A parallel program that generates the Möbius sequence

**Citation for published version (APA):**

**Document status and date:**
Published: 01/01/1988

**Document Version:**
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

**Please check the document version of this publication:**
- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

**Link to publication**

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

**Take down policy**
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
A Parallel Program That Generates
the Möbius Sequence
by
Tom Verhoeff

January 1988
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
All rights reserved
ter: 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 \( n \), \( n \geq 1 \), let \( \pi(n) \) denote the number of (distinct) prime divisors of \( n \). Since 1 is not considered prime, we have \( \pi(1) = 0 \). For \( n \geq 2 \) we have \( \pi(n) \geq 1 \); for example: \( \pi(2) = 1 \), \( \pi(4) = 1 \), and \( \pi(6) = 2 \). The Möbius function \( \mu \) is defined for positive integers by

\[
\mu(n) = \begin{cases} 
0 & \text{if } (\exists m : m > 1 : m^2 | n) \\
(-1)^{\pi(n)} & \text{otherwise.}
\end{cases}
\]
where \( m^2 \mid n \) 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 \( \mu(n : n \geq 1) \) 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 \( \text{MobSeq} \) has one external communication port: an integer output port \( b \). The communication behavior of the program \( \text{MobSeq} \) 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 \( (i \geq 0) \) along port \( b \) is denoted by \( b(i) \). The input-output relation (or i/o-relation for short) of the program \( \text{MobSeq} \) is specified by the equation

\[ b(i) = \mu(i + 1) \quad \text{for} \quad i \geq 0. \]

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

```plaintext
com TrivMobSeq(b hint):
  [x : int
   ;x := 1 ;(b !\mu(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.

*Parallel Program for Möbius Sequence*
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 \( n \geq 1 \)

\[
(S \ d : d \mid n : \mu(d)) = U(n),
\]

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 \( \mu \):

\[
\mu(n) = U(n) - (S \ d : d < n \land d \mid n : \mu(d)).
\]

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 \( m \):

\[
G(m,n) = (S \ d : d < m \land d \mid n : \mu(d)) \quad \text{for} \ 1 \leq m \leq n.
\]

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

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

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 \( e \) and \( d \) alternate, starting along \( c \). The i/o-relation for \( M_j \) is \( (i \geq 0) \)

\[
c(i) = G(i + 1, i + j),
\]

\[
d(i) = \mu(i + 1).
\]

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

We shall now derive the program for \( M_j, j \geq 1 \). 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 \( p.e \) 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:

\[
c(0) = \begin{cases} 
& \text{i/o-relation of } M_j \\
& G(1, j)
\end{cases}
\]

\[
0,
\]

and for \( i \geq 1 \)

\[
c(i) = \begin{cases} 
& \text{i/o-relation of } M_j \\
& G(i + 1, i + j)
\end{cases}
\]

\[
\begin{align*}
& G(i, i + j) + \text{if } i \mid (i + j) \text{ then } \mu(i) \text{ else } 0 \\
& G(i, i + j) + \text{if } i \mid j \text{ then } d(i - 1) \text{ else } 0
\end{align*}
\]

\[
\begin{cases} 
& p.c \text{ satisfies i/o-relation of } M_{j+1}, \text{ hence, } p.c(i) = G(i + 1, i + j + 1) \\
& p.c(i - 1) + \text{if } i \mid j \text{ then } d(i - 1) \text{ else } 0
\end{cases}
\]

\[
Parallel \ Program \ for \ Möbius \ Sequence
\]
For the internal output \( p.d \) we have \( (i \geq 0) \):

\[
p.d(i) = \begin{cases} 
\mu(i + 1) & \text{if } i \mid j \text{ then } d(i - 1) \text{ else } 0 \text{ fi } \text{ for } i \geq 1, \text{ and} \\
0 & \text{for } i \geq 0.
\end{cases}
\]

Summarizing these results we now have

\[
c(0) = 0, \\
c(i) = p.c(i - 1) + \begin{cases} 
0 & \text{if } i \mid j \text{ then } d(i - 1) \text{ else } 0 \text{ fi } \text{ for } i \geq 1, \text{ and} \\
0 & \text{for } i \geq 0.
\end{cases}
\]

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

\[
\text{com } M_j(c \text{ tint}, d \text{ ?int}):
\]

\[
\begin{align*}
\text{sub } p : M_{j+1} \text{ bus} \\
&[i : x, y : \text{int} \\
&i := 0 : c!0 \\
&(d ?x : p.c ?y \\
&: p.d !x : i := i + 1 : c!i \mid j \text{ then } y + x \text{ else } y \text{ fi} \\
&)^
\]
\]

\[
\text{moc}
\]

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 $j$, $j: \text{int}$, 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 $i \mod j$ is not a unit-time operation. Defining $a \mod b$ by

$$(\text{Eq} : a = q b + a \mod b) \land 0 \leq a \mod b < b,$$

we observe that

$$i \mod j = 0.$$

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

$$

f(i) = i + 1.
$$
for \(i \geq 0\). We can now refine the previous program for \(M_j\). Regarding the external output \(c\) we have for \(i \geq 1\)

\[
c(i) = \begin{cases} 
\text{see above derivation} & \text{if } i \mid j \text{ then } d(i-1) \text{ else } 0 \\
\text{property of divisibility} & \text{if } (-j) \mod i = 0 \text{ then } d(i-1) \text{ else } 0 \\
\text{i/o-relation of } M_j & \text{if } e(i-1) = 0 \text{ then } d(i-1) \text{ else } 0
\end{cases}
\]

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

\[
p.e(i) = \begin{cases} 
\text{\(p.e\) satisfies i/o-relation of } M_{j+1} & \\
(-j-1) \mod (i+1) & \text{mod } \\
\text{property of mod} & \text{if } (-j) \mod (i+1) = 0 \text{ then } i \text{ else } (-j) \mod (i+1)-1 \\
\text{i/o-relation of } M_j & \text{if } e(i) = 0 \text{ then } f(i)-1 \text{ else } e(i)-1 \\
\text{distribution} & \text{if } e(i) = 0 \text{ then } f(i) \text{ else } e(i) \\
-f(i-1)
\end{cases}
\]

For the new internal output \(p.f\) we derive for \(i \geq 0\)

\[
p.f(i) = \begin{cases} 
\text{\(p.f\) satisfies i/o-relation of } M_{j+1} & \\
i+1 & \text{i/o-relation of } M_j \\
f(i)
\end{cases}
\]

Parallel Program for M"obius Sequence
Summarizing these results we now have, for \( i \geq 0 \),

\[
\begin{align*}
    c(0) &= 0, \\
    c(i + 1) &= p.c(i) + \text{if } e(i) = 0 \text{ then } d(i) \text{ else } 0, \\
    p.d(i) &= d(i), \\
    p.e(i) &= \text{if } e(i) = 0 \text{ then } f(i) \text{ else } e(i) - 1, \text{ and} \\
    p.f(i) &= f(i).
\end{align*}
\]

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

\[
\text{com } M_j(c \text{ int} , d ? \text{ int} , e ? \text{ int} , f ? \text{ int}):
\]

\[
\begin{align*}
    \text{sub } p : M_{j+1} \text{ bus} \\
    [w , x , y , z : \text{ int}] \\
    :c!0 \\
    :(d ?w , e ?x , f ?y , p.c ?z \\
    :\text{if } x = 0 \text{ then } x , z := y , z + w \text{ fi} \\
    ;p.d !w , p.e !(x - 1) , p.f !y , c !z \\
    )^
\end{align*}
\]

\[
\text{moc}
\]

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 \( j \geq 1 \) 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 \(i \geq 0\). The program then directly derives from the definition of \(U\):

\[
\text{com } USeq(a \text{int}) : a!1 ; (a!10)^* \text{ moc}
\]

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 \(i \geq 0\) this yields:

\[
\begin{align*}
r.d(i) &= \mu(i + 1), \\
r.e(i) &= (-1) \mod (i + 1) = \{\text{property of mod}\} i, \text{ and} \\
r.f(i) &= i + 1.
\end{align*}
\]

For \(MobSeq\)'s external output \(b\) we have for \(i \geq 0\):

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

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

\[
\text{com } MobSeq(b \text{int}) : \\
\text{sub } q : USeq, r : M_1 \text{ bus} \\
{x,y,z : \text{int}} \\
:x := 0 \\
:q.a?y, r.c?z ; y := y - z \\
:b!y, r.d!y, r.e!x, r.f!x+1 ; x := x + 1 \\
)
\]

\text{moc}

Parallel Program for Möbius Sequence
2. RESPONSE TIME

The response time of the program for \textit{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 \( \sigma_j \) 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 \( j \geq 1 \) and \( i \geq 0 \), by

\[
\sigma_j(c,0) = j-1,
\]

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

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

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 \( \sigma \) (and similarly for ports \( p.d \) and \( d \)). Thus we need to verify

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

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

in order for \( \sigma \) to be an admissible 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

\[
(f * g)(n) = \left( \sum_{km = n} f(k)g(m) \right) \quad \text{for } n \geq 1.
\]

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 \( n \geq 1 \), then the Theorem of the Appendix can be succinctly expressed as \( \mu * E = U \); that is, \( \mu \) and \( E \) are each other’s inverse under convolution. The derivation in Section 1 shows how to solve \( \mu \) 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 \( \phi \) satisfies the equation \( \phi * E = I \), where \( I \) is defined by \( I(n) = n \) for \( n \geq 1 \).

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 \((i \geq 0)\)

\[
a(i) = f(i+1) \quad \text{and} \quad b(i) = (f * \mu)(i+1).
\]

A solution could be:

*Parallel Program for Möbius Sequence*
com ConvMob (a : int, b : int):

sub r : M₁ bus
[x, y, z : 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
)
]

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.

Parallel Program for Möbius Sequence
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


APPENDIX

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

Lemma 0

\[ d \mid n \Rightarrow \pi(d) \leq \pi(n) \]

Lemma 1

\[ \mu(d) \neq 0 \iff d \text{ is the product of } \pi(d) \text{ distinct primes} \]

Parallel Program for Möbius Sequence
Lemma 2

\((S_{d : d} \mid n \land \mu(d) \neq 0 \land \pi(d) = i : 1) = (\pi(n) \text{ choose } i)\)

Theorem

\((S_{d : d} \mid n : \mu(d)) = U(n)\)

Proof

We derive

\((S_{d : d} \mid n : \mu(d))\)

= \{ algebra \}

\((S_{d : d} \mid n \land \mu(d) \neq 0 : \mu(d))\)

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

\((S_{i : 0 \leq i \leq \pi(n)} : (S_{d : d} \mid n \land \mu(d) \neq 0 \land \pi(d) = i : \mu(d)))\)

= \{ definition of \( \mu \) \}

\((S_{i : 0 \leq i \leq \pi(n)} : (S_{d : d} \mid n \land \mu(d) \neq 0 \land \pi(d) = i : (-1)^i))\)

= \{ Lemma 2 \}

\((S_{i : 0 \leq i \leq \pi(n)} : (\pi(n) \text{ choose } i)(-1)^i)\)

= \{ Binomial Theorem \}

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

= \{ definition of \( U \) \}

\(U(n)\)

(End of Proof)
In this series appeared:

<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The formal specification and derivation of CMOS-circuits</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a computer for evaluation of flow films</td>
</tr>
<tr>
<td>85/04</td>
<td>T. Verhoeff, H.M.J.L. Schols</td>
<td>Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate</td>
</tr>
<tr>
<td>86/01</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems</td>
</tr>
<tr>
<td>86/02</td>
<td>G.A. Bussing, K.M. van Hee, M. Voorhoeve</td>
<td>ELISA, A language for formal specifications of information systems</td>
</tr>
<tr>
<td>86/03</td>
<td>Rob Hoogerwoord</td>
<td>Some reflections on the implementation of trace structures</td>
</tr>
<tr>
<td>86/04</td>
<td>G.J. Houben, J. Paredaens, K.M. van Hee</td>
<td>The partition of an information system in several parallel systems</td>
</tr>
<tr>
<td>86/05</td>
<td>Jan L.G. Dietz, Kees M. van Hee</td>
<td>A framework for the conceptual modeling of discrete dynamic systems</td>
</tr>
<tr>
<td>86/06</td>
<td>Tom Verhoeff</td>
<td>Nondeterminism and divergence created by concealment in CSP</td>
</tr>
<tr>
<td>86/07</td>
<td>R. Gerth, L. Shira</td>
<td>On proving communication closedness of distributed layers</td>
</tr>
<tr>
<td>Year</td>
<td>Author(s)</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>86/09</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>Full abstraction of a real-time denotational semantics for an OCCAM-like language</td>
</tr>
<tr>
<td>86/10</td>
<td>J. Hooman</td>
<td>A compositional proof theory for real-time distributed message passing</td>
</tr>
<tr>
<td>86/11</td>
<td>W.P. de Roever</td>
<td>Questions to Robin Milner - A responders commentary (IFIP86)</td>
</tr>
<tr>
<td>86/12</td>
<td>A. Boucher, R. Gerth</td>
<td>A timed failures model for extended communicating processes</td>
</tr>
<tr>
<td>86/14</td>
<td>R. Koymans</td>
<td>Specifying passing systems requires extending temporal logic</td>
</tr>
<tr>
<td>87/01</td>
<td>R. Gerth</td>
<td>On the existence of a sound and complete axiomatizations of the monitor concept</td>
</tr>
<tr>
<td>87/02</td>
<td>Simon J. Klaver, Chris F.M. Verberne</td>
<td>Federatieve Databases</td>
</tr>
<tr>
<td>87/03</td>
<td>G.J. Houben, J. Paredaens</td>
<td>A formal approach to distributed information systems</td>
</tr>
<tr>
<td>87/04</td>
<td>T. Verhoeff</td>
<td>Delay-insensitive codes - An overview</td>
</tr>
<tr>
<td>87/05</td>
<td>R. Kuiper</td>
<td>Enforcing non-determinism via linear time temporal logic specification</td>
</tr>
<tr>
<td>87/06</td>
<td>R. Koymans</td>
<td>Temporele logica specificatie van message passing en real-time systemen (in Dutch)</td>
</tr>
<tr>
<td>87/07</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems with real-time temporal logic</td>
</tr>
<tr>
<td>87/08</td>
<td>H.M.J.L. Schols</td>
<td>The maximum number of states after projection</td>
</tr>
<tr>
<td>87/10</td>
<td>T. Verhoeff</td>
<td>Three families of maximally nondeterministic automata</td>
</tr>
<tr>
<td>87/11</td>
<td>P. Lemmens</td>
<td>Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model</td>
</tr>
<tr>
<td>87/12</td>
<td>K.M. van Hee, A. Lapinski</td>
<td>OR and AI approaches to decision support systems</td>
</tr>
<tr>
<td>87/13</td>
<td>J. van der Woude</td>
<td>Playing with patterns, searching for strings</td>
</tr>
<tr>
<td>87/14</td>
<td>J. Hooman</td>
<td>A compositional proof system for an occam-like real-time language</td>
</tr>
<tr>
<td>87/15</td>
<td>G. Huizing, R. Gerth, W.P. de Roever</td>
<td>A compositional semantics for statecharts</td>
</tr>
<tr>
<td>87/16</td>
<td>H.M.M. ten Eikelder, J.C.F. Wilmont</td>
<td>Normal forms for a class of formulas</td>
</tr>
<tr>
<td>87/17</td>
<td>K.M. van Hee, G.J. Houben, J.L.G. Dietz</td>
<td>Modelling of discrete dynamic systems framework and examples</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>87/18</td>
<td>C.W.A.M. van Overveld</td>
<td>An integer algorithm for rendering curved surfaces</td>
</tr>
<tr>
<td>87/19</td>
<td>A.J. Seebregts</td>
<td>Optimalisering van file allocatie in gedistribueerde database systemen</td>
</tr>
<tr>
<td>87/20</td>
<td>G.J. Houben, J. Paredaens</td>
<td>The $R^2$-Algebra: An extension of an algebra for nested relations</td>
</tr>
<tr>
<td>87/21</td>
<td>R. Gerth, M. Codish, Y. Lichtenstein, E. Shapiro</td>
<td>Fully abstract denotational semantics for concurrent PROLOG</td>
</tr>
<tr>
<td>88/01</td>
<td>T. Verhoeff</td>
<td>A Parallel Program That Generates the Möbius Sequence</td>
</tr>
<tr>
<td>88/02</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable Specification for Information Systems</td>
</tr>
<tr>
<td>88/03</td>
<td>T. Verhoeff</td>
<td>Settling a Question about Pythagorean Triples</td>
</tr>
</tbody>
</table>