A Parallel Program That Generates the Möbius Sequence

by

Tom Verhoeff

88/01

January 1988

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science of Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven
The Netherlands

editor: F.A.J. van Neerven

A Parallel Program That Generates the Möbius Sequence

TOM VERHOEFF

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-Mail address: mcvax.UUCP!eutrc3!wstomv

January 1988

ABSTRACT

A CSP-like parallel program that generates the Möbius sequence is derived from its specification. The program has constant response time. It can be generalized to other sequences based on arithmetical functions, like the Euler function.

0. INTRODUCTION

We start by defining the Möbius sequence and specifying, in the style of [0], a computation that generates this sequence. In the major section of this paper we derive a parallel program from that specification. We analyze the response time of the resulting program. We also indicate how the program can be generalized to generate other sequences. Finally, we summarize the design techniques that were applied.

For integer n, $n \geqslant 1$, let $\pi(n)$ denote the number of (distinct) prime divisors of n. Since 1 is not considered prime, we have $\pi(1)=0$. For $n \geqslant 2$ we have $\pi(n) \geqslant 1$; for example: $\pi(2)=1$, $\pi(4)=1$, and $\pi(6)=2$. The *Möbius function* μ is defined for positive integers by

$$\mu(n)= \begin{cases} 0 & \text{if } (\mathbf{E}\, m : m>1 : m^2 \mid n) \\ (-1)^{\pi(n)} & \text{otherwise.} \end{cases}$$

where $m^2 | n$ means "m^2 is a positive divisor of n". For instance, we have $\mu(1) = 1$, $\mu(2) = -1$, $\mu(4) = 0$, and $\mu(6) = 1$. The sequence $\mu(n : n \geqslant 1)$ is called the *Möbius sequence*.

We now give a specification for a program that generates the Möbius sequence. In the next section we shall derive a parallel program satisfying this specification.

The program *MobSeq* has one *external communication port*: an integer *output* port b. The *communication behavior* of the program *MobSeq* is specified by the regular expression

$$b^*.$$

That is, an unbounded sequence of communications along port b is possible. The value of the i-th communication ($i \geqslant 0$) along port b is denoted by $b(i)$. The *input-output relation* (or i/o-relation for short) of the program *MobSeq* is specified by the equation

$$b(i) = \mu(i+1) \qquad \text{for } i \geqslant 0.$$

The following trivial "solution" gives an idea of what our program texts look like.

```
com TrivMobSeq(b !int) :
    [[x : int
    ;x := 1 ;(b !μ(x) ;x := x+1)*
    ]]
moc
```

We aim at a program that has *constant response time* under the assumption that integer addition and comparison are unit-time operations. Roughly speaking, this means that there is a fixed amount of time between successive external communications. The amount of computation required to determine $\mu(n)$, however, increases with n. Our program, therefore, will activate more and more processes and distribute the computation among them. Constant response time is achieved because the processes work harmoniously in parallel. This cooperation resembles that of a *systolic array*.

1. DERIVATION

In this section we derive a parallel program from the above specification. The derivation goes through a number of refinement steps that isolate design decisions. In the concluding section we summarize, in more general terms, the design techniques that we applied. We start our derivation by recalling from Number Theory that for $n \geq 1$

$$(\mathbf{S}\, d : d \mid n : \mu(d)) = U(n),$$

where $U(1) = 1$ and $U(n) = 0$ for $n > 1$ (see Appendix for a proof). Since $n \mid n$ we now can write a recurrent relation for μ:

$$\mu(n) = U(n) - (\mathbf{S}\, d : d < n \wedge d \mid n : \mu(d)).$$

Computing $U(n)$ is simple; it will be done in a subprocess, which is designed at the end of this section. The computation of the quantified sum is delegated to another subprocess, which will be our main concern in the rest of the derivation. Aiming at a program with constant response time, this subprocess should not do the entire summation sequentially, since the domain of the quantified sum increases with n. Therefore, we introduce a sequence of processes M_j, $j \geq 1$, where M_j has a subprocess of type M_{j+1}. We still have the freedom to specify processes M_j. For that purpose we generalize the quantified sum by replacing the first occurrence of the variable n by a new variable m:

$$G(m,n) = (\mathbf{S}\, d : d < m \wedge d \mid n : \mu(d)) \quad \text{for } 1 \leq m \leq n.$$

Hence, we have

$$\mu(n) = U(n) - G(n,n) \quad \text{for } n \geq 1,$$

$$G(1,n) = 0 \quad \text{for } n \geq 1,$$

and

$$G(m+1,n) = G(m,n) + \mathbf{if}\ m \mid n\ \mathbf{then}\ \mu(m)\ \mathbf{else}\ 0\ \mathbf{fi} \quad \text{for } 1 \leq m < n.$$

Process M_j, $j \geq 1$, is specified as follows. M_j has two external communication ports: an integer output port c and an integer input port d. Its communication behavior is

$$(c ; d)^*,$$

that is, communications along ports c and d alternate, starting along c. The i/o-relation for M_j is ($i \geqslant 0$)

$$c(i) = G(i+1, i+j),$$

$$d(i) = \mu(i+1).$$

When fed with the values of μ, process M_j will produce the indicated partial sums; M_1 produces the desired values $G(n, n)$ for $n \geqslant 1$.

We shall now derive the program for M_j, $j \geqslant 1$. Let p be its subprocess of type M_{j+1}. M_j has two *internal* communication ports to its subprocess p: one input port and one output port, denoted by $p.c$ and $p.d$ respectively. This means that values coming from $p.c$ can be used, but also that the proper values for $p.d$ must be supplied—all in accordance with p's specification, of course. We first deal with the external output c, distinguishing the first and succeeding occurrences:

$c(0)$

= { i/o-relation of M_j }

$G(1, j)$

= { property of G }

0,

and for $i \geqslant 1$

$c(i)$

= { i/o-relation of M_j }

$G(i+1, i+j)$

= { property of G }

$G(i, i+j) +$ **if** $i \mid (i+j)$ **then** $\mu(i)$ **else** 0 **fi**

= { property of divisibility and i/o-relation of M_j }

$G(i, i+j) +$ **if** $i \mid j$ **then** $d(i-1)$ **else** 0 **fi**

= { $p.c$ satisfies i/o-relation of M_{j+1}, hence, $p.c(i) = G(i+1, i+j+1)$ }

$p.c(i-1) +$ **if** $i \mid j$ **then** $d(i-1)$ **else** 0 **fi**.

For the internal output $p.d$ we have ($i \geqslant 0$):

$$p.d(i)$$

$= \quad \{ p.d$ satisfies i/o-relation of $M_{j+1} \}$

$$\mu(i+1)$$

$= \quad \{$ i/o-relation of $M_j \}$

$$d(i).$$

Summarizing these results we now have

$$c(0) = 0,$$

$$c(i) = p.c(i-1) + \textbf{if } i \mid j \textbf{ then } d(i-1) \textbf{ else } 0 \textbf{ fi} \qquad \text{for } i \geqslant 1, \text{ and}$$

$$p.d(i) = d(i) \qquad \text{for } i \geqslant 0.$$

Taking into account the desired communication behaviors of M_j and p, we thus get as program text for M_j:

```
com M_j (c !int, d ?int) :
    sub p : M_j+1 bus
    [[i , x , y : int
    ;i := 0 ;c!0
    ;(d ?x ;p.c ?y
     ;p.d !x ;i := i+1 ;c!if i | j then y+x else y fi
     )*
    ]]
moc
```

Restricted to the ports c and d this program exhibits the communication behavior required of M_j, and restricted to the ports $p.c$ and $p.d$ it adheres to M_{j+1}'s communication behavior. Notice that the communication actions in the program are ordered more restrictively than necessary: for example, $d\,?x$ and $p.c\,?y$ could be done concurrently without violating any of the specifications.

There are, however, two problems with the above program for M_j. For one thing the computation refers to j and therefore the program is not a recursive program in the usual sense. This could be remedied by distributing the value of j as part of the computation (add local variable j, j : int, and initial communications $d\,?j\,;p.d\,!(j+1)$; of course, this derives from a properly changed specification for M_j). But this phenomenon also disappears when dealing with the second problem.

The second problem is that computing $i \mid j$ is not a unit-time operation. Defining a **mod** b by,

$$(\mathbf{E}\, q : : a = qb + a \ \mathbf{mod}\ b) \wedge 0 \leq a \ \mathbf{mod}\ b < b,$$

we observe that

$$i \mid j \equiv j \ \mathbf{mod}\ i = 0.$$

M_j's subprocess M_{j+1} is therefore interested in $(j+1)\,\mathbf{mod}\ i$, which is easily computed from $j\ \mathbf{mod}\ i$. Working with the less obvious but as useful value of $(-j)\,\mathbf{mod}\ i$ turns out to give a slightly more compact program. Hence, we introduce another external input port e (and internal output port $p.e$) to distribute the values of $(-j)\,\mathbf{mod}\ i$. Furthermore, to eliminate the local computation for variable i we introduce external input port f that distributes i.

The adapted specification for M_j is as follows. M_j, $j \geq 1$, has four external communication ports: integer output port c and integer input ports d, e, and f. Its communication behavior is given by the extended regular expression

$$(c ; d, e, f)^*,$$

where the comma indicates arbitrary interleaving of the communications along ports d, e, and f (expressing the possibility of concurrency). The i/o-relation is given by the equations

$$c(i) = G(i+1, i+j),$$

$$d(i) = \mu(i+1),$$

$$e(i) = (-j)\,\mathbf{mod}\ (i+1), \text{ and}$$

$$f(i) = i+1,$$

for $i \geqslant 0$. We can now refine the previous program for M_j. Regarding the external output c we have for $i \geqslant 1$

$c(i)$

= { see above derivation }

$p.c(i-1) + \textbf{if } i \mid j \textbf{ then } d(i-1) \textbf{ else } 0 \textbf{ fi}$

= { property of divisibility }

$p.c(i-1) + \textbf{if } (-j) \textbf{ mod } i = 0 \textbf{ then } d(i-1) \textbf{ else } 0 \textbf{ fi}$

= { i/o-relation of M_j }

$p.c(i-1) + \textbf{if } e(i-1) = 0 \textbf{ then } d(i-1) \textbf{ else } 0 \textbf{ fi}$.

The internal output $p.d$ is computed as before. For the new internal output $p.e$ we derive for $i \geqslant 0$

$p.e(i)$

= { $p.e$ satisfies i/o-relation of M_{j+1} }

$(-j-1) \textbf{ mod } (i+1)$

= { property of **mod** }

$\textbf{if } (-j) \textbf{ mod } (i+1) = 0 \textbf{ then } i \textbf{ else } (-j) \textbf{ mod } (i+1) - 1 \textbf{ fi}$

= { i/o-relation of M_j }

$\textbf{if } e(i) = 0 \textbf{ then } f(i) - 1 \textbf{ else } e(i) - 1 \textbf{ fi}$

= { distribution }

$\textbf{if } e(i) = 0 \textbf{ then } f(i) \textbf{ else } e(i) \textbf{ fi} - 1$

For the new internal ouput $p.f$ we derive for $i \geqslant 0$

$p.f(i)$

= { $p.f$ satisfies i/o-relation of M_{j+1} }

$i+1$

= { i/o-relation of M_j }

$f(i)$

Summarizing these results we now have, for $i \geqslant 0$,

$$c(0) = 0,$$

$$c(i+1) = p.c(i) + \textbf{if } e(i) = 0 \textbf{ then } d(i) \textbf{ else } 0 \textbf{ fi},$$

$$p.d(i) = d(i),$$

$$p.e(i) = \textbf{if } e(i) = 0 \textbf{ then } f(i) \textbf{ else } e(i) \textbf{ fi} - 1, \text{ and}$$

$$p.f(i) = f(i).$$

Taking into account the communication behaviors of M_j and p, we thus get as program text for M_j:

```
com  Mj (c !int, d ?int, e ?int, f ?int) :
       sub  p : Mj+1  bus
       [[w , x , y , z  : int
       ;c !0
       ;(d ?w , e ?x , f ?y , p.c ?z
        ;if x = 0 then x , z := y , z +w  fi
        ;p.d !w , p.e !(x −1), p.f !y , c !z
        )*
       ]]
moc
```

The above program for M_j has as primitive operations only communication actions and integer comparison, addition, and subtraction. Notice also that the computation of M_j now no longer refers to j. Hence, the indices can be omitted (from M) and we have an ordinary recursive program. This program, therefore, satisfies for all $j \geqslant 1$ the specification of M_j (which does contain j). We are only interested in M_1, but to realize that specification we introduced the others.

Let us now deal with the simpler subprocess *USeq* of *MobSeq* that computes $U(n)$. We work from the following specification for *USeq*. *USeq* has one external integer output port

a, communication behavior a^*, and i/o-relation $a(i) = U(i+1)$ for $i \geqslant 0$. The program then directly derives from the definition of U:

com *USeq* $(a\,!\text{int}) : a\,!1 \;; (a\,!0)^*$ **moc**

The program for *MobSeq* is now a matter of combining *USeq* and M_1. Let q be the subprocess of type *USeq* and let r be the subprocess of type M_1. *MobSeq* must supply r with the proper input values in order to have it produce the sequence $G(n,n)$. Denoting the internal output ports to r by $r.d$, $r.e$, and $r.f$ the obligation of *MobSeq* is obtained by instantiating the corresponding i/o-relations of M_j with $j = 1$. For $i \geqslant 0$ this yields:

$r.d(i) = \mu(i+1)$,

$r.e(i) = (-1)\,\mathbf{mod}\,(i+1) = \{\text{ property of } \mathbf{mod} \}\; i$, and

$r.f(i) = i+1$.

For *MobSeq*'s external output b we have for $i \geqslant 0$:

$$b(i) = \mu(i+1) = U(i+1) - G(i+1, i+1) = q.a(i) - r.c(i).$$

Combining this knowledge with the required communication behaviors gives rise to the following program text for *MobSeq*:

```
com MobSeq (b !int) :
      sub q : USeq , r : M1 bus
      [[x , y , z : int
      ; x := 0
      ; (q.a ?y , r.c ?z ; y := y − z
        ; b !y , r.d !y , r.e !x , r.f !(x + 1) ; x := x + 1
        )*
      ]]
moc
```

2. RESPONSE TIME

The response time of the program for *MobSeq* is critically dependent only on the response time of the program for M_1. We analyze the response time of M_1 by giving a *sequence function* σ_j for M_j that indicates at what moments the communications could be scheduled, taking into account the ordering imposed by the program. The i-th communication along port c of M_j is scheduled at "time" $\sigma_j(c, i)$.

Since the communications along ports d, e, and f can all take place "at the same time", due to concurrency, we consider only ports c, d, $p.c$, and $p.d$. For these ports the program of M_j imposes the ordering expressed by

$$c;(d, p.c ; p.d, c)^*.$$

We therefore suggest the sequence function defined, for $j \geqslant 1$ and $i \geqslant 0$, by

$$\sigma_j(c, 0) = j-1,$$

$$\sigma_j(d, i) = \sigma_j(p.c, i) = 2i + j, \text{ and}$$

$$\sigma_j(p.d, i) = \sigma_j(c, i+1) = 2i + j + 1.$$

Because the communication actions along port $p.c$ of M_j coincide with those along port c of M_{j+1}, they must have been scheduled at the same time by σ (and similarly for ports $p.d$ and d). Thus we need to verify

$$\sigma_j(p.c, i) = 2i + j = \sigma_{j+1}(c, i) \text{ and}$$

$$\sigma_j(p.d, i) = 2i + j + 1 = \sigma_{j+1}(d, i)$$

in order for σ to be an admissable sequence function.

From this sequence function we can derive that M_1 produces $G(i+1, i+1)$ at moment $\sigma_1(c, i) = 2i$. Hence, the amount of time between external outputs is constant, that is, M_1 has constant response time. Furthermore, we see that M_j is activated at moment $\sigma_j(c, 0) = j-1$. Solving $2i = j-1$ for j, tells us that $2i+1$ subprocesses have been activated when M_1 does its i-th external output.

We should point out, however, that such a sequence function places only an upper bound on the response time complexity of the parallel program.

3. GENERALIZATION

Integer functions on the positive integers are *arithmetical functions*. The Möbius function is an example. For an introduction to the theory of arithmetical functions consult [1]. We treat only a very small part of it in this section.

The *(Dirichlet) convolution* of arithmetical functions f and g is defined by

$$(f*g)(n) = (\mathbf{S}\ k, m : km = n : f(k)g(m)) \qquad \text{for } n \geqslant 1.$$

The result is again an arithmetical function. Convolving is associative and symmetric. The function U, defined at the beginning of Section 1, is the unit: $f*U = f$.

If we define the arithmetical function E by $E(n) = 1$ for $n \geqslant 1$, then the Theorem of the Appendix can be succinctly expressed as $\mu*E = U$; that is, μ and E are each other's inverse under convolution. The derivation in Section 1 shows how to solve μ from $\mu*E = U$. It would equally apply to the problem of solving g from the equation $g*E = f$ for arbitrary given arithmetical function f. Since the solution of this equation is $f*\mu$ (convolve both sides with $E^{-1} = \mu$), we have a way of computing $f*\mu$. For example, the *Euler* function ϕ satisfies the equation $\phi*E = I$, where I is defined by $I(n) = n$ for $n \geqslant 1$.

A generalized specification for program *ConvMob* could be: integer input port a and integer output port b, communication behavior $(a;b)^*$, and i/o relation $(i \geqslant 0)$

$$a(i) = f(i+1) \text{ and}$$

$$b(i) = (f*\mu)(i+1).$$

A solution could be:

com *ConvMob* $(a\,?\text{int}, b\,!\text{int})$:

  sub $r : M_1$ **bus**

  $[\![x, y, z : \text{int} ; x := 0$

  $;(a\,?y, r.c\,?z\ ; y := y - z$

   $; b\,!y, r.d\,!y, r.e\,!x, r.f\,!(x+1) ; x := x + 1$

   $)^*$

  $]\!]$

moc

A nice challenge is finding a parallel program with constant response time that computes the Dirichlet convolution of two arbitrary arithmetical functions.

4. CONCLUSION

We would like to conclude by summarizing the design techniques that have made their appearance in our derivation. In hindsight they very much resemble techniques familiar from sequential programming and functional programming.

The first technique is the introduction of subprocesses to isolate concerns. We have no general heuristics to obtain the specifications of the subprocesses from those of the original process. A second technique is the introduction of an infinite nested chain of subprocesses. Their specification can often be obtained by generalizing the original specification, for example, by the introduction of a new variable. This resembles the way in which invariants are derived from the postcondition when designing a repetition for a sequential program. In order to define the infinite nested chain by a recursive program it is necessary to find a suitably parameterized specification. Finally, we have seen that the introduction of additional ports can be helpful to improve the efficiency of a program. This resembles the introduction of auxiliary variables and the strengthening of an invariant for a sequential repetition, or the introduction of additional parameters in a recursive function of a functional program.

Formal methods are important in the design of good programs. This is even more true for the design of parallel programs, because any operational approach is bound to confuse the designer; our mind cannot cope with the operational complexity of concurrency. Although we do not claim to have presented the ultimate tools for the design of parallel programs, we do think that our approach gives further insight in the requirements of a useful formalism.

ACKNOWLEDGMENTS

Rudolf Mak suggested the problem of writing a parallel program to generate the Möbius sequence from its recurrent relation. Martin Rem and other members of his VLSI club have critically examined earlier presentations of this material.

REFERENCES

[0] M. Rem, "Trace theory and systolic computations", **in** G. Goos and J. Hartmanis (eds.), PARLE: Parallel Architectures and Languages Europe, Proceedings 1987, Vol. I, *Lecture Notes in Computer Science 258*, Springer-Verlag, 1987, pp. 14-33.

[1] P.J. McCarthy, *Introduction to Arithmetical Functions*, Springer-Verlag, 1986, Ch. 1.

APPENDIX

The following three lemmas follow from the Fundamental Theorem of Arithmetic (unique prime factorization).

Lemma 0

$$d \mid n \Rightarrow \pi(d) \leqslant \pi(n)$$

Lemma 1

$\mu(d) \neq 0 \equiv d$ is the product of $\pi(d)$ distinct primes

Lemma 2

$$(\mathbf{S}\ d : d \mid n \ \wedge\ \mu(d) \neq 0 \ \wedge\ \pi(d) = i : 1) \ = \ (\pi(n) \ \mathbf{choose}\ i)$$

Theorem

$$(\mathbf{S}\ d : d \mid n : \mu(d)) = U(n)$$

Proof We derive

$(\mathbf{S}\ d : d \mid n : \mu(d))$

$=$ { algebra }

$(\mathbf{S}\ d : d \mid n \ \wedge\ \mu(d) \neq 0 : \mu(d))$

$=$ { term grouping according to $\pi(d)$, using Lemma 0 }

$(\mathbf{S}\ i : 0 \leqslant i \leqslant \pi(n) : (\mathbf{S}\ d : d \mid n \ \wedge\ \mu(d) \neq 0 \ \wedge\ \pi(d) = i : \mu(d)))$

$=$ { definition of μ }

$(\mathbf{S}\ i : 0 \leqslant i \leqslant \pi(n) : (\mathbf{S}\ d : d \mid n \ \wedge\ \mu(d) \neq 0 \ \wedge\ \pi(d) = i : (-1)^i))$

$=$ { Lemma 2 }

$(\mathbf{S}\ i : 0 \leqslant i \leqslant \pi(n) : (\pi(n) \ \mathbf{choose}\ i)(-1)^i)$

$=$ { Binomial Theorem }

$(1-1)^{\pi(n)}$

$=$ { definition of U }

$U(n)$

(End of Proof)

--

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films
85/04	T. Verhoeff H.M.J.L. Schols	Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate
86/01	R. Koymans	Specifying message passing and real-time systems
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specifications of information systems
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several parallel systems
86/05	Jan L.G. Dietz Kees M. van Hee	A framework for the conceptual modeling of discrete dynamic systems
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers

86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arum Kumar	Compositional semantics for real-time distributed computing (Inf. & Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responders commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of a sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J. Paredaens	A formal approach to distributed information systems
87/04	T. Verhoeff	Delay-insensitive codes - An overview
87/05	R. Kuiper	Enforcing non-determinism via linear time temporal logic specification

87/06	R. Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch)
87/07	R. Koymans	Specifying message passing and real-time systems with real-time temporal logic
87/08	H.M.J.L. Schols	The maximum number of states after projection
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics
87/10	T. Verhoeff	Three families of maximally nondeterministic automata
87/11	P. Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model
87/12	K.M. van Hee A. Lapinski	OR and AI approaches to decision support systems
87/13	J. van der Woude	Playing with patterns, searching for strings
87/14	J. Hooman	A compositional proof system for an occam-like real-time language
87/15	G. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas
87/17	K.M. van Hee G.J. Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples

87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces
87/19	A.J. Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen
87/20	G.J. Houben J. Paredaens	The R^2-Algebra: An extension of an algebra for nested relations
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples