

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

Franz Puntigam

Vienna University of Technology  
Vienna, Austria

`franz@complang.tuwien.ac.at`

`http://www.complang.tuwien.ac.at/franz/`

---

# Kinds of Synchronization

**Internal Synchronization:** Object as server is responsible

Example: `synchronized`, `wait`, and `notify` in Java

**Required External Synchronization:**

Object requires synchronization to be provided by clients

Determines when a method can be invoked

Today usually specified only as comment

**Provided External Synchronization:**

Clients provide it by invoking methods in a specific order

Clients usually provide more synchronization than required

---

# Conflicting Synchronization

Possible conflicts between internal and provided external sync.

Example: invoke put – put – get on buffer with single slot

Clients must avoid conflicting synchronization

**Simple Mutex Sync.** ensures just mutual exclusion

Cannot be in conflict with provided synchronization

**Dependent Internal Sync.** depends on program state

Clients must be aware of it to avoid conflicting sync.

Goal of this work:

Types specify required external and dependent internal sync.

---

# Process Types

- Static types of expressions in a process calculus translated to more conventional object-oriented languages
- Required external synchronization based on tokens

```
class Window {  
    Window[displ]() { ... }  
    void iconify() with displ -> icon { ... }  
    void display() with icon -> displ { ... }  
    void foo (Window[displ -> icon] w) {  
        w.iconify()           // w of type Window[icon]  
        w.display()           // w of type Window[displ]  
        w.iconify()           // w of type Window[icon]  
    }  
}
```

---

## Issues to be Addressed

- Internal synchronization based on tokens
- Tokens on “this”
- Tokens on instance variables shared among threads
- Race-free programs
- Subtyping considering synchronization

---

# Internal Synchronization

- “when ...” manipulates pool of internal tokens per object
- Execution suspended until required tokens available in pool
- “atomic” ensures atomic execution (simple mutex sync.)  
“atomic” can be nested; “when t->t” cannot be nested

```
[empty(10)] class BufferDyn {  
    void increment() when ->empty(10) { ... }  
    void decrement() when empty(10)-> { ... }  
    void put(Elem e) when empty->filled {atomic{ ... }}  
    Elem get() when filled->empty {atomic{ ... }}  
}
```

---

## Tokens on “this”

- Instead of removing/adding tokens move them to/from the self-reference “this”

```
class InternExtern {  
    void makeI() when ->t {this[->t]{... this.makeE() ...}}  
    void makeE() with ->t {...}  
    void useI() when t-> {this[t->]{... this.useE() ...}}  
    void useE() with t-> {this[t->]{... this.useE() ...}}  
}
```

---

# Dependent Tokens

- Tokens of instance variable depend on removed/added tokens in environment and on tokens of “this”

```
class BufferStat {  
    void increment() with ->empty(10) { ... }  
    void put(Elem e) with empty->filled {atomic{ ... }}  
    Elem get() with filled->empty {atomic{ ... }}  
}  
[pe(50)] class Proxy {  
    BufferStat[empty for pe][filled for pf] buffer  
    Proxy(BufferStat[empty(50)->] b) {buffer = b}  
    void put(Elem e) when pe->pf {buffer.put(e)}  
    Elem get() when pf->pe {return buffer.get(e)}  
}
```



---

## Writing to Instance Variables

- Need to know all tokens possibly available in environment
- Fixed-point algorithm computes upper bounds of tokens
- Dependent token spec. determines tokens to be provided
- Worst-case assumption if several upper bounds apply

```
void update(BufferStat[empty(50) filled(10) ->] b)
    when pe(40)->pe(40) {atomic{buffer = b}}
```

---

# Race-Free Programs

- Checked for each class separately
- Compute upper bounds of tokens for considered class
- Two shared variable accesses in different methods cannot influence each other if
  - no upper bound includes the union of all tokens to be removed (by `with` and `when` clauses) and of “`this`” of both methods
  - or a variable access occurs in an `atomic` statement

---

# Subtyping and Tokens

- Substitution principle requires for a subtype  $s$  of a type  $t$ :
  - instance of  $s$  accepts at least all messages in all orders accepted by instances of  $t$  (required external sync.)
  - dependent internal synchronization in instance of  $s$  can block execution only if that in instance of  $t$  does so
- Synchronization in subtype can only be weaker (except for simple mutex sync.)
- Subtyping statically decidable:
  - Init. internal and external tokens of  $s$  include those of  $t$
  - Tokens to left of “ $\rightarrow$ ” in  $t$  include corr. tokens in  $s$
  - Tokens to right of “ $\rightarrow$ ” in  $s$  include corr. tokens in  $t$

---

# Guarantees

- Compiler ensures:
  - Provided external sync. satisfies required external sync.  
(no external tokens needed at run time)
  - Overridden sync. cannot add synchronization conflicts
  - Race-free programs through atomic method execution
- Compiler does not ensure absence of sync. conflicts  
(Enforcement would be too restrictive)
- Signatures give info needed to avoid conflicting sync.

---

# Conclusions

- Interface specifies who is responsible for synchronization
- Specific contributions:
  - improved internal synchronization
  - external tokens on `this`
  - external tokens on shared instance variables
- Future work:
  - state inference for shared instance variables
  - implementation issues (`atomic` and internal tokens)