An Experimental Implementation of Action-Based Concurrency

XIAO-LEI CUI, EMIL SEKERINSKI

JULY 2009

SQRL REPORT 58

MCMASTER UNIVERSITY

Abstract

This paper reports on an experimental implementation of action-based concurrent objects. Concurrency is expressed by allowing objects to have actions with a guard and a body, in addition to methods and variables. Condition synchronization is expressed by method guards protecting entry into an object. An action of an object can execute any time when its guard is true, with the restriction that at most one action or methods can execute within an object, thus guaranteeing atomicity. In principle, all objects can execute in parallel. The appeal of this model is that it is conceptually simple for programmers and comes with a verification and refinement theory based on atomic actions that is a straightforward extension of that for sequential programs. The concurrency model is suited for tightly coupled shared-memory processors. The implementation makes use of a recent thread library, NPTL, that supports a large number of kernel threads. Each active object has its own thread that cycles through the actions, evaluates their guards, and executes an enabled action. Syntactic constraints restrict when guards need to be re-evaluated. We compare the efficiency of the generated code with that of C/Pthreads, Ada, and Java using classical concurrency examples.

1 Introduction

In the action system model of concurrency, a program is described by a set of actions of the form $p \rightarrow S$, where p is a predicate, the *guard*, and S is a statement, the *body* of the action. If the guard of action A is true, then A is *enabled*. In the *sequential execution model*, any enabled action $A_i = p_i \rightarrow S_i$ of the system is selected for execution as long as an action is enabled. In case more than one is enabled, the selection is nondeterministic. In Dijkstra's notation this is expressed as:

$$\textbf{do}\, g_1 \rightarrow S_1 \,[\!]\, \ldots \,[\!]\, g_n \rightarrow S_n\, \textbf{od}$$

In the *concurrent execution model*, any two actions that operate on distinct variables may be executed in any order or concurrently, which represents the interleaving model of concurrency. The execution of an action is atomic, i.e. without interference from other actions. Reasoning in the concurrent execution model can be reduced to reasoning in the sequential execution model, allowing concurrent programs to be designed and analyzed as if they were sequential. The action system model can be used to describe and analyze shared-variable concurrency, message-passing, and distributed systems [**?**, **?**, **?**].

As the sequential execution model simplifies design and analysis, the question arises if action systems can be included in a programming language. In a naive concurrent implementation action guards have to constantly re-evaluated. If done in hardware, all guards can be evaluated in parallel; indeed, action systems have been used as a specification language for the development of efficient hardware implementations [**?**, **?**, **?**, **?**]. A production system is a kind of an action system in which guards are matched against against a memory consisting of value-attribute pairs and action bodies are updating that memory. While production systems can be implemented with concurrency, their use is for expert systems rather than as a general programming model [**?**]. An extension of the Oberon-2 language by actions is reported in [**?**]. However, that implementation uses the sequential execution model, hence concurrency is not exploited.

In this paper we report on the implementation of ABC Pascal, an object-based language with Action-Based Concurrency. The name also suggest that the language is meant to be particularly simple and suitable for explaining concepts of concurrency. If one allows objects to evolve autonomously, objects become a natural "unit" of concurrency. By attaching actions to objects, rather than having "global" actions, objects become action systems and an object-oriented program becomes the parallel composition of objects. We illustrate this with the example

of an aquarium in ABC Pascal. An action $p \rightarrow S$ is written as **when** *b*; *S* within a class declaration:

```
const W = 400;
class Fish
  var x, d: integer; r: boolean;
  procedure setPace (p: integer); {p > 0}
    begin d := p end;
  action moveRight when (x + d < W) and r;
    begin x := x + d end;
  action moveLeft when (x − d ≥ 0) and not r;
    begin x := x − d end;
  action changeToRight when (x < W − 1) and not r;
    begin r := true end;
  action changeToLeft when (x > 0) and r;
    begin r := false end;
  begin x := 0; d := 5; r := true end;
var aquarium: array [1..17] of Fish;
```

Objects with actions are called *active objects*. After an active object is declared, first its initialization, as given by the begin-end block at the end of the declaration, is executed and then any of its enabled actions can be executed or any of its method (procedures) can be called, with the restriction that there can be only one activation of an action or method. Actions can have a name, but the name does not carry any meaning. After the declaration of *aquarium*, 17 fish objects will be active; they do not need to be created separately.

Communication between objects is expressed through method calls and synchronization by method guards. For example a bounded buffer is expressed as:

```
const S = 10;
class Buffer
  var b: array [0 .. S − 1] of integer;
    in, out, n: integer;
  procedure put (x: integer) when n < S;
    begin b[in] := x; in := (in + 1) mod S; n := n + 1 end;
  procedure get (var x: integer) when n > 0;
    begin x := b[out]; out := (out + 1) mod S; n := n − 1 end;
  begin in := 0; out := 0; n := 0 end;
var b: Buffer;
```

Thus calls to *put* and *get* may block. For example, producers and consumers with buffer *b* in between would contain potentially blocking calls:

```
action produce;
  var x: integer;
  begin x := ... ; b.put(x) end;
action consume;
  var y: integer;
  begin b.get(y); ... y ... end;
```

Several models of objects with actions have been proposed. In the *joint actions* model, an action can involve several objects [**?**]; while allowing certain communication patterns to be expressed more abstractly, we restrict to actions attached to single objects to allow an efficient implementation. In the Seuss approach objects have methods and actions, with the restriction that guarded methods can appear only as the first statement in an action of a method [**?**]. Thus an action like *produce* above is forbidden. This model is used for a *reduction* property which states that every execution in the concurrent execution model corresponds to an execution in the sequential model. In OO-action systems the enabledness of an action is determined by the guard as well as the body [**?**, **?**]. For action *produce*, if *b.put*(*x*) is not enabled, the action *produce* is not enabled. In a direct implementation, the state of an object would have to be rolled back in order to ensure atomic execution. If calls to other objects would have been called in the meantime, their state would need to be rolled back as well, making an implementation inefficient. In ABC Pascal this is relaxed: methods and action are atomic only up to method calls. A formal model is discussed in [**?**].

Briot et al. give a classification of concurrency in object-oriented programming, based on the *level of concurrency*, *autonomy of objects*, and the *acceptance of messages* [**?**]. The level of concurrency in ABC Pascal can be classified as *serial*, like in POOL [**?**, **?**] as only one method or action in an object may be active. This is unlike *quasi-concurrent* objects, as in ABCL/1 [**?**], where several method activations may coexist, but at most one is not suspended and in unlike *fully concurrent* objects as in Actors [**?**]. ABC Pascal objects would be classified as *autonomous* rather than *reactive*, as they may be active without receiving a method call; in Java all objects are reactive and autonomous activity is expressed through threads [**?**]. The acceptance of messages is *implicit rather than* explicit as in Ada and POOL; in those languages each object has a *body* that controls entry into the object through a *rendezvous*. The communication between objects is through *synchronous method calls*, as in Ada, POOL, and Java, rather than through *message*

queues as in Actors.

As in ABC Pascal every object can potentially be active, there can be a many active objects. An implementation of similar language by translation to the Java Virtual Machine is reported in [**?**]. There, a fixed number of (pre-empted) *worker threads* repeatedly look for enabled actions in objects. While this limits the number of threads and keeps the memory overhead small, the situation could arise in which all worker threads get blocked and new threads would need to be created. While this is unlikely to occur in practice, the possibility of a false deadlock makes this scheme unattractive.

The recent Native POSIX Thread Library is claimed to support hundred thousands of threads [**?**]. The implementation of ABC Pascal was started with the goal of exploring the possibility of having one kernel thread per active object and delegating all scheduling to the operating system. To make a meaningful assessment, several classical concurrency examples were implemented in ABC Pascal, C/Pthreads, Java, Ada and the performance compared. Ada was chosen as its concurrency constructs are more similar to ABC Pascal than C/Pthreads and Java.

The next section gives further examples and elaborates on methodology. Section 3 sketches the implementation. Section 4 discusses the performance.

2 ABC Pascal

Program structure. A program in ABC Pascal starts with the program name, followed by *constant*, *type*, *class*, *variable*, *procedure* declarations, and the *main body* in that order, except that class and variable declarations may alternate. All except class declarations are as in standard Pascal. A class consists of variables, procedures, actions, and an initializing statement. Class variables and class procedures are called *fields* and *methods*. All fields are private, i.e. can be accessed only within the actions and methods of the object. All methods are public, i.e. can be called from other procedures or the main program. (The only use of action names would be for overriding in subclasses, which is currently not supported.) The scope of a constant, type, variable, and class starts from the point of the declaration. Procedures can only be called from the main body and thus are never executed concurrently. Procedures can be recursive but methods not. Variables can be of class type, i.e. *objects*, of primitive type, or of a structured type. Primitive types are *boolean* and *integer* and structured types are records and arrays. The initialization of an object is executed before any method is called or action selected. Actions and methods can have guards, global procedures cannot. If a

guard is left out it is assumed to be *true*. A program terminates when the main body has terminated, all actions that were initiated have terminated, and all actions are disabled.

Examples. We illustrate the language by some classical examples. For the reader-writer problem, we assume that access to the database by readers and writers is co-ordinated through an arbiter *rwa* which ensures that only a single writer or multiple readers can access the data. A reader would execute the sequence *rwa.start_read* ; *read access* ; *rwa.end_read* and a writer *rwa.start_write* ; *write access* ; *rwa.end_write*.

```
class RW_arbiter
  var rw: integer; {−1: one writer; 0: idle; > 0: #readers}
  procedure start_read when rw ≥ 0;
    begin rw := rw + 1 end;
  procedure end_read;
    begin rw := rw − 1 end;
  procedure start_write when rw = 0;
    begin rw := −1 end;
  procedure end_write;
    begin rw := 0 end;
  begin rw := 0 end;

var rwa: RW_arbiter;
```

For the dining philosopher problem we have forks, philosophers, and a butler as objects. When philosophers start to compete for the forks, the butler acts as the host and ensures that one less philosopher is seated then there are seats.

```
program DP;
  const ROUNDS = 100000; SEATS = 5;

  class Fork
    var up: boolean;
    procedure pickup when not up;
      begin up := true end;
    procedure putdown;
      begin up := false end;
    begin up := false end;
```

```
class Host
  var occupants: integer;
  procedure enter_sitdown when occupants < SEATS − 1;
    begin occupants := occupants + 1 end;
  procedure getup_leave;
    begin occupants := occupants − 1 end;
  begin occupants := 0 end;

var butler: Host; F: array [0 .. SEATS − 1] of Fork;

class Philosopher
  var seat: integer; awake: boolean;
  procedure wakeup(s: integer) ;
    begin seat := s; awake := true end;
  action start when awake;
    var r: integer;
    begin r := 0;
      while r < ROUNDS do
        begin
          butler.enter_sitdown;
          F[(seat + 1) mod SEATS].pickup;
          F[seat].pickup;
          F[(seat + 1) mod SEATS].putdown;
          F[seat].putdown;
          butler.getup_leave;
          r := r + 1;
        end;
      awake := false
    end;
  begin awake := false end;

var P: array [0 .. SEATS − 1] of Philosopher;
  s: integer;

begin s := 0;
  while s < SEATS do
    begin P[s].wakeup(s); s := s + 1 end
```

end.

The third example is a priority queue that allows integers to be added and the least stored integer to be retrieved. This can be implemented by keeping a sorted sequence such that the minimum is always at the end and can be retrieved in constant time. However, inserting in a sorted sequence would take linear time. The implementation below allows quick insertion as well and sorts in the "background". Retrieving the minimum may need to be delayed until the complete sequence is sorted again.

```
const CAPACITY = 10;
class PriorityQueue
  var a: array [0 .. CAPACITY − 1] of integer;
    n: integer; {total count}
    m: integer; {index to not sorted additions}
    s: integer; {sorting position}
  procedure add (u: integer);
    begin a[n] := u; n := n + 1 end;
  procedure min (var u: integer) when s = n;
    begin n := n − 1; u := a[n]; m := n; s := n end;
  action swap when s < n;
    var h: integer;
    begin
      if (s > 0) and (a[s − 1] < a[s]) then
        begin h := a[s − 1]; a[s − 1] := a[s]; a[s] := h;
          s := s − 1
        end
      else begin m := m + 1; s := m end
    end;
  begin n := 0; m := 0; s := 0
  end PriorityQueue;
```

Grammar and Restrictions. The concrete grammar is as follows.

selector ::= { "." *ident* | "[" *Expression* "]" }
factor ::= *ident selector* | *integer* | "(" *Expression* ")" | **not** *factor*
term ::= *factor* { ("*" | **div** | **mod** | **and**) *factor*}
SimpleExpression ::= ["+" | "−"] *term* {("+" | "−" | **or**) *term*}

Expression ::= *SimpleExpression* [(” = ” | ” ≠ ” | ” < ” | ” ≤ ” |
” > ” | ” ≥ ”) *SimpleExpression*]

assignment ::= *ident selector* ” := ” *Expression*
ActualParameters ::= ”(” [*Expression* {”,” *Expression*}] ”)”
ProcedureCall ::= *ident selector* [*ActualParameters*]
CompoundStatement ::= **begin** *statement* { ”;” *statement* } **end**
IfStatement ::= **if** *Expression* **then** *Statement* [**else** *Statement*]
WhileStatement ::= **while** *Expression* **do** *Statement*
Statement ::= [*assignment* | *ProcedureCall* |
CompoundStatement | *IfStatement* | *WhileStatement*]
IdentList ::= *ident* { ”,” *ident* }
ArrayType ::= **array** ”[” *Expression* ”..” *Expression* ”]” **of** *type*
FieldList ::= [*IdentList* ” : ” *type*]
RecordType ::= **record** *FieldList* { ”;” *FieldList* } **end**
type ::= *ident* | *ArrayType* | *RecordType*
FPSection ::= [**var**] *IdentList* ” : ” *type*
FormalParameters ::= ”(” [*FPSection* { ”;” *FPSection* }] ”)”
guard ::= [**when** *Expression*]
ProcedureDeclaration ::= **procedure** *ident* [*FormalParameters*]
guard ”;” *declarations CompoundStatement*
ActionDeclaration ::= **action** *ident guard* ”;”
declarations CompoundStatement
ClassDeclaration ::= **class** *ident* [**var** { *IdentList* ” : ” *type* ”;” }]
{ *ProcedureDeclaration* ”;” } { *ActionDeclaration* ”;” }
CompoundStatement ”;”
declarations ::= [**const** { *ident* ” = ” *Expression* ”;” }]
[**type** { *ident* ” = ” *type* ”;” }]
{ *ClassDeclaration* ”;” | **var** { *IdentList* ” : ” *type* ”;” } }
{ *ProcedureDeclaration* ”;” }

program ::= **program** *ident* ”;” *declarations CompoundStatement*

Predefined types are the primitive types *boolean* and *integer*. Predefined procedures include *write*. Methods can access a global variable only if the variable is an object. Procedures can access all kinds of global variables, objects, primitive, and structured. Procedures can call methods but methods cannot call procedures. These rules guarantee that any data that may be accessed concurrently is encapsu-

lated in an object.

Reference parameters, indicated by **var**, can be of arbitrary type, value parameters must be of a primitive type. Action guards can only access the fields of the object and method guards can only access the fields of the object and value parameters of the method. That is, guards must not contain calls to other objects. These syntactic restrictions are the key for an efficient implementation: the value of a method or action guard can only be affected by method and actions of the same object. Hence guards or procedures that were false at the time of the call and guards of all actions only need to be re-evaluated at the end of an action or method.

Verification Let b, p, q, r be boolean expressions and S, T be statements. We let $\{p\}S\{q\}$ stand for the partial correctness assertion: if P holds initially and S terminates, Q will hold finally. The language does not allow for expressions with side-effects. For simplicity, we assume that all expressions evaluate without error. We include **skip** as the empty statement explicitly. The common rules hold:

$$\begin{array}{rcl}
(p \Rightarrow q) & \Rightarrow & \{p\}\,\textbf{skip}\,\{q\} \\
(p \Rightarrow q[x\backslash e]) & \Rightarrow & \{p\}\,x := e\,\{q\} \\
\{p \wedge b\}\,S\,\{q\} & \Rightarrow & \{p\}\,\textbf{when}\,b; S\,\{q\} \\
\{p\}\,S\,\{q\} \wedge \{q\}\,T\,\{r\} & \Rightarrow & \{p\}\,S;T\,\{r\} \\
\{p \wedge b\}\,S\,\{q\} \wedge (p \wedge \neg b \Rightarrow q) & \Rightarrow & \{p\}\,\textbf{if}\,b\,\textbf{then}\,S\,\{q\} \\
\{p \wedge b\}\,S\,\{q\} \wedge \{p \wedge \neg b\}\,T\,\{q\} & \Rightarrow & \{p\}\,\textbf{if}\,b\,\textbf{then}\,S\,\textbf{else}\,T\,\{q\} \\
\{p \wedge b\}\,S\,\{p\} \wedge (p \wedge \neg b \Rightarrow q) & \Rightarrow & \{p\}\,\textbf{while}\,b\,\textbf{do}\,S\,\{q\}
\end{array}$$

The rules can be extended with those for assignments to array elements and record fields and for procedure calls in a standard way. We leave out the details. The notation $C.init$, $C.meth$, $C.act$ refers to initialization, the method *meth* and action *act* of class C.

Definition 1 (Class Invariant) *Let C be a class and p a boolean expression over the fields of C. Then p is invariant of C if following conditions hold:*

(a) Initialization: *The initialization establishes the invariant:*

$$\{true\}\,C.init\,\{p\}$$

(b) Methods: *Every method meth preserves the invariant:*

$$\{p\}\, C.meth\, \{p\}$$

(c) Actions: *Every action act preserves the invariant:*

$$\{p\}\, C.act\, \{p\}$$

This extends the common definition of a class invariant for sequential programs by adding condition (c) for actions. We argue that the definition is sound. As methods and actions are executed atomically, there is no possibility for interference, hence sequential reasoning is sufficient. The language does not allow for recursive method calls and for re-entrant calls, i.e. calls that go to another object and back, hence the class invariant does not have to be established before a call to another object. For example, an invariant of class *PriorityQueue* is $s \leq m \leq n$. Note that this definition requires that the invariant is only over the fields of one object. This is not suitable for proving a *global* invariant, for example that in program *DP* at most $SEATS - 1$ forks are in state *up*.

Refinement Concurrency can either be part of the requirements and present from the beginning of the development, like for an aquarium, or be an implementation decision. Consider the classes *Doubler* and *DelayedDoubler*:

```
class Doubler
  var x: integer;
  procedure store (u: integer);
    begin x := 2 × u end;
  procedure retrieve (var u: integer);
    begin u := x end;
  begin end;

class DelayedDoubler
  var y: integer; d: boolean;
  procedure store (u: integer);
    begin y := u; d := false end;
  procedure retrieve (var u: integer) when d;
    begin u := y end;
  action double when not d;
    begin y := 2 × y; d := true end;
  begin d := true end;
```

Both classes allow an integer to be stored and its double to be retrieved. In *Doubler* the operation of doubling is performed when the number is stored. In *DelayedDoubler* instead a "background" action is enabled that perform the doubling, allow a call to *store* to return quicker. The *retrieve* method needs to be suspended until the doubling occurs, which is controlled by the additional variable d. This is a universal pattern. For example, when data is written to a file, the programmer is given the illusion that this happens instantly, but typically the data is stored in a cache and a background action is enabled. The same holds for database operations. In each case, the goal is to increase the responsiveness and utilize the resources better. We say that *DelayedDoubler* is a refinement of *Doubler*, written as *Doubler* $\sqsubseteq$ *DelayedDouber*; every observation of *DelayedDoubler* is also possible with *Doubler*. Refinement may change the variables of a class and may introduce actions. Class refinement is defined in [**?**], based on the theory of data refinement of remote procedures of [**?**]. Class refinement is proved using a refinement relation, expressed as a predicate. We can take:

$$R \equiv (d \wedge x = y) \vee (\neg d \wedge x = 2 \times y)$$

We present the proof conditions as a checklist without going into the formalism. Classes *Doubler* and *DelayedDoubler* are abbreviated by *D* and *DD*.

(a) Initialization Refinement: D.init has to be refined by *DD.init* through *R*.

(b) Method Refinement: D.store has to be refined by *D.store* and *D.retrieve* has to be refined by *D.retrieve* through *R*.

(c) Method Enabledness: Whenever *D.store* is enabled, either *DD.store* or *DD.double* has to be enabled.

(d) Action Refinement: DD.double refines **skip** through *R*.

(e) Action Termination: DD.double eventually disables itself.

Class refinement has been applied to the development of relaxed balanced AVL trees from a sequential specification [**?**]. These are trees that allow a temporary imbalance for the purpose of quicker insertion.

3 Implementation

The ABC Pascal compiler is a derivative of a compiler for Pascal that was developed for teaching purposes [**?**]. It is a recursive-descent one-pass compiler that

generates IA-32 assembly language and uses stack-based code generation without optimization of register allocation. The generated code makes calls to the pthreads library.

We explain the translation scheme for classes. Every class corresponds to a record type with all the fields of the class and two additional fields, a *mutex* and a *condition variable*. The initialization corresponds to a global procedure that takes a reference to the record as a parameter, initializes the mutex and condition variable, and executes the initialization body. Every method corresponds to a procedure that takes a reference to the object record as an additional parameter. Every method locks the object record at entry and unlocks it at exit. If a method has a guard, it is evaluated at entry after obtaining the lock. If the guard it true, execution continues, otherwise the condition variable is used to suspend execution. Before exiting a method, the condition variable is used to signal all possibly blocked callers to re-evaluate the guard. Consider following class without actions:

class *C*
  var *f*: *F*;
  procedure *m*(*g*: *G*) **when** *p*;
    begin *S* **end**;
  begin *T* **end**;

This corresponds to:

type *C_rec* = **record** *mtx*: *Mutex*; *cv*: *Cond*; *f*: *F* **end**
procedure *C_m*(**var** *c*: *C_rec*; *g*: *G*);
  begin
    lock(*c.mtx*); **while not** *pp* **do** *cond_wait*(*c.cv*, *c.mtx*);
    SS;
    cond_broadcast(*c.cv*, *c.mtx*); *unlock*(*c.mtx*)
  end;
procedure *C_init*(**var** *c*: *C_rec*);
  begin
    mutex_init(*c.mtx*); *cond_init*(*c.cv*);
    TT
  end

Here pp, SS, TT stand for p, S, T with ever occurrence of f replaced by $c.f$. A method call $c.m(e)$ of object c of class C corresponds to $C\_m(c,e)$.

Now consider that class C has also actions. Then the initialization creates an *object thread*, leading to one thread per active object. That thread repeatedly locks

the object record, evaluates the action guards, executes an enabled action if there is one and waits otherwise, and unlocks the object record.

```
class C
  ...
  action a when q;
    U;
  action b when r;
    V;
  begin T end;
```

In this corresponds to:

```
procedure C_thread(var c: C_rec);
  begin
    while true do
      begin
        lock(c.mtx);
        if qq then UU
        else if rr then VV
        else
          cond_wait(c.cv, c.mtx);                    (*)
        unlock(c.mtx)
      end
  end;
procedure C_init(var c: C_rec);
  begin
    mutex_init(c.mtx); cond_init(c.cv);
    TT;
    thread_create(C_thread, c)
  end
```

Here *thread_create*(*C_thread*, *c*) means that procedure *C_thread* is started with parameter *c* as a new thread. Again, qq, rr, UU, VV stand for q, r, U, V with every occurrence of f replaced by $c.f$.

This scheme of implementing actions has two deficiencies. First, as action guards are evaluated in a specific order, some are being preferred over others. Fairness of the selection can be achieved by keeping an index to the last executed action and starting guard evaluation from the next action in a cyclic fashion.

The second deficiency is that if the main body terminates, all object threads

would terminate as well. The main body cannot *join* all object threads, as they never terminate. Our implementation solves this by keeping in a global variable a count of the number of object threads that are waiting at (*) in *C_thread*. If main body has finished and that count reaches the number of created objects, the whole program terminates. For this, at the end of the main body a loop is added that in a specific interval checks if the count has reached the number of created objects. Before (*) that count is incremented and after (*) the count is decremented. Atomic IA-32 instructions are used for incrementing and decrementing.

In the NPTL implementation of the pthreads standard, a mutex occupies 24 bytes and a condition variable occupies 48 bytes. The index of the last executed action adds another four bytes. Thus every object occupies 76 bytes in addition to the size of the declared fields.

4 Performance

Four programs were implemented in each of ABC Pascal, C/Pthreads, Ada, and Java, where *N* refers to the number of active objects (threads) and *R* to the number of rounds. The number of rounds was selected such that the execution time would be in the seconds and any loading latency would be negligible.

RW The reader-writer problem, with *N* being the total number of readers and writers.

CC The car control problem. Cars come from either the north or south and have to cross a bridge. The bride allows car in one direction allow and has a maximal capacity. Here *N* is the total number of all cars.

DP The dining philosopher problem, with *N* being the number of philosophers.

MRA The multiple resource allocator problem. Several users compete for up to 4 resources out of 10 in a random manner. Here *N* is the number of users.

The measurements were obtained running Suse Linux 10.1 for i686 on Intel Centrino Duo at 1.0 GHz. The C programs were compiled with gcc 4.1.0, Ada programs with gnat 4.1.0 and Java with Sun's HotSpot Client VM of Java 1.5. The C programs use the same NPTL implementation of pthreads as ABC Pascal. All results are in seconds.

Car Control (CC)			
	$R = 5000$ $N = 600$	$R = 10000$ $N = 100$	$R = 500$ $N = 1000$
ABC Pascal	9.78	3.44	0.84
Java	3.60	1.27	0.92
Ada	46.75	5.87	8.77
C	3.58	1.02	0.24

Dining Philosophers (DP)		
	$R = 200000$ $N = 5$	$R = 100000$ $N = 5$
ABC Pascal	2.18	1.21
Java	4.41	2.09
Ada	5.44	2.80
C	1.42	0.72

Multiple Resource Allocator (MRA)			
	$R = 5000$ $N = 500$	$R = 5000$ $N = 100$	$R = 500$ $N = 500$
ABC Pascal	49.88	7.42	5.36
Java	14.60	2.95	1.69
Ada	44.54	8.91	4.70
C	16.60	2.13	1.95

Reader-Writer (RW)			
	$R = 5000$ $N = 600$	$R = 10000$ $N = 100$	$R = 500$ $N = 1000$
ABC Pascal	7.10	2.36	2.91
Java	3.65	1.28	0.93
Ada	16.19	5.32	3.81
C	4.58	1.23	0.57

These numbers represent the averages of 30 runs. The standard deviation was in most cases small enough that it is not reported. The largest deviation was for the Ada implementation of CC with $R = 5000, N = 600$. There the 95 percent confidence interval is $(42.2, 51.3)$. More extensive tables are given in [**?**].

5 Discussion

In most cases, ABC Pascal is as efficient as or more efficient than Ada and not as efficient as C or Java. Given that optimization was not attempted, this gives hope that further performance improvements are possible. For example, [**?**] report on success in eliminating sychronization in Java programs. On the other hand, even with the current compiler the performance is not so weak that the model of action-based concurrent objects needs to be dismissed. The four examples consist mainly of communication. Considering that in a practical application more computation is involved, the relative slowdown due to use of actions would be less. We plan to continue with improving our implementation.

A common issue with a large number of threads is the need for creating stacks with a fixed *stack size*. If the stack size is chosen too small, threads run out of stack space, if the stack size is too large, not enough objects can be created. Indeed, with the C/Pthreads programs we had to adjust the default stack size. In ABC Pascal this is not an issue, as recursive procedures can be called only form the main program. Actions can only call actions of previously declared objects, such that the maximal call depth can be statically determined and the smallest amount of stack space allocated. However, if this restriction is relaxed, the problem of the stack size needs to be addressed.

Acknowledgements. The name ABC Pascal was suggested by Daniel Zingaro.

A C, Ada, Java Sources of Dining Philosophers

C Version Using Monitors

```
#include <pthread.h>
#define TRUE 1
#define FALSE 0

#define ROUNDS 100000
#define SEATS 5

typedef struct {
  int up;
  pthread_mutex_t mutex;
```

```
  pthread_cond_t forkdown;
} Fork;

void pickup(Fork *f) {
  pthread_mutex_lock(&f->mutex);
  while (f->up)
    pthread_cond_wait(&f->forkdown, &f->mutex);
  f->up = TRUE;
  pthread_mutex_unlock(&f->mutex);
}

void putdown(Fork *f) {
  pthread_mutex_lock(&f->mutex);
  f->up = FALSE;
  pthread_mutex_unlock(&f->mutex);
  pthread_cond_signal(&f->forkdown);
}

void fork_init(Fork *f) {
  f->up = FALSE;
  pthread_mutex_init(&f->mutex, NULL);
  pthread_cond_init(&f->forkdown, NULL);
}

typedef struct {
  int occupants;
  pthread_mutex_t mutex;
  pthread_cond_t notfull;
} Host;

void enter_sitdown(Host *h) {
  pthread_mutex_lock(&h->mutex);
  while (h->occupants == SEATS - 1)
    pthread_cond_wait(&h->notfull, &h->mutex);
  h->occupants++;
  pthread_mutex_unlock(&h->mutex);
}
```

```
void getup_leave(Host *h) {
  pthread_mutex_lock(&h->mutex);
  h->occupants--;
  pthread_mutex_unlock(&h->mutex);
  pthread_cond_signal(&h->notfull);
}

void host_init(Host *h) {
  h->occupants = 0;
  pthread_mutex_init(&h->mutex, NULL);
  pthread_cond_init(&h->notfull, NULL);
}

Fork F[SEATS];
Host butler;

void *Philosopher(void *arg) {
  int seat = (int) arg;
  int r;
  for (r = 0; r < ROUNDS; r++) {
    enter_sitdown(&butler);
    pickup(&F[seat]);
    pickup(&F[(seat + 1) % SEATS]);
    putdown(&F[seat]);
    putdown(&F[(seat + 1) % SEATS]);
    getup_leave(&butler);
  }
}

int main() {
  pthread_t p[SEATS];
  pthread_attr_t attr;
  int stack_size;

  pthread_attr_init(&attr);
  pthread_attr_getstacksize(&attr, &stack_size);
  pthread_attr_setstacksize(&attr, stack_size/8);
```

```
  int s;
  for (s = 0; s < SEATS; s++) fork_init(&F[s]);
  host_init(&butler);
  for (s = 0; s < SEATS; s++)
    pthread_create(&p[s], &attr, Philosopher, (void *) s);
  for (s = 0; s < SEATS; s++)
    pthread_join(p[s], NULL);
}
```

Ada Version

```
procedure DP is
  ROUNDS: constant := 100000;
  SEATS: constant := 5;
  type Seat_Index is mod SEATS;

  task type Philosopher is
    entry start (s: Seat_Index);
  end Philosopher;

  protected type Fork is
    entry pickup;
    procedure putdown;
  private
    up: Boolean := False;
  end Fork;

  protected type Host is
    entry enter_sitdown;
    procedure getup_leave;
  private
    occupants: Natural := 0;
  end Host;

  P: array (Seat_Index) of Philosopher;
  F: array (Seat_Index) of Fork;
  butler: Host;
```

```
task body Philosopher is
   seat: Seat_Index;
begin
   accept start (s: Seat_Index) do
      seat := s;
   end;
   for round in 1 .. ROUNDS loop
      butler.enter_sitdown;
      F(seat + 1).pickup;
      F(seat).pickup;
      F(seat + 1).putdown;
      F(seat).putdown;
      Butler.getup_leave;
   end loop;
end Philosopher;

protected body Fork is
   entry pickup when not up is
   begin
      up := True;
   end pickup;
   procedure putdown is
   begin
      up := False;
   end putdown;
end Fork;

protected body Host is
   entry enter_sitdown when occupants < SEATS - 1 is
   begin
      occupants := occupants + 1;
   end enter_sitdown;
   procedure getup_leave is
   begin
      occupants := occupants - 1;
   end getup_leave;
end Host;
```

```
begin
   for s in Seat_Index loop
      P(s).Start(s);
   end loop;
end DP;
```

Java Version

```
class Fork {
   private boolean up = false;

   synchronized void pickup() {
      while (up) {
         try {wait();
         } catch (InterruptedException e) {}
      }
      up = true;
   }
   synchronized void putdown() {
      up = false;
      notifyAll();
   }
}

class Host {
   private int occupants = 0;

   synchronized void enter_sitdown() {
      while (occupants == DP.SEATS − 1) {
         try {wait();
         } catch (InterruptedException e) {}
      }
      occupants++;
   }
   synchronized void getup_leave() {
      occupants−−;
      notifyAll();
   }
```

```
}

class Philosopher extends Thread {
  private int seat;
  private Host butler;
  private Fork[] f;

  public void run() {
    for (int r = 0; r < DP.ROUNDS; r++) {
      butler.enter_sitdown();
      f[(seat + 1) % DP.SEATS].pickup();
      f[seat].pickup();
      f[(seat + 1) % DP.SEATS].putdown();
      f[seat].putdown();
      butler.getup_leave();
    }
  }
  Philosopher(int s, Host b, Fork[] f) {
    seat = s;
    butler = b;
    this.f = f;
  }
}

public class DP {
  static final int ROUNDS = 100000;
  static final int SEATS = 5;

  public static void main(String[] args) {
    Philosopher[] p = new Philosopher[SEATS];
    Host butler = new Host();
    Fork[] f = new Fork[SEATS];
    for (int s = 0; s < SEATS; s++) {
      f[s] = new Fork();
      p[s] = new Philosopher(s, butler, f);
    }
    for (int s = 0; s < SEATS; s++)
      p[s].start();
```

```
  }
}
```

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1986.

[2] Jonathan Aldrich, Emin Gün Sirer, Craig Chambers, and Susan J. Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*, 47:91–120, 2003.

[3] Pierre America. Pool-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, Computer Systems Series. MIT Press, Cambridge, MA, 1987.

[4] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, October/December 1989.

[5] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, May/June 1999.

[6] Ralph Back, Martin Büchi, and Emil Sekerinski. Action-based concurrency and synchronization for objects. In T. Rus and M. Bertran, editors, *Transformation-Based Reactive System Development, Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software*, Lecture Notes in Computer Science 1231, pages 248–262, Palma, Mallorca, Spain, 1997. Springer-Verlag.

[7] Ralph Back and Reino Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[8] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In Johan Jeuring, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pages 68–95, Marstrand, Sweden, 1998. Springer-Verlag.

[9] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. Developing object-based distributed systems. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *3rd IFIP International Conference on Formal Methods for Open*

Object-based Distributed Systems (FMOODS'99), pages 19–34. Kluwer, 1999.

[10] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[11] Martin Büchi and Emil Sekerinski. A foundation for refining concurrent objects. *Fundamenta Informaticae*, 44(1):25–61, 2000.

[12] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[13] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. Technical report, Red Hat, Inc., February 2005.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley, 3rd edition edition, 2005.

[15] Hannu-Matti Järvinen and Reino Kurki-Suonio. DisCo specification language: Marriage of action and objects. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 142–151, Arlington, Texas, May 1991. IEEE Computer Society Press.

[16] Dirk Kalp, Milind Tambe, Anoop Gupta, Charles Forgy, Allen Newell, Anurag Acharya, Brian Milnes, and Kathy Swedlow. Parallel OPS5 user's manual. Technical report, Department of Computer Science, Carnegie Mellon University, November 1988.

[17] Leslie Lamport. The temporal logic of actions. *ACM Transactions of Programming Languages and Systems*, 16(3):872–923, 1994.

[18] Xiao lei Cui. *An Experimental Implementation of Action-Based Concurrency*. M.sc. thesis, McMaster University, January 2009.

[19] Kevin Lou. *A Compiler for an Action-Based Object-Oriented Programming Language*. Master's thesis, McMaster University, 2004.

[20] Jayadev Misra. A simple, object-based view of multiprogramming. *Formal Methods in System Design*, 20(1):23–45, January 2002.

[21] Juha Plosila and Kaisa Sere. Action systems in pipelined processor design. In *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 156–166. IEEE Computer Society, April 1997.

[22] Daniel L. Rosenband and Arvind. Hardware synthesis from guarded atomic actions with performanc specifications. In *International Conference on Computer Aided Design*, pages 783–790, San Jose, CA, 2005.

[23] Emil Sekerinski. Verification and refinement with fine-grained action-based concurrent objects. *Theoretical Computer Science*, 331(2–3):429–455, February 2005.

[24] Emil Sekerinski and Daniel Zingaro. Pascal0 compiler. Technical report, McMaster University, 2007.

[25] Kaisa Sere and Marina Waldén. Data refinement of remote procedures. *Formal Aspects of Computing*, 12(4):278–297, December 2000.

[26] J. Staunstrup and M. R. Greenstreet. From high-level descriptions to VLSI circuits. *BIT*, 28(3):620–638, September 1988.

[27] Jie Yan. *Concurrent Object Refinement and its Application to the Design and Implementation of Relaxed Balanced AVL Trees*. M.sc. thesis, McMaster University, April 2004.

[28] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *ACM Conference on Object Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, Vol 21, No 11, pages 258–268, 1986.