Advanced Thread Synchronization in Java

Christian Heinlein

Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

Abstract. Thread synchronization in Java using `synchronized` methods or statements is simple and straightforward as long as mutual exclusion of threads is sufficient for an application. Things become less straightforward when `wait()` and `notify()` have to be employed to realize more flexible synchronization schemes. Using two well-known examples, the bounded buffer and the readers and writers problem, the snares and traps of hand-coded synchronization code and its entanglement with the actual application code are illustrated. Following that, interaction expressions are introduced as a completely different approach where synchronization problems are solved in a declarative way by simply specifying permissible execution sequences of methods. Their integration into the Java programming language using a simple precompiler and the basic ideas to enforce the synchronization constraints specified that way are described.

1. Introduction

Thread synchronization in Java using `synchronized` methods or statements is simple and straightforward as long as mutual exclusion of threads is sufficient for an application. Things become less straightforward when `wait()` and `notify()` (or `notifyAll()`) have to be employed to realize more flexible synchronization schemes. In that context, things become even more complicated by the fact that *monitors* and *condition variables* − which have been related, but *separate* entities in the original monitor concept [8] − have been merged into a single unit, namely an object [12, 13]. In practice that means that every Java object used as a monitor possesses only a single, implicit condition variable which is the object itself. Therefore, monitor-based solutions of synchronization problems using two or more different condition variables − for example, the rather straightforward solution of the well-known bounded buffer problem shown in Fig. 1 − are difficult to convert to functionally equivalent and comparably comprehensive Java programs. Guaranteeing (and proving) the correctness of such code is further complicated by the fact that `notify()` (or `notifyAll()`) does neither suspend the executing thread (either immediately or at the time it leaves the monitor, i. e., the `synchronized` method or statement it is currently executing) nor immediately resumes the notified thread. (In other words, `notify()` does not put the notified thread in the *running*, but only in the *runnable* state.) That means that the notifying thread (or even any other currently running thread) might execute the monitor code several times again before the notified thread is actually able to continue. This behaviour, which is in contrast to the `signal` semantics of the original monitor concept, usually improves performance as it demands fewer thread switches, but on the other hand increases the typically already high enough possibility of undesired race conditions even further.

Using the bounded buffer problem already mentioned above, Sec. 2 illustrates these problems for a typical and not too complicated real-world example. Following that,

```
monitor Buffer;
  const
    N = ...;     (* buffer size *)
  var
    buf: array [0 .. N-1] of integer; (* buffer *)
    n: integer := 0; (* number of currently occupied slots *)
    p: integer := 0; (* index of next slot for put *)
    g: integer := 0; (* index of next slot for get *)
    notempty, notfull: condition; (* buffer conditions *)

  procedure put(x: integer);
  begin
    if n = N then wait(notfull);
    buf[p] := x;
    p := (p + 1) mod N;
    n := n + 1;
    if n = 1 then signal(notempty);
  end;

  procedure get(var x: integer);
  begin
    if n = 0 then wait(notempty);
    x := buf[g];
    g := (g + 1) mod N;
    n := n - 1;
    if n = N - 1 then signal(notfull);
  end;
end.
```

Figure 1: Monitor-based solution of the bounded buffer problem in a Pascal-like language

Sec. 3 introduces the basic idea of *interaction expressions* and demonstrates their use to derive a concise and clear solution of the bounded buffer problem in a few minutes. In Sec. 4, solutions based on interaction expressions of another well-known synchronization problem, namely the readers and writers problem, are presented and compared with handwritten synchronization code.

Following these deliberately extensive motivating sections, Sec. 5 actually describes the extensions to the Java programming language which are necessary to use interaction expressions in Java programs and their basic implementation in a precompiler. Section 6 briefly describes the implementation of an accompanying library class which actually performs the requested synchronization at run time. Finally, Sec. 7 concludes the paper with a summary and outlook.

2. The Bounded Buffer Problem

Figure 2 shows a transformation of the monitor-based solution of the bounded buffer problem into Java code, where the two explicit condition variables `notempty` and `notfull` of Fig. 1 have been merged into the single implicit condition variable asso-

```
class Buffer {
  private final int N = ...;
  private int [] buf = new int[N];
  private int n = 0, p = 0, g = 0;

  public synchronized void put(int x) {
    if (n == N) wait();
    buf[p] = x;
    p = (p + 1) % N;
    n++;
    if (n == 1) notify();
  }

  public synchronized int get() {
    if (n == 0) wait();
    int x = buf[g];
    g = (g + 1) % N;
    n--;
    if (n == N - 1) notify();
    return x;
  }
}
```

Figure 2: Erroneous solution of the bounded buffer problem in Java

ciated with the current object `this`.[1] This code would behave correctly if a notified thread would be guaranteed to continue execution immediately after the notifying thread has left the monitor. But since such behaviour is not guaranteed in Java, the following erroneous execution sequence might happen in principle and is actually observed in practice:

- Two consumer threads, C_1 and C_2, call `get()` on an empty buffer and thus will be blocked by `wait()`.
- Then, a producer thread P executes `put()` causing one of the consumer threads, C_1 or C_2, to become awakened by `notify()`.
- However, if the currently running thread P has not yet exceeded its time slice, it is usually allowed to continue running before the awakened thread is actually resumed. Therefore, it might happen that P executes another `N` calls of `put()` before it becomes blocked by `wait()` because the buffer has run full.
- Now, the previously awakened consumer thread is actually resumed and completes its execution of `get()`. Because the buffer has become full in the mean time, `notify()` will be executed at the end of `get()` which is assumed to awaken a producer thread waiting for the buffer to become `notfull` (cf. Fig. 1).

[1] To simplify the examples, the fact that `wait()` might throw an `InterruptedException`, which must be either caught or declared in the headers of `put()` and `get()`, is ignored.

- However, it might actually happen that `notify()` awakens the second consumer thread. (Because this thread has been blocked for a longer time than the producer thread *P*, this is indeed quite probable.)
- Before this thread is actually resumed, however, the currently running consumer thread might execute further calls of `get()` until it becomes blocked by `wait()` due to an empty buffer.
- Now, the second consumer thread is actually resumed and completes its execution of `get()`, despite the fact that the buffer is currently empty!

To avoid such subtle errors, a "brute force" solution resembling the idea of *conditional critical regions* [2] might be employed (cf. Fig. 3). This is certainly correct, but rather inefficient as `notifyAll()` always awakens *all* waiting threads while only *one* of them can actually proceed afterwards. In particular, awakening threads executing the same method (`put()` or `get()`) as the notifying thread is completely unnecessary.

```
class Buffer {
  private final int N = ...;
  private int [] buf = new int[N];
  private int n = 0, p = 0, g = 0;

  public synchronized void put(int x) {
    while (n == N) wait();
    buf[p] = x;
    p = (p + 1) % N;
    n++;
    notifyAll();
  }

  public synchronized int get() {
    while (n == 0) wait();
    int x = buf[g];
    g = (g + 1) % N;
    n--;
    notifyAll();
    return x;
  }
}
```

Figure 3: Correct, but inefficient solution of the bounded buffer problem

To complete the example, Fig. 4 shows a correct and maximally efficient solution of the problem, which even improves the original monitor solution by allowing one producer and one consumer thread to operate on the buffer concurrently. However, neither finding nor verifying this solution is a simple task, and the chance for overlooking another subtle detail is extremely high.

```
class Buffer {
  private final int N = ...;
  private int [] buf = new int[N];
  private int p = 0, g = 0;
  private int empty = N, full = 0;
  private Object plock = new Object(), glock = new Object();

  public void put(int x) {
    synchronized (plock) {
      while (empty == 0) plock.wait();
      buf[p] = x;
      p = (p + 1) % N;
      empty--;
    }
    synchronized (glock) {
      full++;
      glock.notify();
    }
  }

  public int get() {
    int x;
    synchronized (glock) {
      while (full == 0) glock.wait();
      x = buf[g];
      g = (g + 1) % N;
      full--;
    }
    synchronized (plock) {
      empty++;
      plock.notify();
    }
    return x;
  }
}
```

Figure 4: Correct and maximally efficient solution of the bounded buffer problem

3. Interaction Expressions

Apart from these correctness and efficiency considerations, all solutions presented so far suffer from an unfavourable intermixing of the actual application code with bothering synchronization details. Even in the original monitor solution, more than half of the code of the methods `put()` and `get()` deals with synchronization. If correct execution sequences of these methods could be enforced otherwise, they might actually be reduced to the following few lines:

```
void put(int x) {
  buf[p] = x;
  p = (p + 1) % N;
}

int get() {
  int x = buf[g];
  g = (g + 1) % N;
  return x;
}
```

Of course, one might argue rightly that developing sophisticated synchronization schemes for such short and simple methods is a waste of time − both for the programmer who has to write the code and for the run-time system which executes it, because the latter might be able to schedule threads more efficiently in practice if the methods were simply declared `synchronized`, even if this is too restrictive from a theoretical point of view. If reading and writing buffer elements is more costly, however, e. g., if extensive file operations are involved, unnecessary mutual exclusion actually becomes a considerable performance brake.

Separating different aspects of an application and implementing them independently, is advocated as a general principle by the current trend of *aspect-oriented programming* [10, 11]. The particular idea to separate low-level synchronization details from the more abstract parts of an application, is quite old, however. In the early 1970s, *path expressions* have been proposed as a simple formalism to describe *permissible execution sequences* of Pascal procedures and by that means to indirectly specify *synchronization conditions* for them [3]. Similarly, *synchronization expressions* have been developed in the mid 1990s to synchronize statements of concurrent C programs [5]. Due to several severe restrictions of these and other related formalisms, yet another similar approach called *interaction expressions* has been developed recently [6, 7].

The basic idea is always the same, however: A formalism based on extended regular expressions is used to describe synchronization conditions for statements, procedures, or methods separately from the actual application code in a very flexible and comfortable way. The *language* of such an expression (or set of expressions), i. e., the set of *words* it accepts [9], is interpreted as the set of *permissible execution sequences* of these code units. To actually enforce the constraints specified by such expressions, they are transformed into some suitable state model by the compiler or an appropriate precompiler. Furthermore, all methods (or other code units) occurring in one or more expressions are bracketed by a prologue and an epilogue which perform corresponding state transitions if they are currently permitted or otherwise wait until they become permitted.

To give a concrete example, consider the interaction expression

```
expr * ( |[int x] put(x) - get() );
```

where the operators `-` and `*` denote sequence and repetition, respectively, and `|[int x] put(x)` is roughly equivalent to `put()`.[2] Furthermore, interaction expressions are introduced by the keyword `expr` and terminated by a semicolon. Because

[2] Because methods might be overloaded, the subexpression `|[int x] put(x)`, specifying that `put(x)` might be executed with *any* `int` value `x`, must be used instead of simply `put()` to correctly refer to the method with the signature `put(int x)`.

this expression is almost equivalent to the regular expression (*put get*)* possessing the language

$$\{\langle\rangle, \langle put, get\rangle, \langle put, get, put, get\rangle, \ldots\},$$

it specifies that the methods `put()` and `get()` have to be executed in alternating sequence starting with `put()`. This is the required behaviour, if the buffer would possess just a single slot, i. e., for the case `N = 1`.

For `N = 2`, two such sequences might be executed *concurrently and independently* (which can be expressed by the parallel composition operator `+`), if groups of `put()`s and groups of `get()`s are both executed sequentially (which can be expressed by repetition operators `*`). This leads to the following set of interaction expressions:

```
// Allow two concurrent alternating sequences
// of put() and get().
expr * ( |[int x] put(x) - get() )
   + * ( |[int x] put(x) - get() );

// Enforce sequential execution
// of multiple put()s and multiple get()s.
expr * |[int x] put(x);
expr * get();
```

To specify for any `N` that `N` alternating sequences of `put()` and `get()` might be executed concurrently, a *multiplier expression* `+{N}` *`body`* can be used to abbreviate the expression *`body`* `+ ... +` *`body`* for any subexpression *`body`*:

```
// Allow N concurrent alternating sequences
// of put() and get().
expr +{N} * ( |[int x] put(x) - get() );
```

In practice, this expression avoids buffer underflows (because each execution of `get()` requires a preceding execution of `put()`) as well as buffer overflows (because each execution of `put()` in one of the `N` concurrent alternating sequences requires a successive execution of `get()` before another execution of `put()` is permitted in the same sequence).

Compared with the explicit synchronization shown in Fig. 4, whose development is time-consuming, error-prone, and hard to understand and verify, developing the equivalent interaction expressions shown above is a straightforward task taking a few minutes after a little training of the formalism (which can be done instead of strenuously studying textbooks on the synchronization of Java threads to avoid the pitfalls outlined in Sec. 2).

Remark: Because this paper does not constitute a tutorial on interaction expressions, but rather a description of their integration and typical use in Java, the formalism itself is not explained in more detail. Nevertheless, the examples given in the present and subsequent section are intended to give the reader a taste of the formalism's expressiveness and applicability.

4. The Readers and Writers Problem

The well-known readers and writers problem − a data object might be accessed by several readers simultaneously, while a writer needs exclusive access [4] − is another example of a synchronization problem where simple mutual exclusion is unsatisfactory when the read and write operations take non-neglectable time. Figure 5 shows a possible solution in Java, which is fairly compact and comprehensive.[3] On the other hand, Fig. 6 shows an equivalent solution with interaction expressions[4] which is even more compact and comprehensive as well as less error-prone and much easier to adapt to additional requirements. To specify, for instance, that the first operation must be `write()` (to guarantee proper initialization of the data object), this is simply achieved by replacing the interaction expression with the following:

```
expr write() - * ( # read() | write() );
```

If additional operations `create()`, `open()`, `close()`, and `destroy()` are introduced, the expressions

```
expr rw() = write() - * ( # read() | write() );
expr oc() = * ( open() - rw() - close() );
expr * ( create() - oc() - destroy() );
```

(where `rw()` and `oc()` are *interaction macros* whose calls are replaced by their right hand side in subsequent interaction expressions or macros) can be used to specify the permitted execution sequences in a simple and natural way without disturbing any of the method bodies, while extending the solution of Fig. 5 in an equivalent manner would require the introduction of several auxiliary "state variables" and substantial extensions to the methods involved.

```
class ReadWrite {
  int n = 0;    // Number of currently executing readers.
  ......        // Other data fields.

  public void read() {
    synchronized (this) { n++; }
    ......      // Actual read operation.
    synchronized (this) { if (--n == 0) notifyAll(); }
  }

  public synchronized void write() {
    while (n > 0) wait();
    ......      // Actual write operation.
  }
}
```

Figure 5: Solution of the readers and writers problem in Java

[3] To simplify the example, the parameters of `read()` and `write()` are omitted.
[4] The operator # permits any number of `read()`s to be executed concurrently, while the typical pattern `*(...|...)` (where | denotes choice) specifies that readers and writers are mutually exclusive.
They keyword `sync` indicates that the methods `read()` and `write()` are subject to synchronization by interaction expressions (cf. Sec. 5).

```
class ReadWrite {
  ......        // Other data fields.

  // Interaction expression to synchronize read() and write().
  expr * ( # read() | write() );

  public sync void read() {
    ......      // Actual read operation.
  }

  public sync void write() {
    ......      // Actual write operation.
  }
}
```

Figure 6: Solution of the readers and writers problem with interaction expressions

5. Java Language Extensions

To actually allow a programmer to solve synchronization problems occurring in a Java program by means of interaction expressions, two extensions to the Java programming language are necessary:

1. Methods might be declared `sync` to indicate that they are subject to synchronization by interaction expressions.
   Only methods declared that way are allowed to appear in interaction expressions, and a `sync` method of a superclass (or an interface) must not be overridden by a non-`sync` method in a subclass (or an implementing class).
2. Classes (and, in a limited way, interfaces, too) might contain interaction expressions (and macros) introduced by the keyword `expr`.
   In addition to being declared `sync`, the methods appearing in an interaction expression must be accessible according to the usual rules of the language. That means, for example, that `public sync` methods of a class might appear in interaction expressions of any class, while `private sync` methods might only appear in interaction expressions of their own class.

Figure 7 shows an EBNF grammar for interaction expressions and their integration with the standard Java grammar. Here, **`boldface`** indicates terminal symbols, i. e., keywords and literal characters like, e. g., **`static`** and `#`, while *`italics`* denote non-terminal symbols. More specifically, upper-case names like *`Expression`* or *`Type`* refer to non-terminals of the Java grammar, while the lower-case name *`expr`* refers to a non-terminal introduced here.

To simplify the presentation, it is assumed that unary operators bind more tightly than multipliers and quantifiers, which in turn bind more tightly than binary operators. The latter are presented in the grammar in decreasing order of precedence. To actually enforce these precedence rules in a parser generator like JavaCC [18], the grammar has to be rewritten to contain a separate production for every level of operator precedence.

```
ClassBodyDeclaration
  : ......
  | [ static ] expr expr ;       // Interaction expression.
  | { public|protected|private|static|abstract|final }
    expr MethodDeclarator [ = expr ] ; // Interaction macro.

expr
  // Atomic expression.
  : MethodInvocation

  // Unary operators.
  | ? expr | expr ?   // Option.
  | * expr | expr *   // Sequential iteration (repetition).
  | # expr | expr #   // Parallel iteration.

  // Binary operators.
  | expr - expr       // Sequential composition (sequence).
  | expr + expr       // Parallel composition.
  | expr | expr       // Disjunction (choice).
  | expr & expr       // Conjunction.
  | expr @ expr       // Synchronization (weak conjunction).

  // Multipliers.
  | ( - | + ) { Expression } expr

  // Quantifiers.
  | ( + | | | & | @ ) [ Type Identifier { [ ] } ] expr

  // Bracketed expressions.
  | ( expr ) | [ expr ] | { expr }
```

Figure 7: Grammar of interaction expressions

Using a rather simple precompiler, programs written in the extended language (called JavaX) are transformed to pure Java code along the following lines:

- The body of a `sync` method is bracketed by calls to the methods `sync.prolog()` and `sync.epilog()` which are statically defined in a library class `sync`.[5]
  In principle, both of these methods receive the name of the `sync` method as a `String` and an array of `Object` instances containing its actual parameters (including the implicit `this` parameter, unless the method is `static`). Actually, this information is combined into a single object of type `sync.Expr` by the library method `sync.activity()`. For technical reasons explained below, `sync.activity()` is actually called by a so-called *shadow method* of the original `sync` method which is needed for several additional purposes, too. The shadow method receives the same parameters as the original method plus an additional dummy parameter of type `sync.Dummy` to obtain a different method signature.

[5] Note that `sync` constitutes a keyword in the extended language JavaX and thus cannot be used as an identifier there. Therefore, name collisions with the name of the library class cannot arise.

For example, Fig. 8 shows the code generated for the method:

```
public sync void read() {
  // body of read
}
```

While `sync.prolog()` has to check whether the method in question is currently permitted by all interaction expressions and, if it is not, wait until it becomes permitted, `sync.epilog()` simply registers the fact that the method execution has finished.

- An interaction expression introduced by the keyword `expr` is transformed to an initializer block which constructs an operator tree representation of the expression at run time which is passed to the library method `sync.enable()`.
  For example, Fig. 9 shows the code generated for the expression:

  ```
  expr * ( # read() | write() );
  ```

  Here, the shadow methods of the `sync` methods `read()` and `write()` are used to

```
public sync.Expr read(sync.Dummy x) {
  return sync.activity("read", new Object [] { this });
}

public void read() {
  sync.Expr expr = read(sync.dummy);
  sync.prolog(expr);
  try {
    // body of read
  }
  finally { sync.epilog(expr); }
}
```

Figure 8: Transformation of a `sync` method

```
{
  sync.enable(
    sync.unary('*',
      sync.binary('|',
        sync.unary('#',
            read(sync.dummy)
        ),
        write(sync.dummy)
      )
    )
  );
}
```

Figure 9: Transformation of an interaction expression

conveniently obtain `sync.Expr` objects representing these methods in an operator tree. Furthermore, by generating code whose correctness depends on the existence of these shadow methods, the precompiler elegantly delegates to the Java compiler the task of checking that only `sync` methods are used in interaction expressions: If a non-`sync` method is used, no corresponding shadow method will be found causing the Java compiler to report an error.[6]

Another reason for employing shadow methods here is the fact that it is hard or even impossible for the precompiler to distinguish the call of an *instance* method `b()` for an object `a` (which receives `a` as an implicit parameter) from the call of a *static* method `b()` of a class `a` (which does not receive an implicit parameter), as both are written `a.b()`.[7] By simply replacing the original call `a.b()` with the call `a.b(sync.dummy)` of the shadow method, this task is again delegated to the Java compiler.

- An interaction macro definition is transformed to a *shadow* method definition which constructs and returns an operator tree representation of the expression on the right hand side of the definition.

  For example, Fig. 10 shows the code generated for the macro definition:

  ```
  expr oc() = * ( open() – rw() – close() );
  ```

  Once again, transforming method invocations in interaction expressions to calls of the corresponding shadow methods significantly simplifies the precompiler's job as it need not distinguish "real" method invocations from interaction macro calls. In the example above, `open()`, `rw()`, and `close()` are all transformed in the same way to corresponding shadow method invocations without needing to know that `open()` and `close()` are normal `sync` methods while `rw()` is another interaction macro.

- All other Java code is left unchanged.

```
sync.Expr oc(sync.Dummy x) { return
  sync.unary('*',
    sync.binary('–',
      open(sync.dummy),
      sync.binary('–',
        rw(sync.dummy),
        close(sync.dummy)
      )
    )
  );
}
```

Figure 10: Transformation of an interaction macro definition

[6] The fact that such an error message will not be completely self-evident to a programmer at first glance, is a typical shortcoming of a precompiler-based approach which is acceptable though.

[7] Because the precompiler is designed to transform a single JavaX source file without consulting any other JavaX, Java, or class file, it is indeed impossible to distinguish these cases in general, since `a` might be a field of a superclass defined in another source file or a class imported by a "type-import-on-demand declaration," respectively.

6. Implementation of the Accompanying Library Class

The code generated by the precompiler relies on several types and static methods defined in the library class `sync`. Roughly, these methods can be categorized as follows:

- Public methods for constructing operator trees to represent interaction expressions at run time, e. g., `activity()`, `unary()`, and `binary()`.
- Public methods providing the essential operations of the library:
  - `enable()` to activate an interaction expression, i. e., to add it to an internal set of expressions;
  - `prolog()` to check whether (resp. to wait until) a `sync` method is permitted by all activated interaction expressions and to register that the method has started execution;
  - `epilog()` to register that a `sync` method has finished execution.
- Private methods implementing an *operational model* of interaction expressions consisting of (hierarchically structured) *states*, *state transitions*, and *state predicates*. These methods, which constitute the core of the library, are based on a precise operational semantics of interaction expressions which is in turn equivalent to the formal semantics of the formalism. Detailed complexity analyses have shown that the operational model, which has been specifically optimized for performance, behaves sufficiently well in practice even for complicated expressions [6, 7].

Enabling an expression via `enable()` actually means to compute and store its *initial state*.

Calling `prolog()` at the beginning of a `sync` method results in performing *state transitions* for all activated expressions containing the method in question.[8] If all resulting states are *valid*, `prolog()` returns immediately, allowing the body of the `sync` method to be executed. Otherwise, if one of the resulting states is *invalid*, the state transitions are undone by restoring the previous states of the expressions, and `prolog()` suspends the current thread until another thread has executed `epilog()`. Afterwards, the state transitions are repeated in the new state and, depending on their validness, `prolog()` returns or waits again, and so on.

Calling `epilog()` at the end of a `sync` method results in performing similar state transitions for all activated expressions containing the method, too. In contrast to `prolog()`, these transitions will always yield valid states because terminating a method is always permitted by interaction expressions. Afterwards, all threads which have been suspended during an execution of `prolog()` are resumed causing their state transitions to be repeated, as described above.

To avoid race conditions in the library itself, all state transitions are performed inside appropriate `synchronized` statements.

[8] To efficiently determine those expressions which contain the method in question, the internal set of expressions mentioned above is actually split into a large number of very small sets which are associated with individual classes containing static `sync` methods and objects of classes containing instance `sync` methods. By that means, only one or two of these small sets have to be actually processed at a time.

7. Summary and Outlook

Using two well-known examples − the bounded buffer and the readers and writers problem −, it has been argued that hand-coded synchronization in Java is cumbersome and error-prone if mutual exclusion is too simplistic. Furthermore, synchronization and actual application code are usually entangled in an unfavourable way which complicates later modifications or extensions. In contrast, interaction expressions constitute a powerful and easy-to-use tool to specify synchronization requirements separate from the application code in a straightforward and natural way.

By appropriately extending the grammar of Java, it has been possible to incorporate the formalism into the language using a rather simple precompiler. The actual synchronization is performed by generic library methods implementing a formally verified operational model of interaction expressions based on states, state transitions, and state predicates. Extensive complexity analyses guarantee efficient run time behaviour in principle, even for complicated expressions.

On the other hand it is obvious and shall not be hidden that synchronization based on interaction expressions requires some computational overhead compared to simple `synchronized` methods or statements which will not be worthwhile for very short and fast methods. If the average execution time of the methods in question is large enough to think about more sophisticated synchronization schemes, however, interaction expressions can be successfully employed to obtain comprehensive and correct solutions in a few minutes. Furthermore, the advantages of clearly separating synchronization details from the actual application logic, cannot be overemphasized.

Of course, synchronization in general as well as synchronization of parallel programs in particular has been a research area for several decades, and even the less common idea to use expression-based formalisms for that purpose is not really new (cf. Sec. 3). However, all comparable formalisms suggested so far (e. g., path expressions [3], synchronization expressions [5], event expressions [15, 14], and flow expressions [16, 1]) suffer from either a very limited range of operators provided, a severe lack of orthogonality and generality, or the absence of a practically usable (i. e., sufficiently efficient) implementation [17]. Therefore, interaction expressions have been developed as a unification and extension of these formalisms which nevertheless possesses an efficient (and formally founded) implementation [6, 7]. Integrating such a formalism into Java using a simple precompiler which does not really need to “understand” Java in detail, but merely performs some “local” source code transformations, appears to be original.

While the core implementation of interaction expressions is very mature, its efficient integration into the Java run time environment is still in a prototypical stage. In particular, different strategies to minimize the duration of critical sections inside `sync.prolog()` and `sync.epilog()` calls (and to completely eliminate unnecessary ones) have to be explored. For example, it might be advantageous to employ optimistic concurrency control protocols instead of traditional locking schemes to synchronize transactions comprising multiple state transitions.

References

[1] T. Araki, N. Tokura: “Flow Languages Equal Recursively Enumerable Languages.” *Acta Informatica* 15, 1981, 209−217.

[2] P. Brinch Hansen: “A Comparison of Two Synchronizing Concepts.” *Acta Informatica* 1 (3) 1972, 190−199.

[3] R. H. Campbell, A. N. Habermann: “The Specification of Process Synchronization by Path Expressions.” In: E. Gelenbe, C. Kaiser (eds.): *Operating Systems* (International Symposium; Rocquencourt, France, April 1974; Proceedings). Lecture Notes in Computer Science 16, Springer-Verlag, Berlin, 1974, 89−102.

[4] P. J. Courtois, F. Heymans, D. L. Parnas: “Concurrent Control with “Readers” and “Writers”.” *Communications of the ACM* 14 (10) October 1971, 667−668.

[5] L. Guo, K. Salomaa, S. Yu: “On Synchronization Languages.” *Fundamenta Informaticae* 25 (3+4) March 1996, 423−436.

[6] C. Heinlein: *Workflow and Process Synchronization with Interaction Expressions and Graphs*. Ph. D. Thesis (in German), Universität Ulm, 2000.

[7] C. Heinlein: “Workflow and Process Synchronization with Interaction Expressions and Graphs.” In: *Proc. 17th Int. Conf. on Data Engineering (ICDE)* (Heidelberg, Germany, April 2001). IEEE Computer Society, 2001, 243−252.

[8] C. A. R. Hoare: “Monitors: An Operating System Structuring Concept.” *Communications of the ACM* 17 (10) October 1974, 549−557.

[9] J. E. Hopcroft, J. D. Ullman: *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin: “Aspect-Oriented Programming.” In: M. Aksit (ed.): *ECOOP’97 − Object-Oriented Programming* (11th European Conference; Jyväskylä, Finland, June 1997; Proceedings). Lecture Notes in Computer Science 1241, Springer-Verlag, Berlin, 1997.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: “An Overview of AspectJ.” In: J. Lindskov Knudsen (ed.): *ECOOP 2001 − Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001.

[12] D. Lea: *Concurrent Programming in Java. Design Principles and Patterns* (Second Edition). Addison-Wesley, Reading, MA, 2000.

[13] S. Oaks, H. Wong: *Java Threads*. O’Reilly, Sebastopol, CA, 1999.

[14] W. F. Ogden, W. E. Riddle, W. C. Rounds: “Complexity of Expressions Allowing Concurrency.” In: *Proc. 5th ACM Symp. on Principles of Programming Languages*. 1978, 185−194.

[15] W. E. Riddle: “An Approach to Software System Behavior Description.” *Computer Languages* 4, 1979, 29−47.

[16] A. C. Shaw: “Software Description with Flow Expressions.” *IEEE Transactions on Software Engineering* SE-4 (3) May 1978, 242−254.

[17] A. C. Shaw: “On the Specification of Graphics Command Languages and Their Processors.” In: R. A. Guedj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce (eds.): *Methodology of Interaction* (IFIP Workshop on Methodology of Interaction; Seillac, France, May 1979). North-Holland Publishing Company, Amsterdam, 1980, 377−392.

[18] A. Williams: “Java Parsing Made Easy.” *Web Techniques* 9/2001, September 2001, www.webtechniques.com/archives/2001/09/java.