TECHNICAL REPORT 95-11

Schematic: A Concurrent Object-Oriented Extension to Scheme

Kenjiro Taura and Akinori Yonezawa

Dec 1995

DEPARTMENT OF INFORMATION SCIENCE
FACULTY OF SCIENCE, UNIVERSITY OF TOKYO

7–3–1 HONGO, BUNKYO-KU TOKYO, 113 JAPAN

TECHNICAL REPORT 95-11

TITLE
Schematic: A Concurrent Object-Oriented Extension to Scheme

AUTHORS
Kenjiro Taura and Akinori Yonezawa

KEY WORDS AND PHRASES
concurrent object-oriented programming language, Scheme, process calculus, linearizable objects

ABSTRACT
A concurrent object-oriented extension to the programming language Scheme, called Schematic, is described. Schematic supports familiar constructs often used in typical parallel programs (`future` and higher-level macros such as `plet` and `pbegin`), which are actually defined atop a very small number of fundamental primitives. In this way, Schematic achieves both the convenience for typical concurrent programming and simplicity and flexibility of the language kernel. Schematic also supports concurrent objects which exhibit more natural and intuitive behavior than the "bare" (unprotected) shared memory, and permit more concurrency than the traditional Actor model. Schematic will be useful for intensive parallel applications on parallel machines or networks of workstations, concurrent GUI programming, distributed programming over network, and even concurrent shell programming.

REPORT DATE	**WRITTEN LANGUAGE**
Dec 1995	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
17	36

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
To Appear in Proceedings of Object Based Parallel and Distributed Computation, Lecture Notes in Compute Science

DISTRIBUTION STATEMENT
First issue 40 copies.

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Schematic: A Concurrent Object-Oriented Extension to Scheme

Kenjiro TAURA andAkinori YONEZAWA
Department of Information Science
University of Tokyo
{tau,yonezawa}@is.s.u-tokyo.ac.jp

1 Introduction

Programmers in the world, we believe, will begin to use *concurrent* languages to do their everyday tasks, including demanding and intensive computation, distributed programming over network, user interface programming and even text file processing. As concurrent programming is in general harder than sequential programming, concurrent languages are often considered as 'special-purpose.' There are, however, many evidences and trends that support the above prospect. First, parallel machines will become ubiquitous. There is a strong economical demand that parallel intensive applications should run not only on dedicated parallel machines (such as CM5, AP1000, T3D, and Paragon), but also on networks of workstations [5, 33]. Recent research [2] has demonstrated some technical evidences which support the idea. Second, there are classes of applications which can be more naturally expressed in concurrent (in particular, object-oriented) languages such as GUI or distributed programming. GUI programmers specify their programs by a set of "responses" to events (input), which can be modeled as a set of objects each method of which describes a response to a specific event. This model is independent of whether objects are concurrent or not, but concurrent objects have much more provisions for handling multiple events in parallel, because accesses to concurrent objects are somehow arbitrated[1] by the runtime system. In distributed programs, concurrent objects give us a natural abstraction of remote data. In summary, concurrent languages serve as a vehicle both for driving parallel machines more easily *and* expressing certain problems more naturally.

Based on the above observation, we designed and implemented an extension to the programming language Scheme, called Schematic. This paper focuses on

[1]This does not necessarily mean *all* accesses to them are serialized, as described in Sect. 5.

its language design. The extension is *concurrency* and *object-orientation*—we add a set of flexible primitives for concurrency and a safe means for dealing with mutable data structure. We believe the design of Schematic interests two types of concurrent language designers. First, designers who wish to extend an already popular sequential language into parallel one will be interested in how Schematic naturally *integrates* powerful concurrent primitives into existing sequential features such as function calls. Second, those who design a new parallel language, perhaps based on a concurrent calculus, will be interested in how concurrent primitives + a set of simple syntactic tricks can give us a concise and familiar syntax for sequential or simple parallel program constructs. They are beneficial for lowering the learning barrier of the language without sacrificing the simplicity of the computation model and implementations of the language.

The range of target applications includes intensive applications (irregular symbolic or algebraic computation, in particular[2]), interactive applications (GUI in particular), and distributed programming over networks. For irregular intensive applications, Schematic supports very efficient fine-grain thread creation and communication. In our previous work [30, 32], we have demonstrated runtime techniques for creating and scheduling excess parallelism within a processor with very low overhead (a local thread creation + reply value communication approximately take ten RISC instructions + function call overhead). For GUI applications, we are currently working on a GUI library where each widget is now a concurrent object and multiple events can be delivered simultaneously. Since method invocations on a widget is arbitrated by the runtime system, almost no further complication is added from the programmer's point of view, while he/she performs concurrent processing of multiple events.

After giving a brief overview of Schematic in Sect. 2 and some background in Sect. 3, we introduce basic concurrency primitives and concurrent object-oriented extension in Sect. 4 and Sect. 5, respectively. Sect. 6 demonstrates some examples which highlight the main features of Schematic. Sect. 7 compares Schematic to a wide range of other languages. We finally conclude and summarize the current status in Sect. 8.

2 Schematic Overview

The following is the summary of key extensions key extensions made to Scheme.

Channels: As the fundamental primitive for synchronization, we provide first-class *channels.* A channel is a data structure on which synchronizing

[2]We are not saying that numeric programs are regular. In fact, it is widely known that many numerics benefit from support of irregular data structures [8, 12, 13] and this leads to many proposals of extensions to C++ [6, 7, 11, 19]. The reason why we did not include numerics from the main target applications is just that in our initial implementation, floating point numbers have boxed (hence slow) representation for the simplicity. We are also working on a similar, but statically typed language called ABCL/*f*[31], which focuses on the performance of irregular numerics in fine-grain concurrent object-based languages.

read/write can be performed. Channels can be passed to other processes or stored in any data structure.

Future: As the fundamental construct for creating parallelism, we introduce a variant of the *future* construct originally proposed by Halstead [14]. The result value of a future expression is a channel, which we call *reply channel* of the future expression, via which the result of the invocation can be extracted.

Explicit Reply: The reply channel of an invocation is visible from the invoked process and subject to any first-class manipulation. This feature allows us to construct many flexible communication/synchronization patterns in a natural way. For example, by an explicit reply channel, multiple invocations can share a single reply channel, or an invocation can delegate the reply channel to another invocation.

Concurrent Objects: Concurrent objects are supported as a convenient and recommended way for sharing *mutable* data structures among concurrent processes. A concurrent object is a data structure where a method invocation can be regarded as an *instantaneous* transaction on that object, in the sense that methods never observe intermediate state of other transactions and when multiple method invocations are performed almost simultaneously, the effect of a method invocation is never lost.

Concurrent Accesses: While achieving the instantaneousness of a method invocation, we still allow a certain amount of concurrency between multiple method invocations on a single concurrent object. In particular, we guarantee that read-only methods are never blocked by other methods.

3 Background

This section briefly surveys related work which directly influenced the design of Schematic. A thorough comparison to other concurrent languages is given in Sect. 7.

3.1 Concurrent Calculi

Concurrent calculi, such as HACL [20] and π-calculus [21], have been drawing much attention and some languages have been designed based on them [23, 24]. The goal is to identify the 'core' language which expresses various computation patterns by a small number of fundamental primitives. In their simplest term, both HACL and π-calculus are based on channels communicating via processes. Channels are first-class citizens which can be passed to other processes, sent through another channel, and stored into data structures. When a process tries to get a value from a channel and another process tries to put a value in the same channel, a communication will take place via the channel.

Although these concurrent calculi are simple and powerful, expressing everything in the pure calculi is tedious. For example, a sequential function call would be expressed by two processes (the caller and the callee) communicating the result value via a channel. Thus the practical concern when designing a language based on them is how to incorporate familiar constructs (e.g., sequential/parallel function calls) into the language, while keeping the purity of the core.

The design of Schematic achieves both the simplicity of the core and familiar/convenient syntax for frequently used idioms such as `future` calls. A `future` call, for example, is understood as a combination of a channel creation and a process invocation. Even higher-level constructs are realized using channels or futures (and can be defined as macros, as in Scheme).

The semantics of Schematic can be understood by encoding it into an untyped subset of HACL. Our optimizing compiler which is currently under development uses the untyped subset of HACL as the intermediate language and we are now investigating the analysis and optimization on the simple intermediate language.

3.2 Linearizable Objects

Herlihy et.al [17] defined "linearizability," which captures and formalizes "intuitively correct behavior of objects" in the presence of concurrent processes. An execution of a program consists of a sequence (history) of events, each of which is either an invocation or a termination (acknowledgement) of a method invocation. A history is linearizable if events can be reordered, preserving the order of method invocations in the original history, to a *sequential* history, a history in which method executions do not overlap.

They also showed that achieving linearizable schedules neither requires blocking other processes (mutual exclusion) nor centralized scheduler (as is required in serializability). In particular, they have shown that a history is linearizable if subhistory for each object is linearizable. An implementation of an object is called linearizable if it only permits linearizable history.

In a different paper, Herlihy [16] proposed an implementation of linearizable objects which never result in deadlock. Our implementation of concurrent objects is an instance of an implementation of linearizable objects. Herlihy's scheme is more permissive than ours with respect to deadlock, but will be less efficient than ours because of extra memory store due to the provision for possible retries.

4 Basic Parallelism and Synchronization Primitives

The driving principle of the design of Schematic is the view that *a function/method invocation is, whether it is sequential or asynchronous, just a special case of a process creation.* More precisely, when we have some way for process creation and communication between processes, and we regard a Scheme lambda expression as (the definition of) a process, a function call is achieved by invoking a process which will put the result value to a communication medium. A sequential call just tries to get the result value immediately, while an asynchronous call at a later time.

In Schematic, processes as well as the medium of process communication, which we call *channel*, are first-class entities, just as functions are first-class in Scheme. This guarantees the flexibility of Schematic in the sense that whatever can be expressed in HACL or π-calculus has an obvious counterpart in Schematic[3]. This is of course true for other languages which support first-class channels and processes, but Schematic more carefully *integrates* the extensions to Scheme than some other extensions to sequential languages such as Concurrent ML [25], and more concisely expresses simple and frequent cases than languages based on concurrent calculi such as PICT [24].

4.1 Channels

Channels are the fundamental entities which realize synchronization and communication between processes. Channels can be explicitly created via the following form:

```
(make-channel),
```

though they are most often *implicitly* created as the result of a process creation as will be described in Sect. 4.3.

Let c be a channel, we can perform following operations on c:

- `(touch` c`)`—extracts a value from c. The value is supplied to the enclosing expression.
- `(reply` x c`)`—puts x in c. The enclosing expression immediately gets an unspecified value.

There may be multiple pairs of `touch/reply` performed on a single channel, in that case the extracted value is an arbitrary one which has been put until that time.

[3]This is not strictly true for π-calculus because writing a value to a channel in Schematic is asynchronous, while it is *synchronous* in π-calculus, in the sense that writing a channel in a π-calculus specifies a post action which is executed after the reply has been completed. We presume this rarely makes difference in practice, and a synchronous call can be emulated by composing asynchronous ones, although it is tedious.

4.2 Processes (or `lambda`)

A process in Schematic is an analogue of a function in functional languages, and in fact Schematic uses a `lambda` expression to express a process. A process should not be confused with a running computation which is created when a process is invoked by `future` expression as will be explained in the next Subsection.

A process expression in Schematic can have the same form as a lambda expression in Scheme. That is,

(lambda (*args* ···) *exprs* ···).

This represents a process which, when invoked, evaluates *exprs* and `replies` the value to a channel, which we call *the reply channel* of the invocation. Neither the reply channel nor the reply operation does appear in this program text explicitly, but when desired they can be explicited by:

(lambda (*args* ···) (:reply-to *r*) *exprs* ···).

Here, (:reply-to *r*) declares the name of the reply channel as *r*. In this case, the value of *exprs* is not automatically replied to *r*. The programmer can reply a value whenever appropriate in *exprs* or defer the reply until a later time or even forever.

In essence, we add an extra parameter to each lambda expression, the parameter which represents the location where the result value should be stored. The reply channel as well as the reply operation to the reply channel is usually implicit and this case subsumes the lambda expression in Scheme. Explicit reply gives the programmer the ability to *decouple* the termination of a process and the delivering the result value; a process may reply a value earlier than its termination and continue some computation, reply values multiple times, or defer the reply until some synchronization/resource constraints are satisfied.

Notice that the implicit case is obtained by the following simple syntactic equivalence:

(lambda (*args* ···) *exprs* ···)
≡ (lambda (*args* ···) (:reply-to *c*) (reply (begin *exprs* ···)
c)).

That is, if the reply channel is implicit, a lambda expression replies the last evaluated value to the reply channel implicitly.

4.3 Process Invocation (or `future`)

Suppose f be a process (`lambda` expression). We can invoke f by the following form:

(future (f *args* ···)).

This actually performs the following.

- create a new channel as the reply channel of the invocation,
- fork f,
- pass arguments and the reply channel to f.
- the entire expression (future (f *args* $\cdots$)) evaluates to the reply channel.

A reply channel may or may not be explicit in the body of f, but it is, at least conceptually, created whenever a process is invoked.

A future form may specify a reply channel explicitly, allowing multiple invocations to share the same reply channel or delegate the reply channel of an invocation to another invocation. This is expressed by:

(future (f *args* $\cdots$) :reply-to r),

which might be read as "invoke a process which will reply a value to r." When the clause :reply-to r is omitted, a new channel is created. That is,

(future (f *args* $\cdots$))
$\equiv$ (future (f *args* $\cdots$) :reply-to (make-channel)).

For example, the following fragment invokes (f x) and (g x) which share a single reply channel and then touches it.

```
(let ((c (make-channel)))
  (future (f x) :reply-to c)
  (future (g x) :reply-to c)
  (touch c))
```

In effect, this fragment waits for whatever arrives first to c and discards the other.

An ordinary sequential function call is derived by:

(f *args* $\cdots$) $\equiv$ (touch (future (f *args* $\cdots$))).

That is, a familiar syntax of function call in Scheme is now interpreted as the particular form of a process invocation and communication.

4.4 Higher-Level Constructs

We also provide even higher-level constructs than futures. They include:

plet: a parallel version of let, which evaluates all bound values in parallel.

pcall: a parallel version of apply (evaluates all arguments in parallel)

pbegin: a parallel version of `begin`, which evaluates all subexpressions in parallel.

pmap: a parallel version of `map`, which applies the given function to all elements of the list in parallel.

pfor-each: a parallel version of `for-each`.

They can be defined by simple macros.

5 Concurrent Object-Oriented Extension

Schematic extends Scheme with concurrent objects which serve as a stylized means for using mutable (updatable) data structure safely in concurrent applications. Scheme does have updatable data structures (cons cells, strings, vectors, symbols are all updatable), but they are not ready to be shared by concurrent processes, in the sense that an interleaving execution of multiple transactions on a single data may result in a state which can never be reached by a non-interleaving execution.

A concurrent object exhibits simpler and more intuitive behavior than the 'bare' shared memory. The most important property is the instantaneousness of a method invocation. That is, from the programmer's point of view, all the actions involved in a method execution appear to take effect at some *point* between its invocation and termination. Hence the programmer never has to reason about how potential *interleaving* executions of concurrent method invocations[4] may affect the result, despite that the implementation may schedule multiple method invocations in parallel.

Realizing an instantaneous method invocation itself is a very trivial matter—we could serialize all the method invocations on an object, as in the traditional Actor model [1]. But this loses a significant amount of concurrency and narrows the range of provably deadlock-free programs (*i.e.,* broadens the range of programs which potentially result in deadlock).

Hence other goals we must achieve are a reasonable amount of *guaranteed* concurrency (deadlock freedom) and a *simple* model in which the programmer can reason about possible object states and potential deadlocks.[5] The object model of Schematic achieves instantaneous methods, while guaranteeing a certain amount of concurrency between method invocations. In particular, read-

[4]Here, we say method invocations M and M' are concurrent if there are no data dependencies that guarantee they never overlap. Notice that this definition is independent of any implementation or scheduling strategy that determines if they are really scheduled in parallel.

[5]The importance of the simple model should not be underestimated. As long as there remains some possibility of deadlock or starvation in the language, a model or a system which judges possibility of deadlock *must be told to the programmer.* In particular, language designers must not pursue to guarantee *maximum* concurrency at the cost of the simplicity of the model.

only methods are never blocked by other methods. We will explain details in Sect. 5.2.

5.1 Classes and Methods

5.1.1 Defining Classes.

A class is defined by `define-class` and a method by `define-method` or `define-method!`. For example,

```
(define-class point ()
  x
  y)
```

defines class called `point`, each instance of which has slots called `x` and `y`. What follows after the class name is the list of inherited classes. For example,

```
(define-class color-point (point)
  color)
```

defines `color-point` class, each instance of which now has slot `color` in addition to `x` and `y`.

A `define-class` implicitly defines a function with the class name which creates an instance of the class. For example, an instance of `point` class is created by:

```
(point 2.0 3.0).
```

5.1.2 Defining Methods.

The following defines a method which returns the distance between the point and the origin.

```
(define-method point (distance self)
  (sqrt (+ (* x x) (* y y))))
```

This can be, as usual, called by

```
(distance p),
```

where p is an instance of `point` (or whatever inherited from `point`).

Explicit reply channels can be used in methods as well. For example, `distance` method could also be written by:

```
(define-method point (distance self)
  (:reply-to r)
  (reply (sqrt (+ (* x x) (* y y))) r)),
```

though this is just a clumsy coding style of the previous simpler definition.

5.1.3 Updating States.

Updating the state of an object is expressed by `become` construct which specifies new values for updated slots as well as the result of the entire expression. For example, the following method increments `x` and `y` by `dx` and `dy` respectively, and returns `#t`.

```
(define-method! point (move! self dx dy)
  (become (redraw! self) :x (+ x dx) :y (+ y dy)))
```

The first argument of a `become` (`(redraw! self)` in this case) is called *result expression* of the `become` and specifies which value the `become` is evaluated to, whereas the rest part the updated values for slots. Values for unchanged slots can be simply omitted.

Notice that we used `define-method!` above, rather than just `define-method`. The rule is that a `become` cannot appear in the body of `define-method`. We put a further restriction on the position of `become` inside the body of a `define-method!`. That is, `become` should be performed *as the last action of the method.* For example, we permit

```
(define-method! class (method self ...)
  (if ...
      (become ...)
      (become ...)))
```

and,

```
(define-method! class (method self ...)
  (let ((x ...))
    ...
    (become ...))),
```

while we reject

```
(define-method! class (method self ...)
  (+ (become ...) 10)).
```

In other words, `(become ⋯)` can appear only at such places that the value of the `become` becomes the value of the entire method body.

The intention is to prohibit updating instance variables multiple times in a method, so that a method can naturally be regarded as a single transaction. However, we have not yet identified a set of simple syntactic (or static) rules which reject the violated use of `become`s and accept most reasonable programs. In this paper we simply assume that `become` only appears in the right places or that a method automatically terminates when the first `become` is performed.

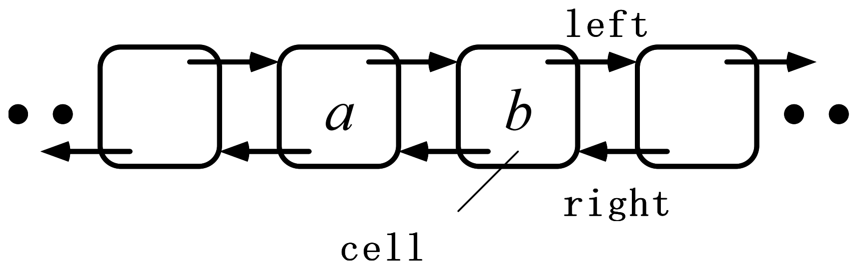

Figure 1: Cell objects linked by `left` and `right` fields.

5.2 Concurrency Semantics

Concurrency semantics refers to the way in which the programmer reasons about deadlock and liveness. Notice that it does not tell the programmer which pair of methods are really scheduled in parallel.

To illustrate the problem, consider a possible description of a relaxation step on a one dimensional mesh.

```
(define-class cell ()
  value
  new-value
  left          ; cell object
  right)        ; cell object

(define-method cell (current-value self)
  value)

(define-method! cell (relax! self)
  (let ((lv (current-value left)) (rv (current-value right)))
    (become #t :new-value (- (+ lv rv) (* 2 value)))))
```

Many cell objects are created and form a doubly linked list via `left` and `right`, as in Fig. 1. A relaxation step updates `value` of `self` using current `value`s of `left` and `right`. In the traditional Actor model, invoking `relax!`s in parallel may result in deadlock. The situation is more formally illustrated as the follows [17]. An execution of a program is a history of events. An event is either an invocation of a method on an object or a termination of a method on an object. We write an invocation of method M on object o by $\langle \text{inv } M\ o \rangle$ and a termination of method M on object o by $\langle \text{ack } M\ o \rangle$.[6]

An execution of the above program may proceed as follows (a and b reference each other as illustrated in Fig.1).

[6] We slightly changed the notation in [17] for the purpose here.

$\langle$inv `relax!` $a\rangle$
$\langle$inv `relax!` $b\rangle$
$\langle$inv `current-value` $b\rangle$

Here, it is impossible to complete any of three pending invocations by appending a corresponding ack event, under the scheduling constraint from data dependence (*i.e.,* $\langle$ack `current-value` $x\rangle$ must precede $\langle$ack `relax!` $x\rangle$) and mutual exclusion (*i.e.,* two methods on a single object cannot overlap).

We refine the traditional Actor model in two ways. First, we classify methods into two types, *i.e.,* those defined by `define-method` (which we call `method` below) and those defined by `define-method!` (which we call `method!` below). A `method` is guaranteed to always progress and can overlap with other `methods` or `method!`s. Second, we breakdown an execution of a `method!` into two stages. One is the evaluation of the result expression of `become` and the other is whatever remains in the method body. Let us call the evaluation of a result expression *after-stage* and the other*before-stage*. In the `move!` method, for example,

```
(define-method! point (move! self dx dy)
  (become (redraw! self) :x (+ x dx) :y (+ y dy)))
```

the before-stage consists of evaluating `(+ x dx)` and `(+ y dy)` and updating the instance variables. After-stage invokes `redraw!` method for self. Our relaxed mutual exclusion rule is that a before-stage of an invocation cannot overlap with before-stages of other invocations, but an after-stage can. Hence, this example does not deadlock.

The first rule says that, as far as deadlock is concerned, we only have to consider `method!`s. The above relaxation example is safe because invoking a `relax!` per an object obviously never causes any interleaving on a single object. The second rule says that, a `method!` can release the lock associated to an object earlier than its termination. In the result expression, we can safely wait for the acknowledgement from other computation, which may include a recursive call to self.

When an execution still results in a situation where no further progress is possible, we simply deadlock and never retry or abort unlike schedulers in many database systems.

5.3 Consistency Semantics

First, the state of an object is *atomically* updated at the point between the before-stage and the after-stage of a `method!`. That is, a method never observes part of slots updated, while others yet unchanged, no matter how many slots are being updated. Second, a value of a slot within a method never changes. A method continues to observe, or 'freezes,' the slot values at the beginning of the method, no matter how the execution is long and likely to be interleaved with other method executions. This is true even if a method calls another method to `self` inside the body, hence is *necessarily* interleaved with another method.

Hence a state observed in a method is determined by the sequence of `become` completed before the method in question is invoked. For example, consider a simple counter object given below.

```
(define-class counter ()
  value)

(define-method counter (get-value self)
  value)

(define-method! counter (add-value! self x)
  (become value :value (+ value x)))
```

Letting c be a counter with initial `value` 0, consider

```
(let ((r0 (future (add-value! c x)))
      (r1 (future (add-value! c x))))
  (touch r0)
  (touch r1)
  (get-value c)).
```

The first `touch` operation returns either 0 or `x`, since `r1` may or may not have been completed at the *invocation* of the first `add-value` (We do not guarantee to preserve the order of asynchronous invocations). The same holds for the second `touch`, but one of `r0` or `r1` gets 0 and the other `x`. This is the case because before-stages of two `method!`s on a single object never overlap. Hence the final `get-value` always gets $2 \times$ `x`.

A care must be taken if we call a `method!` to self from another `method`. Since the slot values are persistent throughout the method body, the effect of the `method!` cannot be observed in the outer method.

6 Examples

6.1 Concurrent Tree Updating

This example demonstrates how the concurrency semantics of our model, the notion of before/after-stage in particular, allows natural description of a concurrent data structure. Consider a binary tree search algorithm where each node of the binary tree is a concurrent object. Here is the definition of each node object.

```
(define-class bintree-node ()
  ;; this node associates mapping
```

```
  ;; between KEY ↔ VALUE
  key
  value
  ;; children (#f when it does not exist)
  left
  right)
```

Each node has its key and associated value. It holds that the key of the left child is less than that of self and the key of the right child is greater than that of self. Hence binary search operation is very straightforward.

```
;;;
;;; Lookup the value associated for K.
;;;

(define-method bintree-node (lookup self k not-found)
  (cond ((= k key) value)                ; found
        ((< k key)
         ;; look for the left subtree if K < KEY
         (if left
             (lookup left k not-found)
             not-found))
        (else
         ;; look for the right subtree if K > KEY
         (if right
             (lookup right k not-found)
             not-found))))
```

Since this operation does not update the tree, we use `define-method`, hence multiple `lookup` invocations can simultaneously operate on a single tree. The following method associates element `val` with key `k`.

```
;;;
;;; Establish association K ↔ VAL, maintaining the
;;; following invariant:
;;;    "KEY of LEFT < KEY of SELF < KEY of RIGHT"
;;;

(define-method! bintree-node (insert! self k val)
  (cond ((< k key)
         (if left
             ;; if there is already left child, delegate this value
             ;; to the child, unlocking self
             (become (insert! left k val))
             ;; if there is no left child, create it
```

```
        (become #t :left (make-leaf-bintree-node k val)))
    ((= k key)
     (format #t "Warning conflicting key (~s ~s)~%" key value)
     (become #f))
    (else
     ;; the same algorithm as the first case, but for the right child
     (if right
         (become (insert! right k val))
         (become #t :right (make-leaf-bintree-node k val))))))
```

This method first finds the appropriate place to which we insert the item and then installs a new node to the place. An interesting case happens in internal nodes; an internal node recursively calls `insert!` method for an appropriate child *after* it unlocks `self` for subsequent requests. This is expressed by

```
(become (insert! left k val))
```

at line 6 and

```
(become (insert! right k val))
```

at line 15. As has been described in Sect. 5.2, these recursive calls are in the after-stage of the method, *i.e.,* executed after `self` has been unlocked.

6.2 Synchronizing Objects

To demonstrate the expressive power of explicit reply channels, consider an implementation of an object which embodies an application-specific synchronization constraint. That is, upon a method invocation, the object defers the reply of the invocation until certain synchronization constraints are satisfied by subsequent methods. Simply blocking computation *inside* the method does not work, because this excludes subsequent method invocations, thus blocks the original computation forever!

For a simple example, consider implementing a "barrier synchronization" object. A set of processes shares a barrier object and each process invokes `finished!` method on the barrier object when its local computation has been done. `Finished!` method does not reply any acknowledgement to the process *until the last invocation reaches the object.*

Here is the definition of `barrier` class.

```
(define-class barrier ()
  ;; number of finished! to wait
  n
  ;; number of finished! so far processed
  count
  ;; list of reply channels
  waiters)
```

An instance of a `barrier` class has three instance variables `n, count, waiters` where `n` is the number of `finished!` calls to be synchronized, `count` the number of `finished!` which have been made, and `waiters` the list of reply channels of previous calls. When synchronization is realized, that is, n'th call to this object is made, it replies value `#t` to all the channels in `waiters` as well as the current reply channel.

Method `finished!` facilitates explicit reply channel for deferring the replies.

```
;;;
;;; When this  finished! is the last call, it unblocks
;;; all the waiters by explicitly calling  reply, otherwise
;;; it does not reply anything so that the caller is blocked.
;;;

(define-method! barrier (finished! self)
  (:reply-to r)
  (if (= (+ count 1) n)
      ;; reply #t to every channels
      (become (for-each (lambda (x) (reply #t x)) (cons r waiters)))
      ;; reply nothing
      (become #t :n n :count (+ count 1) :waiters (cons r waiters))))
```

In the above, the reply channel is named `r`. If `(+ count 1)` $\neq$ `n` (*i.e.,* this is not the last invocation), the method stores `r` in list `waiters`, replying nothing to `r`. In the last invocation, the method serves a reply for every reply channel so far received.

7 Comparison to Other Languages

Schematic can be related to several groups of other concurrent languages. First, Schematic is a language whose computation model is based on a concurrent calculus which gives us the foundations of aggressive compiler optimizations. Second, Schematic supports concurrent objects which allow/guarantee more concurrency than the traditional Actors. Third, Schematic is an extension of a popular sequential language, which already has a philosophy to be preserved. They are summarized below in order.

7.1 Languages Based on Concurrent Calculi

7.1.1 PICT.

PICT [24] is a concurrent language based on π-calculus [21]. Its design goal is to have a simple language directly based on π-calculus which also supports frequently used higher-level idioms as syntactic rules (just as Scheme is based on λ calculus and has higher-level idioms such as `do` loop). Although its language design is still evolving, there seems no constructs which directly support future or even sequential function calls. Schematic has the same goal and demonstrates that, by looking at function calls and lambda expressions of Scheme in a slightly different way, a language with a very small number of fundamental primitives can at the same time provide convenient constructs (such as `future` or `plet`) for typical cases.

7.2 Concurrent Extensions to Sequential Languages

Extending a sequential language to yield a concurrent dialect has many practical advantages. Among others, we compare Schematic with Multilisp and Concurrent ML.

7.2.1 Multilisp.

Multilisp [14] is the language which originally embodies the `future` construct. The central idea of `future` that a future expression returns something which later becomes the result value is adopted not only in parallel Lisps but also in some concurrent object-oriented languages [18, 31, 34, 36].

Schematic also supports a variant of `future`. An apparent difference between the `future` in Multilisp and the one in Schematic is that in Multilisp, producer-consumer synchronization of a future invocation is implicit in value reference, whereas Schematic requires explicit `touch` operations. For example, invoking `(f x)` and `(g y z)` in parallel and then adds the two results is written by

```
(+ (future (f x)) (future (g y z))),
```

in Multilisp, while it is written by

```
(let ((l (future (f x))) (r (future (g y z))))
  (+ (touch l) (touch r)))
```

in Schematic.[7]

Informally, the Multilisp view of a future is that what is immediately returned by a future expression is a placeholder object, which later *becomes* the result value for itself, whereas the Schematic view is that a future expression returns a placeholder *into which the result value is stored.*

There are tradeoffs between the implicit and the explicit version. The implicit version, as the above example indicates, often results in a terse expression but loses some flexibility. By making `touch` explicit, we can distinguish a reference to the placeholder itself from the reference to the value which is stored in

[7]As far as this particular example is concerned, `plet` would express it more nicely.

the placeholder by the program text. This not only guarantees fast value reference without additional compiler analysis [29], but also gives us more expressive power by making the placeholder first-class citizens. Examples have been given in Sect. 6.

Another difference is their positions on shared mutable data. Multilisp provides Scheme builtin data as the basis for mutable data and some atomic memory operations such as `replace-if-eq` (analogue of *compare & swap*). No higher-level mechanisms for defining safe mutable data are provided. Schematic supports and encourages the use of concurrent objects to represent mutable data, concurrent accesses to which are arbitrated by the runtime system.

7.2.2 Concurrent ML.

Concurrent ML (CML) [25] extends SML by first-class channels and fork (`spawn`), whereas Schematic extends Scheme by first-class channels, fork (`future`), and concurrent objects. To put concurrent objects aside, the main difference is that CML does not support any higher-level concurrent primitives (parallel calls or even `futures`).

Consider how to do two CML function calls `f x` and `g x` in parallel. Since the results must now be extracted from a channel, let us define a 'wrapper' function which takes a channel and sends the result of `f x` to the channel.

```
fun wrapper f x c = send (f x, c)
```

What remains is to create two channels, spawn two `wrappers`, and wait for the result.

```
let c0 = channel ()
and c1 = channel ()
in
   (spawn (fn () => wrapper f x c0);
    spawn (fn () => wrapper g x c1);
    accept c0; accept c1)
end
```

Presumably, a fragment like this will appear very often and should be more stylized, as in Schematic. In fact, a restricted version of future can be defined in CML by

```
fun future f x =
  let c = channel ()
  in
    (spawn (fn () => send (c, f x)); c)
  end.
```

Except that it can only invoke a unary function, the above future takes any function and any argument and returns the reply channel. This is more monolithic and less flexible than `futures` in Schematic, in that a future now always creates a reply channel and the caller loses the chance to specify a reply channel.

Given that a function is the fundamental building block of CML programs, CML should support and encourage a convenient way for invoking functions in parallel. Schematic is designed based on this principle, while leaving chances to construct customized communication structure whenever desired.

7.3 Concurrent Object-Oriented Languages

A *concurrent object* refers to data that embodies some access arbitration mechanisms so that an execution of a method never observes inconsistent state of an object. Several object models have been proposed and they differ in the degree of concurrency on a single object, therefore the range of deadlock free programs.

7.3.1 Actors and Early Concurrent Object-Oriented Languages.

The original Actor model [1] and some early concurrent object-oriented languages such as ABCL/1 [35, 36] and Cantor [4] achieves the instantaneousness of a method execution by mutually excluding all the method invocations on an object. This is often explained by "an autonomous object which has its own thread and message queue." Although the traditional Actor model gives us the instantaneousness and a very simple model in which the programmer reasons about deadlock, it is often criticized to serialize too much. This not only loses performance gain which is otherwise possible by exploiting parallelism, but also enforces unnatural description of algorithms to solely avoid potential deadlock.

7.3.2 Concurrent Aggregates.

Concurrent Aggregates (CA) [9, 10] supports *aggregates* in addition to regular objects. A regular object is a traditional Actor and an aggregate is internally composed of multiple objects, but externally viewed as if it were a single object. By processing multiple method invocations on an aggregate by multiple internal objects, an aggregate can serve as a non-serializing object. Maintaining the consistency among multiple internal objects, if required, is the responsibility of the programmer.

7.3.3 UFO and Sympal.

An object in more recent languages such as UFO [26, 27] and Sympal [3] allows/guarantees more parallelism than the traditional Actor. Schematic also belongs to this category and UFO, Sympal, and Schematic are common in many ways. First they support multiple paradigms, in the sense that they do not force programmers to use concurrent objects wherever concurrency is required.

This avoids serializing computation which does not require shared mutable data. Second, a method in those languages allows subsequent methods on an object to overlap with the current method after the current method reaches a certain point. In UFO, the compiler statically identifies a point after which instance variables are never updated and unlocks the object when the execution reaches that point. Sympal expresses updating instance variables by the `finally` construct which performs all updates in the method all at once. `Finally` also takes another expression (called "continuation" in [3]) which becomes the result of the method and evaluating the continuation expression can overlap with subsequent methods. The object model of Schematic is most close to that of Sympal and supports `become` expressions which have almost the same semantics as `finally`.

7.3.4 C++ Dialects.

Here we only discuss C++ dialects which support *objectwise* concurrency control mechanisms and do not discuss a notable data-parallel extension pC++ [6].

CC++ [7] does not directly support concurrent objects, but the similar effect can be achieved by `atomic` member functions. By declaring a member function as `atomic`, the member function locks/unlocks the object at invocation/termination as in the traditional Actors. Thus the object model of CC++ has the same problems with early concurrent object-oriented languages. Non-atomic functions can run concurrently with others, but this merely leaves consistency issues for the programmer.

Objects in ICC++ [11] allows two methods M and M' to operate on a single object in parallel if there are no read/write nor write/write conflicts between them *on any instance variable of the object.* The main difference between ICC++ and the UFO/Sympal/Schematic group is that the ICC++ model performs mutual exclusion on a per instance variable basis, rather than per object basis.

The range of programs which are guaranteed to be scheduled without deadlock do not seem quite different between ICC++ and Schematic. A foreseeable problem with the ICC++ object model is that each object now potentially has to have multiple locks to serialize only conflicting methods. The worst case requires a lock per instance variable and removing redundant locks requires global information on the source code.

8 Summary and Current Status

The design of Schematic, a concurrent object-oriented extension to Scheme, has been presented. Just as most part of Scheme can be understood in terms of a very simple calculus (the λ-calculus), most part of Schematic can be understood in terms of a simple *concurrent* calculus (HACL). To make it really practical, Schematic also supports and encourages the use of familiar paradigms (*i.e.,* `futures` and concurrent objects) as well, achieving both the simple core of the language and the conciseness/convenience in typical programs.

A prototype on top of a sequential Scheme (`Scheme->C`) has been implemented and is running on AP1000 and AP1000+ massively parallel processors [15, 28]. We had developed an RNA secondary structure prediction algorithm [22],[8] which is essentially a parallel tree search with application-specific priority and a load-balancing control scheme, and Barnes-Hut Nbody algorithm. Experiments on an AP1000+ system (SuperSparc 50 Mhz $\times$ 256) indicated an usable performance, though many more improvements are necessary.

Further information is available via:

`http://web.yl.is.s.u-tokyo.ac.jp/pl/schematic.html`.

References

[1] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, 1986.

[2] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[3] Yariv Aridor. *An Efficient Software Environment for Implicit Parallel Programming with a Multi-Paradigm Language.* PhD thesis, the Senate of Tel-Aviv University, 1995.

[4] W. C. Athas and C. L. Seitz. Cantor user report version 2.0. Technical report, Computer Science Department, California Institute of Technology, 1987.

[5] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of Supercomputing*, pages 588–597, 1993.

[7] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 11, pages 281–313. The MIT Press, 1993.

[8] This paper describes algorithms and results by message passing C on CM5, and we are now preparing the result in Schematic.

[8] Andrew Chien, M. Straka, Julian Dolby, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. A case study in irregular parallel programming. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[9] Andrew A. Chien. *Concurrent Aggregates (CA)*. PhD thesis, MIT, 1991.

[10] Andrew A. Chien and William J. Dally. Concurrent aggregates (CA). In *Proceedings of the Second ACM SIGPLAN Symposium on Princeples & Practice of Parallel Programming*, pages 187–196, Seattle, Washington, March 1990.

[11] Andrew. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high performance parallel computing. In *Proceedings of the Second International Symposium on Object Technologies for Advanced Software (To appear)*, 1996.

[12] High Performance Fortran Forum. *HPF-2 Scope of Activities and Motivating Applications*, 1994.

[13] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulation of the Barnes-Hut method for n-body simulations. In *Proceedings of Supercomputing '94*, pages 439–448, 1994.

[14] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, April 1985.

[15] Kenichi Hayashi, Tunehisa Doi, Takeshi Horie, Yoichi Koyanagi, Osamu Shiraki, Nobutaka Imamura, Toshiyuki Shimizu, Hiroaki Ishihata, and Tatsuya Shindo. AP1000+: Architectural support of put/get interface for parallelizing compiler. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, pages 196–207, 1994.

[16] Maurice P. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[17] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[18] Waldemar Horwat, Andrew A. Chien, and William J. Dally. Experience with CST: Programming and implementation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, Portland, Oregon, July 1989.

[19] Yutaka Ishikawa. The MPC++ Programming Language V1.0 Specification with Commentary Document Version 0.1. Technical Report TR–94014, RWC, June 1994. `http://www.rwcp.or.jp/people/mpslab/mpc++/mpc++.html`.

[20] Naoki Kobayashi and Akinori Yonezawa. Higher-order concurrent linear logic programming. In *Proceedings of Workshop on Theory and Practice of Parallel Programming (TPPP)*, volume 907 of *Lecture Notes in Computer Science*, pages 137–166. Springer Verlag, 1994. `http://web.yl.is.s.u-tokyo.ac.jp/pl/hacl.html`.

[21] Robin Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[22] Akihiro Nakaya, Kenji Yamamoto, and Akinori Yonezawa. RNA secondary structure prediction using highly parallel computers. *Comput. Applic. Biosci. (CABIOS) (to appear)*, 11, 1995.

[23] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In *Proceedings of Workshop on Theory and Practice of Parallel Programming (TPPP)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer Verlag, 1994.

[24] Benjamin C. Pierce and David N. Turner. PICT: A programming language based on the Pi-Calculus. Technical report in preparation; available electronically, 1995.

[25] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

[26] John Sargeant. United functions and objects: An overview. Technical report, Department of Computer Science, University of Manchester, 1993.

[27] John Sargeant. Uniting functional and object-oriented programming. In Shojiro Nishio and Akinori Yonezawa, editors, *Proceedings of First JSSST International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1993.

[28] Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Low-latency message communication support for the AP1000. In *The 19th Annual International Symposium on Computer Architecture*, pages 288–297, 1992.

[29] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*, chapter 3, pages 47–87. The MIT Press, 1991.

[30] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 218–228, 1993. `http://web.yl.is.s.u-tokyo.ac.jp/pl/schematic.html`.

[31] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation –. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, number 18 in Dimacs Series in Discrete Mathematics and Theoretical Computer Science, pages 275–292. American Mathematical Society, 1994. `http://web.yl.is.s.u-tokyo.ac.jp/pl/schematic.html`.

[32] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. *StackThreads*: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Proceedings of Workshop on Theory and Practice of Parallel Programming (TPPP)*, number 907 in Lecture Notes in Computer Science, pages 121–136. Springer Verlag, 1994. `http://web.yl.is.s.u-tokyo.ac.jp/pl/schematic.html`.

[33] Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-latency communication over ATM networks using active messages. *IEEE Micro*, 15(1):46–53, February 1995.

[34] William Weihl, Eric Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson Hsieh, Anthony Joseph, Carl Waldspurger, and Paul Wang. PRELUDE: A system for portable parallel software. Technical Report MIT/LCS/TR-519, Laboratory for Computer Science, Massachusetts Institute of Technology, 1991.

[35] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System—Theory, Language, Programming, Implementation and Application.* The MIT Press, 1990.

[36] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA '86 Conference Proceedings*, pages 258–268, 1986.