Conditional Concurrency Combinators

Paweł T. Wojciechowski
Poznań University of Technology
60-965 Poznań, Poland
Pawel.T.Wojciechowski@cs.put.edu.pl

ABSTRACT

We introduce the calculus of conditional concurrency combinators, allowing the programmer to declare synchronization code separately from the main code. The calculus comprises a set of novel concurrency combinators embedded in a small language which has both object-oriented and functional features. The concurrency combinators can be seen as behavioural types which can be used to express synchronization policies for concurrent threads, such as mutual exclusion (or atomicity), barrier synchronization, and policy revocation. Our language constructs allow conditional and complex synchronization policies to be declared for classes, objects, and expressions. In the paper, we present preliminary results of ongoing work. We define the calculus and explain its semantics informally. We also discuss the advantages of our language constructs by using them to solve a few example classical synchronization problems.

Keywords: concurrency, synchronization, atomicity, language design, lambda calculus.

1. INTRODUCTION

With the proliferation of multicore processors parallel programming, once confined to high-end servers and scientific computing, has become a mainstream concern. Unfortunately, writing parallel programs using traditional synchronization primitives is notoriously difficult, time consuming, and error-prone. Thus, to bring parallel programming into the mainstream of software development, we are in an urgent need of better programming models. In this paper, we propose novel language primitives that can be used for synchronization of concurrent threads in programs, and for controlling access of threads to shared data. The key idea of our design is to separate a program's functional behavior and any synchronization constraints imposed on it, and to use a small set of concurrency combinators to express the desired synchronization policy.

Multithreaded programming is considerably more difficult than implementing sequential programs. Traditional concurrency constructs, such as monitors with condition variables, are used by the programmers to express synchronization at the very low level of individual accesses to shared data. This approach compromises both a good understanding of the synchronization and of the application logic since it is hard to see at first glance what the code does. The notions of semantic rôles such as producers and consumers, which are essential for the understanding of a given synchronization policy, tend to dissolve as lines of code accumulate, just as the logical essence of a sequential program gets lost when expressed in, say, an assembly language. Moreover, synchronization constructs are usually entangled with instructions of the main program, which means that the correctness of concurrent behaviour is bound to the correctness of the entire application; this feature complicates code maintenance, composition, and reuse.

In this paper, we introduce a small language (or calculus) of *conditional concurrency combinators*. Our language allows the programmers to declare the required synchronization policy for concurrent threads using a set of language constructs. They can be used for expressing mutual exclusion (or atomicity), barrier synchronization, and policy revocation. The synchronization policy can be declared for any code, for all objects of a given class, for particular objects, and for a given expression. If required, the policy can be locally revoked for some code. Complex synchronization policies can be defined using a conjunction of combinator arguments, and a conjunction of synchronization constraints. Moreover, we can specify a condition for each policy declaration using any program variables. Such a policy will be enforced at runtime if the condition evaluates to true. The main advantages of our language are:

- The declared synchronization policy can be modified and customized by the programmer without changing the main program code. Any changes to program code, which may require to revise synchronization policy, are not subjected to an inspection of the whole program but only of declared synchronization constraints. Thus, our language supports code reuse, making programming easier and less error-prone.
- The possibility to declare synchronization at different levels of abstraction (classes, objects, and expressions) enhances expressiveness of our language when compared to traditional synchronization constructs. For example, the revoke combinator allows atomicity to be locally relaxed if required for progress or dirty reads. For illustration, we present a few example problems solved using concurrency combinators in Section 5.

LaME '12, June 13, 2012, Beijing, China.

In the paper, we report on ongoing work and present our language design. Firstly, we motivate our design in Section 2 by discussing a race-free program expressed using traditional synchronization constructs, atomic blocks, and concurrency combinators. Then, we explain the semantics informally in Section 3, and present the abstract syntax of our language in Section 4. In Section 5, we illustrate the expressiveness of our language using some classical synchronization problems. Finally, we outline related work in Section 6, and conclude in Section 7. In the future, we would like to formalize the operational semantics and a type system, and develop techniques for reasoning about correctness of programs expressed in the calculus.

In our previous work [27], we designed a type system to verify if a declared synchronization policy matches a program. For example, if a policy is such that two expressions must be executed in parallel or in a certain order but none execution of a program can exhibit such behavior, the type checking fails. In this paper, we revise and extend the initial idea from [27], and present a different set of concurrency combinators. They are more flexible and practical. There is also no need to verify satisfiability of combinators statically since they can always be satisfied at runtime. While similar work exists (see [10, 5, 22, 21, 17] among others) and example language constructs have been proposed (see [20, 21, 16, 17]), to our best knowledge, the calculus for declarative synchronization presented in this paper is novel. We discuss some related work in Section 6.

2. MOTIVATING EXAMPLE

Below is a class `SyncCounter` expressed in a Java-like language, in which the construct `synchronized` is used to avoid data races on variable `c` accessed by three methods of this class:

```
public class SyncCounter {
  private int c = 0;
  public synchronized void increment() { c++; }
  public synchronized void decrement() { c--; }
  public synchronized int value() { return c; }
}
```

The synchronized keyword specifies (declaratively) that the methods should not run at the same time on any object of this class. Thus, it is not possible for any concurrent invocations of these methods on the same object to interleave. This guarantees that changes to the state of the object are applied consistently. Thus, synchronized methods enable a simple strategy for preventing memory consistency errors.

In our language of conditional concurrency combinators, we can express the above program as follows:

```
class SyncCounter {
  c = 0
  increment() {c := c+1}
  decrement() {c := c-1}
  value() {c}
  sync SyncCounter.ANY isol [ANY]
}
```

Annotating a class `SyncCounter` with the atomicity (or isolation) combinator `SyncCounter.ANY isol [ANY]` using a keyword `sync` has two effects. First, if an object of this class is visible to more than one thread, all its methods are executed atomically. This corresponds to annotating all methods of class `SyncCounter` with the `synchronized` keyword. Similarly to `synchronized`, we do not require the implementation to use locks. We only require that the concurrent execution of synchronized methods is serializable. Second, any concurrent accesses of objects' instance fields are synchronized (or race-free). This corresponds to declaring `c` as a non-volatile variable in a Java-like language. We explain the syntax in more detail in Section 3.

Our language allows for more expressiveness. For example, more subtle synchronization policies can be declared using various combinator arguments, which include the universal quantifier `ANY` and the names of object fields and methods. Below, we declare synchronized accesses to variable `c`, and the self-isolation of methods:

```
class SyncRelaxed {
  c = 0
  increment() {c := c+1}
  decrement() {c := c-1}
  value() {c}

  sync SyncRelaxed.c isol [ANY]
    ∧ SyncRelaxed.increment isol SyncRelaxed.increment
    ∧ SyncRelaxed.decrement isol SyncRelaxed.decrement
}
```

The above policy is weaker than before: The concurrent executions of `increment` on the same object are isolated (serializable), and also the concurrent executions of `decrement` on the same object are isolated. However, the concurrent execution of `increment` and `decrement` can be arbitrarily interleaved. If both methods would happen to read the same value of `c`, no results of their execution will be visible.

We allow conditional concurrency combinators to be used as typing declarations of arbitrary expressions. For instance, consider implementing a shared buffer within an array. The core of a Java-style design could be:

```
public synchronized int get() {
  int result;
  while (items == 0)
    wait ();
  items --;
  result = buffer[items];
  notifyAll ();
  return result;
}
```

Assuming that `get` is a method of a class `P`, we can express the above program in our language using a conditional concurrency combinator, as below:

```
int get() {
  sync (items != 0) P.get isol [ANY]
  in
    items := items - 1;
    buffer[items]
}
```

The code following `in` will be executed atomically, and its execution is blocked until condition `items != 0` evaluates to true. Conditions may contain arbitrary Boolean expressions, which can refer to objects and variables. Below we express the above program using the `atomic` construct [6], implementing *conditional critical regions (CCR)* relying on *transactional memory (TM)* [9, 23]:

```
public int get() {
  atomic (items != 0) {
    items --;
    return buffer[items];
  }
}
```

Although both programs appear similar, there are important differences which we explain below. The TM-based CCR is a synchronization construct designed to replace locks by transactional memory. The majority of TM implementations developed for shared memory multiprocessors (see e.g., [6, 7, 19] among others) are based on optimistic concurrency control. There are also TM frameworks [8] which can support a variety of concurrency control algorithms. In optimistic TMs, atomic transactions run in parallel and any conflicting transactions are automatically rolled back and reexecuted. TM avoids typical lock-induced deadlocks (but see [2] and Section 5). But irrevocable I/O actions may be forbidden inside atomic transactions. Some TMs also support a 'retry' or 'abort' construct which we do not have.

TM implementations obey either *weak* or *strong* atomicity [2]. The former semantics is guaranteed with respect to transactional code only, while the latter semantics requires that every non-transactional access to a variable is serialized with respect to all concurrent transactions that share the same variable. The weak atomicity gives some additional expressiveness but it is generally unsafe since any non-transactional code can read uncommitted data, and so potentially invalidate invariants about code atomicity. On the other hand, strong atomicity ensures strict safety but can unnecessarily restrict concurrency.

Concurrency combinators give more flexibility since they allow both strong and weak semantics. In particular, strong atomicity can be revoked for code that can read uncommitted state without invalidating any important invariants. For example, we used `ANY` to express isolation of `get` with respect to any other code (strong atomicity). Alternatively, we could declare atomicity of `get` with respect to some code only, thus weakening atomicity.

We can also relax declared atomicity with respect to some code. For example, consider a method `Q.dirty_read` which scans `buffer`. To increase concurrency, we locally revoke atomicity of `get` with respect to `dirty_read`, as below:

```
sync
  P.get !isol [Q.dirty_read]
in
  e
```

The revocation of atomicity of `get` is observed only by the executions of method `dirty_read` invoked by expression `e`.

Both atomic transactions and concurrency combinators improve over locks due to the declarative style of expressing synchronization. To express a synchronization policy, such that a consumer is blocked until a buffer is nonempty and a producer is blocked if a buffer is full, a method `put` also has to be protected by CCR (or synchronized by a concurrency combinator), much like when using locks with condition variables. However, the lock-based synchronization is not local—to verify the correctness of `get`, the code of `put` must also be inspected to check if both methods synchronize using exactly the same lock. On the other hand, the correctness of `put` and `get` can be verified independently if declarative synchronization is used.

In [2], the authors invalidate the intuition that transactions are strictly safer than lock-based critical sections, that strong atomicity is strictly safer than weak atomicity, and that transactions are always composable. Despite these subtleties, `atomic` supports the declarative style, which simplifies programming. However, it does not allow for much more expressiveness than traditional synchronization constructs, such as locks (or monitors) with condition variables.

On the other hand, conditional concurrency combinators are more flexible (albeit for the cost of additional complexity compared to `atomic`). By using the combinator `P.get isol [ANY]`, we declared `get` to be atomic with respect to any other code that may share data. Then, we were able to locally revoke this policy for some concurrent code. This allows to increase concurrency when it is safe to do so. However, contrary to open-nesting in TM [18], no conflicts can occur and so our approach is safer to use for non-experts. Policy relaxation can also be used to avoid deadlock. We demonstrate this feature in Section 5, where we also demonstrate the barrier synchronization combinator.

Unfortunately, it is not clear yet how to efficiently implement concurrency combinators. It is likely that any practical implementation will be a trade-off between expressiveness and efficiency. We leave this problem for future work.

3. CONCURRENCY COMBINATORS

An expression `sync` SP `in` e declares synchronization policy SP for evaluation of expression e. When e reduces to a value, SP ceases. The value is returned as the value of the whole expression `sync` SP `in` e. The policy is defined using *concurrency combinators*, denoted a, b, and c. The simplest form of concurrency combinators is X s $[Y]$, where s describes synchronization to be enforced on combinator arguments X and Y. If `sync` SP `in` e is defined as part of a class, declared synchronization SP is valid for all objects of this class, and e (of type `Unit`) is executed just after object creation; the part `in` e can be omitted.

An expression `sync` (*condition*)a `in` e can be used to define *conditional concurrency combinators*: the execution of e synchronized by concurrency combinator a is blocked until *condition* is true. We use some syntactic sugar: `sync` a `in` e means `sync` (`true`)a `in` e, and `sync` (*condition*)a means `sync` (*condition*)a `in` (). We also allow a conditional of the form `if` *condition* `then` a `else` b, which returns a combinator a if *condition* is true and b otherwise.

Concurrency combinators are *compositional*: expressions with a declared synchronization policy can be nested in other expressions guided by another synchronization policy. However, we require an invariant that any inner policy cannot be invalidated by an outer policy. This requirement can be enforced by a type system (future work).

The arguments of combinators can be either expressions, or object fields and methods, or compounds of these things. The arguments can be *specific*: e.g., $o.n$ denotes a field or method n of some object o. They can also be *generic*: e.g., o.`ANY` denotes *any* fields/methods of object o, $P.n$ denotes a field/method n of *any* object of class P, P.`ANY` denotes *any* fields/methods of *any* object of class P, and `ANY` denotes any code. A concurrency combinator is valid for all possible instances of its arguments, e.g., the combinator $P.n$ `isol` `[`Q`.ANY]` declares that a field (or a method) n of any object of class P will be isolated with respect to any field *and* any method of any object of class Q.

In the most general case, `ANY` can be used to denote *any* instance of any expression among those that must be synchronized by a given combinator, where the precise semantics of `ANY` depends on the used combinator. For instance, $o.n$ `isol` [`ANY`] declares *strong* atomicity of $o.n$, i.e., the execution of method n (or access of field n) on object o must be atomic with respect to any other code.

Complex combinator arguments can be created using a conjunction $\oplus$. For instance, a combinator $X\ s\ Y \oplus Z$ declares synchronization constraint s between X and a compound of Y and Z; the $\oplus$ binds stronger than synchronization names. This policy does *not* declare any synchronization constraints between Y and Z (which can be executed by the same or different threads).

Compound combinator arguments can be defined using an expression `let` $A \leftarrow X$ `in` e, where a new compound name A binds in e; the name A can be used to define synchronization policies. $\wedge$ can be used to define a conjunction of synchronization policies to be enforced in the expression e following. The above idioms allow a complex synchronization policy to be defined. For instance,

```
let A ← X ⊕ Y in
sync X isol Y ∧ Z isol A in
  e
```

declares mutual isolation between X and Y and also between Z and a compound of X and Y, to be enforced in e.

Below we define primitive concurrency combinators of the form $X\ s\ [Y]$, where $X \neq$ `ANY` and expressions referred to by X and Y are assumed to be executed by separate threads (otherwise no synchronization is necessary):

The *atomicity* (or *isolation*) combinator X `isol` $[Y]$ declares X to be atomic (or isolated) with respect to Y, i.e., the entire set of operations that X contains appears to Y to take place indivisibly.

The *barrier* combinator $X \bowtie [Y]$, where $Y \neq$ `ANY`, declares a synchronization barrier for X: the last write in X will be blocked until another concurrent thread writes in Y for the last time (Y is not blocked). $\bowtie$ could also be encoded using conditional `isol` (Section 5).

The *revoke* combinator $X\ !\,s\ [Y]$ declares a synchronization policy that revokes any valid synchronization constraint s declared for X with respect to Y, where the constraint is either atomicity ($s =$ `isol`), or barrier ($s = \bowtie$). The revocation of s affects Y only, and does not change any synchronization held between X and any other code. Moreover, any synchronization policy defined for Y (or other code) and any components of X remains operational.

Let p be either s or $!\,s$. We use syntactic sugar $X\ p\ Y$ for $X\ p\ [Y]\ \wedge\ Y\ p\ [X]$. We also write $X\ p$ `self` for $X\ p\ X$. Combinators $X\ p\ Y$ are symmetric, i.e., $X\ p\ Y \equiv Y\ p\ X$, while $X\ p\ [Y]$ $(X \neq Y)$ is not, but $X\ p\ [X] \equiv X\ p\ X$. X `isol` Y declares atomic (or isolated) execution of X and Y that is equivalent to a serial execution of X and Y.

The implementation may rely either on the optimistic concurrency control (e.g. using transactional memory), or the pessimistic concurrency control, or some combination of the two. In the former case, conflicting atomic blocks are rolled back and reexecuted. However, we assume a reasonable implementation of combinators: any concurrent accesses of objects that are race free are not instrumented, and so can be executed in parallel. A *race* occurs if two threads access the same data and at least one access is a write operation.

4. THE CALCULUS OF COMBINATORS

In this section, we define a class-based object calculus of conditional concurrency combinators. It builds on the call-by-value λ-calculus extended with basic object features. The syntax of our calculus is in Figure 1. Below we describe language constructs that were not explained in the last section about concurrency combinators. We use the following notation: P, Q range over class names; f ranges over object field names, and m ranges over method names.

Types include the base type `Unit` of unit expressions, which abstracts away from concrete ground types for basic constants (integers, floats, etc.), the type of `Boolean` values, and the type $\overline{t} \to t'$ of functions and class methods.

A *class* has declarations of its name (e.g. `class` P) and the class body $\{\overline{f} = \overline{v}, \overline{M}, e_s\}$, where $\overline{f} = \overline{v}$ is a sequence of fields (data containers) accessible via names $\overline{f}$ and instantiated to values $\overline{v}$, $\overline{M}$ is a sequence of object methods, and e_s is an optional synchronization expression `let` $A \leftarrow X$ `in` e or `sync` $(condition)a$ `in` e. Class inheritance and object constructor methods can be added to the calculus, in the style of Featherweight Java (FJ) [11].

A *method* of the form $t\ m\ F$ has declarations of a type t of the value that it returns, its name m, and its body F. Objects can refer to their own methods with *this.m*, where *this* is a special variable (we usually omit *this* for simplicity). A method's body is a function abstraction of the form $\overline{x} : \overline{t} = \{e\}$ (we adopted the C++ or Java notation, instead of the usual $\lambda\overline{x} : \overline{t}.e$ from the λ-calculus).

Access control is not modelled (all fields and methods are public). To be able to express synchronization on the same variable among all objects of a given class, the calculus could be extended with `static` to mark these variables.

A *value* is either an empty value () of type `Unit`, a null object (), an object instance, e.g. `new` P, a Boolean value `true` or `false`, or a function abstraction, e.g. $\overline{x} : \overline{t} = \{e\}$. Values are first-class, i.e., they can be passed as arguments to functions and methods, and returned as results or extruded outside objects. (Typing could be used to forbid extruding functions that contain the *this* references.)

Basic expressions e are mostly standard and include variables, values, field/method selectors, function/method applications, `let` binders, field assignment $e := e$, and Boolean expressions (the latter are omitted in Figure 1). We can write, e.g., $x.f := v$ to overwrite a field f of object x with a value v, or we can write, e.g., $x.m\ v$ to call a method m of object x. We use syntactic sugar $e_1; e_2$ (sequential execution) for `let` $x = e_1$ `in` e_2 (for some x, where x is fresh).

The calculus allows multithreaded programs by including an expression `fork` e, which spawns a new thread for the evaluation of expression e. This evaluation is performed only for its effect; the result of e is never used. `fork` can also be used to express asynchronous object calls, as in `fork` $A.m\ v$.

5. SYNCHRONIZATION PROBLEMS

To demonstrate expressive power of our language, we use concurrency combinators to provide solutions to example classical synchronization problems.

5.1 Shared/Exclusive Locking

Below we explain how to express the synchronization policy known as shared/exclusive locking (or readers/writers locking). Most commonly this policy is used when some shared data are being read and written by various threads. Multiple threads are allowed to read the data concurrently, but a thread modifying the data must do so when no other thread is accessing the data.

With locks, this policy is implemented using four methods: any thread wanting to read data calls `AcquireShared`, then reads the data, then calls `ReleaseShared`. Similarly any

Variables	x, y, z, o	$\in$	*Var*		
Comb. arg. names	A, B	$\in$	*Lab*		
Class names	P, Q	$\in$	*Lab*		
Field names	f				
Method names	m				
Selector names	n	$\in$	*Sel*	::=	$f \mid m$
Types	t			::=	$P \mid \texttt{Unit} \mid \texttt{Boolean} \mid \overline{t} \to t'$
Combinator args	X, Y			::=	$e \mid e.\texttt{ANY} \mid P.n \mid P.\texttt{ANY} \mid \texttt{ANY} \mid X \oplus Y \mid A$
Combinators	a, b, c			::=	$X\ \texttt{isol}\ [Y] \mid X \bowtie [Y] \mid X\ \texttt{!isol}\ [Y] \mid X\ \texttt{!}\bowtie\ [Y] \mid a \wedge b \mid \texttt{if}\ e\ \texttt{then}\ a\ \texttt{else}\ b$
Funct. abstractions	F			::=	$\overline{x} : \overline{t} = \{e\}$
Methods	M			::=	$t\ m\ F$
Classes	K	$\in$	*Class*	::=	$\texttt{class}\ P\ \{f_1 = v_1, \ldots, f_k = v_k,\ M_1, \ldots, M_n\} \mid$
					$\texttt{class}\ P\ \{f_1 = v_1, \ldots, f_k = v_k,\ M_1, \ldots, M_n,\ e_s\}$
Values	v, w	$\in$	*Val*	::=	$() \mid \texttt{new}\ P \mid \textbf{true} \mid \textbf{false} \mid F$
Expressions	e	$\in$	*Exp*	::=	$x \mid v \mid e.n \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid e := e \mid \texttt{fork}\ e \mid e_s$
Sync. expressions	e_s	$\in$	Exp_s	::=	$\texttt{let}\ A \leftarrow X\ \texttt{in}\ e \mid \texttt{sync}\ (e)a\ \texttt{in}\ e$

We work up to alpha-conversion of expressions throughout, with $\overline{x}$ binding in e in an expression $\overline{x} : \overline{t} = \{e\}$, and x binding in e in an expression $\texttt{let}\ x = e\ \texttt{in}\ e$.

Figure 1: The class-based object calculus of conditional concurrency combinators

thread wanting to modify the data calls `AcquireExclusive`, then modifies the data, then calls `ReleaseExclusive`. The implementation can range from a simple one to a fairly complicated one, dealing with spurious wake-ups, spurious conflicts and starvation (see e.g. [1], for example code).

Consider the RW class defining the `read` and `write` methods to be called by readers and writes. By using concurrency combinators, we can define the scheduling policy for all objects of this class, as follows.

```
class RW {
  v = 0
  read () = { v }
  write (x:Int) = {v := x}

  sync RW.write isol RW.read ∧ RW.write isol RW.write
}
```

The synchronization code takes only one line. Moreover, the shared/exclusive policy defined for objects of class `RW` can be easily refined or overwritten for some objects of class `RW` in some expressions. Below we illustrate the idea:

```
let o = new RW in
  sync
    o.write isol o.write ∧ o.write !isol o.read
  in e
```

For evaluation of expression e, we defined a new local read/write synchronization policy for an object `o` of class `P`. The shared/exclusive policy inherited from class `P` is overwritten by a simpler policy that defines an exclusive locking only for writers, and revokes synchronization between readers and writes. This new policy is valid only for object `o`.

We decided (perhaps controversially) that this policy change is valid only for the evaluation of expression `e` (including any threads spawned by `e`). If object `o` is accessed outside the scope of `e`, the original policy declared in `RW` is valid. The rationale behind this design choice is to support static verification. To verify correctness of components independently of other code which may be unknown, the synchronization policy should be declared for the components (and so inferred statically), not obtained dynamically.

Note that the above program can also be expressed as:

```
let o = new RW in
  sync
    o.write !isol o.read
  in
    e
```

since the synchronization constraint `o.write isol o.write` is already inherited by object `o` from the declaration of class `P`. Accesses to any other objects of class `P` obey the original shared/exclusive synchronization policy.

5.2 Five Philosophers

Below is an example implementation of the classical five philosophers problem: We define a class `Fork` implementing two methods `take` and `putdown` a fork, parameterized by philosophers who want to acquire the forks:

```
class Fork {
  p = ()
  take (x:Phil) = {p:=x}
  putdown () = {p}
}
```

Then, we define a class `Phil` implementing a method `take_forks` that tries to acquire two forks at once. This method should be atomic to avoid deadlock which might occur if all philosophers would take either all left or all right forks only. Thus, we extend the class definition with a concurrency combinator declaring `take_forks` to be executed in isolation with respect to any concurrent calls of this method:

```
class Phil {
  take_forks (left : Fork, right : Fork) = {
    left.take(this); right.take(this);
    ...  /* eating */
    left.putdown(); right.putdown()
  }
  sync Phil.take_forks isol self
}
```

Note that forks are obsolete in the above program, since the isolation policy imposed on method `take_forks` already ensures mutual exclusive of "eating". To express the required synchronization policy using forks, we can define it as below:

```
class Phil {
  take_forks (left : Fork, right : Fork) = {
    sync
      left.ANY ⊕ right.ANY isol self
    in
      left.take(this); right.take(this);
      ...  /* eating */
      left.putdown(); right.putdown()
  }
}
```

The above means that for each object of class `Fork`, a compound of its two methods `take` and `putdown` in the body of method `take_forks` will be executed atomically. This is enforced by the synchronization policy declared locally for objects `left` and `right`.

5.3 Composition and Deadlock

Composition of code containing traditional synchronization constructs is a source of potential problems. Consider the following program, expressed using conditional critical regions (alternatively, we could use locks):

```
atomic {
  x=1;
}

atomic(y==1) {
  ...
}

atomic(x==1) {
    y=1;
}
```

Suppose that these atomic blocks are part of some three methods correspondingly, `m1`, `m2`, and `m3`, which can be executed by concurrent threads, and initially `x=y=0`. Now, if methods `m1` and `m2` are composed into a compound required to be executed atomically, we obtain the following code:

```
atomic {
  atomic {
    x=1;
  }
  atomic(y==1) {
    ...
  }
}

atomic(x==1) {
  y=1;
}
```

Unfortunately, the code obtained in this way prevents some interleaving of threads executing atomic blocks. The last atomic block requires `x==1` before it can assign `y` to `1`. On the other hand, the atomic block that assigns `x` to `1` is part of the outermost atomic block that gets stuck waiting for the condition `y==1` to be true. Since the outermost block does not commit, the assignment of `x` to `1` cannot be visible, which leads to deadlock. To get around this, the programmer might want to locally relax atomicity. This must be done with care to avoid breaking the invariants on which the methods rely. Unfortunately, traditional synchronization mechanisms lack language support for it.

Below we use our language to declare the above synchronization policy for methods `m1`, `m2`, and `m3`, and then we relax the original policy to prevent deadlock. For simplicity, all methods are part of the same class `P`:

```
class P {
  x = 0
  y = 0
  m1() = {sync P.m1 isol [ANY] in x=1}
  m2() = {sync (y==1) P.m2 isol [ANY] in ...}
  m3() = {sync (x==1) P.m3 isol [ANY] in y=1 }
}

let A ← P.m1 ⊕ P.m2 in
sync A isol [ANY] in
sync A !isol [P.m3] in
  e
```

We used conditional concurrency combinators to declare the atomic execution of `m1`, `m2`, and `m3`, guarded by conditions on `x` and `y`. Next, we extended this policy and defined a compound `A` of methods `m1` and `m2` to be executed atomically (this corresponds to the outermost `atomic`). To avoid deadlock, we then revoked the atomicity of compound `A` with respect to `m3`. The final policy is valid in expression `e`.

`A` remains atomic with respect to any other code. Thus, all invariants that require the atomicity of `A` and do not depend on values of `x` and `y` will be preserved. The revoke combinator only weakens the atomicity of `A`, the atomicity defined for `m1`, `m2`, and `m3` will *not* be cancelled.

5.4 Nested Atomic Blocks

Consider transaction nesting like below:

```
atomic {
  ...
  atomic { ... }
  ...
}
```

The most reasonable semantics is closed-nesting: The effects of a nested transaction are not externally visible until the outermost transaction completes. However, this approach is not modular. We explain this using an example of synchronization barrier that appeared in [24, 25].

Below we define a method `barrier`. A thread calling method `barrier` is blocked until `NUMTHREADS` concurrent threads will call this method (the barrier is then reached):

```
void barrier() {
  atomic { count++; }
  atomic(count == NUMTHREADS) {
    /* Barrier reached */
  }
}
```

The problem begins when the barrier routine happens to be used inside a different critical section, possibly protecting completely distinct data:

```
atomic {
  ... barrier(); ...
}
```

In this case, no thread will ever exit the barrier, since its effects are prevented from being seen by other threads. The intended behavior is that the execution of the outer atomic section will *not* be atomic, but will instead be interrupted, allowing other threads to observe its results (i.e. the result of `count++`), and itself observing the results of other threads at the point of evaluation of the inner atomic sections guard (i.e. `count==NUMTHREADS`). But this behaviour is contradictory to the closed-nesting semantics. It is hard to see what would be a reasonable semantics for the atomic sections in this case.

On the other hand, concurrency combinators allow the programmer to locally revoke the atomicity policy (with care to avoid invalidation of important program invariants) which solves this problem. Below we invoke method `barrier` inside an atomic method `m`:

```
class P {
  m() = {
    sync P.m isol [ANY] in
    sync P.m !isol [P.barrier] in
      ...
      barrier()
      ...
  }
}
```

Since method `barrier` is also atomic, we locally revoked the atomicity policy declared for method `m` for the execution of `barrier` (for other code method `m` appears atomic). This, however, does not affect any synchronization defined for method `barrier`.

5.5 Barrier Synchronization

Barrier synchronization can be encoded using the conditional atomicity combinator (the method `barrier` in the last section gives the idea). However, since barrier synchronization is a common idiom, our language has a corresponding combinator. The combinator can be used to declare a synchronization barrier put on any variable or object field in any expression modifying this variable/field.

For example, we can declare a synchronization barrier in an expression `e`, as follows:

```
sync o.barrier ⋈ self in
  e
```

where `barrier` is a field of an object `o` of some class `P` (initialized to 0). Now, any write to `barrier` on this object by a thread executing `e` will be suspended until all concurrent threads forked by `e` (if there are any) either reach this barrier or complete without writing to `barrier`.

We can use a conditional to bound the number of threads in a barrier, as follows. Below is the code executed by concurrent threads in order synchronize on a barrier:

```
if (o.barrier =< NUMTHREADS) then
  o.barrier := o.barrier + 1; e1
else
  sync o.barrier ! ⋈ self in
  o.barrier := 0; e2
```

If the number of writes to the instance field `barrier` reaches `NUMTHREADS`, the barrier is revoked in expression `e2`.

By extending the calculus with access modifier `static`, and replacing `counter=0` by `static counter=0`, the barrier could be declared and revoked for all objects of class `P` using a combinator argument `P.barrier`.

6. RELATED WORK

There is a vast amount of work on various synchronization constructs for concurrent languages but relatively little work was done on declarative synchronization for shared-memory multithreaded systems. The work on separation of concerns (see [10, 14, 5, 22, 16, 17] among others), in particular on separation of concurrency aspects, is the most relevant here. Below we discuss example work in this area.

For a long time, the object-oriented community has been pointing out, that the concurrency control code interwoven with the code of classes can represent a serious obstacle to class inheritance (see [15]). Milicia and Sassone [16, 17] addressed the inheritance anomaly problem in Java, and proposed an extension of this language with a linear temporal logic for expressing synchronization constraints on object method calls. The language support of declarative synchronization proposed in this paper shares common features with their approach but allows the programmer to express arbitrarily complex synchronization policies.

Frølund and Agha [5] proposed a language for specifying multi-object coordination in the actor model, expressed in the form of constraints. However, the constraints can only be declared for invocations of a group of objects. Ramirez *et al.* [20, 21] proposed a constraint logic language for expressing temporal constraints between marked points in concurrent programs, and demonstrated their approach using Java extended with the syntax for marking. However, these languages have limited expressiveness. Conditional concurrency combinators can refer to any variables and code, and allow composite synchronization policies to be expressed.

In *aspect-oriented programming (AOP)*, the programmers can specify the various *concerns* (or *aspects*) of a program and some description of their relationship, and use the AOP tools to *weave* [12] or compose them together into a coherent program. The aspect code is executed before and after the execution of pointcuts, where a *pointcut* usually corresponds to a method invocation. Hürsch and Lopes [10] identified various other concerns, including synchronization. Lopes [14] developed a programming language D, which allowed thread synchronization to be expressed as a separate concern. The AOP tools have been developed for popular programming languages (e.g., AspectJ [13]). However, the AOP approach does not yet provide the expressiveness of a declarative synchronization language.

In [26], we described *rôle-based synchronization (RBS)* for the OCaml language. RBS assumes assigning semantic rôles (specific for a given synchronization problem) to concurrent threads, and using abstract types for expressing constraints between rôles. The RBS differs from the approach described in this paper. Firstly, the synchronization behaviour is attached to rôles (or threads), not to code fragments, which limits the range of synchronization policies to those implemented by types specifying the rôles. Secondly, the synchronization policy can be switched on-the-fly (with dynamic control on the moment of switching).

Flanagan and Qadeer's [4] developed a type system for specifying and verifying the atomicity of methods in multithreaded Java programs. The type system is a synthesis of Lipton's theory of left and right movers (for proving properties of parallel programs) and type systems for race detection. In recent years, many transactional memory systems have also been developed for shared memory multiprocessors (see e.g., [6, 7, 8, 19] among others). They provide language constructs for expressing atomic transactions as an alternative to locks. Contrary to our `isol` combinator, programmers can also rollback transactions.

Recently, type-based synchronization has also been proposed for other models of concurrency: Bocchino Jr. *et al.* [3] describe an object-oriented type and effect system for expressing deterministic parallelism in imperative, object-oriented programs. They have similar goals to ours (simplifying concurrent programming) but the models are very different. While our approach assumes a typical model of

shared memory concurrent systems, their proposal is aimed at deterministic-by-default systems where non-deterministic behaviour must be explicitly declared.

Harris *et al.* [7] proposed *composable memory transactions* in Haskell. The `retry` construct can be used inside an atomic transaction to rollback and reexecute the transaction if some required condition does not hold; the transaction's rollback is blocked until another concurrent thread writes to a shared variable. The `orElse` construct allows alternative code to be executed on rollback. The code chunks containing `retry` or `orElse` can be composed using `atomic`. The type system based on monads forbids the use of I/O actions inside `atomic`. `retry` and `orElse` allow transactions to make progress, e.g., an atomic transaction can throw an exception; the transaction is then aborted with no effect. The concurrency model based on 'retry' is elegant but it forbids the use of irrevocable I/O actions, so it may limit the range of potential applications. In our calculus, we propose the CCR-style conditional concurrency combinator, and the policy revocation mechanism. The programmer can use the latter to revoke the declared synchronization policy if some condition is not satisfied. The policy revocation can be valid for selected code only. Our guess is that this approach is promising despite the additional complexity and care needed to ensure safety. However, we need more research on this.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have focused on foundations, and presented a calculus that allows us to study the problems of declarative synchronization. The key idea of our design was to provide a set of concurrency combinators that could be composed into desired synchronization policies. Our calculus also allows to locally revoke the declared policy. In the paper, we showed that the local policy revocation allows the problems of deadlock and nesting of synchronized code to be solved in an elegant way.

We think that our calculus can be a useful basis for work on different problems of parallel programming for multicore systems. For example, it could be an intermediate language between some low-level implementation mechanisms and a language used to express high-level abstractions. As future work, we will define the operational semantics formally. We also plan to design an abstract machine for the implementation of concurrency combinators.

Acknowledgments We would like to thank anonymous reviewers for useful comments that helped us to improve the paper. This work has been partially supported by the Polish Ministry of Science and Higher Education within the European Regional Development Fund, Grant No. POIG.01.03. 01-00-008/08.

8. REFERENCES

[1] A. Birrel. An introduction to programming with C# threads. Technical Report TR-2005-68, Microsoft Research, May 2005.

[2] C. Blundell, E. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2):17–21, Nov 2006.

[3] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proc. of OOPSLA '09: the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2009.

[4] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. of PLDI '03: the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[5] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proc. of ECOOP '93: the 7th European Conference on Object-Oriented Programming*, volume 627 of *LNCS*, July 1993.

[6] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. of OOPSLA '03: the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[7] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. of PPoPP '05: the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[8] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proc. of OOPSLA '06: the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2006.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of ISCA '93: the 20th International Symposium on Computer Architecture*, May 1993.

[10] W. Hürsch and C. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Feb. 1995.

[11] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. of OOPSLA '99: the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nov. 1999.

[12] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proc. of ECOOP '03: the 17th European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*. Springer, July 2003.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[14] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Dec. 1997 (1998).

[15] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[16] G. Milicia and V. Sassone. Jeeg: A programming language for concurrent objects synchronization. In *Proc. of ACM Java Grande/ISCOPE Conference*, Nov. 2002.

[17] G. Milicia and V. Sassone. Jeeg: Temporal constraints

for the synchronization of concurrent objects. Tech. Report RS-03-6, BRICS, Feb. 2003.

[18] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proc. of PPoPP '07: the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2007.

[19] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proc. of OOPSLA '08: the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Sept. 2008.

[20] R. Ramirez and A. E. Santosa. Declarative concurrency in Java. In *Proc. of HIPS 2000: the 5th Workshop on High-Level Parallel Programming Models and Supportive Environments*, May 2000.

[21] R. Ramirez, A. E. Santosa, and R. H. C. Yap. Concurrent programming made easy. In *Proc. of ICECCS 2000: the 6th IEEE International Conference on Engineering of Complex Computer Systems*, Sept. 2000.

[22] S. Ren and G. A. Agha. RTsynchronizer: Language support for real-time specifications in distributed systems. In *Proc. of the 2nd ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, June 1995.

[23] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of PODCS '95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1995.

[24] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *Proc. of OOPSLA '07: the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2007.

[25] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. General and efficient locking without blocking. In *Proc. of MSPC '08: ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, Mar. 2008.

[26] V. Tanasescu and P. T. Wojciechowski. Role-based declarative synchronization for reconfigurable systems. In *Proc. of PADL '05: the 7th Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *LNCS*, pages 52–66. Springer, Jan. 2005.

[27] P. T. Wojciechowski. Concurrency combinators for declarative synchronization. In *Proc. of APLAS '04: the 2nd Asian Symposium on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 163–178. Springer, Nov. 2004.