Client and Server Synchronization Expressed in Types

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

ABSTRACT

Synchronization based on semaphores, monitors, or atomic actions is in general not visible through object types. Often we use unnecessary or even conflicting synchronization in clients and servers because of missing information. Process types – a kind of behavioral types – solve this problem so far only partially: Types specify synchronization required from clients, but they hide the servers' synchronization. In this paper we propose a solution based on dynamic token management (in addition to static token management already existing in process types). This approach extends conventional object types in a straightforward way, supports subtyping under consideration of synchronization, ensures race-free programs, and still does not expose implementation details to clients.

Categories and Subject Descriptors

D.3.3 [**Programming languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Synchronization, Types

Keywords

Client and server synchronization, synchronization conflicts, race avoidance, subtyping

1. INTRODUCTION

We usually think of synchronization as a purely dynamic concept: If there is a dependence between two control flows, then one of the corresponding threads must wait until the other thread has caught up to meet at a synchronization point. Since threads usually run asynchronously and at statically unpredictable speed, it is only possible to decide at run time whether a thread must wait. However, these considerations are valid only at a low level (close to the hardware)

point of view. From the programmers' higher level point of view it is quite often not clear whether there exist dependences between threads or not. Using explicit synchronization as with monitors, semaphores, etc. programmers must add much more synchronization (to get mutual exclusion) than actually necessary and thereby increase run-time overhead and the danger of deadlocks and other undesirable program behavior. Good knowledge about the complete program and its synchronization structure can reduce the problem. Unfortunately, in many cases such knowledge is not available (especially for independently developed components), and it is very difficult to derive the necessary information just from source code.

We address this problem by adding information about synchronization to object types regarded as contracts between clients and servers [19]. A type specifies

client synchronization to be guaranteed by clients (the server assumes it as given), and

server synchronization the server handles by itself.

For buffers and similar simple cases we typically use server synchronization where only the server decides when to perform an operation. Sometimes we prefer client synchronization for better control over synchronization. For example, in Java the class `Hashtable` provides server synchronization while the (otherwise similar) newer class `HashMap` requires client synchronization. Clients having sufficient information about dependences and synchronization can get much better performance from `HashMap` than from `Hashtable`.

Of course, clients must know about required client synchronization, and they need also knowledge about server synchronization. For example, a buffer with a single slot can perform `put` and `get` operations only in alternation, no matter who deals with synchronization. In the case of client synchronization, clients must ensure to invoke `put` and `get` only in alternation, and the server can perform the buffer operations without further synchronization. In the case of server synchronization, clients must avoid conflicting synchronization as in the invocation sequence 'put put get' within a single thread (resulting in a deadlock).

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 in the form of token sets as proposed for process types [24, 25]. This type system allows us to specify client synchronization in types and statically ensures that clients observe required dependences.

$$P ::= unit^*$$
$$unit ::= \texttt{class } c \; [\texttt{< } c^+]_{\text{opt}} \; \texttt{is } [\texttt{token } x^+]_{\text{opt}} \; def^+ \quad | \quad \texttt{type } c \; [\texttt{< } c^+]_{\text{opt}} \; \texttt{is } [\texttt{token } x^+]_{\text{opt}} \; decl^+$$
$$decl ::= \texttt{method } m \; (\; par^* \; [\texttt{with } ctok]_{\text{opt}} \;) \; [: t]_{\text{opt}} \; [\texttt{when } ctok]_{\text{opt}}$$
$$def ::= \texttt{var } v^+ : t \quad | \quad decl \; \texttt{do } s^+ \quad | \quad \texttt{creator } m \; (\; par^* \;) : t \; \texttt{do } s^+$$
$$par ::= v : c \, [[\; ctok \;]]_{\text{opt}} \; [\texttt{< } tok^+ \texttt{ >}]_{\text{opt}}$$
$$ctok ::= tok^+ \texttt{ -> } tok^* \quad | \quad \texttt{-> } tok^+$$
$$tok ::= x \; [.i]_{\text{opt}}$$
$$t ::= c \; [[\; tok^+ \;]]_{\text{opt}} \; [\texttt{< } tok^+ \texttt{ >}]_{\text{opt}}$$
$$s ::= v : t \texttt{=} e \quad | \quad v \texttt{=} e \quad | \quad e \quad | \quad \texttt{return } [e]_{\text{opt}} \quad | \quad \texttt{fork } e$$
$$e ::= \texttt{this} \quad | \quad v \quad | \quad c \quad | \quad e.m \, (\; e^* \;) \quad | \quad i \quad | \quad \texttt{null}$$

$$
\begin{aligned}
c &\in \text{ class and type names} \\
x &\in \text{ token names} \\
m &\in \text{ message selectors} \\
v &\in \text{ variable names} \\
i &\in \text{ natural number literals}
\end{aligned}
$$

Figure 1: Syntax of of the Language

The paper is structured as follows: In Section 2 we introduce client synchronization, and in Section 3 we add server synchronization. Next, in Section 4 we discuss relationships between subtyping and synchronization. In Section 5 we give examples of how static analysis can help to ensure proper synchronization. In Section 6 we motivate some future work. Finally, in Section 7 we point out related work before we draw our conclusions in Section 8.

2. CLIENT SYNCHRONIZATION

Figure 1 shows the grammar of a simple language we use as showcase. We differentiate between classes and types without implementation (beginning with `type`, similar to interfaces in Java). To create a new object we invoke a creator in a class. Type annotations follow after ":". Token declarations (these are token names following the keyword `token`), tokens occurring within square brackets in types, and `with`-clauses together determine required client synchronization.

The following example shows how tokens allow us to specify synchronization constraints:

```
type E is ...       -- type of buffer elements
type Buffer is          -- type of the buffer
   token empty filled   -- token declarations
   method put (e:E with empty->filled)
   method get (with filled->empty): E
```

According to the `with`-clause in `put` we can invoke `put` only if we have a token `empty`; this token is removed on invocation, and `filled` is added on return. For a variable x of type `Buffer[empty]` – a buffer with a single token `empty` abstracting over an empty buffer slot – we can invoke `x.put(...)`. This invocation changes the type of x to `Buffer[filled]`. Afterwords we can invoke `x.get`, then again `x.put(...)`, and so on. Static type checking enforces `put` and `get` to be invoked in an instance of type `Buffer[empty]` only in alternation.

Instances of `Buffer[empty.8 filled.7]` have at least 8 filled and 7 empty slots. Suffixes give token numbers. Single tokens need no suffix, and token lists are regarded to be multi-sets. Hence, the type is equivalent to

```
Buffer[empty.3 filled empty.5 filled.6].
```

An instance accepts `put` and `get` in all sequences such that the buffer never contains more that 15 (this is $3 + 1 + 5 + 6$)

or less than zero elements. Token sequences on both sides of `->` in `with`-clauses are multi-sets.

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

```
class Test is
   method copyE (b:Buffer[empty filled->filled.2])
      do e:E = b.get()
         b.put(e)
         b.put(e)
```

Arrows in parameter types relate tokens required on invocation with those available on return. Let y be a variable of type `Buffer[empty.2 filled.2]` and x one of type `Test`. We can invoke `x.copyE(y)` since y has the required tokens. This method gets an element from the buffer, assigns it to the local variable e (declared in the first statement), and puts e twice into the buffer. Within `copyE` the buffer is known to have at least an empty and a filled slot on invocation and two filled slots on return. Removing the tokens to the left of `->` in the formal parameter type on invocation causes the type of y to become `Buffer[empty filled]`, and adding the tokens to the right on return causes it to become `Buffer[empty filled.3]`.

Parameter passing does not produce or consume tokens. Tokens just move from argument types to formal parameter types on invocation and from formal parameter types to argument types on return. Only `with`-clauses add and remove tokens. As a basic principle, each object produces and consumes only its own tokens.

The statement 'fork x.copyE(y)' spawns a new thread executing `x.copyE(y)`. Since the current thread continues execution without waiting for termination of the new thread, the new thread cannot return any parameter as well as any token. The type of y changes on invocation from `Buffer[empty.2 filled.2]` just to `Buffer[empty filled]`. Everything to the right of `->` in the formal parameter type has to be ignored. Thereby the old type of y is split into two types – the new type of y and the type of the formal parameter b. Both threads can invoke methods in the same buffer without affecting each other concerning type information. Through y and through b we can invoke `put` and `get` as if the buffer had only two slots although it has at least four slots. Of course, through y we can get elements from the buffer put there through b and the other way around.

Assignment resembles parameter passing when spawning threads: We split an assigned value's type into two types such that one split type equals the variable's type, and the remaining type becomes the assigned value's new type. Tokens move from the value's to the variable's type. For example, if `v` is of type `Buffer[empty.2]` and `y` of type `Buffer[empty.2 filled.2]`, then the execution of 'v = y' causes the type of `y` to become `Buffer[filled.2]`.

Explicit result types of creators play an important role for introducing tokens:

```
class Buffer1 < Buffer is
   var elt: E    -- single slot of buffer
   method put (e:E with empty->filled)
      do elt = e
   method get (with filled->empty): E
      do return elt
   creator new(): Buffer1[empty]
      do null
```

Class `Buffer1` inherits the tokens from `Buffer`. An invocation of `Buffer1.new()` returns a new instance with a token `empty`. No other token is available for the new object. Since invocations of `put` and `get` consume a token before they issue another one, there exists always at most one token for this object. This property ensures that `put` and `get` can be invoked only in alternation beginning with `put`. Hence, no empty buffer slot can be read and no filled one overwritten, and we need no further synchronization even if several threads access the buffer.

As presented here, client synchronization is of a very static nature. We anticipate all synchronization at compilation time. Clients in our buffer example must statically know that all needed tokens are available in types. Especially for buffers such knowledge is often not available. In contrast, clients of the following type (of very similar structure) usually know about the needed tokens statically:

```
type Window is
   token shown, iconified
   method show (with iconified->shown)
   method iconify (with shown->iconified)
```

An invocation of `show` shows a previously iconified window on a screen, and `iconify` replaces the shown window with an icon. To invoke these methods we press corresponding buttons. Since the two buttons of the same window cannot be active at the same time, the buttons can statically know the window's state, and we need no further synchronization.

3. SERVER SYNCHRONIZATION

Solutions like `Buffer1` move responsibility for synchronization to clients which easily becomes tedious when repeated several times in larger examples. In such cases we need a dynamic kind of synchronization (in addition to the static synchronization provided by `with`-clauses). We understand server synchronization as dynamic counterpart to the approach presented in Section 2.

To add server synchronization we associate each object with a multi-set of tokens to be manipulated dynamically. Result types of creators specify initial multi-sets for objects created by these creators within angles (see Figure 1). Tokens to the left of `->` in `when`-clauses of methods must be available and are removed from this multi-set 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 by executing methods of the same object. Checks for the availability of these tokens occur only at run time.

The following variant of our buffer example uses only server synchronization to ensure mutual exclusion and to avoid buffer overflow and underflow:

```
type BufferDyn is
   token empty filled
   method put(e:E) when empty -> filled
   method get(): E when filled -> empty


class BufferDyn50 < BufferDyn is
   token sync
   var buff: ArrayOfEs
   var in out: Natural
   creator new(): BufferDyn50<sync empty.50>
      do buff = ArrayOfEs.new(50)
         in = 0
         out = 0
   method put (e:E) when sync empty -> sync filled
      do in = in.incrementModulo (buff.size())
         buff.set (in e)
   method get(): E when sync filled -> sync empty
      do out = out.incrementModulo (buff.size())
         return buff.get(out)
```

The creator `new` introduces just a single token `sync`. Both `put` and `get` remove this token at the begin and issue a new one on return. The type `BufferDyn<empty.50>` exposes essential (but not all) parts of server synchronization – there are at least 50 buffer slots – to the clients. Tokens within angles are those returned by creators that shall be visible to clients. Different from tokens in square brackets these tokens do not change on method invocations.

Compared to other synchronization concepts our solution has the following advantages:

- We treat underflow and overflow checks rather directly without any need for `wait` and `notify` operations and without extra variables holding the number of filled slots.

- We have clear scopes – in this paper just methods, but extending the language with `when`-statements having clear start and end points is straightforward.

- We can use different tokens to allow unrelated methods of an object to be executed simultaneously, and we can allow several (but a restricted number of) simultaneous executions of the same method.

- Clients see essential parts of server synchronization which can help to avoid conflicting synchronization (especially together with subtyping, see Section 4).

- Nonetheless servers need not expose implementation details to clients (like the values of `in` and `out` and the current number of filled slots).

- Simple synchronization just to ensure mutual exclusion (like spin locks in the Linux kernel) need not be exposed to clients (see Section 4).

- We use essentially the same concept (tokens) for client and server synchronization.

The next example shows how client synchronization and server synchronization can interact:

```
class StaticAndDynamic is
    token t
    method makeDynamic (with t->) when ->t
        do ...
    method makeStatic  (with ->t) when t->
        do ...
    creator new(): StaticAndDynamic<t>
        do ...
```

No matter how often and from how many threads we invoke `makeDynamic` and `makeStatic` there always exists only a single token `t` for each instance (either in the dynamic token set after invoking `makeDynamic` or in a static type after invoking `makeStatic`), and the two methods will be sequentially executed in alternation. When considering just client or server synchronization alone we cannot see such behavior. This (abstract) example makes clear that it is essential to specify synchronization conditions by sets of tokens at both sides of `->`; it is less important for proper synchronization where the tokens belong to – `with`-clauses or `when`-clauses. Hence, programmers have complete freedom to select between all combinations of client synchronization with server synchronization.

If we omitted the `with`-clauses in `StaticAndDynamic`, then instances correspond to semaphores where `makeStatic` is equivalent to P and `makeDynamic` to the V operation. Our approach to server synchronization is at least as powerful as semaphores.

We imagine to implement `when`-clauses by semaphore operations, for example, those provided by Linux. When creating an object we introduce a new semaphore set containing a semaphore for each token name occurring in `when`-clauses of the class. We initialize them with the token numbers specified in the creator. On method invocation we simultaneously decrement the semaphore values according to the token numbers specified to the left of `->` in the `when`-clause, causing the process to sleep until all required tokens are available. On return we increment the semaphore values as specified to the right. In practice, semaphore operations provided by the operating system are not the best solution for our purpose: They are slow, and semaphore values are limited to small numbers. Thus we recommend to use a specific implementation in the language allowing us to avoid unnecessary semaphore operations.

4. SIMPLE MUTEXES AND SUBTYPING

A token is a *simple mutex token* of a class if there is a number $n > 0$ such that

- each creator of the class specifies exactly n occurrences of the token within angles,

- and each `when`-clause of a method in the class either does not contain the token or contains exactly n occurrences of the token at both sides of `->`.

For example, `sync` is a simple mutex token of `BufferDyn50`. The only purpose of such tokens in server synchronization is to ensure mutual exclusion of methods containing the tokens in their `when`-clauses. Since these tokens usually cause locking just for a short period of time, they are in general not of interest to clients and need not be exposed to clients.

Furthermore, the availability of simple mutex tokens need not be checked in cases where only a single thread can access the object, and where the thread accessing the object has already removed these tokens from the same object. This exception from the general rule (execution blocks until all required tokens are available) supports, for example, recursive invocations of methods requiring simple mutex tokens in their `when`-clauses. There is no such exception for other tokens. While executing `put` we regard the used buffer slot as neither filled nor empty.

Of course, we can use more conventional synchronization (for example, the `synchronized`-statement in Java) instead of simple mutex tokens to avoid exceptional cases. In the rest of this paper we prefer to use simple mutex tokens together with the exceptions that

- their availability need not be checked if no concurrent accesses can occur (for the purpose of optimization),

- and non-availability must not cause blocking of the thread that caused these tokens to be removed (to avoid unnecessary deadlocks for recursive invocations).

We do so because the programming system gets some (at a first glance possibly unexpected, positive as well as negative) properties that we can easily analyze in our setting. Using an additional synchronization concept we would get essentially the same properties, but they are less visible. In practical use programmers may prefer an additional synchronization concept over these exceptions.

In our language a type s is subtype of a type t if

[**conventional subtyping**] the usual conditions for subtype relationships hold (this is, s specifies members corresponding to those in t, parameter types are contravariant, result types are covariant, and subtyping is reflexive and transitive);

[**synchronization conditions**] tokens in `with`-clauses as well as in `when`-clauses of corresponding methods are related as follows (where empty clauses are omitted):

1. all tokens to the left of `->` in s occur also to the left of `->` in t (except if 4 applies),

2. all tokens to the right of `->` in t occur to the right of `->` in s (except if 3 applies),

3. tokens that occur on both sides of `->` in t and do not occur to the left of `->` in s need not occur to the right of `->` in s;

4. if s is a class, then simple mutex tokens of s in a `when`-clause in s need not occur in the corresponding `when`-clause in t;

[**tokens**] and the multi-sets of tokens (in square brackets and angles) associated with s contain at least all tokens in corresponding multi-sets of t.

According to this definition `Buffer[empty.2 filled]` and `Buffer1[empty]` are subtypes of `Buffer[empty]` (because subtypes can be associated with more tokens than supertypes), and `BufferDyn50<sync empty.50>` is a subtype of `BufferDyn<empty.10>` and of `BufferDyn<empty>`.

Concerning client synchronization, subtypes specify essentially the same or more sets of acceptable message sequences (acceptable orderings of method invocations) than supertypes. Corresponding results for a similar setting have been shown in [25]. Subtypes can relax the client synchronization constraints expressed in supertypes, but cannot make them stronger. 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 [16, 28]. If a client ensures that the client synchronization constraints for a method in a supertype are satisfied, then the client synchronization constraints for all corresponding methods in subtypes are also satisfied. We can safely invoke a method in an instance of a subtype where we expect to invoke the corresponding method in an instance of a supertype without breaking client synchronization (or any other property ensured by the type system).

Server synchronization does not restrict the acceptability of messages, this is, we do not regard unsatisfied server synchronization constraints as a type error. Nonetheless subtypes can just relax server synchronization constraints, but cannot make them stronger (except through simple mutex tokens). As a consequence and for the same reasons as subtyping is sound for client synchronization we get the following desirable property:

> If clients of an object do not enforce synchronization conflicting with server synchronization (as specified in the object's type), then clients still do so after replacing the object with an instance of a subtype of the object's type.

For example, let the only client of a buffer invoke `put` three times before invoking `get`. There is a synchronization conflict if the buffer has less than three slots. Hence, if we substitute an instance of `BufferDyn<empty.2>` for one of `BufferDyn<empty.3>`, then we introduce a potential synchronization conflict. Subtyping prevents this undesirable case. By substituting an instance of `BufferDyn<empty.3>` for one of `BufferDyn<empty.2>` we cannot introduce additional conflicts.

Simple mutex tokens are not considered because from the clients' point of view their 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 this context it is important to support recursive invocations of methods requiring simple mutex tokens. Otherwise it would be necessary for clients to know about simple mutex tokens in order to avoid possibly recursive invocations.

5. STATIC ANALYSIS AND PROPER SYNCHRONIZATION

The concept of tokens can be quite helpful in the static analysis of programs. We give some examples of what we can achieve just by analyzing local token information in a class. Static type checking ensuring proper client synchronization (or something quite similar) is discussed in [24, 25]. Here we want to concentrate on further aspects.

First we need a treatable notation for possibly infinite multi-sets of tokens (or simply token sets): A token set (denoted by $\rho, \sigma, \tau, \ldots$) is a finite set of elements $x.i$ where x is a token name and i a natural number $i \geq 1$ or the special symbol ∞ such that no two elements have the same token

name. The symbol ∞ specifies that there is no upper limit on the token number. For example, $\{\texttt{empty.2}, \texttt{filled.1}\}$ and $\{\texttt{empty.}\infty\}$ are token sets, but $\{\texttt{empty.1}, \texttt{empty.2}\}$ is not. Each multi-set of tokens can be converted to this form. For representing synchronization constraints as specified in `with`-clauses and `when`-clauses we use token sets as preconditions and postconditions on both sides of the arrow. For example, $\{\texttt{empty.1}\} \rightarrow \{\texttt{filled.1}\}$ corresponds to `empty->filled`. We denote the subset relationship on token sets by $\sqsubseteq$, token set union by $\oplus$, and token set difference by $\ominus$ (see Figure 2). In integer operations we assume $i + \infty = \infty = \infty + i$, $\infty - i = \infty$, $i \leq \infty$, and $i < \infty$ where i is a number or the symbol ∞.

5.1 Upper Bounds on Token Sets

An important part of some analyses is the computation of upper bounds of token sets that can occur in a computation. Figure 2 shows a fixed point algorithm to extend initial token sets S_{in} to upper bounds of reachable token sets $U^A(S_{\text{in}})$. For each token set ρ in S we construct an updated token set ρ' according to each precondition/postcondition pair where ρ satisfies the precondition. If there is not already a token set covering ρ' in S, then we add ρ' to S. Thereby we must avoid endless sequences of updates: If ρ' differs from ρ just by specifying more tokens (and no number of tokens of any name has become smaller, this is $\rho \sqsubseteq \rho'$), then repeated applications of the same update lead to unlimited token numbers. In this case we abbreviate the computation by increasing corresponding token numbers to ∞ in a single step. A fixed point has been reached if no update adds a new token set to S. This algorithm always terminates for finite input because each token set added to S contains either a smaller number of tokens for some name or the token number ∞ (which does not change anymore).

Usually we apply this algorithm for client synchronization and server synchronization together. We distinguish between tokens used for client and server synchronization (by indexes if necessary). For example, for `StaticAndDynamic` (see Section 3) we invoke the algorithm with $S_{\text{in}} = \{\{\texttt{t}_s.1\}\}$ (the set containing the token set introduced by invoking the only creator) and $A = \{\{\texttt{t}_c.1\}\rightarrow\{\texttt{t}_s.1\}, \{\texttt{t}_s.1\}\rightarrow\{\texttt{t}_c.1\}\}$ (preconditions and postconditions in `with`-clauses and `when`-clauses together). The result $U^A(S_{\text{in}}) = \{\{\texttt{t}_c.1\}, \{\texttt{t}_s.1\}\}$ shows that there can always be at most one token $\texttt{t}_c$ for client synchronization or one token $\texttt{t}_s$ for server synchronization, but never a $\texttt{t}_c$ and a $\texttt{t}_s$ at the same time.

5.2 Race Avoidance

For simplicity we use only a single sufficient (but not always necessary) criterion to ensure proper synchronization of shared accesses to instance variables: No preconditions in `with`-clauses and `when`-clauses of two methods accessing the same variable can be satisfied at the same time. To be concrete, if σ and τ are preconditions of two methods, then $\forall \rho \in U^A(S_{\text{in}}) : \sigma \oplus \tau \not\sqsubseteq \rho$ is a sufficient condition to ensure that the two methods never can be executed concurrently. To check this criterion we compare all combinations of preconditions of each two methods accessing the same variable with upper bounds of token sets constructed from the class. We analyze each class separately. All needed information is available in the code of the class and the super-classes (in case of inheritance). We need no aliasing information.

In general the above criterion ensures even more: There

Figure 2: Computing Upper Bounds of Reachable Token Sets

cannot be any two overlapping (or recursive) executions of methods accessing the same variables even in the sequential case. Hence, we can treat instance variables more efficiently as if they were local variables. However, exceptions on the use of simple mutex tokens cause this property to be violated. They allow overlapping executions to occur in sequential program code.

5.3 Continuity

If class `BufferDyn50` (see Section 3) had just a `put` method and no `get` method, then execution would be blocked after putting 50 elements into the buffer even without conflicting client synchronization. Such cases cannot happen for continuous classes: A class is continuous in a strict sense if in each reachable object state there exist sequences of method invocations such that each method in the class eventually becomes executable. For example, `BufferDyn50` is continuous in this sense. However, we do not need continuity for client synchronization. We relax continuity: A class is continuous if for all overlapping method invocations acceptable according to client synchronization (sequential invocations only if required by client synchronization to exclude synchronization conflicts) there exist further acceptable method invocations causing all invoked methods to become executable (according to server synchronization).

To check continuity we use the set A_c of all precondition/postcondition pairs of `with`-clauses in a class and the set A_{cs} of all pairs where preconditions and postconditions of `with`-clauses and `when`-clauses are put together. Then, a class is continuous if for each creator returning tokens τ_{cr} (in square brackets and angles together), for each two methods with $\sigma_c \to \tau_c$ and $\sigma_c' \to \tau_c'$ in the `with`-clauses and $\sigma_s \to \tau_s$ and $\sigma_s' \to \tau_s'$ in the `when`-clauses, respectively, and with $R(\sigma, \tau, A) = \{\rho \in U^A(\{\tau\}) \mid \sigma \sqsubseteq \rho\}$ the following conditions hold:

- if $R(\sigma_c, \tau_{cr}, A_c) \ne \emptyset$, then $R(\sigma_c \oplus \sigma_s, \tau_{cr}, A_{cs}) \ne \emptyset$

  (this is, from all initial states all acceptable methods can become executable);

- for each $\rho \in R(\sigma_c \oplus \sigma_s, \tau_{cr}, A_{cs})$ with $R(\sigma_c', \rho', A_c) \ne \emptyset$ we have $R(\sigma_c' \oplus \sigma_s', \rho', A_{cs}) \ne \emptyset$, where ρ' is the updated token set: $\rho' = (\rho \ominus (\sigma_c \oplus \sigma_s)) \oplus \tau_c \oplus \tau_s$

  (this is, after updating reachable states according to all pairs of preconditions and postconditions of executable methods still all acceptable methods can become executable).

Continuity is no sufficient condition to ensure that all invoked methods will eventually be executed: There is no guarantee that clients invoke methods that have to be executed before. For example, although `BufferDyn50` is continuous we can invoke `put` 51 times, and the last invocation does not become executable except if we invoke also `get`. Furthermore, continuity does not ensure the absence of deadlocks and livelocks.

5.4 Conflict Prevention

While it is easy to statically ensure proper client synchronization by means of type checking, it is difficult to statically prevent conflicting synchronization in clients and servers. Nonetheless we can statically find some kinds of conflicting synchronization. For example, if we know that all clients of a server belong to the same single thread, then the exactly same analysis for server synchronization as we usually use for client synchronization reliably detects all synchronization conflicts. Difficulties arise from possible synchronization dependences between several threads. We regard the analysis of such dependences as future work.

6. CLIENT VERSUS SERVER: COMPETITION VERSUS COOPERATION

The separation between client and server synchronization is not new and is in use in many (or virtually all) concurrent systems. However, since current programming languages express just server synchronization in a program and mention required client synchronization only in the documentation (if at all), client synchronization is less visible than server synchronization. Nonetheless client synchronization is and

always was an important part of synchronization. This proposal does not aim in introducing completely new forms of synchronization. Its goal is to give programmers a language concept to explicitly express (so far often implicit) requirements for client synchronization in program code. Support of static program analysis ensuring proper synchronization is a welcome supplement. Because of more complete synchronization information such analysis is simpler than in conventional systems.

For the practical use of the proposed concepts it is important to know when to use client synchronization and when server synchronization. Server synchronization adds some synchronization overhead at run time while client synchronization puts all the burden to clients. For classes like `Window` in Section 2 it is easy for the client to provide for the necessary synchronization. In other cases this is much more difficult. There is a competition between client and server synchronization using just the language concepts proposed so far in this paper. Programmers decide for either client or server synchronization, and it is quite difficult to change this decision afterwards. We need more cooperation between client and server synchronization.

In the following we propose some extensions of our language which we consider to be future work. These extensions are based on simple and well-known language concepts, but details have not been worked out sufficiently. These extensions should increase the flexibility, causing the competition between client and server synchronization to vanish.

The first extension is just genericity as in this example:

```
type Buffer (type T) is
   token empty filled
   method put (e:T with empty->filled)
   method get (with filled->empty): T
```

Adding genericity is highly dependent on details of the programming language: Type parameters can carry tokens that may be needed for static type checking. With heterogeneous translation of genericity (as with templates in C++) an implementation is straightforward. However, with homogeneous translation (as in Java) we have to add more information to type parameters to be able to statically check the availability of required tokens.

We can dramatically improve the flexibility and expressiveness by specifying token numbers dynamically:

```
class BufferImpl (type T) < Buffer(T) is
   token sync
   var buff: Array(T)
   var in out: Natural
   creator new (i: Natural)
              : BufferImpl(T)[empty.i]<sync>
      do buff = Array(T).new(i)
         in = 0
         out = 0
   method put (e:T with empty->filled)
            when sync->sync
      do in = in.incrementModulo (buff.size())
         buff.set (in e)
   method get (with filled->empty): T
            when sync->sync
      do out = out.incrementModulo (buff.size())
         return buff.get(out)
```

As in this example the token numbers in types need not be known at compile time. However, we need some mechanism to statically ensure the availability of required tokens. For example, a statement sequence

```
   x = BufferImpl.new(i)
   if i>0 then
      x.put(...)
```

provides sufficient information to safely invoke `put`. It is easy to specify syntactic constraints allowing a compiler to ensure that all requirements are satisfied, but it is difficult and highly language dependent to do so in a way avoiding surprises for the programmer.

Finally, we regard "dependent tokens" as very important in a flexible language based on types:

```
class BufferDyn (type T) is
   token emptyD filledD
   var v: Buffer[empty if emptyD]
               [filled if filledD]
   method put (e:T) when emptyD->filledD
      do v.put(e)
   method get (): T when filledD->emptyD
      do return v.get()
   creator new (i: Natural u: Buffer(T)[empty.i])
               : BufferDyn(T)<emptyD.i>
      do v = u
```

The tokens in the type of the instance variable `v` depend on knowledge about tokens (as specified in `with`-clauses and `when`-clauses) in the corresponding instance of `BufferDyn`. Within `put` we assume the type of `v` to contain at least one `empty` token on invocation (because of an `emptyD` to the left of `->` in the `when`-clause), and there must be a `filled` on return (because of `filledD` to the right of `->`). Within `get` we assume an `filled` on invocation and must provide an `empty` before return. In the creator we must initialize `v` with an instance of `Buffer` specifying at least as many `empty` and `filled` as `emptyD` and `filledD` are introduced.

Dependent tokens are a new and not yet sufficiently explored, but hopefully very natural and intuitive concept in a language based on tokens. As a requirement for this concept, overlapping accesses to corresponding variables must not be supported. With race avoidance as discussed in Section 5.2 this requirement is automatically satisfied except if we avoid races by simple mutex tokens (or similar other synchronization statements).

The examples in this section show how we expect client and server synchronization to cooperate: In `Buffer` and `BufferImpl` we use client synchronization (except for a simple mutex token) to avoid unnecessary synchronization overhead at run time in cases where clients have enough information to deal with synchronization. Clients that do not want to deal with synchronization use an instance of `BufferDyn` in addition to an unsynchronized buffer. Also classes like `HashMap` in Java (together with `synchronizedMap`) use this concept to combine the advantages of client and server synchronization without duplicated implementations. The description of `HashMap` in the API contains a big warning saying that clients are responsible for synchronization. For `Buffer` and `BufferImpl` we need no warning because static type checking ensures that clients properly deal with synchronization. We need also no warning in classes with server synchronization because the interface provides enough information about conflicting synchronization that can lead to deadlocks.

7. RELATED WORK

A huge number of language features for synchronization has been proposed, most of them concentrating on server synchronization [3, 10]. Conventional synchronization concepts like semaphores express synchronization directly while other concepts express what groups of operations must be executed in isolation [9, 15]. Client synchronization was addressed only recently [13, 20, 23, 24].

The `with`-clauses and `when`-clauses in our approach resemble assertions (preconditions and postconditions) in Eiffel [17]. Especially preconditions can be regarded as synchronization conditions (for server synchronization) that must be satisfied before entering a routine [4, 18]. 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 [12]. A disadvantage is the missing separation between client and server 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. For example, using preconditions we must explicitly check for the availability of filled or empty slots in a buffer while this information is implicit in the number of available tokens. The `when`-clauses (synchronization conditions) of protected types (monitors) in Ada [11] are similar to preconditions in Eiffel except that Ada does not support subtyping on protected types.

Many proposals ensure race-free programs [2, 3, 8]. Some approaches depend on explicit type annotations [8] (like our approach) while others perform type inference [2]. Usually only simple locks (corresponding to synchronization based on simple mutex tokens) are considered; there is no notion of state in types. Such techniques often lead to more locks because no approach can accurately decide between necessary and unnecessary locks. Program optimization can remove some unnecessary locks [5, 27]. Unfortunately, we usually must analyze complete programs for good results. An important step is the elimination of locks from objects occurring just in sequential code – an optimization not considered in the present paper. In our approach, clients promise to satisfy synchronization conditions – something that program analyzers often cannot find out easily. Thus, we consider a combination of our approach with program optimizations for race-free programs to be important future work.

The Fugue protocol checker [7] 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 client synchronization) as well as specify pre- and postconditions. Since there is no concept resembling type splitting (as in our approach) and no complete aliasing information, the checker cannot always statically ensure that methods are invoked in a specified order. In these cases the checker introduces pre- and postconditions to be executed at run time.

Process types were developed as abstractions over expressions in process calculi [23, 24, 25]. Static type checking ensures that only acceptable messages can be sent and thereby enforces client synchronization. This work already deals with support of genericity and dynamically specified token numbers – concepts that have to be reconsidered in more general settings.

Process types allow us to specify arbitrary constraints on the acceptability of messages. We consider types to be partial behavior specifications [16, 19]. They are especially valuable in order to specify the behavior of software components [1, 14, 22].

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 [21, 22]. Other approaches consider dynamic changes of message acceptability, but do not guarantee message acceptability in all cases [4, 6, 26]. Few approaches ensure all sent messages to be acceptable [13, 20]. 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.

The major contribution of the present work is to propose a clear separation between client synchronization and server synchronization expressing both kinds of synchronization in types. Such type information specifies responsibilities and distributes responsibilities over classes [19]. The consideration of synchronization in subtype relations is an important step toward reliable and conflict-free synchronization at the presence of object substitution.

8. CONCLUSIONS

Differentiation between server synchronization and client synchronization allows us to clearly specify who is responsible for which synchronization. Except of the simplest kind of synchronization (through simple mutex tokens) clients need information about synchronization, no matter whether clients or servers are responsible for it. We express this information by tokens in types. Subtyping ensures that substitutions of instances of subtypes for instances of supertypes cannot add new conflicts between server synchronization and client synchronization. The concept of tokens in types supports rather simple local program analyses, for example to ensure proper client synchronization, to avoid races, and to guarantee continuity.

9. REFERENCES

[1] F. Arbab. Abstract behavior types: A foundation model for components and their composition. *Science of Computer Programming*, 55(1–3):3–52, 2005.

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

[3] P. Brinch-Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[4] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–101, Sept. 1993.

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

[6] J.-L. Colaco, M. Pantel, and P. Salle. A set-constraint-based analysis of actors. In *Proceedings FMOODS'97*, Canterbury, United Kingdom, July 1997. Chapman & Hall.

[7] R. DeLine and M. Fähndrich. The fugue protocol checker: Is your software baroque? Technical report,

Microsoft Research, 2004.
http://www.research.microsoft.com.

[8] F. Flanagan and M. Abadi. Types for safe locking. In *Proceedings ESOP'99*, Amsterdam, The Netherlands, Mar. 1999.

[9] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA'93*, pages 388–402, Anaheim, California, USA, Oct. 2003. ACM.

[10] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[11] ISO/IEC 8652:1995. Annotated ada reference manual. Intermetrics, Inc., 1995.

[12] J. L. W. Kessels. An alternative to event queues for synchronization in monitors. *Communications of the ACM*, 20(7):500–503, July 1977.

[13] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[14] E. A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3):210–237, Aug. 2004.

[15] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[16] B. Liskov and J. M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, Oct. 1993. Proceedings OOPSLA'93.

[17] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[18] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, Sept. 1993.

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

[20] E. Najm and A. Nimour. A calculus of object bindings. In *Proceedings FMOODS'97*, Canterbury, United Kingdom, July 1997. Chapman & Hall.

[21] F. Nielson and H. R. Nielson. From CML to process algebras. In *Proceedings CONCUR'93*, number 715 in Lecture Notes in Computer Science, pages 493–508. Springer-Verlag, 1993.

[22] O. Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28(10):1–15, Oct. 1993. Proceedings OOPSLA'93.

[23] F. Puntigam. Type specifications with processes. In *Proceedings FORTE'95*, Montreal, Canada, Oct. 1995. IFIP WG 6.1, Chapman & Hall.

[24] F. Puntigam. Coordination requirements expressed in types for active objects. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP'97*, number 1241 in Lecture Notes in Computer Science, pages 367–388, Jyväskylä, Finland, June 1997. Springer-Verlag.

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

[26] A. Ravara and V. T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Proceedings Euro-Par'97*, number 1300 in Lecture Notes in Computer Science, pages 554–561. Springer-Verlag, 1997.

[27] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI '03*, pages 115–128. ACM Press, 2003.

[28] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP'88*, number 322 in Lecture Notes in Computer Science, pages 55–77. Springer-Verlag, 1988.