warning otherwise (except for tokens associated with instance variables and class variables – another reason to issue a warning). However, even if there is no warning it is up to the programmer to ensure that this code is actually executed. Furthermore, the programmer must ensure that tokens do not move in cycles instead of being used in real work.
- If there is a token $t_1 = x{:}\sigma?\tau$ (with σ different from the declared type of x) in an annotation of a reference, then there is also a token $t_2 = x{:}\rho?\sigma'$ with $\sigma' \leq \sigma$ anywhere in the system. The execution of a method invoked by using t_2 can wake up a thread waiting for a lock according to t_1. Provided that t_2 does not get lost, this property is ensured by the rules in Table 3 and by the way how creators introduce tokens: The initial token for x can only be of the form $x{:}\tau'?\tau'$ or $x{:}\tau''$ which is then replaced by $x{:}\tau'?\tau'$. Further tokens for x can be introduced only by applying the last rule in Table 3 which ensures that there are always tokens with cycles in types as in $x{:}\tau_1?\tau_2, \dots, x{:}\tau_{n-1}?\tau_n, x{:}\tau_n?\tau_1$. By subtyping the types to the right of "?" can only become more concrete, those to the left less concrete.
- Method invocations using t_1 and t_2 (as above) do not block each other because of nonterminating computations. The execution of each method invoked using these tokens must terminate in finite (and for practical reasons short) time. Programmers have to ensure that there are no infinite loops in such method executions.
- Method invocations using t_1 and t_2 do not block each other because of missing

concurrency. The invocations shall occur in different threads. The compiler shall warn if two different ?-tokens of the same variable name occur in the same method. Because of insufficient aliasing information there will be false positives, that is, the variables in the tokens can belonging to different objects. Furthermore, in some cases it is useful for t_1 and t_2 to temporarily occur in the same method, for example in an initialization phase while starting concurrent threads.

- Method invocations using t_1 and t_2 do not block each other because of a deadlock. The compiler shall give a warning where deadlocks are possible:
  - For each method m the compiler computes the set S_m of tokens consisting of each required token of the form $x{:}\sigma$ (or $*x{:}\sigma$) in the access constraint or the annotation of a formal parameter of m for which there is an ensured token $x{:}\tau$ (or $*x{:}\sigma$) in the same access constraint or annotation.
  - For each method m the compiler computes the set T_m of tokens consisting of each required token of the form $x{:}\sigma?\tau$ in the access constraint or the annotation of a formal parameter of m and each implicit token possibly needed while executing m.
  - The compiler gives a warning if for a method m there is an $x{:}\tau$ or $*x{:}\tau$ in S_m and a $y{:}\sigma?\rho$ in T_m where x does not precede y in the order of variable names (as assumed given in Section 2.4).

  Without warning there can be no deadlock because all locks must be acquired in a specific order; there can be no cycles. However, the compiler will find false positives (as above), and complete avoidance of cycles is a very restrictive property. Sometimes developers prefer possible (but unlikely) deadlocks over very expensive program refactoring to avoid cycles.

The compiler can perform most checks using just information visible within a class. Only in two cases related to deadlock prevention we need a global program analysis:

- To keep the probability of false deadlock warnings as small as possible we can pinpoint the global order of variable names only when we know all dependences.
- To compute the set T_m we have to determine all (not only local) implicit tokens possibly needed while executing m.

We hope to be able to develop incremental techniques for these cases in future work. Otherwise it may often be preferable to switch off deadlock prevention.

Sometimes we have to terminate computations that otherwise continue forever. In this case we need just the opposite of continuity. We propose a simple solution by disallowing further synchronization using a specific variable: The invocation of a designated method with x as parameter causes x to enter *stop mode* – a specific kind of lock – and afterwards each thread that wants to acquire a lock on x gets an exception instead. A variable x automatically enters stop mode if an exception causes loss of a token $x{:}\sigma?\tau$ with σ and τ different from the declared type of x.

4 Related and Future Work

In the proposed approach, concurrency is based on rather conventional threads, locks, and values of shared variables. In this respect there are many similarities with the SCOOP model of concurrency [22]. Both, the SCOOP model and our model, disclose information on the variables used for synchronization. However, the way how and the time when such information becomes available to clients is different: Every client can get access to such variables at runtime in the SCOOP model while access control in our model causes much information to be available at compilation time and usually only few clients actually access the variables at runtime. The access control mechanism adds a further dimension to concurrent programming and reduces the importance of synchronization at the presence of static information.

There are many approaches to ensure unique access [13], most of them by avoiding aliases. The token-based approach used in the present work has been developed from a process type model [26,27,28] for active objects – essentially an object-oriented variant of linear types [20]. This concept restricts the way how to access objects without preventing aliasing [31]. Tokens express all information needed to avoid synchronization conflicts.

Many proposals ensure race-free programs [4,8,15]. Some approaches depend on explicit type annotations [15] while others perform type inference [4]. Such techniques can lead to more locks because no approach accurately decides between necessary and unnecessary locks. Program optimization can remove some unnecessary locks [9,34]. Unfortunately, we usually must analyze complete programs for good results.

There is also much work on deadlock prevention [1,15,19]. A major problem of all such proposals is that static deadlock prevention considerably affects the flexibility of the language. The approach proposed in the present work is no exception. Therefore, we argue that a potential deadlock found by a compiler is no more than just a hint for the programmer to have a closer look into the code.

Synchronization based on tokens has a long tradition: Petri Nets have been explored for nearly half of a century as a basis of synchronization [24]. In general, expressing states by abstract tokens has clear (both practical and theoretical) advantages over expressing them more concretely by values in instance variables: Tokens are much easier tractable than concrete states especially when used in a static analysis. Many proposals use tokens to express abstract object states [6,10,33]. In our approach we combine abstract tokens (giving us tractable static access control) with concrete variable values (for dynamic synchronization).

Goals of the author's early work on process types [26,27] were to express information about synchronization in an Actor-like language [2] in types and to consider synchronization in subtyping. Because of the semantic simplicity of the Actor model it was rather easy to achieve these goals. Later attempts to get similar results for Java-like concurrent languages turned out to be much more difficult, not only because of higher semantic complexity, but also because these languages consider concurrency to be independent of the object model. A concept similar to the one expressing synchronization in the Actor-like language became more useful as an access control mechanism, and a new concept for synchronization was needed [30].

There are several approaches related to process types. Especially linear types [20] based on the π-calculus [23] are well-known. Since there is no natural notion of message acceptability in the π-calculus as it is in the Actor model, static checking of linear types has to prevent deadlocks and thus is more restrictive than process type checking that can ensure message acceptability without preventing deadlocks.

The Fugue protocol checker [10,11] uses a different approach to specify client-server protocols: Rules for using interfaces are recorded as declarative specifications. These rules can constrain the order of method invocations as well as specify pre- and post-conditions. Tokens in this protocol checker represent typestates. Other than in process types and in the approach proposed in this paper, tokens can be used only for unique references. Since there is no concept resembling type splitting (as in process types) and no concept of token duplication (as used for ?-tokens in our approach), the Fugue protocol checker cannot statically ensure all methods to be invoked in specified orders at the presence of aliasing. In these cases the checker introduces pre- and post-conditions to be executed at runtime. Process types can statically ensure type safety in cases where the Fugue protocol checker can perform only dynamic checks. There is a number of further similar approaches to express (abstract) object states in types and check protocol compatibility [6,7,32,33,35].

Several programming languages [5,14,25] were developed based on the Join calculus [16]. For example, in Polyphonic C# [5] we combine methods that must be executed simultaneously to a chord which is executed as a single unit. Clients can see how methods in a chord are synchronized. Since only one method in a chord is executed synchronously and all other methods are asynchronous, only specific forms of synchronization are supported. Communication in Polyphonic C# and similar languages resembles that of the rendezvous concept in Ada [18]. There is no way to constrain method invocations as with our token concept, and there is no obvious way to use chords in controlling aliasing.

In the author's previous work on token-based synchronization, tokens always have been separate entities not carrying values and without relationship to variables. It is a major new contribution to regard tokens as static abstractions of concrete variables that can be locked. The variables provide the missing link between static tokens and dynamic synchronization. Different kinds of tokens were developed as a direct consequence. Those presented here turned out to be useful, some other considered kinds were rejected because they either were not consistent with more important kinds or turned out to be less useful.

In previous work the author often used dependent types (these are types depending on values) to increase the flexibility of the system. In the present work we avoid dependent types as much as possible because they are difficult to deal with. We get the necessary flexibility by dynamically changing tokens based on results of dynamic type queries (`instanceof`).

In contrast to those in Java, threads in our approach need not have any identity:

The thread that acquired a lock is in no way privileged compared to other threads because only the (statically checked) availability of tokens counts, not the thread identity. This property is an important step towards modularity of synchronization. Locking variables instead of objects represents a further step: We keep concurrency independent from object-oriented factorization. Concerning modularity it is important not to distinguish between access control and synchronization in methods: The code is rather stable because most likely we need not change a method if we replace synchronization with access control or move the point of synchronization.

The present work is in an early stage. There exist only fragments of a prototype implementation, and a rigorous practical evaluation of the proposed approach has not yet been carried out. An evaluation is important future work. In further future work we want to address (among others) the following topics:

- Currently we use types of variables to express object states in tokens, and changes of types to express state changes. We want to investigate more elaborate ways of expressing object states and state changes allowing us to encode arbitrary pre-conditions and more precisely defined state modifications into tokens.
- As discussed in this article we have to analyze the whole program to find an appropriate ordering of variable names needed for deadlock prevention. We want to develop appropriate annotations that support separate compilation of classes.
- We need a more advanced concept to deal with tokens in exception handling.
- There are approaches to concurrent object-oriented programming that avoid low-level locking and still ensure atomicity [17]. We want to apply such techniques in combination with our approach.

5 Conclusions

Synchronization and access control ensuring exclusive access to shared variables fit together quite naturally. We explored an approach to integrate synchronization into a static access control mechanism based on tokens giving information about exclusive access to variables and supposed types of their values. It is possible and beneficial to annotate methods with access constraints not distinguishing between static access control and dynamic synchronization. Clients decide if they prefer access control or synchronization. Object interfaces provide all the information that clients need to avoid conflicts of dependent synchronization, and simple mutex synchronization can be regarded as an implementation detail. Static type checking guarantees exclusive write-access and consistent read-access to shared variables and thereby ensures race-free synchronization. We get flexibility by using dynamic type information in tokens. The approach supports subtyping and ensures to some extent continuous operation of the system.

References

[1] Abramsky, S., S. Gay and R. Nagarajan, *A type-theoretic approach to detect deadlock-freedom of asynchronous systems*, in: *Theoretical Aspects of Computer Software*, Lecture Notes in Computer

Science **1281** (1997), pp. 295–320.

[2] Agha, G., I. A. Mason, S. Smith and C. Talcott, *Towards a theory of actor computation*, in: *Proceedings CONCUR'92*, Lecture Notes in Computer Science **630** (1992), pp. 565–579.

[3] Aldrich, J., C. Chambers and D. Notkin, *Archjava: Connecting software architecture to implementation*, in: *Proceedings of the 24th International Conference on Software Engineering*, ACM, Orlando, Florida, 2002, pp. 187–197.

[4] Bacon, D. F., R. E. Strom and A. Tarafdar, *Guava: A dialect of Java without data races*, in: *OOPSLA 2000*, 2000.

[5] Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for C#*, ACM Transactions on Programming Languages and Systems **26** (2004), pp. 269–804.

[6] Bierhoff, K. and J. Aldrich, *Lightweight object specification with typestates*, in: *ESEC/FSE-13* (2005), pp. 217–226.

[7] Boyland, J., *Checking interference with fractional permissions*, in: R. Cousot, editor, *Static Analysis: 10th International Symposium*, Lecture Notes in Computer Science **2694** (2003), pp. 55–72.

[8] Brinch-Hansen, P., *The programming language Concurrent Pascal*, IEEE Transactions on Software Engineering **1** (1975), pp. 199–207.

[9] Choi, J.-D., M. Gupta, M. Serrano, V. C. Sreedhar and S. Midkiff, *Escape analysis for Java*, in: *OOPSLA'99*, Denver, Colorado, 1999.

[10] DeLine, R. and M. Fähndrich, *The fugue protocol checker: Is your software baroque?*, Technical report, Microsoft Research (2004), http://www.research.microsoft.com.

[11] DeLine, R. and M. Fähndrich, *Typestates for objects*, in: *ECOOP 2004 – Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science (2004).

[12] Dietl, W., S. Drossopoulou and P. Müller, *Generic universe types*, in: E. Ernst, editor, *ECOOP – Object-Oriented Programming*, Lecture Notes in Computer Science **4609** (2007).

[13] Drossopoulou, S., D. Clarke and J. Noble, *Types for hierarchic shapes*, in: *ESOP*, 2006, pp. 1–6.

[14] Drossopoulou, S., A. Petrounias, A. Buckley and S. Eisenbach, *School: A small chorded object-oriented language*, in: *Proceedings of ICALP Workshop on Developments in Computational Models*, 2005.

[15] Flanagan, F. and M. Abadi, *Types for safe locking*, in: *Proceedings ESOP'99*, Amsterdam, The Netherlands, 1999.

[16] Fournet, C. and G. Gonthier, *The reflexive cham and the join-calculus*, in: *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, 1996, pp. 372–385.

[17] Harris, T. and K. Fraser, *Language support for lightweight transactions*, in: *OOPSLA'03*, ACM, Anaheim, California, USA, 2003, pp. 388–402.

[18] ISO/IEC 8652:1995, *Annotated ada reference manual*, Intermetrics, Inc. (1995).

[19] Kobayashi, N., *A partially deadlock-free typed process calculus*, ACM Transactions on Programming Languages and Systems **20** (1998), pp. 434–482.

[20] Kobayashi, N., B. Pierce and D. Turner, *Linearity and the pi-calculus*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 914–947.

[21] Liskov, B. and J. M. Wing, *Specifications and their use in defining subtypes*, ACM SIGPLAN Notices **28** (1993), pp. 16–28, proceedings OOPSLA'93.

[22] Meyer, B., "Object-Oriented Software Construction," Prentice Hall, 1997, second edition edition.

[23] Milner, R., *The polyadic π-calculus: A tutorial*, in: *Logic and Algebra of Specification* (1992), pp. 203–246.

[24] Murata, T., *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE **77** (1989), pp. 541–580.

[25] Odersky, M., *Functional nets*, in: *Proceedings of the European Symposium on Programming* (2000).

[26] Puntigam, F., *Type specifications with processes*, in: *Proceedings FORTE'95*, IFIP WG 6.1 (1995).

[27] Puntigam, F., *Coordination requirements expressed in types for active objects*, in: M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP'97*, Lecture Notes in Computer Science **1241** (1997), pp. 367–388.

[28] Puntigam, F., "Concurrent Object-Oriented Programming with Process Types," Der Andere Verlag, Osnabrück, Germany, 2000.

[29] Puntigam, F., *From static to dynamic process types*, in: *ICSOFT 2006, First International Conference on Software and Data Technologies (1)* (2006), pp. 21–28.

[30] Puntigam, F., *Internal and external token-based synchronization in object-oriented languages*, in: *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006*, Lecture Notes in Computer Science **4228** (2006), pp. 251–270.

[31] Puntigam, F., *See the pet in the beast: How to limit effects of aliasing*, in: *International Workshop on Aliasing, Confinment and Ownership in object-oriented programming (IWACO 2007)*, Berlin, Germany, 2007.

[32] Strom, R. E. and D. M. Yellin, *Extending typestate checking using conditional liveness analysis*, IEEE Transactions on Software Engineering **19** (1993), pp. 478–485.

[33] Strom, R. E. and S. Yemini, *Typestate: A programming language concept for enhancing software reliability*, IEEE Transactions on Software Engineering **12** (1986), pp. 157–171.

[34] von Praun, C. and T. R. Gross, *Static conflict analysis for multi-threaded object-oriented programs*, in: *PLDI '03* (2003), pp. 115–128.

[35] Yellin, D. M. and R. E. Strom, *Protocol specifications and component adaptors*, ACM Transactions on Programming Languages and Systems **19** (1997), pp. 292–333.