

Internal and External Token-Based Synchronization in Object-Oriented Languages

Franz Puntigam

Technische Universität Wien, Institut für Computersprachen
Argentinerstraße 8, 1040 Vienna, Austria
franz@complang.tuwien.ac.at

Abstract. We expect interfaces in programming languages to expose essential parts of the objects' internal synchronization as well as required external synchronization. Clients need this information to provide required and avoid conflicting synchronization. We propose a mixed static and dynamic token-based approach to uniformly specify internal and external synchronization in a simplified Java-like language. This concept gives us much flexibility on token management, ensures race-free programs without any need for complete aliasing information, and supports static type checking of synchronization using a rich notion of subtyping.

1 Introduction

Synchronization is essentially a means to enforce data dependences in control flows: If there is a dependence between data accesses in two threads of control, then one of the threads must wait until the other has caught up to meet at a synchronization point where data are consistent. Providing proper synchronization is difficult. Omitted synchronization statements can from time to time result in reading inconsistent data (these are races). It is very difficult to catch such bugs. Programmers often apply synchronization and restrict concurrency much more than necessary to prevent the danger of races and because of missing knowledge about data dependences. Unnecessary synchronization affects program efficiency and increases the danger of deadlocks and other undesirable program behavior.

Good knowledge of the whole program and its synchronization structure reduces the problem. Many programs consist of independently developed components, each treating synchronization in its own way. Component interfaces are the best place to provide synchronization information. Substituting a component for another one requires synchronization in the two components to be compatible, this is, their interfaces must be in a subtype relationship. In this paper we deal with synchronization information in interfaces and subtype relations as well as static type checking to ensure race-free programs. We explore interfaces of simple objects, but all results hold also for component interfaces.

Most programming languages support internal synchronization within objects, while required external synchronization (to be provided by clients) can be expressed only as comments. Since especially required external synchronization is important to clients and for subtyping, we propose language support for it.

An important topic is the representation of synchronization information. Direct representation of dependences is hardly tractable and exposes too many implementation details to clients. Therefore, we prefer an abstract representation based on tokens as proposed for process types [1,2]. This type system allows us to specify required external synchronization in types and statically ensures that clients observe required dependences.

In Sect. 2 we analyze internal and external synchronization and introduce a corresponding language. Next, in Sect. 3 we propose a concept of internal synchronization resembling more conventional synchronization, and in Sect. 4 we address the problem of encoding changing synchronization information into the types of instance variables. We consider synchronization information together with subtyping in Sect. 5. A discussion of related work follows in Sect. 6.

2 The Basis: Synchronization, Tokens, and Language

We develop a simple Java-like language using tokens for synchronization as basis for further analysis. First, we explore the synchronization information we need in interfaces. Then, we introduce the language mainly by examples.

2.1 Responsibility for Synchronization

We use a broad notion of synchronization and distinguish between the following kinds based on the roles of objects as clients and servers and their responsibilities:

Internal Synchronization: This is any means to enforce data dependences in control flows within an object as server – synchronization in a narrow sense. Thereby, the execution of a thread can be blocked. In Java we use `synchronized` methods (or statements) together with `wait` and `notify` for internal synchronization. For example, `java.util.Hashtable` uses internal synchronization to ensure consistent updates of its instances. Internal synchronization determines when an invoked method gets executed. Only the object itself is responsible for providing this kind of synchronization.

Required External Synchronization: This is synchronization an object requires to be provided by each client. A method can be invoked only if a synchronization condition is satisfied, and within the method this condition is regarded as satisfied. For example, `java.util.HashMap` requires external synchronization to ensure consistent updates. Required external synchronization determines when a method can be invoked. All clients are responsible for providing required synchronization.

Provided External Synchronization: A client of the object under consideration provides this synchronization. Regarded as server a client does so by internal or external synchronization, or simply by invoking methods in a specific order. Clients usually provide more synchronization than required. Unrequired external synchronization can be in conflict with internal synchronization. For example, we get a deadlock if a simple buffer with a single

slot uses internal synchronization to ensure `put` and `get` to be executed only in alternation, and clients provide external synchronization by invoking `get` only after `put` has been executed twice. Provided external synchronization determines if and when an invocable method actually is invoked.

Clients must know the server's internal synchronization that can be in conflict with provided synchronization. Fortunately, not every internal synchronization can be in conflict with provided synchronization:

Simple Mutex Synchronization: This kind of internal synchronization ensures just that only a single thread can execute a critical section at any time. The execution of the critical section must terminate, and the synchronization condition must not depend on anything else than the number of threads in the critical section. Examples are `synchronized` methods in Java that do not invoke `wait`. It is a good idea to demand such critical sections to terminate in a short period of time; otherwise the execution of the whole system can be blocked. Simple mutex synchronization can delay execution, but cannot prevent it. Hence, simple mutex synchronization is not in conflict with provided external synchronization and need not be visible to clients.

Dependent Internal Synchronization: Synchronization conditions depend on the program state in a more complex way. For example, `synchronized` methods in Java invoking `wait` depend on an invocation of `notify` in another thread. Clients must know about such dependences to avoid conflicting synchronization (through simultaneous executions of methods invoking `notify`).

In simple cases like buffers we usually prefer internal synchronization where only the server decides when to perform an operation. External synchronization gives clients better control. Clients having sufficient information about dependences can get better performance from external than from internal synchronization.

There are useful relationships between the different kinds of synchronization. For example, a proxy as client of an object can convert the object's required external synchronization into internal synchronization in the proxy. Programmers can decide case by case whether they prefer direct access to an object with required external synchronization or indirect access through a proxy providing the required synchronization. Because of such techniques servers requiring external synchronization give clients all possibilities.

2.2 A Simple Language with Tokens

In our language we use tokens as proposed for process types [1,2]. This type system statically ensures that users observe all required external synchronization conditions without any need for complete aliasing information.

Fig. 1 shows the grammar of our language. Throughout this paper, u, \dots, z (possibly quoted and indexed) denote names, t token specifications, c pre- and post-condition pairs on tokens, d dependent tokens, τ types, p formal parameters, s statements, e expressions, and i, \dots, n natural numbers (including zero and the special symbol ∞). We differentiate between classes and interfaces as named

$$\begin{aligned}
P &::= gdef^+ \\
gdef &::= [t^*] \text{interface } u \{ decl^* \} \mid [t^*] \text{class } u \{ ldef^+ \} \\
decl &::= \tau x(p^*) \text{ with } c_1 \text{ when } c_2 \mid \text{void } x(p^*) \text{ with } c_1 \text{ when } c_2 \\
ldef &::= \tau d^* v \mid decl \{s^*\} \mid \text{new}(p^*) \text{ with } \rightarrow t^* \{s^*\} \\
d &::= [t_1^+ \text{ for } t_2^+] \\
p &::= u[c] v \\
c &::= t_1^* \rightarrow t_2^* \\
t &::= (n)x \\
\tau &::= u[t^*] \\
s &::= v = e \mid v.x(w^*) \mid \text{fork } v.x(w^*) \mid \text{with } c \{s^*\} \mid \text{return } e \\
e &::= v \mid v.x(w^*) \mid u.\text{new}(w^*) \mid \text{null}
\end{aligned}$$
Fig. 1. Syntax of our Language

basic units in a program. Subtyping is inferred from the structure of classes and interfaces; there are no explicit supertype specifications. For syntactic simplicity we avoid commas and semicolons as separators. To create a new object we invoke a constructor (beginning with `new`) in a class. As shown by the examples below, initial internal tokens (those specified in front of the `class` or `interface` key word) and `when`-clauses determine internal synchronization. Tokens associated with types and `with`-clauses determine required external synchronization.

In our examples we take the liberty to omit `with`-clauses, `when`-clauses, and square brackets not containing any tokens. We often write just x instead of $(1)x$ (one token of name x) and usually avoid to write down $(0)x$ (no token of name x). The first example shows how to specify required external synchronization:

```

interface Window {
    void iconify() with displayed → iconified
    void display() with iconified → displayed
}

```

We assume windows to be displayed on a screen or just represented by icons. Two methods switch between these states. According to the `with`-clause we can invoke `iconify` only if we have a token `displayed`; this token is removed on invocation, and `iconified` is added on return. For a variable v of type `Window[displayed]` we can invoke `v.iconify()`. This invocation changes the type of v to `Window[iconified]`. Afterwards we can invoke `v.display()`, then again `v.iconify()`, and so on. Simple static type checking enforces the methods to be invoked only in alternation. Typically a button causing an icon to be replaced with a displayed window does not exist at the same time as one causing the window to become an icon. In this case it is very natural to statically express the expected state of the window in its type, and there is no need for synchronization at run time.

Types can be associated with several tokens. For example, let an instance of `Window[(8)displayed (7)iconified]` be a window manager iconifying and displaying any of its at least 15 windows. This object accepts `iconify` and `display` in all sequences such that never more than 15 or less than zero windows are displayed.

2.3 Moving Static Tokens Around

We handle tokens in parameter types in a similar way as in `with`-clauses:

```
class Parameters {
    void foo(Window[displayed iconified → (2)displayed] w) {
        w.iconify() w.display() w.display()
    }
}
```

Arrows in parameter types relate tokens required on invocation with those available on return. Let `v` be a variable of type `Window[(2)iconified (2)displayed]` and `x` one of `Parameters`. An invocation of `x.foo(v)` first iconifies a window and then displays two of them. The variable `w` in `foo` is known to have at least a `displayed` and a `iconified` token on invocation and two `displayed` tokens on return. Removing the tokens to the left of the arrow on invocation causes the type of `v` to become `Window[displayed iconified]`, and adding those to the right on return causes it to become `Window[(3)displayed iconified]`. Tokens move from argument types to formal parameter types on invocation and vice versa on return. Only `with`-clauses (and for this section `when`-clauses, see below) add and remove tokens. Each object produces and consumes only its own tokens.

A statement `'fork x.foo(v)'` spawns a new thread executing `x.foo(v)`. Since execution continues without waiting for termination of the new thread, `fork` statements cannot return result values and tokens. The type of `v` changes on invocation from just to `Window[displayed iconified]`. Everything to the right of the arrow in the formal parameter type is ignored. Thereby, the old type of `v` is split into two types – the new type of `v` and the type of the formal parameter `w`. Both threads can invoke methods in the same object without affecting each other concerning type information.

Assignment resembles parameter passing when spawning threads: We split an assigned value's type into two types. One type becomes the variable's type, and the other becomes the assigned value's new type. Tokens move from the value's to the variable's type. For example, if a variable `w` is expected to be of type `Window[(2)iconified]` and `v` is of `Window[(2)displayed (2)iconified]`, then the execution of `'w = v'` causes the type of `v` to become `Window[(2)displayed]`.

`With`-clauses in constructors play an important role in introducing tokens:

```
class MyWindow {
    int test
    void iconify() with displayed → iconified { test = 0 }
    void display() with iconified → displayed { test = 1 }
    new() with → displayed { test = 1 }
}
```

An invocation of `MyWindow.new` returns a new object with a single token. No other token is available for this object. Since invocations of `iconify` and `display` consume a token before they issue another one, there exists always at most one token. This property ensures that each method invocation switches the value of `test` between 0 and 1, and there cannot be simultaneous accesses of `test`.

2.4 Simplified Internal Synchronization

For the rest of this section we use a simplified view of internal synchronization, and we will reconsider it in the next section. We associate each object with a dynamically manipulated pool of internal tokens. Classes and interfaces specify initial internal tokens of new instances (in front of the key words `class` and `interface`). Tokens to the left of the arrow in `when`-clauses must be available and are removed from this pool before executing the method body, and tokens to the right are added on return. If required tokens are not available, then the execution is blocked until other threads cause the tokens to become available. Checks for the availability of these tokens occur only at run time.

The following buffer example uses internal synchronization to ensure mutual exclusion and to avoid buffer overflow and underflow:

```
[sync (10)empty] class BufferDyn {
    ListElem head
    new() { head = null }
    void increment() when → (10)empty {}
    void decrement() when (10)empty → {}
    void put(Elem e) when sync empty → sync filled { /* add to list */ }
    Elem get() when sync filled → sync empty { /* get from list */ }
}
```

Instances get some `empty` tokens and a single token `sync` on creation. Both `put` and `get` remove `sync` at the begin and issue a new one on return and thereby ensure exclusive access to `head`. Tokens `filled` and `empty` dynamically ensure that a buffer never contains more than the maximum or less than zero elements. Execution blocks until these conditions are satisfied. The maximum capacity of the buffer can be changed by (repeatedly) invoking `increment` and `decrement`.

2.5 Usual Uses of Tokens and Infinity

The above examples use tokens as counters in a similar way as semaphores are essentially counters, no matter whether we use internal or external synchronization. More often we use tokens as binary semaphores where at most one token of some name can exist for each object. Names of such tokens usually abstract over (specific aspects of) the objects' current states as in the next example:

```
class ShowStates {
    new() with → justCreated {...}
    void init1() with justCreated → partlyInitialized {...}
    void init2() with partlyInitialized → (∞)ready {...}
    void doSomething() with ready → {...}
}
```

Immediately after creation an instance is in state `justCreated`. On execution of `init1` (which can occur at most once) the state changes to `partlyInitialized`, and on executing `init2` (at most once) to `(∞)ready` – an unlimited number of tokens

ready. We can invoke `doSomething` as often as we want, even simultaneously. As with-clause of `doSomething` we can have $(n)\text{ready} \rightarrow (m)\text{ready}$ with any $n \geq 1$ and $m \geq 0$ without changing the semantics.

To meet our expectations we define (for all natural numbers n including 0 and ∞) $\infty \geq n$ to be true, $\infty + n = \infty$, and $\infty - n = \infty$ (implying $\infty - \infty = \infty$).

As in the example we usually use unlimited numbers of tokens to indicate that corresponding (aspects of) object states do not change anymore. Major reasons for using ∞ instead of 1 as token numbers include

- no limitation of simultaneous execution,
- and type splitting (for external synchronization) where each of the split types has full information about available tokens.

In type splitting for assignment and forking we usually must decide which client gets which tokens. With unlimited token numbers all clients can have complete information. For example, `ShowStates[(∞)ready]` can be split into twice the same type because of $\infty - \infty = \infty$.

3 Internal Synchronization Reconsidered

3.1 Atomic Actions and Their Problems

Token-based internal synchronization as introduced in the previous section and in [3] has desired properties especially for dependent internal synchronization (see Sect. 2.1) and some weaknesses for simple mutex synchronization and in relating internal with required external synchronization:

Atomic Actions: When-clauses (and with-clauses) specify atomic actions. Synchronization is guaranteed by removing tokens at the begin and adding new tokens only at the end. Atomic actions are quite valuable in programming because they clearly specify points in the program where we expect object states to be consistent. Between these points we have to regard states as inconsistent. This principle is enforced also for nested (possibly recursive) method invocations. Unfortunately, if the `when`-clause of the outer method invocation removed tokens needed by the inner invocation, then the execution can easily be in a deadlock: The outer invocation waits for termination of the inner invocation, and the inner invocation waits for tokens to be added at the end of the outer invocation. In general, such situations are erroneous: We expect the object to be in a consistent state on invocation of the inner method, but actually it is possibly in an inconsistent state. Therefore, program termination or raising an exception is useful program behavior in this case. However, if we use tokens just to ensure mutual exclusion (this is simple mutex synchronization), then missing tokens do not imply inconsistent states. Provided that the missing tokens were removed within the same thread, we expect the execution to continue in this case. To improve the model we must distinguish between dependent internal synchronization and simple mutex synchronization.

This: Tokens for internal and external synchronization are strictly separated: Only with-clauses manipulate external tokens, and only when-clauses internal tokens. The self-reference `this` naturally breaks the separation: `This` implicitly refers to an instance of the most specific type of the object it occurs within. We can regard internal tokens to be at the same time tokens in the implicit type of `this`. When invoking a method through `this`, the with-clause of the invoked method takes required tokens from the pool of internal tokens and puts added tokens back to this pool. The use of `this` as formal parameter can also manipulate internal tokens. Unfortunately, requiring internal tokens from the pool can block the execution (if required tokens are not available). Thereby, synchronization points occur within atomic actions – not necessarily at the begin of atomic actions. We regard such blocking as undesirable. To improve the model we must statically ensure availability of all tokens needed in the implicit type of `this`.

3.2 Thread-Specific Token Pools

With a more advanced concept of internal synchronization we treat both problems: In addition to the general pool of internal tokens in each object we use a token pool per thread and object. We redefine the semantics of when-clauses such that they just move tokens from the general token pool to the thread-specific token pool on invocation and vice versa on return instead of removing and adding them. There is no need to wait for the availability of tokens in the general pool if they are already in the thread-specific pool. Only tokens in the thread-specific pool are regarded as available in the implicit type of `this`. The following abstract example demonstrates the new semantics of when-clauses:

```
[sync] class InternExtern {
    void makeIntern() when sync → sync dep { ... this.makeExtern() ... }
    void makeExtern() with → dep when sync → sync {...}
    void useIntern() when sync dep → sync { ... this.useExtern() ... }
    void useExtern() with dep → when sync → sync {...}
    ...
}
```

We assume that each of the four methods accesses a shared critical resource protected by an internal token `sync` that ensures mutual exclusion. With a token `dep` we let `useExtern` only be invocable as often as `makeExtern` was invoked before. An invocation of `makeIntern` causes the token produced in an invocation of `this.makeExtern` to become available as internal token, and an invocation of `useIntern` consumes an internal token `dep` in an invocation of `this.useExtern`.

On invocation of `makeIntern` the when-clause first looks if `sync` is available in the thread-specific token pool. In this case execution immediately continues with the method body. Otherwise execution is blocked until `sync` becomes available in the general pool; then `sync` will be removed from the general pool and added to the thread-specific pool before execution continues. When invoking `makeExtern`

through this in the body of `makeIntern`, `sync` will be available in the thread-specific pool and the body of `makeExtern` can be executed without delay. On return from `makeExtern` a new token `dep` specified in the `with`-clause is added to the thread-specific pool because the tokens in the implicit type of `this` correspond to the thread-specific pool. On return from `makeIntern` the tokens `sync` and `dep` must be available in the thread-specific pool as specified to the right of the arrow in the `when`-clause. Then, `dep` is removed from the thread-specific pool and added to the general pool in any case, and `sync` is removed from the thread-specific and added to the general pool only if it was not available in the thread-specific pool when invoking `makeIntern`; otherwise `sync` remains in the thread-specific pool.

On invocation of `useIntern` the token `dep` must be moved from the general to the thread-specific pool even if there is already such a token: We expect the thread-specific pool to contain the same tokens on invocation and return. Since the execution of `useIntern` just removes `dep`, this token would not be in the thread-specific pool on return if it was taken from this pool.

Note that we use `sync` in `makeExtern` and `useExtern` only to demonstrate nesting of `when`-clauses. We omit them where we need no mutual exclusion.

Quite often methods like `makeExtern` and `useExtern` need not be used from outside. In this case we can simplify the syntax by using `with`-statements instead of `with`-clauses in methods:

```
void makeIntern() when sync → sync dep { ... with → dep {...} ... }
void useIntern() when sync dep → sync { ... with dep → {...} ... }
```

`With`-statements are essentially syntactic sugar inlining methods that would otherwise be invoked through `this`. Tokens in `with`-statements are taken from and added to the thread-specific pool.

3.3 New Semantics of `when`-Clauses

We differentiate between three kinds of tokens in `when`-clauses:

Remove-Tokens occur only to the left of the arrow.

Add-Tokens occur only to the right of the arrow.

Through-Tokens occur on both sides of the arrow.

This is the new semantics of the `when`-clause in each method:

- On method invocation the `when`-clause moves these tokens from the general to the thread-specific pool of the current thread:
 - all remove-tokens and
 - through-tokens not yet being in the thread-specific pool.
 Execution blocks until all tokens to be moved are simultaneously available in the general pool. Then, these tokens are moved atomically. To be concrete, they must be removed from the global pool in an indivisible step while adding them to the thread-specific pool need not be atomic.
- On return from the method the `when`-clause moves these tokens from the thread-specific to the general pool (not necessarily atomically):

- all add-tokens and
 - through-tokens moved to the thread-specific pool on method invocation.
- Static type checking ensures that all tokens occurring to the right of the arrow will be available in the thread-specific pool on return from the method if tokens to the left were available on method invocation. To perform static type checking we treat **this** as any other instance variable and assume the tokens to the left of the arrow in the **when**-clause as being available in the type of **this** at the begin of checking the method. After checking the method, the type of **this** must contain at least all tokens to the right. Each instance variable is assumed to have some tokens at the begin and some (other) tokens at the end of method checking (see Sect. 4).

We regard tokens of some name as relevant for internal synchronization in a class or interface if at least one token of this name occurs to the left of the arrow in any **when**-clause in the class or interface. Accordingly we regard tokens as relevant for required external synchronization if at least one token of this name occurs to the left of the arrow in any **with**-clause. All tokens occurring in **when**-clauses must be relevant for internal synchronization. Other tokens cannot influence internal synchronization. In contrast, tokens (to the right of the arrow) in **with**-clauses must be relevant for internal or required external synchronization: Methods invoked through **this** can add such tokens to the internal token pool.

The semantics of **when**-clauses imposes a natural differentiation between tokens possibly relevant for dependent internal synchronization and those relevant only for simple mutex synchronization: Tokens relevant for internal synchronization are possibly relevant for dependent internal synchronization if they

- occur as remove-tokens,
- or are also relevant for required external synchronization.

Add-tokens are necessarily relevant for both internal and required external synchronization; otherwise they cannot become available in the type of **this**. All other tokens relevant for internal synchronization (they can only be through-tokens) are relevant only for simple mutex synchronization. Such tokens need not be considered in subtyping (see Sect. 5). In all examples used so far, **sync** is relevant only for internal synchronization while internal tokens of other names are possibly relevant for dependent internal synchronization.

The semantics of **when**-clauses is compatible with semantics of more conventional synchronization concepts: Synchronized methods in Java correspond essentially to methods with **when**-clauses containing only a single token relevant for simple mutex synchronization (monitor concept). The **wait**-operation can easily be simulated using remove-tokens, and the **notify**-operation with add-tokens.

4 Accessing Instance Variables

Instance variables are declared only once, but they can be accessed (possibly simultaneously) in several methods, each expecting different tokens to be available

in the variables' types at different points in time. Uses of variables can cause changes of the tokens encoded into types. In this section we address corresponding problems and show how to avoid races.

4.1 Read-Access to Instance Variables with Dependent Tokens

In Sect.2.1 we mentioned that a proxy can convert required external synchronization into internal synchronization. When implementing such synchronization proxy we face a problem: We need an instance variable referring to the object requiring external synchronization. Tokens in the type of this variable change over time as does the state of the object. Because of internal synchronization in the proxy the tokens in the variable's type depend on internal tokens available in the proxy. The next example shows the implementation of a buffer and a proxy using dependent tokens to connect the external tokens of the buffer with the internal tokens of the proxy:

```
interface BufferStat {
    void put(Elem e) with empty → filled
    Elem get() with filled → empty
}
[sync] class BufferStatImpl {
    ListElem head
    new() { head = null }
    void increment() with → (10)empty {}
    void put(Elem e) with empty → filled when sync → sync { /* add to list */ }
    Elem get() with filled → empty when sync → sync { /* get from list */ }
}
[(50)pe] class Proxy {
    BufferStat[empty for pe][filled for pf] buffer
    new(BufferStat[(50)empty →]) { buffer = b }
    void put(Elem e) when pe → pf { with pe → pf { buffer.put(e) } }
    Elem get() when pf → pe { with pf → pe { return buffer.get() } }
}
```

In class `BufferStatImpl`, internal synchronization ensures mutual exclusion, and required external synchronization avoids over- and underflows. As we will see in Sect. 5 `BufferStatImpl` is a subtype of `BufferStat`, an interface not showing simple mutex synchronization.

Dependent tokens in the type of `buffer` specify that within each method in `Proxy` we assume an `empty` to be available for each `pe` in the corresponding `with`-clause, and a `filled` for each `pf`. In the `with`-statement in `put` the variable is of type `BufferStat[empty]` at the begin and of `BufferStat[filled]` at the end. An execution of `buffer.put(e)` causes the type change. In the `with`-statement in `get` the type of `buffer` changes from `BufferStat[filled]` to `BufferStat[empty]`. On return from the constructor, `buffer` must be of type `BufferStat[(50)empty]` because of the initial internal tokens (and, in general, tokens in the constructor's `with`-clause). By strictly coupling tokens in the variable's type to the containing object's initial tokens and all changes through `with`-clauses we ensure the object's tokens

to actually reflect the variable's tokens. Static type checking guarantees this property.

We need `with`-statements in `put` and `get` because dependent tokens depend only on tokens in `with`-clauses. It is impossible to assume dependences on tokens in `when`-clauses since several atomic actions can see the same tokens (occurring in a thread-specific pool) causing tokens in the variables' types to be implicitly duplicated. Token duplication would destroy soundness.

Dependent tokens support concurrent read accesses. For example, instances of `Proxy` support up to 50 simultaneous executions of `put` and `get`, each reading and changing the state encoded in the type of `buffer`.

We compute the tokens in the type of instance variables by repeatedly applying dependences (`for`-clauses) specified in variable declarations in arbitrary ordering. We start with the multi-set of tokens in the `with`-clause. If we have the tokens to the right of a `for`-clause in this multi-set, then we delete these tokens from the multi-set and add the tokens to the left of the `for`-clause to the tokens assumed to be available for the variable. We repeat this step as long as there are appropriate `for`-clauses. To ensure the results to be unique we require that if at least one token of some name occurs to the right of `for` in a dependence, then no token of this name occurs in another dependence in the same variable declaration. Furthermore, there must be at least one token to the right of `for`.

4.2 Write-Access to Instance Variables with Dependent Tokens

Dependent tokens require the absence of concurrent or overlapping accesses to instance variables when writing them. Neither `put` nor `get` can write `buffer`. When added to `Proxy` the following method writes `buffer`:

```
void update(BufferStat[(50)empty →] b) when (50)pe → (50)pe
  { with (50)pe → (50)pe { buffer = b } }
```

Because each instance of `Proxy` always has at most 50 tokens there are no tokens left that would allow `put`, `get`, or another invocation of `update` to run simultaneously. Since each instance has at most 50 tokens it is sufficient for the assigned value to provide 50 tokens `empty`. In general, we must find out

- that no concurrent access can occur when writing a variable (see Sect. 4.3),
- which tokens can be available for an object,
- and which tokens must be provided when writing to an instance variable.

There is a simple fixed-point algorithm to compute upper bounds of token sets that can become available for an object [3]. It extends initial token sets to upper bounds according to each pre-/post-condition pair in `with`-clauses where the precondition is satisfied. For example, when applied to `Proxy` the algorithm computes the set of token sets

$$\{[(i)pe \ (50 - i)pf] \mid 0 \leq i \leq 50\}$$

and applied to `BufferStatImpl` we get

$$\{[sync \ (\infty)empty \ (\infty)filled]\}.$$

The algorithm uses as input only information available in a single class (and hence supports separate compilation) and is accurate in the sense that

- if it generates a token set containing only a finite number of tokens, then we can construct clients invoking methods such that this set of tokens actually becomes available,
- and if it generates a token set containing an unlimited (∞) number of tokens, then we can select arbitrary numbers i and construct clients invoking methods such that more than i of these tokens become available.

We apply this algorithm to find out which token sets can become available for an instance of a class. From each of these sets we compute the tokens expected to be available in the type of an instance variable at the end of an atomic action (this is the end of a method or `with`-statement). Unfortunately, in general we do not know exactly which one of the token sets applies in the current situation. This is, we cannot know at compilation time (and probably we do not know at run time, too) which tokens are available for the object within the whole system. Thus, we use a conservative approximation to compute the tokens expected to be available: We compute the maximum of tokens within all sets returned by our algorithm that satisfy the precondition (this is, that contain all tokens to the left of the arrow in the `with`-clause). Static type checking ensures corresponding tokens to be available in the variable's type at the end of the method.

In instances of `Proxy` we can write to `buffer` whenever we have 50 tokens – any mixture of tokens `pe` and `pf`. For `update` we require $(50)pe$; hence, writing is actually possible. However, adding a method `increment` to `Proxy` (as in `BufferStatImpl`) would cause `update` to be no longer type-safe since in this case unlimited numbers of tokens can become available.

4.3 Variable Accesses and Race Avoidance

We must ensure not to write an instance variable simultaneously with other accesses of the same variable. A single criterion is sufficient to ensure this property: No preconditions in `with`-clauses and `when`-clauses of two methods accessing the same variable can be satisfied at the same time if at least one of the methods writes to the variable. This criterion implies race-free programs.

As basis of a corresponding analysis we use again upper bounds of token sets that can become available for an object. In contrast to the fixed-point algorithm applied in Sect. 4.2 we compute these sets by extending initial token sets according to each pre-/post-condition pair in `with`- and `when`-clauses (not just `with`-clauses). Otherwise the algorithm remains unchanged.

For (1) each instance variable in the analyzed class, (2) each method writing to this variable, and (3) each method accessing the variable (reading or writing, including the method considered in (2)) we build the union of all tokens occurring to the left of the arrow in the `with`- and `when`-clauses of the methods considered in (2) and (3). If the resulting token set is covered by a token set returned by the fixed-point algorithm, then it is possible that the methods considered in (2) and (3) run concurrently. We regard this case as a program error. Otherwise

writing of instance variables cannot occur simultaneously with other accesses to the variable because there cannot exist enough tokens allowing us to do so. We analyze each class separately. All needed information is available in the code of the class, and we need no aliasing information.

By including information in *when*-clauses we get more flexibility. To ensure the absence of races it is not necessary for atomic actions to run completely in isolation: If an action A is executed in the same thread as another thread B and B starts after A and terminates before B, then B is nested into A. In contrast to *with*-clauses, *when*-clauses support nested actions. Values written to instance variables in A before starting B are visible in B, and values written in B are visible in A after termination of B. Nonetheless, type consistency is ensured because dependent tokens depend only on preconditions of *with*-clauses which do not support nesting. It is impossible to write to the same variable in A and B because at most one of the actions can have got all tokens necessary to do so.

5 Subtyping

5.1 Definition of Subtyping with Tokens

Subtyping has to consider internal as well as required external synchronization information. If we use an instance of a subtype where an instance of a supertype was expected, then the instance of the subtype

- accepts at least all method invocations in all orders that clients of an instance of the supertype can invoke, and
- the *when*-clause of an invoked method in the subtype can block execution for a possibly unlimited amount of time only if also the corresponding *when*-clause in the supertype can do so.

These conditions are necessary to ensure that clients knowing only supertypes have enough information to provide all required external synchronization and to avoid conflicting synchronization. Synchronization conditions in subtypes can only be less restrictive than corresponding conditions in supertypes.

In Fig. 2 we give a formal definition of subtyping. A type τ_1 is subtype of τ_2 in a program P if $P; \emptyset \vdash \tau_1 \leq \tau_2$ holds. Beyond the usual conditions for subtype relationships (contravariant formal parameter types, covariant result types, etc.) we require

- subtypes to have at least the same (or more) internal and external tokens as supertypes, respectively,
- *when*- and *with*-clauses in subtypes to contain at most the same (or less) tokens to the left of the arrow as corresponding clauses in supertypes,
- and *when*- and *with*-clauses in subtypes to contain at least the same (or more) tokens to the right of the arrow as corresponding clauses in supertypes.

Two exceptions from these rules improve the flexibility of subtyping:

$$\begin{array}{c}
 \frac{}{P; \Gamma \vdash \tau \leq \tau} \quad \frac{P; \Gamma \vdash \tau_1 \leq \tau_2 \quad P; \Gamma \vdash \tau_2 \leq \tau_3}{P; \Gamma \vdash \tau_1 \leq \tau_3} \quad \frac{}{P; \Gamma, \tau_1 \leq \tau_2 \vdash \tau_1 \leq \tau_2} \\
 \frac{RI_P^{u|v} \vdash Tok_P^u \geq Tok_P^v \quad \forall dec_2 \in Sig_P^v \cdot \exists dec_1 \in Sig_P^u \cdot P; \Gamma, u[] \leq v[] \vdash u.dec_1 \leq v.dec_2}{P; \Gamma \vdash u[] \leq v[]} \\
 \frac{P; \Gamma \vdash u[] \leq v[] \quad RE_P^{u|v} \vdash t_1^* \geq t_2^*}{P; \Gamma \vdash u[t_1^*] \leq v[t_2^*]} \\
 \frac{P; \Gamma \vdash \tau_1 \leq \tau_2 \quad P; \Gamma \vdash p_2^* \leq p_1^* \quad RE_P^{u|v} \vdash c_2 \geq c_1 \quad RI_P^{u|v} \vdash c_4 \geq c_3}{P; \Gamma \vdash u.\tau_1 x(p_1^*) \text{ with } c_1 \text{ when } c_3 \leq v.\tau_2 x(p_2^*) \text{ with } c_2 \text{ when } c_4} \\
 \frac{}{P; \Gamma \vdash \epsilon \leq \epsilon} \quad \frac{P; \Gamma \vdash u_1[] \leq u_2[] \quad RE_P^{u_1|u_2} \vdash c_1 \geq c_2 \quad P; \Gamma \vdash p_1^* \leq p_2^*}{P; \Gamma \vdash u_1[c_1] v p_1^* \leq u_2[c_2] v p_2^*} \\
 \frac{R \vdash (n_1^1)x_1 \dots (n_j^1)x_j t_1^* \rightarrow (n_1^3)x_1 \dots (n_j^3)x_j \geq (n_1^2)x_1 \dots (n_j^2)x_j \rightarrow (n_1^4)x_1 \dots (n_j^4)x_j t_2^*}{\forall 1 \leq i \leq j \cdot x_i \in R \Rightarrow (n_i^1 \geq n_i^2 \wedge n_i^4 + n_i^1 - n_i^2 \geq n_i^3)} \\
 \frac{\forall 1 \leq i \leq j \cdot x_i \in R \Rightarrow n_i^1 \geq n_i^2}{R \vdash (n_1^1)x_1 \dots (n_j^1)x_j t^* \geq (n_1^2)x_1 \dots (n_j^2)x_j} \\
 Tok_P^u = \text{initial tokens declared in class } u \text{ in program } P \\
 Sig_P^u = \text{set of method signatures in class } u \text{ in program } P \\
 RI_P^{u|v} = \{\text{tokens relevant for dependent internal synchronization in } u \text{ and } v \text{ in } P\} \\
 RE_P^{u|v} = RI_P^{u|v} \cap \{\text{tokens relevant for external synchronization in } u \text{ and } v \text{ in } P\}
 \end{array}$$

Fig. 2. Subtyping

- We consider only tokens relevant for external synchronization as well as tokens possibly relevant for dependent internal synchronization (see Sect. 3.3), and we consider only tokens relevant for both the subtype and the supertype. Tokens relevant only for simple mutex synchronization as well as tokens relevant only in one of the two types need not be considered.
- If a when- or with-clause in a subtype does not contain some token to the left of the arrow that occurs there in the corresponding clause in the supertype, then the clause in the subtype need not contain this token also to the right of the arrow. Invoking the method in the subtype where we expect to invoke the method in the supertype simply does not touch this token while we expect it to be removed on invocation and added on return.

According to this definition, `BufferStat[empty]` is subtype of `BufferStat[(2)empty]`, and `BufferStatImpl[(i)empty]` is subtype of `BufferStat[(j)empty]` (k)filled] for all $i \leq j$. However, the types `BufferStat[empty]`, `BufferStat[filled]`, and `BufferDyn` are not related by subtyping.

5.2 Semantics of Subtyping

Concerning required external synchronization, subtypes specify essentially the same or more sets of acceptable message sequences (supported orders of method invocations) than supertypes [2]. Subtypes cannot strengthen synchronization constraints. This restriction is a direct consequence of the substitution principle:

An instance of a subtype can be used where an instance of a supertype was expected [4,5]. If a client provides the required external synchronization when invoking a method in a supertype, then the required external synchronization of all corresponding methods in subtypes are also satisfied.

We need not consider tokens irrelevant in the supertype because they can be relevant in the subtype only for methods not invocable according to the supertype. In this case the substitution principle does not apply. We need not consider tokens irrelevant in the subtype because no method depends on them; the subtype supports more orders of method invocations than the supertype.

Internal synchronization does not restrict message orders. Nonetheless subtypes must not strengthen internal synchronization to get this property: If sequences of method invocations in an instance of a supertype do not enforce synchronization conflicting with internal synchronization, then these sequences of invocations do not enforce conflicting synchronization in an instance of a subtype. For example, an internally synchronized buffer can safely substitute another one with less slots, but substituting one with more slots can lead to unexpected deadlocks because of stronger synchronization constraints. If two objects of two types execute the same method invocations in the same order, then the instance of the subtype always contains at least the same (or more) internal tokens than the instance of the supertype after executing the same methods. This is because the instance of the subtype has at least the same initial internal tokens, and each method removes the same or less tokens on invocation and adds the same or more tokens on return than the instance of the supertype.

Of course, this property holds only for synchronization possibly conflicting with the *when*-clauses in corresponding methods in sub- and supertypes. Subtyping cannot avoid that a different implementation of the method in the subtype introduces errors like conflicting synchronization when invoking further methods. Currently static type checking does not avoid conflicting synchronization.

For internal synchronization we consider only tokens relevant in both the subtype and the supertype for essentially the same reason as for required external synchronization. However, we consider only tokens possibly relevant for dependent internal synchronization. Simple mutex synchronization need not be considered because from the client's point of view its only effect is to delay execution for a finite amount of time under the simplifying assumption that all methods depending on these tokens terminate in finite (and for practical reasons short) time. It is the programmer's obligation to ensure termination.

Subtyping does not consider instance variables because they are only accessible within objects containing them (at the presence of dependent tokens). Otherwise we would not be able to resolve dependences between tokens. It is always possible to circumvent this restriction by using setter and getter methods.

6 Related Work and Contribution

A huge number of language features for synchronization has been proposed, most of them concentrating on server synchronization [6,7]. Conventional

synchronization concepts like semaphores express synchronization directly while other concepts express what groups of operations must be executed in isolation [8,9]. Petri Nets have been explored for nearly half of a century as a basis of token-based synchronization [10]. External synchronization and especially static type checking of external synchronization was addressed only recently [11,12,13,14,1].

With- and *when-*clauses in our approach resemble assertions in Eiffel [15]. Especially preconditions can be regarded as synchronization conditions (for internal synchronization) that must be satisfied before entering a routine [16,17]. They use Boolean expressions as synchronization conditions rather directly. Similar as in our approach, synchronization conditions in subtypes can be less restrictive, and unnecessary exposure of implementation details to clients can be avoided by assigning names to synchronization conditions. There is no separation between internal and required external synchronization. While preconditions require all conditions to be explicit in program code (local to an object) they can remain on a more abstract level in our approach. *When-*clauses (synchronization conditions) of protected types in Ada [18] are similar to preconditions in Eiffel except that Ada does not support subtyping on protected types.

The Fugue protocol checker [11] uses a different approach to specify client-server protocols: Rules for using interfaces are recorded as declarative specifications. These rules can limit the order in which methods are called (implying required external synchronization) as well as specify pre- and post-conditions. Since there is no complete aliasing information and no concept resembling type splitting (as in our approach), the checker cannot statically ensure all methods to be invoked in specified orders. In these cases the checker introduces pre- and post-conditions to be executed at run time.

Many proposals ensure race-free programs [19,6,20]. Some approaches depend on explicit type annotations [20] while others perform type inference [19]. Usually only simple mutex synchronization is considered. Such techniques can lead to more locks because no approach accurately decides between necessary and unnecessary locks. Program optimization can remove some unnecessary locks [21,22]. Unfortunately, we usually must analyze complete programs for good results.

Process types [14,1,2] were developed as abstractions over expressions in object-oriented process calculi like Actors [23]. Static type checking ensures that only acceptable messages can be sent and thereby enforces required external synchronization. Process types allow us to specify arbitrary constraints on the acceptability of messages. We consider types to be partial behavior specifications [4,24] especially valuable to specify the behavior of software components [25,26,27].

There are several approaches similar to process types. Some approaches ensure subtypes to show the same deadlock behavior as supertypes, but do not enforce message acceptability [28,27]. Other approaches consider dynamic changes of message acceptability, but do not guarantee message acceptability in all cases [16,29,30]. Few approaches ensure all sent messages to be acceptable [12,13]. All

of these approaches specify constraints on the acceptability of messages in a rather direct way and do not make use of a token concept.

Recently several programming languages [31,32,33] were developed based on the Join calculus [34]. For example, in Polymorphic C# [31] we combine methods like `put` and `get` in a buffer to a chord to be 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 a specific form of internal synchronization is supported. There is no way to express required external synchronization. For example, it is easy to program a buffer with unlimited capacity using chords, but a buffer with limited capacity (even when using only internal synchronization) and a window changing state between iconified and displayed (see Sect. 2) cannot easily be written. Communication in Polymorphic C# and similar languages resembles that of the rendezvous concept while our approach extends monitors.

The present work extends previous work on separating client synchronization from server synchronization [3]. Major improvements over this work are the proposal of a more advanced concept of internal synchronization, the support of dependent types, and a richer notion of subtyping. These contributions are important for the following reasons:

- We regard the new concept of internal synchronization as an extension of more conventional synchronization concepts like that in Java. Synchronized Java methods correspond to methods with a clause “`when sync → sync`”, and `wait` and `notify` can be modelled by `when`-clauses and `with`-statements with conditions of the form “`token →`” and “`→ token`”, respectively. This property can be the basis for slow migration to the new concept which provides more expressiveness and safety than conventional synchronization.
- Dependent types solve an old problem of process types together with instance variables: They allow us to express states in types of variables depending on the states of objects where the variables belong to. Thereby, the coupling between the objects becomes evident, and corresponding type information becomes usable in static type checking.
- It is well-known that subtyping has to consider object behavior to provide substitutability [4]. However, it is still not clear which aspects of the behavior must be considered. Our notion of subtyping distinguishes between simple mutex synchronization (that can be ignored) and dependent internal as well as required external synchronization (which must be considered). Unfortunately, Java-like languages make essentially simple mutex synchronization (through synchronized methods) clearly visible while dependent internal synchronization (through `wait` and `notify`) are less visible and required external synchronization is not even expressible in program code.

Much work remains to be done. The most important work planned for the future is an implementation of the proposed concept. Other future work includes a rigorous formalization of the approach and the development of further consistency checks (for example, to guarantee continuity and liveness properties) on this basis.

7 Conclusions

Differentiation between internal and required external synchronization allows us to clearly specify who is responsible for which synchronization. Clients need all synchronization information except of simple mutex synchronization to use objects as expected. Subtyping considers dependent internal and required external synchronization, but ignores simple mutex synchronization. Dependent tokens in types of instance variables give us the flexibility to use more available synchronization information in providing external synchronization. Static type checking uses all kinds of synchronization information (without any need for aliasing information) to ensure race-free programs while supporting separate compilation.

References

1. Puntigam, F.: Coordination requirements expressed in types for active objects. In Aksit, M., Matsuoka, S., eds.: *Proceedings ECOOP'97*. Number 1241 in *Lecture Notes in Computer Science*, Jyväskylä, Finland, Springer-Verlag (1997) 367–388
2. Puntigam, F.: *Concurrent Object-Oriented Programming with Process Types*. Der Andere Verlag, Osnabrück, Germany (2000)
3. Puntigam, F.: Client and server synchronization expressed in types. In: *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, USA (2005)
4. Liskov, B., Wing, J.M.: Specifications and their use in defining subtypes. *ACM SIGPLAN Notices* **28** (1993) 16–28 *Proceedings OOPSLA'93*.
5. Wegner, P., Zdonik, S.B.: Inheritance as an incremental modification mechanism or what like is and isn't like. In Gjessing, S., Nygaard, K., eds.: *Proceedings ECOOP'88*. Number 322 in *Lecture Notes in Computer Science*, Springer-Verlag (1988) 55–77
6. Brinch-Hansen, P.: The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering* **1** (1975) 199–207
7. Hoare, C.A.R.: Communicating sequential processes. *Communications of the ACM* **21** (1978) 666–677
8. Harris, T., Fraser, K.: Language support for lightweight transactions. In: *OOPSLA'93*, Anaheim, California, USA, ACM (2003) 388–402
9. Liskov, B., Scheifler, R.: Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems* **5** (1983) 381–404
10. Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* **77** (1989) 541–580
11. DeLine, R., Fähndrich, M.: The fugue protocol checker: Is your software baroque? Technical report, Microsoft Research (2004) <http://www.research.microsoft.com>.
12. Kobayashi, N., Pierce, B., Turner, D.: Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems* **21** (1999) 914–947
13. Najm, E., Nimour, A.: A calculus of object bindings. In: *Proceedings FMOODS'97*, Canterbury, United Kingdom, Chapman & Hall (1997)
14. Puntigam, F.: Type specifications with processes. In: *Proceedings FORTE'95*, Montreal, Canada, IFIP WG 6.1, Chapman & Hall (1995)

15. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
16. Caromel, D.: Toward a method of object-oriented concurrent programming. *Communications of the ACM* **36** (1993) 90–101
17. Meyer, B.: Systematic concurrent object-oriented programming. *Communications of the ACM* **36** (1993) 56–80
18. ISO/IEC 8652:1995: Annotated ada reference manual. Intermetrics, Inc. (1995)
19. Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: A dialect of Java without data races. In: *OOPSLA 2000*. (2000)
20. Flanagan, F., Abadi, M.: Types for safe locking. In: *Proceedings ESOP'99*, Amsterdam, The Netherlands (1999)
21. Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. In: *OOPSLA'99*, Denver, Colorado (1999)
22. von Praun, C., Gross, T.R.: Static conflict analysis for multi-threaded object-oriented programs. In: *PLDI '03*, ACM Press (2003) 115–128
23. Agha, G., Mason, I.A., Smith, S., Talcott, C.: Towards a theory of actor computation. In: *Proceedings CONCUR'92*. Number 630 in *Lecture Notes in Computer Science*, Springer-Verlag (1992) 565–579
24. Meyer, B.: *Object-Oriented Software Construction*. Second edition edn. Prentice Hall (1997)
25. Arbab, F.: Abstract behavior types: A foundation model for components and their composition. *Science of Computer Programming* **55** (2005) 3–52
26. Lee, E.A., Xiong, Y.: A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing* **16** (2004) 210–237
27. Niestrasz, O.: Regular types for active objects. *ACM SIGPLAN Notices* **28** (1993) 1–15 *Proceedings OOPSLA'93*.
28. Nielson, F., Nielson, H.R.: From CML to process algebras. In: *Proceedings CONCUR'93*. Number 715 in *Lecture Notes in Computer Science*, Springer-Verlag (1993) 493–508
29. Colaco, J.L., Pantel, M., Salle, P.: A set-constraint-based analysis of actors. In: *Proceedings FMOODS'97*, Canterbury, United Kingdom, Chapman & Hall (1997)
30. Ravara, A., Vasconcelos, V.T.: Behavioural types for a calculus of concurrent objects. In: *Proceedings Euro-Par'97*. Number 1300 in *Lecture Notes in Computer Science*, Springer-Verlag (1997) 554–561
31. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems* **26** (2004) 269–804
32. Drossopoulou, S., Petrounias, A., Buckley, A., Eisenbach, S.: School: A small chorded object-oriented language. In: *Proceedings of ICALP Workshop on Developments in Computational Models*. (2005)
33. Odersky, M.: *Functional nets*. In: *Proceedings of the European Symposium on Programming*, Springer-Verlag (2000)
34. Fournet, C., Gonthier, G.: The reflexive cham and the join-calculus. In: *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*. (1996) 372–385