FROM STATIC TO DYNAMIC PROCESS TYPES

Franz Puntigam
Technische Universität Wien
Argentinierstr. 8, 1040 Vienna, Austria
Email: franz@complang.tuwien.ac.at

Keywords: Process types, synchronization, type systems, race-free programs.

Abstract: Process types – a kind of behavioral types – specify constraints on message acceptance for the purpose of synchronization and to determine object usage and component behavior in object-oriented languages. So far process types have been regarded as a purely static concept for Actor languages incompatible with inherently dynamic programming techniques. We propose solutions of related problems causing the approach to become useable in more conventional dynamic and concurrent languagues. The proposed approach can ensure message acceptability and support local and static checking of race-free programs.

1 INTRODUCTION

Process types (Puntigam, 1997) represent a behavioral counterpart to conventional object types: They support subtyping, genericity, and separate compilation as conventional types. Additionally they specify abstractions of object behavior. Abstract behavior specifications are especially desirable for software components, and they can be used for synchronization. Both concurrent and component-based programming are quickly becoming mainstream programming practices, and we expect concepts like process types to be important in the near future. However, so far process types are not usable in mainstream languages:

1. Their basis are active objects communicating by message passing (Agha et al., 1992). Variables are accessible only within single threads. In mainstream languages like Java, threads communicate through shared (instance) variables; one thread reads values written by another. To support such languages we must extend process types with support of shared variables.
2. Process types are static. Object state changes must be anticipated at compilation time. We must adapt process types to support dynamic languages like Smalltalk (using dynamic process type checking).

Support of dynamic languages turns out to be a good basis for supporting communication through shared variables. Hence, we address mainly the second issue and show how dynamic type checking can deal with the first issue.

We introduce the basic static concept of process types for a conventional (Java-like) object model in Section 2. Then, we add support of dynamic synchronization in Section 3 and of shared variables with late type checking in Section 4. Local and static checking of race-free programs is rather easy in our setting as discussed in Section 5.

2 STATIC PROCESS TYPES

Figure 1 shows the grammar of TL1 (Token Language 1) – a simple Java-like language we use as showcase. We differentiate between classes and types without implementations. To create a new object we invoke a creator `new` in a class. Type annotations follow after ":". Token declarations (names following the keyword `token`), tokens occurring within square brackets in types, and `with`-clauses together determine the statically specified object behavior.

The first example shows how tokens allow us to specify constraints on the acceptability of messages:

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

According to the `with`-clause in `put` we can invoke `put` only if we have an `empty`; this token is removed

P ::= *unit**
unit ::= `class` c [`<` c^+]$_{\text{opt}}$ `is` [`token` x^+]$_{\text{opt}}$ *def*$^+$ |
`type` c [`<` c^+]$_{\text{opt}}$ `is` [`token` x^+]$_{\text{opt}}$ *decl*$^+$
decl ::= m`(` *par** [`with` *ctok*]$_{\text{opt}}$ `)` [`:` t]$_{\text{opt}}$
def ::= v `:` c | *decl* `do` s^+ | `new(` *par** `):` t `do` s^+
par ::= v `:` c [`[` *ctok* `]`]$_{\text{opt}}$
ctok ::= *tok*$^+$ `->` *tok** | `->` *tok*$^+$
tok ::= x [`.` n]$_{\text{opt}}$
t ::= c [`[` *tok*$^+$ `]`]$_{\text{opt}}$
s ::= v `:` t `=` e | v `=` e | e | `return` [e]$_{\text{opt}}$ | `fork` e
e ::= `this` | v | c | n | e`.`m`(` e* `)` | `null`

c ∈ class and type names
x ∈ token names
m ∈ message selectors
v ∈ variable names
n ∈ natural number literals

Figure 1: Syntax of TL1

on invocation, and `filled` is added on return. For `x` of type `Buffer[empty]` – a buffer with a single token `empty` – we invoke `x.put(..)`. This invocation changes the type of `x` to `Buffer[filled]`. Next we invoke `x.get()`, then `x.put(..)`, and so on. Static type checking enforces `put` and `get` to be invoked in instances of `Buffer[empty]` in alternation. Type checking is simple because we need only compare available tokens with tokens required by `with`-clauses and change tokens as specified by `with`-clauses (Puntigam, 1997).

The type `Buffer[empty.8 filled.7]` denotes a buffer with at least 8 filled and 7 empty slots. An instance accepts `put` and `get` in all sequences such that the buffer never contains more that 15 or less than zero elements as far as the client knows.

In the next example we show how to handle tokens in parameter types similarly as in `with`-clauses:

```
class Test is
  play(b:Buffer[filled->filled])
  do e:E = b.get() -- b:Buffer[empty]
     e = e.subst() -- another e
     b.put(e)      -- b:Buffer[filled]
  copy(b:Buffer[empty filled->filled.2])
  do e:E = b.get() -- b:Buffer[empty.2]
     b.put(e) -- b:Buffer[empty filled]
     b.put(e) -- b:Buffer[filled.2]
```

Let `y` be of type `Buffer[empty.2 filled.2]` and `x` of type `Test`. We can invoke `x.play(y)` since `y` has the required token `filled`. This routine gets an element from the buffer, assigns it to the local variable `e` (declared in the first statement), assigns a different element to `e`, and puts this element into the buffer. Within `play` the buffer is known to have a single filled slot on invocation and on return. For the type of `b` specified in the formal parameter list it does not matter that the buffer has been empty meanwhile and the buffer contents changed. After return from `play` variable `y` is still of type `Buffer[empty.2 filled.2]`.

Invocations of `copy` change argument types: On return from `x.copy(y)` variable `y` will be of type `Buffer[empty filled.3]`. Removing tokens to the left of `->` on invocation causes the type 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 the argument type to the parameter type on invocation and vice versa on return. Only `with`-clauses can actually add tokens to and remove them from an object system. This is a basic principle behind the idea of tokens: Each object can produce and consume only its own tokens.

A statement '`fork x.copy(y)`' spawns a new thread executing `x.copy(y)`. Since the execution continues without waiting for the new threads, invoked routines cannot return tokens. The type of `y` changes from `Buffer[empty.2 filled.2]` to `Buffer[empty filled]`. The type of `y` is split into two types – the new type of `y` and the type of `b`. Both threads invoke routines in the same buffer without affecting each other concerning type information.

Assignment resembles parameter passing in the case of spawning threads: We split the type of an assigned value into two types such that one of the split types equals the current static type of the variable, and the remaining type becomes the new type of the assigned value. Thereby, tokens move from the value's to the variable's type. If the statically evaluated type of `v` is `Buffer[empty.2]` and `y` is of type `Buffer[empty.2 filled.2]`, then `v=y` causes `y`'s type to become `Buffer[filled.2]`.

Local variables are visible in just a single thread of control. This property is important because it allows us to perform efficient type checking by a single walk through the code although variable types can change with each invocation. Because of explicit formal parameter types we can check each class separately. If variables with tokens in their types were accessible in several threads, then we must consider myriads of possible interleavings causing static type checking to become practically impossible. Instance variables can be shared by several threads. To support instance variables and still keep the efficiency of type checking we require their types to carry no token information. We address this restriction in Section 4.

Explicit result types of creators play a quite important role for introducing tokens into the system:

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

Class `Buffer1` inherits `empty` and `filled` from

def ::= v `:` c | *decl* [`when` *ctok*]$_{\text{opt}}$ `do` s^+ |
`new(` *par** `):` t [`->` *tok*$^+$]$_{\text{opt}}$ `do` s^+

Figure 2: Syntax of TL2 (Differences to TL1)

`Buffer`. An invocation of `Buffer1.new()` returns a new instance with a single token `empty`. No other token is initially available. Since invocations of `put` and `get` consume a token before they issue another one, there is always at most one token for this object. 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. The use of tokens greatly simplifies the implementation. However, this solution is inherently static and does not work in more dynamic environments.

3 DYNAMIC TOKENS

The language TL2 (see Figure 2) slightly extends TL1 with dynamic tokens for synchronization. This concept resembles more conventional synchronization like that in Java. There is no need to anticipate such synchronization at compilation time.

We associate each object with a multi-set of tokens (token set for short) to be manipulated dynamically. TL2 differs from TL1 by optional `when`-clauses in routines and optional initial dynamic tokens (following `->`) in creators. Tokens to the left of `->` in `when`-clauses must be available and are removed before executing the body, and tokens to the right are added on return. Different from `with`-clauses, `when`-clauses require dynamic tokens to be in the object's token set and change this token set. If required dynamic tokens are not available, then the execution is blocked until they become available. Checks for token availability occur only at run time. The following variant of the buffer example uses static tokens to avoid buffer overflow and underflow, and dynamic tokens to ensure mutual exclusion:

```
class Buffer50 < Buffer is
  token sync
  lst: List
  new(): Buffer50[empty.50] ->sync do
      lst = List.new()
  put(e:E with empty->filled)
    when sync->sync do lst.addLast(e)
  get(with filled->empty): E
    when sync->sync do
      return lst.getAndDeleteFirst()
```

The creator introduces just a single token `sync`. Both `put` and `get` remove this token at the begin and issue a new one on return. Clients need not know about the mutual exclusion of all buffer operations. Of course we could use only dynamic tokens which is more common and provides easier handling of buffers.

decl ::= m `(` v* [`with` *ctok*]$_{\text{opt}}$ `)`
def ::= v `:` | *decl* [`when` *ctok*]$_{\text{opt}}$ `do` s^+ |
`new(` v* `):` t [`->` *tok*$^+$]$_{\text{opt}}$ `do` s^+
s ::= v `:=` e | v `=` e | e | `return` [e]$_{\text{opt}}$ | `fork` e

Figure 3: Syntax of TL3 (Differences to TL1–TL2)

Static and dynamic tokens live in mostly independent worlds. Nonetheless we have possibilities to move tokens from the static to the dynamic world and vice versa as shown in the following example:

```
class StaticAndDynamic is
  token t
  beDynamic(with t->) when ->t do null
  beStatic(with ->t) when t-> do null
  new(): StaticAndDynamic[t] do null
```

There always exists only a single token `t` for each instance, no matter how often and from how many threads we invoke `beDynamic` and `beStatic`.

The major advantage of our approach compared to concepts like semaphores and monitors is the higher level of abstraction. It is not so easy to "forget" to release a lock as often occurs with semaphores, and it is not necessary to handle wait queues using `wait` and `notify` commands as with monitors. For static tokens we need not execute any specific synchronization code at all. This synchronization is implicit in the control flow.

4 DYNAMIC TYPING

In TL1 and TL2 we constrained the flexibility of the language to get efficient static type checking: Types of instance variables cannot carry tokens. In this section we take the position that static type checking is no precondition for the token concept to be useful. We want to increase the language's flexibility (by supporting tokens on instance variables) and nonetheless ensure that synchronization conditions expressed in `with`-clauses are always satisfied. An error shall be reported before invocations if required tokens are not available.

Figure 3 shows the grammar of TL3 that differs form TL2 just by missing type annotations on formal parameters and declarations. However, without type annotations there is no explicit information about available tokens. We handle this information dynamically. One kind of type annotation is left in TL3: Types of new instances returned by creators must be specified explicitly because tokens in this type together with `with`-clauses determine which routines can be invoked. Such types are part of behavior specifications. Except of type annotations the following example in TL3 equals `Buffer50`:

```
type BufferDyn is
  token empty filled
  put(e with empty->filled)
  get(with filled->empty)

class Buffer50Dyn < BufferDyn is
  token sync
  lst:
  new():Buffer50Dyn[empty.50]->sync do
      lst = List.new()
  put(e with empty->filled)
    when sync->sync do lst.addLast(e)
  get(with filled->empty)
    when sync->sync do
      return lst.getAndDeleteFirst()
```

The following example gives an intuition about the use of static tokens in a dynamic language. An open window is displayed on a screen or shown as icon:

```
type Window is
  token displ icon closed
  setup(with closed->displ)
  iconify(with displ->icon)
  display(with icon->displ)
  close(with displ->closed)

class WindowImpl < Window is
  new(): WindowImpl[closed] do ...
  ...

class WManager is
  win:
  new(w):WManager do win=w win.setup()
  onButton1() do win.iconify()
  onButton2() do win.close()
  onButton3() do win.display()
```

Some state changes (directly from an icon to closed, etc.) are not supported. Class `WManager` specifies actions to be performed when users press buttons. Under the assumption that a displayed window has only Button 1 and 2 and an icon only Button 3 the constraints on state changes are obviously satisfied. Since the assumption corresponds to the existence of at most one token for each window we need nothing else to ensure a race-free program. We express the assumption by `with`-clauses and dynamically ensure them to be satisfied. The variable `win` must be associated with a (static) token specifying the window's state. In TL1 and TL2 we cannot express such type information that is implicit in TL3.

TL3 deals with dynamic tokens in the same way as TL2. To dynamically handle information about available static tokens we consider two approaches – TL3flex as a simple and flexible approach, and TL3strict as a more restrictive and safer approach.

TL3flex. In TL3flex we tread static tokens in a similar way as dynamic tokens: Each objects contains a pool of static tokens. On invocations tokens to the left of `->` in `with`-clauses are taken from the pool, and on return those to the right are added to the pool. An error is reported if the pool does not contain all required tokens.

This approach is very flexible. Each thread can use all previously issued static tokens no matter which thread caused the tokens to be issued. A disadvantage is a low quality of error messages because there is no information about the control flow causing tokens not to be available. Furthermore, there is a high probability for program runs not to uncover synchronization problems. Thus, program testing is an issue.

TL3strict. To improve error messages and the probability of detecting problems we dynamically simulate static type checking: Instead of storing static tokens centralized in the object we distribute them among all references to the object. On invocation we check and update only tokens associated with the corresponding reference. We must find an appropriate distribution of tokens among references. In TL1 and TL2 the programmer had to determine the distribution by giving type annotations. In TL3strict we distribute tokens lazily as needed in the computation.

Instead of splitting a token set on parameter passing or assignment we associate the two references with pointers to the (unsplit) token set as well as with a new empty token set for each of the two references. Whenever required tokens are not available in the (after assignment or parameter passing empty) token set of a reference we follow the pointers and take the tokens where we find them. New tokens are stored in the references' own token sets. This way all references get the tokens they need (if available) and we need not foresee how to split token sets. Repeated application leads to a tree of token sets with pointers from the leaves (= active references) toward the root (= token set returned by creator). We report an error only if tokens required at a leaf cannot be collected from all token sets on the path to the root. On return of invocations we let actual parameters point to token sets of corresponding formal parameters.

Figure 4 shows an example: Immediately after creating a window there is only one reference `n` to it (a). The box contains the single token in the corresponding token set. When invoking `new` in `WManager` using `n` as actual parameter we construct new token sets for `n` and for the formal parameter `w` (b). When the creator assigns `w` to `win` we add new token sets for `w` and `win` (c). An invocation of `setup` on `win` removes the token `closed` and adds `displ`. On return from the creator we let the token set of `n` point to that of `w` (d). Now only `win` carries the single token. We cannot change the window's state through `n`. Therefore, TL3strict is safer and less flexible than TL3flex.

We can build large parts of the structures shown in Figure 4 already at compilation time by means of ab-

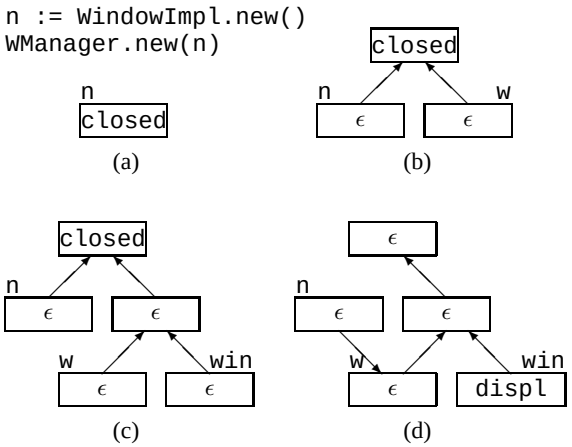

Figure 4: Token Sets per Reference

stract interpretation. Most checks for the availability of tokens can be performed statically. In fact we need dynamic checks of token availability only for tokens associated with instance variables.

5 RACE-FREE PROGRAMS

It is possible to ensure race-free programs just by analyzing the tokens in classes. We use only a single sufficient (but not always necessary) criterion: No two preconditions in `with`-clauses and `when`-clauses of routines accessing the same variable (where an access is a write) can be satisfied at the same time. To check this criterion we compute upper bounds on the token sets that can be constructed from the tokens of new instances. We analyze each class separately.

In the following description of the algorithm to determine upper bounds of token sets we first consider only static tokens as in TL1. We start with the set of token sets declared in the result types of the analyzed class' creators (one token set per creator). For each `with`-clause in the class we repeatedly construct new token sets by removing tokens to the left of `->` and adding those to the right from/to each token set constructed so far containing all required tokens. If a token set contains all tokens occurring in another token set, then we remove the smaller token set. And if a token set differs from the token set from which it was constructed just by containing more tokens, then we increment the token numbers that differ to the special value ∞ indicating infinite grow. Because of this treatment the algorithm always reaches a fixed point. The algorithm is accurate in the sense that

- the token set produced for an instance of the class is always a subset of a token set returned by the algorithm,
- if a token set returned by the algorithm does not contain ∞, then there exists a sequence of invocations producing exactly this token set,
- and if a token set returned by the algorithm contains ∞, then there exist invocation sequences producing corresponding tokens without upper bounds.

In TL2 and TL3 we must consider static and dynamic tokens together to get most accurate results. Since the static and the dynamic world are clearly separated, static and dynamic tokens must not be intermixed. We have to clearly mark each token as either static or dynamic (for example, by an index) and regard differently marked tokens as different. The algorithm starts with one token set for each creator containing both static and dynamic tokens. A new token set is constructed by simultaneously removing and adding tokens as specified in the `with`- and `when`-clause of a routine. The result shows which dynamic tokens can exist together with static tokens. For example, applied to `StaticAndDynamic` (see Section 3) the algorithm returns two token sets, one containing only a static token `t` and the other only a dynamic token `t`; in this case no dynamic token can exist at the same time as a static one.

Once we know the upper bounds it is easy to perform our check of race-free programs as shown in the following pseudo-code:

let U be the upper-bound set of token sets of class c;
for each instance variable v of c
  for each routine r write-accessing v
    for each routine s (read or write) accessing v
      let p be the union of the token sets
                to the left of `->` in r and s;
      if there is a $u \in U$ containing all tokens in p
        then issue a warning about a potential race;
otherwise c is race-free

As an example we apply this check to `Buffer1` (see Section 2). As upper-bound set of token sets S we have $\{\{\texttt{empty}\}, \{\texttt{filled}\}\}$; there is always at most one token `empty` or `filled`. The only instance variable `s` is written in `put` and read in `get`. Hence, r ranges just over `put`, s over `put` and `get`, and p over $\{\texttt{empty}.2\}$ and $\{\texttt{empty}, \texttt{filled}\}$. The class is race-free because no token set in S contains two `empty` or an `empty` and a `filled`.

The set S can become quite large because of combinatorial explosion. For example, S constructed for `Buffer50Dyn` contains 51 different token sets – all possibilities of summing up tokens of two names to 50 tokens. Fortunately, a simple change in the algorithm to compute upper bounds can reduce the size of S considerably: When computing the fixed point we replace all token numbers larger than $2 \cdot n^2 \cdot i$ by ∞, where n is the largest total number of tokens to the left of `->` in the `with`- and `when`-clause of the same routine, and i is the number of different token names

in the class. For `Buffer50Dyn` we have $n = 2$, $i = 3$, $2 \cdot n^2 \cdot i = 24$, and S contains just a single token set $\{\texttt{sync}, \texttt{empty}.\infty, \texttt{filled}.\infty\}$. This optimization does not change the output of the race-freeness check: Soundness is not affected because the multi-set of supposedly reachable tokens in a system can just get larger. No token set p (as in the algorithm) can contain more than $2 \cdot n$ tokens, and a single token of some name can be generated from no more than $n \cdot i$ tokens of another name. Therefore, more than $2 \cdot n^2 \cdot i$ tokens of one name can be ignored for our purpose. Probably there are more accurate estimations, but we expect this simple one to be sufficient because token numbers to the left of `->` are usually small.

All information needed to check race-free classes is explicit in TL1, TL2, and TL3. We need no information about formal parameter types and no aliasing information. No global program analysis is necessary.

6 DISCUSSION, RELATED WORK

The idea of integrating process types into dynamic languages is new and at a first glance unexpected because such types were developed to move dynamic aspects like synchronization to the static language level whenever possible (Puntigam, 1995; Puntigam, 1997; Puntigam, 2000). In some sense the integration of more advanced static concepts into dynamic languages is a consistent further development allowing us to use the appropriate (static or dynamic) concept for each task. Such integration helps us to deepen our understanding of related concepts.

We usually regard synchronization of concurrent threads 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 the 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 at a synchronization point. However, these considerations are valid only at a very 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, rendezvous communication, etc. programmers must add much more synchronization points than are actually necessary. There are optimization techniques that can statically eliminate up to about 90% (about 60% in average) of all locks from Java programs and thereby considerably improve program performance (von Praun and Gross, 2003). Probably even more synchronization points are actually not necessary.

Current programming languages allow programmers to write programs with races although there are many proposals to ensure race-free programs (Bacon et al., 2000; Boyapati and Rinard, 2001; Brinch-Hansen, 1975; Flanagan and Abadi, 1999). Applications of such techniques may lead to further increase of unnecessary synchronization because no approach can accurately decide between necessary and unnecessary locks. Nonetheless, these techniques are very useful because races are an important practical problem in concurrent programming.

Process types were developed as abstractions over expressions in process calculi (Puntigam, 1995). These abstractions specify acceptable messages of active objects and allow the acceptability to change over time (thereby specifying synchronization constraints). Static type checking ensures that only acceptable messages can be sent and enforces all synchronization constraints to be satisfied. In this sense type checking in process types has a similar purpose as ensuring race-free programs. However, process types allow us to specify arbitrary constraints on message acceptability, not just synchronization necessary to avoid races. In fact, the underlying calculi do not support shared data that could suffer from races.

There is a clear tendency toward more and more complex interface specifications going far beyond simple signatures of available routines (Arbab, 2005; de Alfaro and Henzinger, 2001; Heuzeroth and Reussner, 1999; Jacobsen and Krämer, 1998; Lee and Xiong, 2004; Mezini and Ostermann, 2002; Nierstrasz, 1993; Plasil and Visnovsky, 2002; Yellin and Strom, 1997). We consider such interfaces to be partial specifications of object behavior (Liskov and Wing, 1993). They are especially valuable to specify the behavior of software components as far as needed for component composition. Process types are useful as partial behavior specifications (Puntigam, 2003; Südholt, 2005). We regard behavior specifications as the major reason for using process types.

Pre- and postconditions in `with`-clauses allow us to specify a kind of contracts between components (Meyer, 1997; Meyer, 2003). Such contracts clearly specify responsibilities of software and help us to move responsibilities from one component to another. For example, we move the responsibility of proper synchronization from the server to the client if we use `with`-clauses instead of `when`-clauses.

Behavioral types and synchronization of concurrent threads are related topics: Specifications of object behavior cannot ignore necessary synchronization if we expect components composed according to their behavioral types to work together in concurrent environments, and constraints on message acceptability specify a kind of synchronization. The present work allows programmers to decide between synchronization globally visible through the interface (`with`-clauses) and local synchronization regarded as an im-

plementation detail (`when`-clauses). While `with`-clauses just ensure that clients coordinate themselves (for example, through the control flow allowing `m2()` to be invoked only after `m1()`) `when`-clauses ensure proper synchronization using more conventional techniques. Locking does not get visible in interfaces, just synchronization requirements are visible.

There are good reasons for using locking only for local synchronization: Uncoordinated locking easily leads to deadlocks and other undesirable behavior, and it is much easier to coordinate locking within a single unit. The monitor concept supports just local locking for similar reasons. Furthermore, it is very difficult to deal with globally visible locking at the presence of subtyping and inheritance (Matsuoka and Yonezawa, 1993). Process types express just synchronization conditions in interfaces, they do not provide for locking. Another approach directly expresses locking conditions in interfaces (Caromel, 1993; Meyer, 1993). As experience shows, that approach easily leads to undesirable locking where it would be more appropriate to raise exceptions.

There are several approaches similar to process types. Nierstrasz (Nierstrasz, 1993) and Nielson and Nielson (Nielson and Nielson, 1993) define behavioral types where subtypes show the same deadlock behavior as supertypes, but message acceptability is not ensured. Many further approaches consider dynamic changes of message acceptability, but do not guarantee message acceptability in all cases (Caromel, 1993; Colaco et al., 1997; Kobayashi and Yonezawa, 1994; Meyer, 1993; Ravara and Vasconcelos, 1997). Well known in the area of typed π-calculi (Milner et al., 1992) is the work of Kobayashi, Pierce and Turner on linearity (Kobayashi et al., 1999) which ensures all sent messages to be acceptable. Work of Najm and Nimour (Najm and Nimour, 1997) is very similar to process types except that in their approach at each time only one user can interact with an object through an interface (no type splitting). These approaches specify constraints on the acceptability of messages in a rather direct way and do not make use of a token concept. The use of tokens in behavior specifications gives us high expressiveness and flexibility, allows us to express synchronization in a way similar to well-known concepts like monitors and semaphores, and is easily understandable.

7 CONCLUSION

Behavioral types like process types gain more and more importance especially together with component composition. By partially specifying object behavior these types express synchronization in the form of software contracts clearly determining who is responsible for proper synchronization. Process types use simple token sets as abstractions over object states.

In this paper we explored how to add process types to rather conventional object-oriented programming languages. As a showcase we developed the languages TL1 to TL3. Static type checking in TL1 ensures that all conditions in `with`-clauses are satisfied, this is, all required tokens are available. We can synchronize concurrent threads just by waiting for messages. To overcome the restriction, TL2 adds a new dynamic concept of synchronization based on token sets. Neither TL1 nor TL2 can deal with static token sets associated with instance variables because of possible simultaneous accesses by concurrent threads. In TL3 we dispense with static types and apply one of two methods to dynamically ensure the availability of required tokens – a flexible method and one with better error messages and partial support of static type checking. All variables in TL3 have only dynamic types that can implicitly carry tokens. In the three languages we can ensure race-free programs by checking each class separately, without any need of global aliasing information.

Our approach uses token sets for several related purposes – synchronization of concurrent threads and statically and dynamically checked abstract behavior specifications. It is a major achievement to integrate these concepts because of complicated interrelations. The integration is valuable because it gives software developers much freedom and at the same time clear contracts and type safety.

Much work on this topic remains to be done. For example, currently our algorithm can issue warnings about potential races even in purely sequential program parts. Many other approaches to ensure race-free programs put much effort into detecting sequential program parts. By integrating such approaches into our algorithm we expect to considerably improve the accuracy. Most approaches to remove unnecessary locking from concurrent programs also work on sequential program parts (Choi et al., 1999; von Praun and Gross, 2003; Vivien and Rinard, 2001). We expect a combination of the techniques to improve run time efficiency.

REFERENCES

Agha, G., Mason, I. A., Smith, S., and Talcott, C. (1992). Towards a theory of actor computation. In *Proceedings CONCUR'92*, number 630 in Lecture Notes in Computer Science, pages 565–579. Springer-Verlag.

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

Bacon, D. F., Strom, R. E., and Tarafdar, A. (2000). Guava:

A dialect of Java without data races. In *OOPSLA 2000*.

Boyapati, C. and Rinard, M. (2001). A parameterized type system for race-free Java programs. In *OOPSLA 2001*. ACM.

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

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

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

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

de Alfaro, L. and Henzinger, T. A. (2001). Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press.

Flanagan, F. and Abadi, M. (1999). Types for safe locking. In *Proceedings ESOP'99*, Amsterdam, The Netherlands.

Heuzeroth, D. and Reussner, R. (1999). Meta-pr
type system for the dynamic coupling of b
ponents. In *OORASE'99: OOSPLA'99 W
Reflection and Software Engineering*, Bico

Jacobsen, H.-A. and Krämer, B. J. (1998).
pattern based approach to generating sy
tion adaptors from annotated IDL. In *IEE
tional Conference on Automated Software
ing (ASE'98)*, pages 63–72, Honolulu, Haw

Kobayashi, N., Pierce, B., and Turner, D. (1999)
and the pi-calculus. *ACM Transactions on
ming Languages and Systems*, 21(5):914–9

Kobayashi, N. and Yonezawa, A. (1994). Typ
foundations for concurrent object-oriente
ming. *ACM SIGPLAN Notices*, 29(10):31
ceedings OOPSLA'94.

Lee, E. A. and Xiong, Y. (2004). A behavioral ty
and its application in Ptolemy II. *Formal
Computing*, 16(3):210–237.

Liskov, B. and Wing, J. M. (1993). Specification
use in defining subtypes. *ACM SIGPLA
28(10):16–28. Proceedings OOPSLA'93.

Matsuoka, S. and Yonezawa, A. (1993). Anal
heritance anomaly in object-oriented conc
gramming languages. In Agha, G., editor
*Directions in Concurrent Object-Oriented
ming*. The MIT Press.

Meyer, B. (1993). Systematic concurre
oriented programming. *Communications o*
36(9):56–80.

Meyer, B. (1997). *Object-Oriented Software Co*
Prentice Hall, second edition edition.

Meyer, B. (2003). The grand challenge of trusted components. In *ICSE-25 (International Conference on Software Engineering)*, Portland, Oregon. IEEE Computer Press.

Mezini, M. and Ostermann, K. (2002). Integrating independent components with on-demand remodularization. In *OOPSLA 2002 Conference Proceedings*, pages 52–67, Seattle, Washington. ACM.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77.

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

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

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

Plasil, F. and Visnovsky, S. (2002). Behavioral protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076.

Puntigam, F. (1995). Flexible types for a concurrent model.