How to Specify the Flow of Data Accessibility: An OO Way of Concurrent Programming

Franz Puntigam

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

Abstract. Program structures appropriate for concurrency are often in conflict with object-oriented principles. Especially average programmers need high-level language constructs for concurrency with good integration into the object-oriented paradigm. A key concept in this respect is better static knowledge of the data flow. We propose to explicitly specify the flow of data accessibility in a program. This information is sufficient for a compiler to automatically spawn threads and add synchronization as necessary. Programmers regard the specifications just as assertions.

1 Introduction

It is not easy to express concurrency in a programming language mainly because concurrent units access shared data. We need synchronization to avoid data races and observe data dependences. Thereby we serialize parts of the execution in a statically rather unpredictable way. By spawning concurrent threads we cause the execution to be controlled to a large extent by the data flow – or more precisely the flow of accessibility to shared data – instead of the control flow. Most programs clearly express the control flow, but the flow of data accessibility is rather hidden. That is an important difficulty in concurrent programming.

In the present work we propose a technique to make the flow of data accessibility more visible in an object-oriented language. We want to explicitly express for each method in each object interface

- which shared data are accessible in the method,
- the kind of possible accesses to these data:
  - exclusive read-and-write accesses
  - or consistent read-only accesses (without concurrent write in between)
  - or shared accesses that need synchronization,
- and constraints on values the variables are supposed to hold before method invocation and after return (representing data dependences).

Such information is to a large extent sufficient for a compiler to

- decide where to spawn concurrent threads to ensure a continuous data flow,
- decide where to synchronize threads to avoid data races and comply with data dependences (as enforced by `wait` and `notify` in Java),

– check mutual compatibility of sequential and concurrent program parts,
– and check important properties required for continuous operation.

Programmers can profit from this approach in several ways:

– They write neither code for spawning threads nor for synchronization and still have control over concurrency and can express data dependences.
– The additional information in interfaces conforms to Design by Contract [6] – a concept familiar to object-oriented programmers. Specifications concerning concurrency no longer contradict object-oriented design principles.
– Subtyping considers concurrency according to the substitution principle [5].

We show our approach by examples in Sect. 2, discuss assertion checking in Sect. 3, present rather formal techniques in Sect. 4, address inherent problems in Sect. 5, refer to related work in Sect. 6, and give our conclusions in Sect. 7.

2 How it Works

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

```
class Buffer {
   public void put(int i) [!full() -> !empty()] { e[top++] = i; }
   public int get() [!empty() -> !full()] { return e[--top]; }
   public boolean empty() [true] { return (top == 0); }
   public boolean full() [true] { return (top == e.length()); }
   public Buffer(int s) [-> !full()] { e = new int[s]; }
   private int e[], top = 0;
}
```

Square brackets on methods represent *access constraints*. We can read expressions in square brackets as assertions (pre- and post-conditions). Accordingly, `put` is invokable only when the buffer is not full and afterwards it is not empty, and `get` only when the buffer is not empty and afterwards it is not full. The arrow between pre- and post-condition indicates state changes of the buffer; an execution of `get` requires exclusive read-and-write access to `this`. Executions of `empty` and `full` need only read-only access as indicated by the availability of an assertion (`true`) and the absence of `->` in the access constraint. A method without any assertion must not access instance variables. The constructor automatically has read-and-write access and ensures a new buffer not to be full.

Access constraints can also be specified for formal parameters and method results. However, there are no post-conditions (expressions to the right of `->`) in access constraints of method results; only conditions to the left of `->` can occur there. Access constraints on local variables, instance variables and class variables are inferred by the compiler. It would be difficult to specify them explicitly because each occurrence of a variable needs its own assertions.

We regard assertions in square brackets to be tokens as used to represent type-states. Tokens are limited resources, and a compiler can statically guarantee that actions depending on tokens are carried out at run time only if these

tokens are available. For example, the compiler can guarantee that `put` is invoked only in a buffer of type `Buffer[!full()->]` without any run-time checks. The associated access constraint implies exclusive access to a non-full buffer. Moreover, tokens can move from one method to another as needed in the computation in a statically predetermined way. In access constraints of methods and formal parameters, tokens to the left of `->` move from the caller to the callee on invocation, and those to the right from the callee to the caller. If a token occurs only on one side, then it flows only in one direction.

Buffers are useful only if several clients access them concurrently and clients compete for exclusive access. We need a further kind of tokens – *cooperative tokens* – to express competing access as shown in the following code snippet:

```
void produce(Buffer[[!full() -> !empty()] ->] p)
  { while(true) { ...; p.put(...); ... }
void consume(Buffer[[!empty() -> !full()] ->] c)
  { while(true) { ...; i = c.get(); ... }
```

There is no exclusive access to the formal parameters `p` and `c` in `produce` and `consume`. Instead, `produce` has to repeatedly acquire a temporary lock on `p` when `p` is not full to get exclusive access, and then perform actions on `p` that cause `p` not to be empty before releasing the lock. An invocation of `put` on `p` changes the state as required. The syntax of the corresponding token (pre-condition on `p`) is the same as the access constraint of `put`. This token moves from the caller to `produce` and remains there because the token occurs only to the left of `->`.[1]

A method like `produce` would be rather useless without another method like `consume` and vice versa. We express this relationship in tokens: If there is a token `[!full() -> !empty()]` in a thread, then there must be at least one further token `[!empty() -> !full()]` in another concurrent thread. In general, for each token `[`f_1 `->` f_2`]` there must be tokens `[`f_2 `->` f_3`]`, ..., `[`f_{n-1} `->` f_n`]` and `[`f_n `->` f_1`]`, eventually all in different threads; the pairwise different assertions $f_1, \ldots, f_n$ must be arrangeable in a cycle. Furthermore, the availability of a cooperative token `[`f_i `->` f_j`]` is an obligation to repeatedly invoke methods with corresponding access constraints. Otherwise the computation gets stuck with high probability.

The next example shows how the compiler can introduce concurrency:

```
void produce_consume() {
   Buffer b = new Buffer(2); // Buffer[!full()->]
   b.put(1);                 // Buffer[!empty()->]
   int i = b.get();          // Buffer[!full()->]
   produce(b);               // Buffer[[!full() -> !empty()],
                             //        [!empty() -> !full()]]
   consume(b);               // Buffer[[!full() -> !empty()]]
   produce(b);               // Buffer
}
```

[1] The token would move back on return if there was no arrow in the access constraint or the token occurred on both sides of the arrow.

The comments show the types of `b` (including inferred access constraints) after executing the statements on the same lines. First, we construct a new instance of `Buffer` and get the token ensured by the constructor. Because of this token we can invoke `put` on `b` and get another token that allows us to invoke `get`. These statements are executed sequentially without any need of synchronization because all required tokens are available. However, the cooperative token required on the argument of `produce` is not directly available. As discussed in Sect. 4 we can replace an access constraint `[!full()->]` or `[!empty()->]` with `[[!full()->!empty()],[!full()->!empty()],[!empty()->!full()]]`. We need the token required by `produce` twice because `produce` is invoked twice. For the first invocation of `produce` and the invocation of `consume` the compiler has to spawn new threads because these tokens must end up in different threads. To be exact, a new thread is necessary only if the invoked method requires just a single cooperative token on the same parameter; otherwise the invoked method is responsible for spawning the required threads. Each invocation uses up a token until nothing is left.

As shown in this example, thread creation follows a simple algorithm based essentially on the availability of tokens. We expect programmers to be able to understand the algorithm although it is usually advisable to concentrate on the flow of data accessibility instead of the thread structure.

3 Checking Assertions

By regarding assertions as tokens we introduce a token flow that can easily be followed by a compiler. Properties expressed in tokens (except of cooperative tokens) cannot accidentally change because there is no other thread or unknown alias that could modify the object state. However, we have to clarify if a property holds in the first place when introducing a new token or changing a property as in `put` and `get`: After executing the method body we have to dynamically check if the property holds (`!empty()` in `put` and `!full()` in `get` as well as in the constructor). When invoking a method we have to perform a check only in the course of acquiring a lock for a cooperative token.

Dynamic assertion checks can be avoided by using static information accessible by the compiler. We propose a set of static properties and argue that most data dependences needed for synchronization are easily expressible with them. The most important static properties are of the forms $v{:}\tau$, $v{:}c$ and $v{:}c..c'$ where v is a variable, τ a type, and c and c' are constant values. They are satisfied if v holds a value of type τ, value c, and any value in the range from c to c', respectively. After assigning a value of type τ or a constant value to v the compiler has all needed information; otherwise it will issue an error message. An important advantage of static properties are well-defined relationships between them. For example, we have the following implications: `v:3` $\Rightarrow$ `v:2..4` $\Rightarrow$ `v:1..8` $\Rightarrow$ `v:int`. Of course, we can move a token `v:3` to a method that requires a token `v:2..4`. In contrast, we cannot use a token `empty()` where a token `!full()` is required because the compiler does not know about such relationships.

Static properties of the forms v+i and v-i can be used only as post-conditions together with pre-conditions of the form v:j..j' where i, j, j' are integer constants. They specify that the value of v will be incremented or decremented correspondingly. The next variant of our example shows how we can make use of such static properties:

```
class Buffer8 {
   public void put(int i) [top:0..7 -> top+1] { e[top++] = i; }
   public int get() [top:1..8 -> top-1] { return e[--top]; }
   public Buffer8() [-> top:0] { e = new int[8]; }
   private int e[], top = 0;
}
```

This variant is shorter because we state dependences more directly in terms of variable values. The static determination of the buffer size allows us to compile the following statement sequence:

```
Buffer8 b = new Buffer8();    // Buffer8[top:0->]
b.put(1); b.put(2); b.put(3)  // Buffer8[top:3->]
```

The compiler knows that `top` is 3 after three invocations of `put` just from the specifications in the class interface. A corresponding statement sequence would not compile for `Buffer` (instead of `Buffer8`) for several reasons:

- There is no static information about the buffer size. Value 3 can be inappropriate for `top`.
- Unlike `top+1` a post-condition `!empty()` as well as `top:1..8` does not tell us if and how a method invocation modifies `top`.
- There are no invocations of `get` in the code snippet. If there were three invocations of `get`, the compiler could generate concurrent threads so that the buffer size does not matter. No concurrency is necessary for `Buffer8`.

A non-cooperative token specifies two aspects – accessibility (exclusive or consistent read-only) and a property. Accessibility is the more important aspect because with exclusive as well as consistent read-only access we can quite often dynamically determine the property as in the following example:

```
Buffer[empty()->] emptyTest(Buffer[true ->] b)
   { if (b.empty()) { return b; }; return null; }
```

The parameter `b` has an exclusive pre-condition `true` and no post-condition at all. If possible, `emptyTest` returns this parameter with an exclusive assertion `empty()`; otherwise it returns `null` which can be associated with any assertion in the same way as it can be used instead of an instance of any reference type. For each side-effect-free condition in an `if`-statement depending only on a single parameter (or a single variable or `this`) the compiler can assume the condition to be a valid property of this parameter (or variable or `this`) within the body of `if`. Hence, `b` has the needed property in the `return` statement. As special cases, a condition v `instanceof` τ implies the property v:τ, v`==`c implies v:c, and v `>=` c `&&` v `<=` c' implies v:c..c'. In this context it is not useful to distinguish between static and dynamic properties.

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

$$\frac{a \equiv b \quad b \equiv c}{a \equiv c} \qquad \frac{a \equiv b}{b \equiv a} \qquad \frac{a \equiv c \quad b \equiv d}{a, b \equiv c, d} \qquad \frac{f \Rightarrow g}{f, g \equiv f} \qquad \frac{f \Rightarrow g}{f^*, g^* \equiv f^*} \qquad \frac{a \equiv c \quad b \equiv d}{[a\texttt{->}b] \equiv [c\texttt{->}d]}$$

Fig. 1. Equivalence of Tokens Sequences

$$\frac{a \equiv b}{a \leq b} \qquad \frac{a \leq b \quad b \leq c}{a \leq c} \qquad \frac{a \leq c \quad b \leq d}{a, b \leq c, d} \qquad f \leq \epsilon \qquad f^* \leq \epsilon \qquad [f\texttt{->}f] \leq \epsilon$$

$$\frac{f \Rightarrow g}{f \leq g} \qquad \frac{f \Rightarrow g}{f^* \leq g^*} \qquad \frac{f \Rightarrow f' \quad g' \Rightarrow g}{[f\texttt{->}g] \leq [f'\texttt{->}g']} \qquad \frac{\text{check}(g)}{f \leq f, g} \qquad \frac{\text{check}(g)}{f^* \leq f^*, g^*}$$

$$[f\texttt{->}g] \leq [f\texttt{->}f'], [f'\texttt{->}g] \qquad \frac{\text{clean}(f')}{[f\texttt{->}f'], [f'\texttt{->}g] \leq [f\texttt{->}g]}$$

$$\frac{\text{new Proxy}}{f \leq [f\texttt{->}f]} \qquad \frac{\text{clean}(f), \text{ no Proxy}}{[f\texttt{->}f] \leq f} \qquad \frac{\text{write-lock}(f)}{[f\texttt{->}g] \leq f}\ (^\dagger) \qquad \frac{\text{read-lock}(f)}{[f\texttt{->}f] \leq f^*}\ (^\ddagger) \qquad \frac{f \Rightarrow g}{f \leq g^*}\ (^\ddagger)$$

Fig. 2. Subsumption of Tokens Sequences

4 Relations on Tokens and Types

Non-cooperative tokens occurring in access constraints with arrows are exclusive tokens, those in access constraints without arrows are read-only tokens. In this section we discuss tokens without their context available. To distinguish between these token kinds we mark read-only tokens with *. For properties f and g, f^* denotes a read-only token, f an exclusive token, and $[f\texttt{->}g]$ a cooperative token.

A *token sequence* is a possibly empty comma-separated list of tokens of the same kind, this is a sequence of exclusive tokens, a sequence of read-only tokens, or a sequence of cooperative tokens. An access constraint is a square bracket containing either a sequence of non-exclusive tokens or two sequences of tokens that are not read-only (separated by an arrow). Equivalence of token sequences is defined in Fig. 1 where f, g denote properties, a, b, c, d token sequences, and ϵ the empty sequence.

Fig. 2 defines a subsumption relation on token sequences. If $a \leq b$ holds, then we can replace token sequence a with token sequence b where a is available and b is required. Most tokens can be removed, that is, replaced by ϵ. However, tokens of the form $[f\texttt{->}g]$ with $f \not\Rightarrow g$ (f does not imply g or f is not known to imply g) must not be removed because such tokens have to be repeatedly used for continuous operation. Many rules in Fig. 2 depend on conditions like implications between properties. The compiler has to guarantee that these conditions are satisfied at run time when applying these rules. For the condition check(g) the compiler has to ensure that g is satisfied, for example, by an `if`-statement. There is a rule allowing the compiler to replace a single cooperative token by two different cooperative tokens. This rule helps us to ensure that properties in cooperative tokens always build cycles. Another rule can reduce the number of

properties in a cycle if no synchronization depends on the withdrawn property f' as expressed by clean(f').

The rules in the last line of Fig. 2 relate tokens of different kinds. Since all tokens in a token sequence must be of the same kind, they must be replaced simultaneously. The compiler has to introduce a new proxy for synchronization (locking) at the position where exclusive tokens are replaced by cooperative tokens. In this proxy the compiler sets exclusive locks when replacing a corresponding cooperative token with an exclusive token and shared read-only locks when replacing a cooperative token with a read-only token. Rules marked with (†) or (‡) are applicable only for a limited amount of time when invoking methods with access constraints corresponding to the tokens at the right-hand-side of $\leq$. On return the tokens at the left-hand-sides of $\leq$ become valid again. Rules marked with (‡) allow the compiler to invoke several such methods simultaneously.

We regard access constraints as annotations of types, no matter whether they are expressed explicitly (for formal parameters and result types) or inferred by the compiler (for local variables, instance variables and class variables). As a consequence, access constraints play an important role in (behavioral) subtyping:

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

Subtyping of types with such annotations resembles subtyping with assertions where a subtype can have weaker pre-conditions (requiring less restrictive tokens) and stronger post-conditions (ensuring more restrictive tokens) than supertypes [5]. Of course, types promising exclusive access to their instances can be used where only read-only access is needed, but not the other way around.

As usual, types of actual parameters must be subtypes of formal parameter types. In the proposed approach, access constraints associated with these types determine to a large extent when to spawn threads and apply synchronization. Most access constraints associated with actual parameters are inferred by the compiler based on the rules in Fig. 2. Essentially, the compiler

- determines all tokens needed by a variable in method invocations (specified in the method signature) as well as tokens becoming available thereby,
- relates tokens becoming available by one invocation with those needed in the next invocation,
- and finally checks for each assignment if all tokens needed by the left-hand-side can be supplied by the right-hand-side.

The rules have to be applied in the last two steps, and as a side-effect of rule applications the compiler possibly has to add code for generating new synchronization proxies, locking, etc. However, the rules are not deterministic: The compiler has some freedom in spawning threads as the following example shows:

```
void test(Buffer[!full() -> !full()] b)
   { b.put(1); b.get(); }
```

It is easy to invoke `put` and `get` sequentially. However, it is also possible to spawn separate threads for `put` and `get` by transforming tokens as follows:

```
Buffer[!full()->]                                   (b = new Proxy on b)
  ≤ Buffer[[!full()->!full()]]
  ≤ Buffer[[!full()->!empty()], [!empty()->!full()]]
                (invoke put and get, then ensure clean(!b.empty()))
  ≤ Buffer[[!full()->!full()]]
                 (ensure clean(!b.full()) and remove Proxy from b)
  ≤ Buffer[!full()->]
```

The concurrent solution suffers from a large overhead caused by creating and removing a synchronization proxy, acquiring locks, and dynamically ensuring that synchronization no longer depends on `!b.full()` and `!b.empty()`. In each case, `put` and `get` will be executed sequentially.

To avoid such overhead we require from the compiler to introduce cooperative tokens only if there is no solution using non-cooperative tokens. This simple strategy causes the inference of access constraints to become deterministic. Nonetheless, for independent method invocations the rules do not provide enough information to decide between concurrent and sequential execution:

```
void test2(Buffer[!full()->!empty()] b,
           Buffer[!empty()->!full()] c)
   { b.put(1); c.get(); }
```

The two invocations can safely be executed in parallel without any synchronization because they cannot access the same variables. However, it is not clear if parallelism can compensate for the overhead of thread creation in this case.

5 Some Nasty Details

Token-based techniques like the one we use in our approach usually suffer from a number of difficulties. In this section we address the most important ones.

Recursion. Many token-based techniques do not support recursion because we consider methods like `put` to be atomic actions executed in isolation from other methods. An invocation of `put` within `put` could occur in an inconsistent state and thereby compromise isolation. Programmers usually prefer recursion over isolation. Therefore, the approach proposed in this paper supports recursion in the sequential as well as concurrent case. Technically speaking, by synchronization we can convert a cooperative token into an exclusive one that can be used without further synchronization, for example, in a recursive invocation. It is also possible to have several concurrent recursive invocations by introducing a further synchronization proxy and replacing the exclusive token with cooperative tokens. There can simultaneously exist any number of synchronization proxies on the same object, all but at most one of them with an exclusive lock. By the use of assertions as tokens we still can ensure the required isolation. Pre-conditions have to be satisfied also for recursive invocations.

Token Loss. Especially in exceptional cases we easily lose tokens. For example, we lose a token moved into a method if the method terminates with an unexpected exception. We can dynamically reconstruct the property expressed by the assertion of a token, but we cannot reconstruct the accessibility information by hand because usually we do not know anything about other threads that may have got the token from the exceptionally terminated method. Being careless we can lose tokens even in the normal control flow – for example, tokens in access constraints of instance variables belonging to objects ready to be garbage collected or simply being untraceable.

Token loss is caused by many factors to be addressed differently. First of all we have to organize the software so that we do not lose still needed tokens in the normal control flow. For example, we put object references together with corresponding tokens into a repository where users can easily find them together on demand; see the notes on arrays and collections. Expected exceptions can carry object references together with tokens. For unexpected exceptions thrown by the run-time system we have no satisfactory solution, but several approaches:

- The exception handling mechanism creates a repository containing all object references (together with their tokens) available as local variables where the exception occurred. The exception handler is responsible for exploring this repository and making use of needed tokens. This approach causes an overhead for throwing as well as handling exceptions, and tokens associated with instance variables at the time when the exception occurred will probably be untraceable by the exception handler.
- When a method terminates with an exception, the exception handling mechanism automatically returns all accessibility information provided by the caller on invocation (but all corresponding assertions will simply be `true` for exclusive tokens). If values have been assigned to variables annotated with tokens to be returned, the exception handling mechanism has to write `null` to these variables, and if threads have been spawned and locks acquired because of such tokens, then the threads must be terminated and the locks reset. This approach allows for simple implementations of exception handlers, but it is probably difficult to understand which variables will be set to `null`.
- Similar as above, the exception handling mechanism returns accessibility information, but only for tokens provided by the caller *and* to be returned to the caller according to the normal control flow. In this approach, the exception handling mechanism has to set only those variables to `null` that would contain another value or would not be annotated with this token at normal termination of the method. However, some tokens can be lost.

Stopping Continuous Operation. The support of continuously operating systems through cooperative tokens is a main feature of the proposed approach as the `produce_consume` example shows [13]. As long as there is a producer there must also be a consumer and vice versa. Once the continuous operation has been started, the token needed by the consumer cannot come together with the token

needed by the consumer, and hence no rule in Fig. 2 is applicable to combine these tokens again. In the normal case we cannot stop the producer independently from the consumer. To stop the producer (or any continuous operation in general) in exceptional cases we introduce a *stop mode.* After entering stop mode (by invoking a specific method) it is no longer possible to get a lock on the corresponding synchronization proxy. All threads waiting for a lock will get an exception instead. There is no way back from stop mode.

Arrays and Collections. Each occurrence of a variable has its own access constraint computed by the compiler. For arrays and other collections of items the access constraint can differ for each item. Since the compiler does not know array indexes in general, it is impossible to statically compute different access constraints for different indexes. Similar problems arise for other collections of items. There are two ways to solve these problems:

- Each item in the collection is associated with the same access constraint. This approach works quite well if always the same operation is applied to all items in a collection. Otherwise we have to assume that a token is removed from each item if it is removed from one item, and we lose tokens added to only one item (but not all items).
- Each item in a collection is associated with its own access constraint, and these access constraints are managed dynamically. This approach is more flexible and powerful, but access constraints are no longer a purely static concept and tokens are run-time entities. Explicit dynamic checks of the availability of tokens can be helpful (where brackets on `a` denote an array, those on `Buffer` access constraints):

```
Buffer a[] = ...;
if (a[i] instanceof Buffer[!empty()->]) { a[i].get(); }
```

There is an appropriate combination of both approaches: The compiler statically computes tokens available for all items, and the run-time system stores additional tokens of some items in the collection. Tokens computed by the compiler are directly usable while explicit dynamic checks are necessary before using additional tokens. As a generalization it is useful to associate additional tokens to all variables with a corresponding declaration. The additional tokens are accessible through type checks with `instanceof` and through type casts.

6 Related and Future Work

The proposed approach has similarities with the SCOOP model of concurrency [7]. Both, the SCOOP model and our model, use assertions and disclose information on variables for synchronization. However, the way how and the time when such information becomes available to clients is different: In the SCOOP model every client can get access to corresponding variables at run time while in our model much information is statically available and usually only few clients

can access the variables at run time. Tokens give additional information that allows the compiler to automatically spawn and synchronize concurrent threads.

There are many approaches to ensure unique access [3], most of them by avoiding aliases. The token-based approach used in the present work has been developed from a process type model [9–11] for active objects – essentially an object-oriented variant of linear types [4]. This concept restricts the way how to access objects without preventing aliasing [12, 13].

Synchronization based on tokens has a long tradition: Petri Nets have been explored for nearly half of a century as a basis of synchronization [8]. In general, expressing states by abstract tokens has clear (both practical and theoretical) advantages over expressing them more concretely by values in instance variables: Tokens are much easier tractable than concrete states especially when used in a static analysis. Many proposals use tokens to express abstract object states [1, 2, 14]. In our approach we combine abstract tokens with concrete variable values to get more flexibility.

Many programming languages consider concurrency to be orthogonal to the object model. By the use of object accessibility and assertions as tokens we exploit inherent relationships between objects as a basis for concurrency and synchronization. Concurrency is subordinate to object-oriented factorization. Other than in the concurrency model in Java, a thread that acquires a lock in our approach is in no way privileged compared to other threads because only the availability of tokens counts, not the thread identity. This property is an important step towards modularity of synchronization. A further step is the support of recursion while still executing methods atomically and in isolation. Since the compiler deals with concurrency on a class by class basis, concurrency is almost completely modular. However, there remains a small non-modular rest: To prevent deadlocks the compiler needs global information [13]. In this respect our approach does not differ from many other approaches.

The present work is in an early stage. Currently there is no implementation, and no practical evaluation of this approach has been carried out so far. In future work we want to address (among others) the following topics:

- The compiler has some freedom to decide between sequential and concurrent execution. We need appropriate heuristics to make a beneficial decision.
- Our proposal provides a foundation for assured continuous operation [13]. More work is needed to strengthen the guarantees given by the compiler. Deadlock prevention in at least some cases is one of the goals.
- We need more advanced concepts and more experience to deal with time as well as with exceptional cases (like exception handling and stop mode) in a practical setting.

7 Conclusions

We have explored a way to express the flow of data accessibility in object interfaces so that a compiler can use this information to execute methods in concurrent threads and provide synchronization between them. Based on this in-

formation the compiler can always determine if it is necessary or unfeasible to spawn concurrent threads; for independent computations further information is necessary to decide if it is advantageous to do so. Programmers specify the flow of data accessibility essentially in the form of assertions that are considered to be tokens. As a consequence, object-oriented design principles dominate the program structure, not concurrency. The approach supports subtyping and ensures to some extent continuous operation of the system.

References

1. Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *ESEC/FSE-13*, pages 217–226, Lisbon, Portugal, September 2005. ACM Press.
2. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP 2004 – Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, Oslo, Norway, June 2004. Springer-Verlag.
3. Sophia Drossopoulou, David Clarke, and James Noble. Types for hierarchic shapes. In *ESOP*, pages 1–6, 2006.
4. Naoki Kobayashi, Benjamin Pierce, and David Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.
5. Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. Proceedings OOPSLA'93.
6. Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.
7. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition edition, 1997.
8. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
9. Franz Puntigam. Type specifications with processes. In *Proceedings FORTE'95*, Montreal, Canada, October 1995. IFIP WG 6.1, Chapman & Hall.
10. Franz Puntigam. Coordination requirements expressed in types for active objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 367–388, Jyväskylä, Finland, June 1997. Springer-Verlag.
11. Franz Puntigam. *Concurrent Object-Oriented Programming with Process Types*. Der Andere Verlag, Osnabrück, Germany, 2000.
12. Franz Puntigam. See the pet in the beast: How to limit effects of aliasing. In *International Workshop on Aliasing, Confinment and Ownership in object-oriented programming (IWACO 2007)*, Berlin, Germany, July 2007.
13. Franz Puntigam. Synchronization as a special case of access control. *Electr. Notes Theor. Comput. Sci.*, 241:113–133, 2009.
14. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.