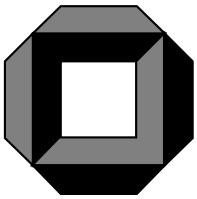
Data Flow Analysis of Parallel Programs

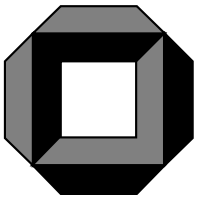
Jürgen Vollmer

Interner Bericht 19/95



Universität Karlsruhe
Fakultät für Informatik

Universität Karlsruhe
Fakultät für Informatik



Data Flow Analysis of Parallel Programs

Jürgen Vollmer
Institut für Programmstrukturen und Datenorganisation

Interner Bericht 19/95
24. März 1995

Abstract

Data flow analysis is the prerequisite of performing optimizations such as *common subexpression eliminations* or *code motion of partial redundant expressions* on imperative sequential programs. To apply these transformations to parallel imperative programs, the notion of data flow must be extended to concurrent programs. The additional source language features are: common address space (shared memory), nested parallel statements (`PAR`), or-parallelism, critical regions and message passing. The underlying interleaving semantics of the concurrently-executed processes result in the so-called *state space explosion* which on first appearance prevents the computation of the *meet over all path* solution needed for data flow analysis.

For the class of one-bit data flow problems (also known as bit-vector problems) we can show that for the computation of the *meet over all path* solution, not all interleavings are needed. Based on that, we can give simple data flow equations representing the data flow effects of the `PAR` statement. The definition of a *parallel control flow graph* leads to an efficient extension of Killdal's algorithm to compute the data flow of a concurrent program. The time complexity is the same as for analyzing a "comparable" sequential program.

Keywords: Data flow analysis, control flow parallelism, parallel languages.

This work has been funded by the ESPRIT project COMPARE, contract number 5399.

A short version of this report will be published in the proceedings of the *International Conference on Parallel Architectures and Compilation Techniques* PACT 95, [Vollmer 95].

Address: Universität Karlsruhe, IPD, Vincenz-Prießnitz-Straße 3, D-76128 Karlsruhe

Email: vollmer@ipd.info.uni-karlsruhe.de

Contents

1 Introduction and Motivation

The key tool for attacking the *Grand Challenge Problems* is parallelism. Parallel hardware is becoming more available and cheaper, but to program these devices is still a difficult task. Hence high-level programming languages are needed which have enough expressive power to implement parallel algorithms. But as usual with high-level languages, when translating them to machine code, some inefficiencies are introduced by the compiler. Therefore compilers have to perform so-called optimizations which improve the program. We may distinguish broadly between two kinds:

- transformations performed on the input language level such as *mapping of SIMD programs to MIMD machines*, *removal of unnecessary synchronization and communication*, *clustering of processes*, *data placement* etc. [Zima *et al* 90, Philippsen *et al* 91, Zimmermann *et al* 94] and
- transformations performed on the machine language level, such as *common subexpression elimination*, *constant folding*, *dead code elimination*, *code motion* etc. (for an introduction: [Aho *et al* 86]).

For the rest of this paper we have only these kinds of optimizing transformations in mind.
Optimizing a program requires analyzing it, and this is often done by solving data flow equations for the program. Traditional data flow analysis (DFA) methods [Kennedy 81] are designed for sequential programs. Hence they may fail when applied to the control flow of parallel programs as shown by [Midkiff *et al* 90]. We give another example showing the problems when "low level" optimizations such as instruction scheduling are performed on parallel programs.

1.1 Machine Model

We assume that other phases of the compiler have done the more "high-level" transformations already, and hence our investigation is based on an imperative language with explicit control flow parallelism, dynamic process creation, and shared memory. As a computing model we assume a MIMD (multiple instruction, multiple data) system, where each process is executed on a separate logical processor.[1] Each processor runs independently of each other and has its own set of registers, which are invisible for other processors. All processors access a shared memory. The access to a single memory cell is atomic, i.e. at a given time only one process may read or write a given cell. We assume an interleaving semantics for the execution of the program with respect to the memory accesses.

1.2 What Might Go Wrong: An Example

This section shows the potential problems, when applying sequential data flow analysis to an explicitly parallel program. The small program[2] given in Table 1 executes the processes P_1 and P_2 in parallel. It is intuitively clear that $critical_1$ and $critical_2$ are never executed at the same time.
On a first glance this program has a *race condition* [Netzer *et al* 92], (both processes read and write the shared variables, without some kind of explicit synchronization) but this race condition is an intended one: The variables `a` and `b` are used to implement a simple synchronization scheme. Extensive work has been done on analyzing parallel programs for potential races[3] but little work has been done to optimize them.
A simple-minded optimizer could perform the following "optimizations" (which would be correct in sequential contexts):

- Propagate `a = 0` and `b = 0` to `IF a = 0` and `IF b = 0` respectively.
- Then the expressions could be statically evaluated to `TRUE`.

[1] A *processor* may be implemented via a time sharing system.
[2] [Lamport 79] presents this problem concerning the design of parallel computers.
[3] for an overview see [Bristow *et al* 88]

a := 0; b := 0; PAR	
(P_1)	(P_2)
a := 1;	b := 1;
IF b = 0	IF a = 0
THEN $critical_1$;	THEN $critical_2$;
a := 0;	b := 0;
ELSE $else_1$	ELSE $else_2$
END	END
END	

Table 1: Part of process synchronization code

- Dead code elimination removes the `IF` and `ELSE` parts.

And as consequence, both, $critical_1$ and $critical_2$ would be executed.
But even without traditional optimizations performed by the compiler, things might go wrong when using an assembler which does instruction scheduling (reordering), to better use the processor's internal parallelism (i.e. the pipelined processing of instructions): The non-optimized code of process body P_1 on a typical RISC processor is given in Table 2. The instruction scheduler could now decide to reorder the instructions, e.g. to insert another instruction between a register load and an immediately following register use instruction (e.g. `ldc 1,r0; st r0,a`) which results in the code for P_1 shown in Table 3a. In this case, it may happen that $critical_1$ and $critical_2$ are both executed, as shown in Table 3b.

ldc	1, r0	Load constant 1 into register r0.
st	r0, a	Store the content of register r0 in memory at address a.
ld	b, r1	Load content of memory at address b into a register.
cmp	r1, 0	Compare a register with a constant, set condition code.
jeq	$then_1$	Conditional branch to $then_1$, if condition code *equal* set.
code of $else_1$		
...		

Table 2: Non-optimized code for Process P_1

Even worse, some processors, such as the *DEC Alpha* [DEC 92], are able to reorder the memory accesses to different addresses to some degree. Hence, even the unchanged code could give the wrong result. To avoid this situation, the *DEC Alpha* offers a *memory barrier* instruction, which delays the processor until all pending memory requests are fulfilled. In our example this instructions must follow every memory access, which results in a significant slow-down of the program.
On a system with distributed memory, the shared memory access may be implemented by calls to the operating system, which transports a value from the memory it is stored in to the destination where it is needed. If these calls are performed asynchronously (e.g. the memory fetch is separated into two calls: a non blocking *ask_for_value(address)* and a blocking *wait_for_value(address)*), the same problem arises.
[Lamport 79] offers a solution which is formalized by [Afek *et al* 93]. The memory access must fulfill the two conditions:

1. Each processor executes memory access request in the order specified by the program.
2. All accesses to a single memory cell are executed in a first-in-first-out queue.

ldc	1, r0
ld	b, r1
st	r0, a
cmp	r1, 0
jeq	$then_1$
...	

3a

time	t_1	t_2	t_3	t_4	...	
$Processor_1$:	ldc 1, r0;	ld b, r1; *(r1 = 0)*	st r0, a;	cmp r1, 0;	...	$critical_1$
$Processor_2$:	ldc 1, r0;	ld a, r1; *(r1 = 0)*	st r0, b;	cmp r1, 0;	...	$critical_2$

3b

Table 3: Code and execution of reordered code

It is obvious that these conditions are too restrictive, since optimizations of "real" independent memory accesses are forbidden.
The transformations shown above are based on information such as the *reaching definitions*[4] or *available expressions*[5]. The reason for the above problems is that they use the wrong information, i.e. the information was calculated in a "sequential context", not considering the parallelism expressed in the program.

1.3 The Problem

Data flow analysis is more or less the estimation of the effects caused by program statements. This estimation is based on two things: an abstraction of the information needed as prerequisite for the optimizing transformation, and the propagation of the information along the statements of the source program. The information is usually represented by the elements of a semi-lattice[6]. The effect of a single program statement is then a function over these semi-lattice values. One execution of a program (up to the point of consideration) represents an execution path. The propagation is modeled by applying these functions in the order given by the statements of such an execution path. Since we are looking for the "worst case information" (only this guarantees that the transformation is correct for all execution paths leading to this program point) we have to consider the *Meet Over all Paths* of these information. [Kildall 73] formalized this idea and gave an efficient algorithm to compute the data flow information for all points of a program.
The data flow information of two statement sequences, without any branches, executed concurrently, is given by the *meet* of the information of all interleavings[7] of the two sequences. It is clear that this may lead to a "state space explosion" [Chow *et al* 94], which makes it, on a very first view, intractable to compute the data flow information implied by a `PAR` statement.

1.4 Contribution of this Work

Hence we have to ask:

- which data flow information is valid after the `PAR` statement, and
- which information is valid before each statement in a process body?

[4] Which definition (assignment) of a variable is valid at the program point under consideration.
[5] Which computed values are valid at some point.
[6] E.g. boolean values for "the information is present or not".
[7] The topological sorting of the simple statements of both sequences.

The main contributions of our paper are:

- The lattice-theoretic based data flow framework is extended to cope with parallel programs. The proposed extension is valid only for the large class of *one-bit* (also known as *bit-vector*) data flow problems. They are based on a two-element semi-lattice.

- Simple bit-vector data flow equations are derived representing the data flow effects of the `PAR` statement.

- The Parallel Control Flow Graph is defined and used as a base for an extension of the well known and efficient iterative data flow analysis algorithm.

Based on these results, data flow analysis of parallel programs is possible and efficient. Then traditional optimizations may be applied to these programs without any restriction.

1.5 Related Work

Current approaches in analyzing the data flow of parallel programs have either a restricted model of shared memory, or even disallow it.

[Reif 84] investigates the data flow of communicating processes, but these do not share memory: processes communicate solely through synchronous channels. [Srinivasan *et al* 91a] describe an efficient method of computing the *Static Single Assignment Form* [Cytron *et al* 89] for explicitly parallel programs with `wait` clauses. The parallel sections must be data-independent, except where explicit synchronization is used. The same is true for [Srinivasan *et al* 91b, Wolfe *et al* 91] who introduce a *Parallel Control Flow Graph* and the *Parallel Precedence Graph* which may form the basis of concrete optimizing algorithms.

[Grunwald *et al* 93] present data flow equations for parallel programs, both with and without synchronization. But this work is restricted to PCF FORTRAN programs, which means that access to shared variables is done only at synchronization points. For process start and process end they assume a copy in/copy out semantics. They don't provide a formal, but intuitive derivation of their data flow equations, which solve only the *reaching definition* problem.

[Cousot *et al* 90] extend abstract interpretation to cope with communicating sequential processes. The problem there is that the resulting "state space" explodes, which makes it intractable. [Chow *et al* 92b] apply abstract interpretation to analysis of parallel programs too, but base their semantics on a *labeled transition system*. [Chow *et al* 92a, Chow *et al* 94] attack this problem using the stubborn set theory [Valmari 90] which decreases the state set using some heuristics. Hence the analysis is accurate for some examples, and less accurate for others.

[Vollmer 94] presents the basic idea of how the number of interleavings may be reduced; the parallel program is represented by its structure tree. [Vollmer *et al* 94] use ideas of this report to prove the *Hierarchical Coincidence Theorem* which is based on a functional representation [Sharir *et al* 81] of the problem.

2 The Sample Language

A simple imperative language will be used in this paper, which has loops, conditional statements, and a statement to execute other statements in parallel (explicit control flow parallelism). Replicators allow dynamic process creation, and processes share memory. Besides parallelism the two basic concepts of mutual exclusion and synchronization should be dealt with. *Critical regions* are basically pieces of the program code which should be executed by at most one process. If two processes wish to interact, they may can synchronize and exchange a message over a *channel* (cf. [Hoare 85]).

2.1 The PAR Statement

The PAR statement executes all processes specified by ProcessBody in parallel and independently. The process executing a PAR statement[8] is suspended until all child processes have terminated. A ProcessBody is a list of statements, which may be replicated. That is: max(UpperBound - LowerBound + 1, 0) processes are forked which all execute the statements following the replicator. Each replicated process gets its private copy of the replicator variable Identifier, which has in each replicated process a unique value in the range [LowerBound .. UpperBound]. Replicated processes are also called *asynchronous forall loops* in other languages. All variables can be accessed in each process. No automatic synchronization is done for the access.

```
Stmt        ::=  PAR ProcessBody//"|" END .
ProcessBody ::=  [Replicator] Stmt//";" .
Replicator  ::=  "[" Identifier ":" LowerBound TO UpperBound "]" .
LowerBound  ::=  Expr .
UpperBound  ::=  Expr .
Expr        ::=  usual expressions.
```

Stmt//";" is a list of statements separated by a semicolon. [Replicator] stands for an optional Replicator part.

2.2 OR–Parallelism: The POR Statement

The parent process of a PAR statement is suspended until all child processes have terminated. However, for some applications it may be better if the parent process is reactivated after the first child process has terminated, instead of waiting for all children. Hence we can distinguish between *and parallelism* (PAR statement) where all children have to terminate and *or parallelism* (POR statement) where only one child has to terminate. The syntax of the POR statement is:

```
Stmt ::= POR ProcessBody//"|" END .
```

If now the first child process c has terminated, all other children of c's parent process receive a "signal" that they should now terminate. The parent process resumes its execution as soon as all children have terminated (either "voluntarily" or forced by the "signal"). Since the child processes are executed independently, it may happen in some program runs that the POR statement is equivalent to the PAR statement, i.e. the child processes all terminate "voluntarily", if they receive the "signal" too late.

2.3 Critical Regions

Critical regions are expressed in the program using the following statements:

```
Stmt          ::=  InitSemaStmt | LockStmt | TryStmt .
InitSemaStmt  ::=  INIT "(" SemaVar ")" .
LockStmt      ::=  LOCK SemaVar DO Stmt//";" END .
TryStmt       ::=  TRY SemaVar DO Stmt//";" [ELSE Stmt//";"] END .
```

INIT(SemaVar) initializes the *semaphore* variable SemaVar, by assigning it a unique identification *sema_id*. Semaphore variables are ordinary variables, i.e. they may be assigned and passed as parameters.
All statements within LOCK statements, and all statements before the ELSE part of TRY statements, for which the semaphore variable contains the same *sema_id*, form a *critical region*. Hence the code of a critical region can be distributed over the entire program text, and even more, the extent of a critical region may vary during the program execution, since it depends on the *sema_id*. A critical region can be "entered" by at most one process at a time.

[8] PAR statements may be nested.

If a process p wants to execute a `LOCK` statement, and another process p' has already entered this region, p is suspended from execution until p' has exited the region.
If a process p wants to execute a `TRY` statement, and another process p' has already entered this critical region, p continues with the execution of the optional `ELSE` part[9] of the `TRY` statement. Note that the statements following `ELSE` are not part of the critical region.
Weaker versions of this concept allow only constant values for semaphore variables, or allow only one critical region. In those cases the critical regions can always be discovered statically.

2.4 Message Passing

Synchronous message passing is expressed in the program by the following statements:

```
Stmt         ::=  InitChnStmt | SendStmt | ReceiveStmt .
InitChnStmt  ::=  INIT "(" ChnVar ")" .
SendStmt     ::=  ChnVar "!" Expr .
ReceiveStmt  ::=  ChnVar "?" Var .
```

`INIT(ChnVar)` initializes the *channel* variable `ChnVar`, by assigning it a unique identification *chn_id*. Channel variables are ordinary variables, i.e. they may be assigned and passed as parameters.
Two processes synchronize and exchange a message (e.g. the value of `Expr` is assigned to the variable `Var`), by executing a `SendStmt` and `ReceiveStmt`. The `chn_id` of both channel variables must be equal. Both processes wait for each other. After exchanging the message they, continue their execution independently.
Weaker versions of these statements have only constants as channel identification, which allows static determination of the communication partners.

3 Lattice-Theoretic Background of Data Flow Analysis

This section gives the lattice-theoretic background of data flow analysis and follows [Kam *et al* 77]. It may be skipped by the reader familiar with the notation.
The source program under consideration is represented as a (sequential) control flow graph[10]:

Definition 1 *A* control flow graph *is a triple $G = (N, E, n_0)$, where N is a finite set of* nodes *(which contains a list of simple statements, such as assignments). $E \subseteq N \times N$ is a set of ordered* edges *between these nodes and n_0 the unique initial node.*
A path *from n_1 to n_k is a sequence of nodes $n_1, n_2, \ldots, n_k$, such that for $1 \leq i < k$ all edges $(n_i, n_{i+1}) \in E$. Such a path has* length *k.*
For a node n pred$[n]$(succ$[n]$) *is the set of* predecessors (successors) *defined as:* pred$[n] = \{n' : (n', n) \in E\}$, *and* succ$[n] = \{n' : (n, n') \in E\}$.
All nodes of a control flow graph are reachable from the initial node, i.e. there is a path from n_0 to each node n. path$[n]$ is the set of all paths from the initial node to n. path(n) is the set of paths from n_0 up to all predecessors of n.

The data flow information is represented as a semi-lattice:

Definition 2 *A* semi-lattice $(L, \sqcap)$ *is a set L with a binary* meet *operation $\sqcap$ such that for all $a, b, c \in L$ the following holds:*

$$\begin{array}{rcll} a \sqcap a & = & a & \textit{Idempotent} \\ a \sqcap b & = & b \sqcap a & \textit{Commutative} \\ a \sqcap (b \sqcap c) & = & (a \sqcap b) \sqcap c & \textit{Associative} \end{array}$$

[9] or the statement following the `TRY` statement, if the `ELSE` part is missing.
[10] Section 10 defines the parallel version of a control flow graph.

For two elements $a, b \in L$, we define:

$$\begin{array}{lcl} a \sqsubseteq b & \Leftrightarrow & a \sqcap b = a \\ a \sqsubset b & \Leftrightarrow & a \sqcap b = a \text{ and } a \neq b \\ a \sqsupseteq b & \Leftrightarrow & a \sqcap b = b \\ a \sqsupset b & \Leftrightarrow & a \sqcap b = b \text{ and } a \neq b \end{array}$$

$(L, \sqcap)$ has a zero-element *$\bot$ (bottom), if $\forall x \in L : x \sqcap \bot = \bot$ and a* one-element *$\top$ (top), if $\forall x \in L : x \sqcap \top = x$. From now on we assume that $(L, \sqcap)$ has a zero-element, but not necessary a one-element. We can extend the $\sqcap$ operation:*

$$\prod_{i=1}^{n} x_i = x_1 \sqcap x_2 \sqcap \ldots \sqcap x_n \text{ with } \prod_{x \in \emptyset} = \top$$

A sequence of elements $x_1, x_2, \ldots, x_n$ of L is called a chain, *if $\forall 1 \leq i < n : x_i \sqsupset x_{i+1}$. $(L, \sqcap)$ is called* bounded[11] *if for all $x \in L$ there is a constant b_x such that each chain starting with x has length at most b_x. If $(L, \sqcap)$ is bounded, we can define for each countably infinite set S of elements of L: $\prod_{x \in S} x = \lim_{n \to \infty} \prod_{i=1}^{n} x_i$. Since S is bounded, there is a number m with $\prod_{x \in S} x = \prod_{i=1}^{m} x_i$*

How a single program statement transforms, by its symbolic execution, the data flow information valid before its execution, is described by a *transfer function*:

Definition 3 *Let $(L, \sqcap)$ be a bounded semi-lattice. A set F of functions on L is called an* monotone function space associated with L, *if the conditions [M1] – [M4] are satisfied. If also [M5] is valid, it is called a* distributive function space associated with L.

[M1] *All functions $f \in F$ are monotone: $\forall x, y \in L : f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)$. This is equivalent to: $\forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.*

[M2] *There is an identity function* $\mathrm{id} \in F$ *with: $\forall x \in L : \mathrm{id}(x) = x$.*

[M3] *F closed under composition: $\forall f, g \in F : f \circ g \in F$.*

[M4] *L is the closure of $\{\bot\}$ with respect to the $\sqcap$ operation and application of functions in F.*

[M5] *All functions are distributive: $\forall x, y \in L : f(x \sqcap y) = f(x) \sqcap f(y)$.*

A *monotone data flow framework* is defined as:

Definition 4 *A* monotone data flow framework *is a triple $D = (L, \sqcap, F)$ where $(L, \sqcap)$ is a bounded semi-lattice and F is a monotone function space associated with L. An* instance *of a monotone data flow framework is a pair $I = (G, M)$ where $G = (N, E, n_0)$ is a control flow graph and $M : N \to F$ is a labeling which maps each node from N onto a function of F.*
If F is a distributive function space, D is called a distributive data flow framework.

The "maximal (or worst case) information reaching a program statement" is given by the following

Definition 5

$$\prod_{p \in \mathrm{path}[n)} f_p(\bot)$$

is called the meet over all path *and represents the "maximal (or worst case) information reaching a node n of the program". f_p is the transfer function of the path p (see below).*

4 Properties of Some DFA Frameworks

First we give some properties of bit-vector data flow frameworks $\mathcal{D}^{\mathcal{B}}$ which is then generalized to $\mathcal{D}^{\mathcal{C}}$. At the end of this section then we apply these results to the transfer functions of statements and statement sequences.

[11] or *of finite length* [Hecht 77].

4.1 Properties of the Boolean Semi-Lattice

Since we restrict our investigation to the class of bit-vector data flow problems[12], we give some general results for the boolean semi-lattice:

Definition 6 *The data flow information of an entity is a value of the set $\mathcal{B}$ (Bool) $\mathcal{B} = \{\top, \bot\}$. For a given binary meet operation $\sqcap$, $\bot \sqsubset \top$ must hold.*

Observation 1: Obviously there are only two different binary operations which can be the *meet* operation of a semi-lattice: $\wedge$ (*boolean and*) and $\vee$ (*boolean or*)[13]. They are given as shown in Table 4.

			$\sqcap = \wedge$, $\top$ = TRUE			$\sqcap = \vee$, $\top$ = FALSE		
a	b	$a \sqcap b$	a	b	$a \wedge b$	a	b	$a \vee b$
$\top$	$\top$	$\top$	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
$\top$	$\bot$	$\bot$	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
$\bot$	$\top$	$\bot$	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
$\bot$	$\bot$	$\bot$	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE

Table 4: The boolean meet operations.

Observation 2: There are only four functions $\mathcal{B} \to \mathcal{B}$: the two constant functions, the identity, and negation.

use	$\mathrm{u}(\top)$	$= \top$	$\mathrm{u}(\bot)$	$= \top$
modify	$\mathrm{m}(\top)$	$= \bot$	$\mathrm{m}(\bot)$	$= \bot$
identity	$\mathrm{id}(x)$	$= x$		
negation	$\overline{\top}$	$= \bot$	$\overline{\bot}$	$= \top$

Obviously, the negation function is not monotone and not distributive. The other three are both monotone and distributive. Often, the constant functions are interpreted, respectively as *use*, which generates or uses some information, and *modify*, which modifies or invalidates it.
We finish this section with:
Observation 3: For any two-element semi-lattice $(\mathcal{B}, \sqcap)$, there is exactly one monotone function space $\mathcal{F}^{\mathcal{B}} =_{def} \{\mathrm{u}, \mathrm{m}, \mathrm{id}\}$ associated with $\mathcal{B}$[14]. It is also distributive. $\mathcal{D}^{\mathcal{B}} = (\mathcal{B}, \sqcap, \mathcal{F}^{\mathcal{B}})$ is called the *one bit data flow analysis framework*. There are only two interpretations of the meet operation: the boolean $\wedge$ and $\vee$ operation, respectively. Their DFA interpretation is: the information valid before a node n must be valid on all ($\wedge$) (at least one ($\vee$)) path reaching n.

4.2 Properties of the Function Space $\mathcal{F}^{\mathcal{C}}$

A slight generalization of the $\mathcal{D}^{\mathcal{B}}$ DFA framework is $\mathcal{D}^{\mathcal{C}}$, for which the following lemma obviously holds:

Lemma 1 *Let $(\mathcal{C}, \sqcap)$ be a bounded semi-lattice, and $\mathcal{F}^{\mathcal{C}}$ a set of functions $\mathcal{C} \to \mathcal{C}$, such that $\mathcal{F}^{\mathcal{C}}$ contains only the identity function* id *and for each element c of $\mathcal{C}$ its constant function* const_c *with* $\forall x \in \mathcal{C}$: $\mathrm{const}_c(x) = c$. *Then $\mathcal{F}^{\mathcal{C}}$ is a distributive function space, and the corresponding* constant data flow framework $\mathcal{D}^{\mathcal{C}}$ *is distributive.*

[12] The name *one-bit* or *bit-vector* problem comes from the implementation technique usually used: the data flow information of each entity under consideration (e.g. variable or expression) is either valid or not (hence the boolean semi-lattice) and this fact may be represented by one bit. Usually several entities are considered at the same time, and their information is coded within a *bit-vector*.

[13] The other possible 14 binary operations over $\mathcal{B}$ do not have the required properties, even if some may have an interesting interpretation such as *xor:* the data flow information is valid, if it is valid in exact one path.

[14] Note: To be a monotone function space associated with $\mathcal{B}$, all three functions are needed.

We consider now "composition chains" of functions $f_n \circ f_{n-1} \circ \cdots \circ f_1(x)$ and show some properties. The following lemmata helps us to compute the data flow information which is valid after a `PAR` statement: if for such a composition chain a predicate P holds for all $x \in C$, then there is a function f_i in it, such that P holds for all x, and all "following" $f_j, j > i$ do not invalidate the predicate.

Lemma 2 *Let $f_1, \cdots, f_n \in \mathcal{F}^{\mathcal{C}}$ and P a predicate over $\mathcal{C}$.*

$$\forall x \in \mathcal{C} : P[f_n \circ f_{n-1} \circ \cdots \circ f_1(x)]$$
$$\textit{iff}$$
$$\exists 1 \leq i \leq n : \forall x \in \mathcal{C} : P[f_i(x)] \textit{ and } \forall i < j \leq n : \forall x \in \mathcal{C} : P[f_j(f_i(x))]$$

The lemma can be proved using induction over the number n of functions in the composition chain. The next lemma is a corollary of the previous one:

Lemma 3 *Let $f_1, \cdots, f_n \in \mathcal{F}^{\mathcal{C}}$. Then:*

$$\exists 1 \leq i \leq n : \forall x \in \mathcal{C} : f_n \circ \cdots \circ f_1(x) = f_i(x) \textit{ and } \forall i < j \leq n : f_j = \mathrm{id}$$

The next lemmata state that under some circumstances the order of the functions of a composition chain may be changed and still return the same value.

Lemma 4 *Let $f_1, \ldots, f_n \in \mathcal{F}^{\mathcal{C}}$. From $\exists 1 \leq i \leq n : \forall x \in \mathcal{C} : f_n \circ f_{n-1} \ldots \circ f_1(x) = f_i(x)$, it follows that for an arbitrary permutation $(k_1, \ldots, k_{i-1})$ of the numbers $1, \ldots, i \Leftrightarrow 1$ holds:*

$$f_n \circ f_{n-1} \circ \ldots \circ f_1(x) = f_n \circ f_{n-1} \circ \ldots f_i \circ f_{k_{i-1}} \circ f_{k_{i-2}} \circ f_{k_1}(x)$$

Proof (Lemma 4) If f_i is a constant function, it returns the same result for all arguments. Hence the order of the functions which form the argument of f_i is not important. If f_i is the identity function, all other functions must also be the identity function, otherwise $f_n \circ f_{n-1} \ldots \circ f_1(x) = f_i(x)$ would not hold for all x. ■

The following lemmata state some properties of the value of composition chains. They answer the question which information is valid before a statement inside a `PAR` statement.

Lemma 5 *Let $f, g \in \mathcal{F}^{\mathcal{C}}$. Then for arbitrary $x \in \mathcal{C}$:*

$$x \sqcap f(x) \sqcap g(f(x)) = x \sqcap f(x) \sqcap g(x)$$

Proof (Lemma 5) If $g = \mathrm{const}_c$ then: $x \sqcap f(x) \sqcap g(f(x)) = x \sqcap f(x) \sqcap g(x)$.
If $g = \mathrm{id}$ then: $x \sqcap f(x) \sqcap g(f(x)) = x \sqcap f(x) = x \sqcap f(x) \sqcap g(x)$. ■

Using induction over the number n of functions in the composition chain we can conclude:

Lemma 6 *Let $f_1, \ldots, f_n \in \mathcal{F}^{\mathcal{C}}$. Then for arbitrary $x \in \mathcal{C}$:*

$$x \sqcap f_1(x) \quad \sqcap \quad f_2 \quad \circ f_1(x) \quad \sqcap \quad \ldots \quad \sqcap \quad f_n \circ f_{n-1} \ldots \circ f_1(x) = x \sqcap f_1(x) \sqcap f_2(x) \sqcap \ldots \sqcap f_n(x)$$

4.3 Properties of Statement Sequences and Composition Chains

From now on, we consider only the $\mathcal{D}^{\mathcal{C}}$ data flow analysis framework. We now connect the functions to a statement sequence, which represents an execution path. Then we state properties of the function composition chains, if two (or more) statement sequences are executed in parallel. This is modeled by considering the set of topological sortings of the statements contained in the sequences.

Definition 7 *Let $s_1, \ldots, s_n$ be simple statements, such as assignments, which are executed in the order $p \equiv \langle s_1; \ldots; s_n \rangle$. Let $f_{s_i} \in \mathcal{F}^{\mathcal{C}}$ be the transfer function connected to the statement s_i. $f_p(x) =_{def} f_{s_n} \circ f_{s_{n-1}} \circ \ldots \circ f_{s_1}(x)$*

Definition 8 *Let $s'_1, \ldots, s'_{n'}$ and $s''_1, \ldots, s''_{n''}$ be simple statements, which are executed in the order $p' \equiv \langle s'_1; \ldots; s'_{n'} \rangle$ and $p'' \equiv \langle s''_1; \ldots; s''_{n''} \rangle$, respectively.* $\mathrm{TopSorts}(p', p'')$ *is the set of statement sequences which result from a topological sorting of the two sequences p' and p''. For two statements s'_i, s''_j no order is defined, and s'_i must be executed before s'_{i+1} (also for the statements s''_j).*

Lemma 7 *Let $s'_1, \ldots, s'_{n'}$ and $s''_1, \ldots, s''_{n''}$ be simple statements, which are executed in the order $p' \equiv \langle s'_1; \ldots; s'_{n'} \rangle$ and $p'' \equiv \langle s''_1; \ldots; s''_{n''} \rangle$, and $f_{s'_i}, f_{s''_j} \in \mathcal{F}^{\mathcal{C}}$ the functions corresponding to the statements s'_i and s''_j, respectively. The following holds:*

$$\bigsqcap_{p \in \mathrm{TopSorts}(p', p'')} f_p(x) = f_{p';p''}(x) \sqcap f_{p'';p'}(x)$$

Where $p;q$ is the concatenation of two statement sequences p and q.

Proof (Lemma 7) We prove this lemma in several steps:

1. Let $p \equiv \langle s_1; \ldots; s_n \rangle \in \mathrm{TopSorts}(p', p'')$. With lemma 3 we have a $1 \leq i \leq n$ where $f_p(x) = f_i(x)$[15] and $\forall i < j \leq n : f_j = \mathrm{id}$. If there are several such i, we use the largest one. Now s_i, the statement determining the value of the path p, may be contained either in p' or p''.

2. The set $\mathrm{TopSorts}(p', p'')$ can be split into two disjoint subsets seq' and seq'', where:
   $seq' =_{def} \{q \in \mathrm{TopSorts}(p', p'') \mid \forall x : f_q(x) = f_i(x) \text{ and } s_i \in p'\}$, i.e. seq' contains all those paths, whose value is determined only by statements contained in p'. seq'' is defined analogously.

   Since $\mathrm{TopSorts}(p', p'') = \mathrm{seq}' \cup \mathrm{seq}''$, it follows that $\bigsqcap_{p \in \mathrm{TopSorts}(p', p'')} f_p(x) = \bigsqcap_{p \in \mathrm{seq}'} f_p(x) \sqcap \bigsqcap_{p \in \mathrm{seq}''} f_p(x)$.

3. Proposition: If $s_i \in p''$, then for all x: $f_p(x) = f_{p';p''}(x)$.

   Proof: The proposition is proved, by reordering the sequence p stepwise: The statements $s_{k_1}, s_{k_2} \in p$ with $1 \leq k_1, k_2 < i$ may be reordered in a way that all statements $s_{k_1} \in p'$ are placed before $s_{k_2} \in p''$ and still fulfill the order constraints of p' and p'', respectively. Lemma 4 guarantees that the value of this reordered sequence is still equal to $f_p(x)$.

   The instructions $s_k \in p'$ with $k > i$ may also be placed before s_i, since $f_k = \mathrm{id}$.

   The statements $s_k \in p''$ with $k > i$ need not be reordered.

   Hence if $s_i \in p''$ then $f_p(x) = f_{p';p''}(x)$. Analogously, if $s_i \in p'$ the $f_p(x) = f_{p'';p'}(x)$.

4. Now the statement sequences from seq' and seq'' may be reordered as shown above, while not changing their value. Hence $\bigsqcap_{p \in \mathrm{seq}'} f_p(x) = f_{p'';p'}(x)$ and $\bigsqcap_{p \in \mathrm{seq}''} f_p(x) = f_{p';p''}(x)$.
   And so: $\bigsqcap_{p \in \mathrm{TopSorts}(p', p'')} f_p(x) = f_{p'';p'}(x) \sqcap f_{p';p''}(x)$.

■

5 Data Flow Analysis of the PAR Statement

We now solve the data flow analysis problem for the PAR statement in two steps:

1. which information is valid before each statement in a process body, and
2. which data flow information is valid after the PAR statement?

But before doing this, we need some more definitions.

[15] f_i is an abbreviation of f_{s_i}.

5.1 Statement Traces Generated by the PAR Statement

If the maximum (worst case) data flow information of some statements is given as the *meet over all paths*, the question arises: how can we compute all paths of a parallel program?
We now consider *statement traces* instead of paths in the control flow graph; this simplifies the presentation of the next results.

Definition 9 *A* statement trace *or* statement execution sequence *is a list of simple statements in the order they are executed by a single run of a program. The set* $\mathrm{seq}[S]$ *is the set of all traces generated by statement* S. $s \in p \in \mathrm{seq}[S]$ *means that* s *is a simple statement contained in sequence* q.

It is well known how the set of traces is constructed for sequential programs:

- For the concatenation of statements $S_1; S_2$[16] the traces produced from S_1 and S_2 are concatenated.
- For an **IF** statement, the traces generated by the **THEN** and **ELSE** parts are united.
- For loops, all n-fold concatenations of the traces produced by the loop body are united.

Since we assumed an interleaving semantics for the execution of the statements of our **PAR** statement, it is obvious that the topological sorting of the traces produced by the process bodies, determines all traces of the **PAR** statement.

Definition 10 *For a given trace* p *the set* $\mathrm{prefixes}(p)$ *is the set of all prefixes of* p. *It includes the empty trace and the entire trace* p. $\mathrm{prefixes}(P)$ *is the extension to a set of traces* P, *so that it contains all prefixes of all traces of* $p \in P$. *For a statement* S, $\mathrm{prefixes}[S]$ *is the abbreviation of* $\mathrm{prefixes}(seq[S])$.

Definition 7 is now extended for arbitrary (composite) statements:

Definition 11 *Let* $S, S_1, \ldots, S_n$ *be arbitrary (composite) statements:* $f_S(x) =_{def} \prod_{p \in \mathrm{seq}[S]} f_p(x)$. $f_{S_1;S_2;\ldots;S_n}(x) =_{def} f_{S_n} \circ f_{S_{n-1}} \circ \ldots \circ f_{S_1}(x)$ *is the function corresponding to the statement sequence* $S_1; \ldots; S_n$.

an obvious lemma is:

Lemma 8 *Let* $S_1, \ldots, S_n$ *be arbitrary statements and let be* $S \equiv S_1; \ldots; S_n$. *Then for distributive functions* $f \in F$: $f_{S_1;S_2;\ldots;S_n}(x) = f_S(x)$, *while for monotone* $f \in F$, *only* $f_{S_1;S_2;\ldots;S_n}(x) \sqsubseteq f_S(x)$ *holds.*

Definition 12 *Let* s *be a statement inside a (nested)* ***PAR*** *statement.* $\mathrm{sibl}[s]$ *is the set of all simple statements which could possibly be executed in parallel to* s. *If* s *is part of a replicated process body, all statements of this process body are also contained in* $\mathrm{sibl}[s]$.

Note that the property $s' \in \mathrm{sibl}[s]$ of s' is a pure syntactical one, which can easily be determined from the source program.

5.2 Which Data Flow Information Reaches a Statement

Let s be a statement inside a **PAR** statement. By definition the information valid before s is given by the meet over all traces reaching s. To answer the question which these traces are, let us examine the following example[17]: **PAR** $s \mid s_1; s_2; \ldots; s_n$ **END**. s may now be the first statement executed in an interleaving produced by this **PAR** statement. On the other hand, s_1 may be executed before s in another interleaving, or $s_1; s_2$ are executed before s, etc. Hence the set of statements executed before s is given by $\mathrm{prefixes}(\langle s_1; \ldots; s_n \rangle)$. Using lemma 6 we can conclude that:

[16] Capital letters S denote composite statements, whereas small letters s denote simple ones.
[17] Note that in these traces some statement within the body of the **PAR** statement may not be included, since they are executed "after" s.

$\prod_{p \in \text{prefixes}((s_1;\ldots;s_n))} f_p(x) = x \sqcap \prod_{i=1}^{n} f_{s_i}(x)$. If the process body S of PAR s | S END which is executed in parallel to s produces more than one execution sequence, the set of execution sequences reaching s is given by: $\bigcup_{p \in \text{seq}[S]} \text{prefixes}(p)$, and hence: $\prod_{p \in \text{prefixes}(\text{seq}[S])} f_p(x) = x \sqcap \prod_t$ is a statement from S $f_t(x)$.
The fact that there may be statements, which are *always* executed before s, does not influence our considerations, since the argument x of the function f_s reflects the information reaching s, if no parallel statements are executed before s.
Since PAR statements may be nested, the following theorem is a consequence of the above:

Theorem 1 *Let s be a simple statement in a program, and* sibl$[s]$ *the set of simple statements possibly executed in parallel to s. Then for $\mathcal{D}^{\mathcal{C}}$ data flow problems, the following information is valid before s:*

$$x \sqcap \prod_{t \in \text{sibl}[s]} f_t(x),$$

where x is the information valid before the ***PAR*** *statement.*

If we use the following definition, we can restate the theorem so that it is easier to use as a base for an implementation.

Definition 13 *For statement s, we define* $\text{in}^{'}[s]$ *as the information reaching s on a "sequential" trace from the program entry. That is, none of siblings of s are executed before s. And let* $\text{in}^{\|}[s]$ *be the information reaching s if all possible traces are considered.*

If a function $f_t \in \mathcal{F}^{\mathcal{C}}, t \in \text{sibl}[s]$ is a constant function, then the value of $f_t(x)$ is independent of x: $f_t(x) = f_t$. Otherwise f_t is the identity, and $f_t(x) = x$. Then $\text{in}^{'}[s]$ is simply the value of x, and the Theorem 1 can be restated as:

$$\text{in}^{\|}[s] = \text{in}^{'}[s] \sqcap \prod_{t \in \text{sibl}[s]} f_t \tag{1}$$

5.3 Which Information is Valid After the PAR Statement

Lemma 7 is the cornerstone for the following theorems. After extending it to sets of paths we obtain:

Theorem 2 *For the $\mathcal{D}^{\mathcal{C}}$ data flow problems and the* ***PAR*** *S_1 | S_2 **END** statement the following holds:*

$$\prod_{p \in \text{seq}[\texttt{PAR } S_1 \mid S_2 \texttt{ END}]} f_p(x) = \prod_{p \in \text{seq}[S_1;S_2]} f_p(x) \sqcap \prod_{p \in \text{seq}[S_2;S_1]} f_p(x)$$

To extend this theorem for PAR statements with more than two process bodies, we define:

Definition 14 *The set of* simple permutations s_perm *of the numbers $1, \ldots, n$ is given by:*
$\text{s\_perm}(1,n) =_{def} \{\langle 1,2,\ldots,n \Leftrightarrow 1,n\rangle, \langle n,2,\ldots,n \Leftrightarrow 1,1\rangle, \langle 1,n,3,\ldots,n \Leftrightarrow 1,2\rangle, \ldots, \langle 1,2,3,\ldots,n,n \Leftrightarrow 1\rangle\}$.
That is, the i^{th} number is exchanged with the last number in the sequence $\langle 1,2,\ldots,n\rangle$.
$\vec{\imath} \equiv \langle i_1, i_2, \ldots, i_n\rangle$ *denotes an element of this set. Let $S_1, \ldots, S_n$ be statement sequences, and $\vec{\imath} \in$* $\text{s\_perm}(1,n)$. *Then $S_{\vec{\imath}}$ is defined to be the statement sequence: $S_{i_1}; \ldots; S_{i_n}$.*

Note that $\text{s\_perm}(1,n)$ has only n elements.
If we have a closer look at the proof of lemma 7, we see that the order of the statements s_k, with $k < i$ has no influence on that result. Hence we have the following lemma:

Lemma 9 *Let $p_1, \ldots, p_n$ statement sequences of simple statements. Then:*

$$\prod_{p \in \text{TopSorts}(p_1,\ldots,p_n)} f_p(x) = \prod_{\vec{\imath} \in \text{s\_perm}(1,n)} f_{p_{i_1};\ldots;p_{i_n}}(x)$$

Note that any other permutation would serve, as long as each statement sequence appears at least once at the end. Hence we have the following theorem:

Theorem 3 *For the $\mathcal{D}^{\mathcal{C}}$ data flow problems and the* **PAR** S_1 | ... | S_n ***END*** *statement:*

$$\prod_{p \in \text{seq}[\texttt{PAR } S_1 \mid \ldots \mid S_n \texttt{ END}]} f_p(x) = \prod_{\vec{i} \in \text{s\_perm}(1,n)} f_{S_{i_1};\ldots;S_{i_n}}(x).$$

It is easy to see that the result is not changed by a replicated process body `[var := lwb TO upb]` S with `upb - lwb + 1 > 0`, since for the result only S is important. If `upb - lwb + 1` $\geq$ `0`, then it may happen that S is never executed, and the theorem should be adjusted accordingly. We will see in the implementation section 10, how this could easily be done.
Now we show how to analyze the other language constructs (`POR, TRY, LOCK, ?, !`). Their analysis is based on the results of this section.

6 Data Flow Analysis of the POR Statement

Let S_1 and S_2 be sequences of simple statements. Then the **POR** S_1 | S_2 **END** statement can be "transformed" into a series of **PAR** statements, where one branch is executed fully, while the other one is executed only partially. During runtime an "oracle" determines which of the **PAR** statement is actually executed. An example is given in Table 5. Under the view of data flow analysis this transformation is valid, if the data flow analysis problem is distributive. Now we can apply the previous results.

POR	PAR	PAR	PAR	PAR	PAR
a := 1;	a := 1;	a := 1;	a := 1;	a := 1;	a := 3;
a := 2;	a := 2;	a := 2;	a := 2;	|	a := 4;
|	END	|	|	a := 3;	END
a := 3;		a := 3;	a := 3;	a := 4;	
a := 4;		END	a := 4;	END	
END			END		

Table 5: Example transformation of a **POR** statement.

We see, that the set of **PAR** statements which result from the transformation of a **POR** S_1 | S_2 **END** statement is given by:

$$\{\texttt{PAR } S_1 \mid p \texttt{ END} | p \in \text{prefixes}[S_2]\} \cup \{\texttt{PAR } p \mid S_2 \texttt{ END} | p \in \text{prefixes}[S_1]\} \tag{2}$$

But what happens if we have arbitrary statements S_1 and S_2, such as loops or conditionals? Nothing, except that the set of prefixes is potentially infinite, but over a finite set of statements.
Since $\mathcal{D}^{\mathcal{C}}$ is distributive, we can apply Theorem 2 and Lemma 6. Let S_1 and S_2 be arbitrary statements, $q \in \text{seq}[S_2]$ a trace of statements of S_2, and $p \in \text{prefixes}(q)$ a prefix of that trace. Let us examine the first set of Equation 2. From Theorem 2 we obtain[18]: $f_{S_1 \| p}(x) = f_{S_1;p}(x) \sqcap f_{p;S_1}(x)$. If we now consider all prefixes of q we have: $\prod_{p \in \text{prefixes}(q)} f_{S_1 \| p}(x) = \prod_{p \in \text{prefixes}(q)} f_{S_1;p}(x) \sqcap f_{p;S_1}(x)$. Using Lemma 6 this equals: $\prod_{s \in q} f_s(f_{S_1}(x)) \sqcap f_{S_1}(f_s(x))$. Extending this to all traces produced by S_2 results in: $\prod_{p \in \text{prefixes}(\text{seq}[S])} f_{S_1 \| p}(x) = \prod_{s \in p \in \text{seq}[S_2]} f_s(f_{S_1}(x)) \sqcap f_{S_1}(f_s(x))$. We still have to consider all statement traces of S_2, but since a trace consists only of simple statements contained in a composite one, this can be reduced to: $\prod_{s \in S_2} f_s(f_{S_1}(x)) \sqcap f_{S_1}(f_s(x))$
The same considerations can be applied to the second part of Equation 2 and we end up with:

[18] $S_1 \parallel S_2$ is a shorthand for the **PAR** S_1 | S_2 **END** statement.

Theorem 4 *For the $\mathcal{D}^{\mathcal{C}}$ data flow problems and the* **POR** $S_1 \mid S_2$ **END** *statement:* $\forall x \in V$:

$$\prod_{p \in \text{seq[POR } S_1 \mid S_2 \text{ END]}} f_p(x) \quad = \quad \prod_{s \in S_1} f_s(f_{S_2}(x) \sqcap f_{S_2}(f_s(x)) \sqcap \prod_{s \in S_2} f_s(f_{S_1}(x)) \sqcap f_{S_1}(f_s(x))$$

Theorem 1 is also valid for the `POR` statement. Theorem 4 may be extended quite easily for any number of process bodies.

7 Data Flow Analysis of Critical Regions

Let $\mathcal{S} \equiv$ `LOCK` var_s `DO S END`, $\mathcal{T} \equiv$ `LOCK` var_t `DO T END`,
$\psi \equiv$ `PAR` ...; $\mathcal{S}$; ... | ...; $\mathcal{T}$; ... `END`, and $\psi' \equiv$ `PAR` ...; `S`; ... |...; `T`; ... `END`.
Using Theorem 2 we have: $f_{\ldots;\mathcal{S};\ldots;\mathcal{T};\ldots}(x) \sqcap f_{\ldots;\mathcal{T};\ldots;\mathcal{S};\ldots}(x) = f_{\psi}(x)$. The sequentialization of $\mathcal{S}$ and $\mathcal{T}$ now reflects the semantics of `LOCK`. Hence we have the following theorem:

Theorem 5 *For the $\mathcal{D}^{\mathcal{C}}$ data flow problems and the* **LOCK** *statements:* $f_{\psi}(x) = f_{\psi'}(x)$*, independently of the value of semaphore variables of the* **LOCK** *statements.*
If the values $\mathbf{var}_s$ *and* $\mathbf{var}_t$ *are equal, then the statements of S can not be executed in parallel to T. This fact may be used when computing* $\text{in}^{\parallel}[s]$ *with* $s \in S$*, and* $s \in T$*, respectively.*

The `TRY` statement can be modeled for the data flow analysis by `IF ... THEN LOCK var DO ... END ELSE ... END`.

8 Data Flow Analysis of Message Passing

If we have a closer look at the statement

```
PAR S'; ch ! expr; S'' | T'; ch ? var; T'' END
```

we could use the knowledge that `S'` (`S''`) is never executed in parallel to `T''` (`T'`), if they use the same channel. But if these statements are contained in loops, this fact does not hold any more:

```
PAR LOOP S'; ch ! expr; S'' END | LOOP T'; ch ? var; T'' END END
```

Now `S'` (`S''`) may be executed in parallel to `T''` (`T'`). Hence we have to consider this worst case situation and we have the following theorem:

Theorem 6 *For the $\mathcal{D}^{\mathcal{C}}$ data flow problems and the* `!` *and* `?` *statements, Theorems 1 and 2 hold.*

9 Bit-Vector Implementation

From now on, we consider only the $\mathcal{D}^{\mathcal{B}}$ data flow analysis framework.
Until now we considered only one program entity, such as a single program variable or a single expression. When implementing data flow analysis, usually all entities are considered at the same time. Hence we are dealing with sets of informations, valid at a program point. Each entity is coded by a number $1 \ldots |\text{entities}|$. For the class of one-bit data flow problems $\mathcal{D}^{\mathcal{B}}$ there is a quite efficient implementation of sets: the bit-vector.

Definition 15 *A bit-vector is the characteristic function* vec *of a finite set of of object numbers* $1 \ldots n = |\text{entities}|$. $\text{vec} : \{1 \ldots n\} \to \{\text{TRUE}, \text{FALSE}\}$ *with* $1 \leq n$*. Usually* $\text{vec}(i)$ *is denoted as* $\text{vec}[i]$.
The set operations $\cup, \cap, \overline{}$ *are defined for bit-vectors, by element wise application of the boolean operations* $\vee, \wedge, \overline{}$*. The set difference is defined by* $a \Leftrightarrow b =_{def} a \cap \overline{b}$.
The empty set $\emptyset$ *is represented by the bit-vector, in which all values are* FALSE.

Usually for each statement and basic block the following four sets are defined [Aho *et al* 86]: *gen*, *kill*, *in* and *out*. *gen* (*kill*) is the set of information generated (invalidated) by this statement/basic block. *in* is the set of information valid immediately before execution of this statement and *out* the information valid immediately after that point.
As seen before, the $\sqcap$ operation is either set union or set intersection. The DFA problems using set union (intersection) as meet operations are called *may problems* (*must problems*), since the information must reach a given program point on at least one (on all) paths leading to that point.
We define now the four sets in terms of the transfer functions. Each statement s has a separate transfer function for each entity e, denoted by f_s^e.

Definition 16 *Let s be a statement,*

$\mathrm{gen}_s[e]$ = TRUE *iff* $f_s^e = \mathrm{u}$ *(use),*
$\mathrm{kill}_s[e]$ = TRUE *iff* $f_s^e = \mathrm{m}$ *(modify),*
$\mathrm{in}_s[e]$ = TRUE *iff* $\prod_{p \in \mathrm{path}[s)} f_p^e(\bot) = \top$ *and*
$\mathrm{out}_s[e]$ = TRUE *iff* $\prod_{p \in \mathrm{path}[s]} f_p^e(\bot) = \top$.
Obviously: $\mathrm{gen}_s \cap \mathrm{kill}_s = \emptyset$

We will first formulate the result of Theorem 1 in the form of Equation 1 using the DFA sets. Note that we are now using $\mathcal{D}^{\mathcal{B}}$:
In the equation $\mathrm{in}^{\parallel}{}_s[e] = \mathrm{in}^{i}{}_s[e] \sqcap \prod_{t \in \mathrm{sibl}[s]} f_t^e$, all f_t are constant functions. The right hand side is $\top$, iff both parts of it evaluate to $\top$. The following equivalences hold:
$\prod_{t \in \mathrm{sibl}[s]} f_t^e = \top \Leftrightarrow \forall t \in \mathrm{sibl}[s] : f_t^e = \top \Leftrightarrow \nexists t \in \mathrm{sibl}[s] : f_t^e = \bot \Leftrightarrow \bigwedge_{t \in \mathrm{sibl}[s]} \mathrm{gen}_s[e] = \mathrm{TRUE} \Leftrightarrow$
$\bigvee_{t \in \mathrm{sibl}[s]} \mathrm{kill}_s[e] = \mathrm{FALSE}$.
We now distinguish:

$\sqcap = \wedge; \top = \mathrm{TRUE}$
$\prod_{t \in \mathrm{sibl}[s]} f_t^e = \top \Leftrightarrow \bigwedge_{t \in \mathrm{sibl}[s]} f_t^e = \mathrm{TRUE} \Leftrightarrow \overline{\bigvee_{t \in \mathrm{sibl}[s]} \mathrm{kill}_s[e]} = \mathrm{TRUE}$

$\sqcap = \vee; \top = \mathrm{FALSE}$
$\prod_{t \in \mathrm{sibl}[s]} f_t^e = \top \Leftrightarrow \bigvee_{t \in \mathrm{sibl}[s]} f_t^e = \mathrm{FALSE} \Leftrightarrow \overline{\bigwedge_{t \in \mathrm{sibl}[s]} \mathrm{gen}_s[e]} = \mathrm{FALSE} \Leftrightarrow$
$\bigvee_{t \in \mathrm{sibl}[s]} \overline{\mathrm{gen}_s[e]} = \mathrm{FALSE} \Leftrightarrow \bigvee_{t \in \mathrm{sibl}[s]} \mathrm{gen}_s[e] = \mathrm{TRUE}$.

And we can state the following theorem:

Theorem 7 *The information of a one-bit DFA problem reaching a statement s can be computed by:*

$$\textit{for } \sqcap = \wedge: \quad \mathrm{in}^{\parallel}{}_s = \mathrm{in}^{i}{}_s \Leftrightarrow \bigcup_{t \in \mathrm{sibl}[s]} \mathrm{kill}_t \qquad \textit{with } \mathrm{in}^{i}{}_{s_0} = \top = \mathrm{TRUE}$$

$$\textit{for } \sqcap = \vee: \quad \mathrm{in}^{\parallel}{}_s = \mathrm{in}^{i}{}_s \cup \bigcup_{t \in \mathrm{sibl}[s]} \mathrm{gen}_t \qquad \textit{with } \mathrm{in}^{i}{}_{s_0} = \top = \mathrm{FALSE}$$

(s_0 is the first statement of the program).

The next step is the adaptation of Theorem 3. Before doing so, we restate the equations for in and out in the sequential case, as they may be found e.g. in [Aho *et al* 86]:
For a simple statement the relation is given by:

$$\mathrm{out}_s = \mathrm{gen}_s \cup \mathrm{in}_s \Leftrightarrow \mathrm{kill}_s \tag{3}$$

For sequential composition of statements $s \equiv s_1; s_2$:

$$\mathrm{out}_s = \mathrm{out}_2 \cup \mathrm{out}_1 \Leftrightarrow \mathrm{kill}_2 = \mathrm{gen}_2 \cup (\mathrm{gen}_1 \Leftrightarrow \mathrm{kill}_2) \cup (\mathrm{in}_s \Leftrightarrow (\mathrm{kill}_1 \cup \mathrm{kill}_2))^{19} \tag{4}$$

and more generally: $s \equiv s_1; \ldots s_n$

$$\mathrm{out}_s = \bigcup_{i=1}^{n} \left(\mathrm{gen}_i \Leftrightarrow \bigcup_{j=i+1}^{n} \mathrm{kill}_j \right) \cup \left(in_s \Leftrightarrow \bigcup_{i=1}^{n} \mathrm{kill}_i \right) \tag{5}$$

For branches in a sequential program:

$$\begin{array}{rcl} \mathrm{in}_s & = & \bigsqcap_{p \in \mathrm{pred}[s]} \mathrm{out}_p \\ \mathrm{out}_s & = & \mathrm{gen}_s \cup \mathrm{in}_s \Leftrightarrow \mathrm{kill}_s \end{array} \tag{6}$$

Using these equations we can compute the right hand side of the equation from Theorem 3: $\bigsqcap_{\vec{i} \in \mathrm{s\_perm}(1,n)} f_{s_{i_1}; \ldots; s_{i_n}}(x)$. We again have to distinguish between may and must problems. Hence the following theorem can be given (the proof is given in the appendix):

Theorem 8 *The information* out_S *of a one-bit DFA problem valid after the* ***PAR*** $S_1 \mid \ldots \mid S_n$ ***END*** *statement* S *can be computed as:*

$$\begin{array}{lcl} for \sqcap = \wedge & : & \mathrm{out}_S = \left(\left(\bigcup_{i=1}^{n} \mathrm{gen}_i \right) \Leftrightarrow \left(\bigcup_{i=1}^{n} \mathrm{kill}_i \right) \right) \cup \left(\mathrm{in}'_S \Leftrightarrow \bigcup_{i=1}^{n} \mathrm{kill}_i \right) \\ for \sqcap = \vee & : & \mathrm{out}_S = \bigcup_{i=1}^{n} \mathrm{gen}_i \cup \left(\mathrm{in}'_S \Leftrightarrow \bigcup_{i=1}^{n} \mathrm{kill}_i \right) \end{array}$$

where in'_S *is the information valid before the* ***PAR*** *statement, and* gen_i *and* kill_i *are the sets corresponding to the process bodies* S_i*.*

Now we have to determine which information is generated and invalidated by a **PAR** statement as a whole. Again we start with the sequential composition of statements (following [Aho *et al* 86]): For $S \equiv S_1; S_2$ we have:

$$\begin{array}{rcl} \mathrm{gen}_S & = & \mathrm{gen}_2 \cup \mathrm{gen}_1 \Leftrightarrow \mathrm{kill}_2 \\ \mathrm{kill}_S & = & \mathrm{kill}_2 \cup \mathrm{kill}_1 \Leftrightarrow \mathrm{gen}_2 \end{array} \tag{7}$$

Hence we can follow the same arguments as for $\mathrm{out}_{\mathrm{PAR}}$, with the simplification that the term $\mathrm{in} \Leftrightarrow (\mathrm{kill} \cup \ldots)$ does not exist. We note here that if the "In-Out-problem[20]" is a may-problem, then the "Gen-Problem[21]" is a may-problem too, while the corresponding "Kill-problem" is a must-problem, and vice versa, if the In-Out-problem is a must-problem. So the next theorem can be stated:

Theorem 9 *The information of a one-bit DFA problem generated and invalidated by a* ***PAR*** $S_1 \mid \ldots \mid S_n$ ***END*** *statement* S *can be computed as:*

$$\begin{array}{lcl} \textit{In-Out-problem } \sqcap = \wedge : & \mathrm{gen}_S & = \left(\bigcup_{i=1}^{n} \mathrm{gen}_i \right) \Leftrightarrow \left(\bigcup_{i=1}^{n} \mathrm{kill}_i \right) \\ & \mathrm{kill}_S & = \bigcup_{i=1}^{n} \mathrm{kill}_i \end{array}$$

[19] The symbol X_{s_i} is abbreviated to X_i.
[20] Which information is valid before/after a statement.
[21] Which information is generated by the statement.

$$\textit{In-Out-problem } \sqcap = \vee: \quad \begin{aligned} \text{gen}_S &= \bigcup_{i=1}^{n} \text{gen}_i \\ \text{kill}_S &= \left(\bigcup_{i=1}^{n} \text{kill}_i\right) \Leftrightarrow \left(\bigcup_{i=1}^{n} \text{gen}_i\right) \end{aligned}$$

Having all these theorems, we see that we have avoided the state-space explosion problem.
These results are given for an "isolated" `PAR` statement. The next section will put them in the context of a *parallel control flow graph* and we will see how this gives us an elegant algorithm for computing the data flow information of a parallel program.

10 The Parallel Control Flow Graph

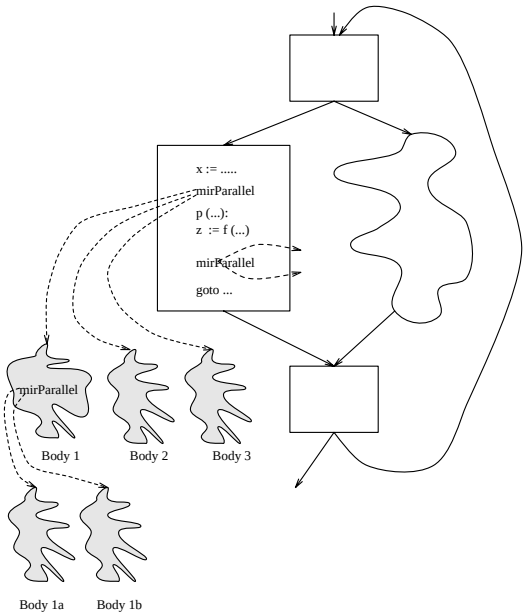
The following explanation is based on our implementation of the parallel language *Modula-P* [Vollmer 89, Vollmer *et al* 92] developed in the COMPARE project. To express parallelism in the intermediate language *CCMIR-P*[22] we define some additional CCMIR statements and define a *parallel control flow graph* (PCFG).
Looking at the results of this theory, we observe that the `PAR` statement and its processes are treated by it as a single statement with a complex behavior. The idea for integrating the analysis of a `PAR` statement is to treat it like other CCMIR statements such as assignment or procedure call, except that it has a more complex DFA behavior. The DFA effects of the `PAR` statement are determined solely by its process bodies.
The central idea of our parallel control flow graph (PCFG) is that it is a forest of disjoint CFG's. Each process body and procedure body constitutes a separate CFG. Since jumps into or out of process bodies are forbidden in the source language, there are no *jump edges* connecting the CFG's.
To form a PCFG of a procedure we connect these separate CFG's by adding *parallel edges* between the CFG of a process body and the `mirParallel` statement containing this process body.
For the program fragment shown in Table 6 the PCFG is given in Figure 1.

```
REPEAT ..;
  IF ..
  THEN
    x:=..;
    PAR
     ..; PAR Body1a | Body1b END; ..;
     | Body2
     | Body3
    END
    p (..); z := f (..);
    PAR Body4a | Body4b END;
  ELSE ..
  END;
UNTIL ..
```

Table 6: A program fragment with nested `PAR` statements.

After parsing a source program, we obtain a list of *all* basic blocks, constituting an entire procedure. There is no specific order in this list. From that, we compute for each basic block the list of predecessors and successors (cf. [Aho *et al* 86]). Each basic block has zero, one or two successors. It has none, if it has as its last statement the `EndProcedure` or `EndProcess` statement. It has one, if the last statement is a

[22] *Common* COMPARE *Medium Intermediate Representation, Parallel extension*; the intermediate language of the COMPARE compilers.



The dashed lines are parallel edges connecting the separate CFG's of the process bodies to their `mirParallel` statement. The grey boxes are CFG's of the process bodies. The other boxes are part of the procedure body's CFG.

Figure 1: A PCFG for the program fragment of Table 6.

`mirGoto`, and two if this is a `mirIf` statement. Entry basic blocks are marked by the `BeginProcedure` and `BeginProcess` CCMIR instructions: their basic blocks have no predecessors. To find the roots of the CFG's we simply scan the procedure's list of all basic blocks for basic blocks having no predecessors. If a replicator specifies that no process should be created, then we draw an extra jump edge from the process body entry to its exit. This solves the problems mentioned in the note to Theorem 3.
Our definition of a PCFG differs from the one given in [Wolfe *et al* 91] in the way that their PCFG has two kinds of nodes: "ordinary" basic blocks and *super nodes (or parallel blocks)*. Such a super node represents the entire process body.

11 Solving the Data Flow Equations of Concurrent Programs

Let's assume we do not have nested `PAR` statements. Then we must analyze a program in the following order:

1. Compute *gen* and *kill* information for all process bodies and the `mirParallel` statement itself.
2. Compute *gen* and *kill* information for all statements of the procedure's CFG.
3. Compute *in* and *out* for all statements of the statements of the procedure's CFG.
4. Since we know the exact information reaching the `mirParallel` statement, we can compute the *in* and *out* information of the valid at the statements of the process bodies: the information reaching the `mirParallel` statement reaches the entry of each process body.

Now it is obvious how to deal with nested `PAR` statements:

1. Visit the deeply-nested process bodies first and compute their *gen* and *kill* information. This corresponds to a depth-first traversal of the PCFG along the parallel edges. This is called *inside-out computation of Gen/Kill.*

2. Compute *gen* and *kill* of all other statements of the procedure's CFG.

3. Compute *in* and *out* for all statements of the procedure's CFG.

4. Visit the `mirParallel` statements from *outside-in* (the reverse order of inside-out) and compute *in* and *out* of the process bodies. Outside-in is the top-down traversal of the PCFG along the parallel edges.

This kind of computation of the data flow information is a mixture of the structural [Babich *et al* 78], [Aho *et al* 86, page 611] (for the effect of the `mirParallel` statement) and iterative (all other statements) method.
A similar idea has been presented for DFA of sequential problems by [Horwitz *et al* 87]. They partition the CFG into strongly connected components *(scc)* and visit them in topological order, while the DFA inside an scc is done using rPOSTORDER *(scc-iteration algorithm).*
The computation of the *in* and *out* information must be done using an iterative algorithm [Kildall 73, Aho *et al* 86]. At a first glance the same seems to be true for the computation of *gen* and *kill*[23]. But this iterative approach for *gen* and *kill* solves a broader problem: it computes for each basic block b the set of *gen* and *kill* information reaching b. But we need only the *gen* and *kill* information which is valid at the end of a process body. We can therefore use a simpler algorithm which combines the computation of the local *gen* and *kill* with the computation of the *gen* and *kill* of a set of basic blocks. We visit the basic blocks in the reverse depth first order, which guarantees that at the end of the process we have the same result than using the iterative method.

11.1 The Inside-Out/Outside-In Algorithm

Most tasks in analyzing a parallel program need to traverse nested `PAR` statements in the previously mentioned *Inside-out/Outside-In* Order. Here is a sketch of an algorithm for doing this.
The assignments to the variable *info* ← ... below an abbreviation, for "compute *info* in some way, using possibly ...".
The function `tInfo doit_io (LIST(mirBasicBlock) bbl)` and `void doit_oi (LIST(mirBasicBlock) bbl, tInfo info)` do the "real work" on the list of basic blocks `bbl`, representing a CFG.

Algorithm 11.1 Inside-Out

TASK: Visit all basic blocks of all processes of the `PAR` statement in Inside-Out order. Compute some information and propagate it Inside-Out.

SIGNATURE: `tInfo visit_in_out (mirParallel stmt)`

METHOD:

```
tInfo info; /* the "information" passed around */
forall processes p in stmt->Processes do
  forall mirParallel stmts s in p do  /* Visit the PAR statements, nested in this process. */
    info ← visit_in_out (s)
  end

  info ← doit_io (process->cfg) /* Visit the body of this process */
end
return info;
```

[23] Note: We have to compute the *gen* and *kill* info for a set of basic blocks. In the sequential DFA, this information is not needed.

■

Algorithm 11.2 Outside-In

TASK: Visit all basic blocks of all processes of the nested `PAR` statement in Outside-In order. Use some information passed from the "outside".

SIGNATURE: `void visit_out_in (mirParallel stmt, tInfo info)`

METHOD:

```
tInfo info; /* the "information" passed around */
forall processes p in stmt->Processes do
  doit_oi (p->cfg, info) /* Visit the body of this process */

  forall mirParallel stmts s in p do  /* Visit the PAR statements, nested in this process. */
    visit_out_in (s, info)
  end
end
```

■

Algorithm 11.3 Driver for Inside-out /Outside-In

TASK: visit all basic blocks of a procedure in Inside-Out and then Outside-In order.

SIGNATURE: `void do_all (mirProcGlobal proc)`

METHOD:

```
tInfo info; /* the "information" passed around */
/* A: Inside-Out, compute information info and propagate it inside-out. */
forall mirParallel stmts s in proc do  /* Visit the all PAR statements Inside/Out. */
  info ← visit_in_out (s)
end
info ← doit_io (proc->cfg) /* Visit the body of this procedure */

/* B: Outside/In, propagate information info from outside-in. */
doit_oi (proc->cfg, info) /* Visit the body of the procedure */
forall mirParallel stmts s in proc do  /* Visit the all PAR statements */
  visit_out_in (s, info)
end
```

■

11.2 Computing Gen/Kill and InS/Out for a CFG

Here we give the algorithms which compute Gen/Kill and InS/Out information for the basic blocks of a CFG. Using the Inside/Out and Outside-In strategies of the previous section, we can compute the DFA information valid for each basic block of the PCFG.

Algorithm 11.4 Gen/Kill of a CFG

TASK: Compute for all basic blocks of a CFG and for the CFG itself the Gen/Kill information. This routine will be called *inside-out* for the procedure under consideration.

SIGNATURE: `void gen_kill_cfg(LIST(mirBasicBlock) cfg, SET *GenOut, SET *KillOut);`

INPUT: `cfg` is the list of basic blocks constituting this CFG; the list is ordered in reverse depth first order, i.e. the first (last) list element is the entry (exit) basic block of this process/procedure.

OUTPUT: `GenOut` and `KillOut` give the *Gen/Kill* information of the entire CFG, i.e. valid after the exit basic block of this CFG.

METHOD:

/* *Initially the Gen/Kill sets of all basic blocks are empty.* */

forall bb **in** cfg **do**

/* *Compute local Gen/Kill information.* */
bb->Gen $\leftarrow$ /* *Gen of this basic block* */;
bb->Kill $\leftarrow$ /* *Kill of this basic block* */;

/* *Gen/Kill reaching this bb* */
/* *If the in/out problem is a* $\sqcap$ *problem, then Gen is a* $\sqcap$ *problem and Kill is a* $\sqcup$ *problem.* */
g $\leftarrow \prod_{pp \in bb \Rightarrow pred} pp \Leftrightarrow Gen$;
k $\leftarrow \bigsqcup_{pp \in bb \Rightarrow pred} pp \Leftrightarrow Kill$;
/* *Don't worry about Gen/Kill of basic blocks which are not computed yet. These basic blocks will be considered later in the* cfg *list, and contribute their Gen/Kill then. This is valid, since we are not interested in a fixed point, but rather in the Gen/Kill valid after the exit basic block of this CFG.* */

/* *Gen/Kill valid at the end of this bb* */
bb->Gen $\leftarrow$ g - bb->Kill;
bb->Kill $\leftarrow$ k - bb->Gen;

end

pp = `LAST(cfg)`; /* *last basic block of the basic block list.* */
GenOut = pp->GenOut; KillOut = pp->KillOut

■

Algorithm 11.5 In/Out of a CFG

TASK: Compute for all basic blocks of a CFG and for the CFG itself the In/Out information. This routine will be called *outside-in* for the procedure under consideration.

SIGNATURE: `void in_out_cfg (LIST(mirBasicBlock) cfg);`

INPUT: `cfg` is the list of basic blocks constituting this CFG; the list is ordered in the reverse depth first order, i.e. the first (last) list element is the entry (exit) basic block of this process/procedure.

OUTPUT: for each basic block, *InS and Out.*

METHOD:

We use the iterative algorithm of [Kildall 73], as given in e.g. [Aho *et al* 86].

/* *Initially the In/Out sets of all basic blocks are empty.* */

entry = `FIRST(cfg)`;

if entry basic block is a process entry basic block **then**

entry->InS = InS of the `mirParallel` statement, containing the process; /* *Everything reaching a* `mirParallel` *statement reaches each of its processes.* */

else

entry->InsS = $\top$; /* *No information is valid at the procedure entry* */

end

/* *Compute fixed point* */

while no changes in any bb->Out **do**

forall bb **in** cfg **do**

bb->InS $\leftarrow \prod_{pp \in bb \Rightarrow pred} bb \Leftrightarrow Out$;
bb->Out $\leftarrow bb \Leftrightarrow Gen \cup bb \Leftrightarrow In \Leftrightarrow bb \Leftrightarrow Kill$;

end

end

■

12 Complexity of this DFA Algorithm

To estimate the complexity of this algorithm, we use as complexity measure the number of visits of a basic block during the iterative computation of the DFA information. A "comparable" sequential program is one where the `PAR` statement and its process bodies are executed sequentially.
For the computation of the *gen* and *kill* sets, we have to visit each basic block once, both in the sequential and parallel case.
During the computation of *in* and *out* we apply the iterative algorithm several times to different (and disjoint) sets of basic blocks: first, we compute the DFA information for the basic blocks of the procedure's CFG. Second, we compute *in* and *out* for the sets of basic blocks corresponding to process bodies. The process bodies are considered in the *outside-in* order of the `PAR` statement.
The number of iterations needed to compute the DFA information is determined by the *loop nestedness* [Hecht *et al* 75] of the source program.
Since we don't have jump edges between basic blocks of different process bodies, we are always computing the DFA information of disjoint sets of basic blocks. Hence the loop nestedness is the same for the parallel and the comparable sequential program. Hence the overall number of basic block visits is equal in the parallel and the comparable sequential program.
As a result the data flow analysis of a parallel program has the same complexity as a comparable sequential one.
This result is supported by [Horwitz *et al* 87], who stated the following theorem:

> An application of the scc iteration will visit no more nodes than an application of rPOST-ORDER iteration.

13 Current and Future Work

Currently the algorithm of [Knoop *et al* 94] for elimination of partial redundancies is implemented in the Compare compiler for the source language *Modula-P* [Vollmer 89, Vollmer *et al* 92], which is an extension of Modula-2 [Wirth 85] with CSP (Communicating Sequential Processes) [Hoare 78].
As part of the project "analyzing and optimizing parallel programs", the next step is to adapt the *static single assignment form* [Cytron *et al* 89] to cope with parallelism, without having the restrictions of [Srinivasan *et al* 91a].

14 Conclusion

This work shows that data flow analysis of parallel programs is possible, and can be done as efficiently as for sequential programs. The novelty is that there is no restriction in the kind shared memory access, nor in the "accuracy" of the resulting DFA information. Hence it is now possible to apply optimizing transformations, which are well known from the sequential context.
To show this, we proved some nice properties of the semi-lattice based data flow frameworks $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^{\mathcal{B}}$, which allowed us to reduce the number of interleavings needed for the computation of the *meet over all paths* solution of the DFA problem. Then we extended these results to bit-vectors, and obtained simple set equations, computing the DFA information valid inside and after a `PAR` statement. Based on that we gave a simple algorithm to compute the DFA information valid at all program points. This algorithm is a slight variant of the usual iterative DFA algorithm, but the basic blocks are visited in a special order: *inside-out and outside-in* of the nesting structure of the `PAR` statement.

Acknowledgments

I want to express many thanks for discussions with Bernhard Steffen and Jens Knoop, both of the University of Passau, and with Helmut Emmelmann.

References

[ACM89] ACM. *16. ACM Symposium on Principles of Programming Languages*, January 1989. Austin, Texas.

[ACM92] ACM. *19. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1992. Albuquerque, New Mexico.

[Afek *et al* 93] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[Aho *et al* 86] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Babich *et al* 78] Wayne A. Babich and Mehdi Jazayeri. The method of attributes for data flow analysis. *Acta Informatica*, 10:345–272, 1978. Part I Exhaustive Analysis, Part II Demand Analysis.

[Bakker *et al* 80] J.W. de Bakker and J van Leeuwen, editors. *Automata, Languages and Programming, 7. Colloquium*, volume 85 of *Lecture Notes in Computer Science*. Springer Verlag, Heidelberg, New York, July 1980. Noordwijkerhout, NL.

[Banerjee *et al* 91] U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors. *Languages and Compilers for Parallel Computing*, volume 589 of *Lecture Notes in Computer Science*. Springer Verlag, Heidelberg, New York, August 1991. 4th International Workshop, Santa Clara, Proceedings.

[Bristow *et al* 88] G. Bristow, C. Drey, B. Edwards, and W. Riddle. Anomaly detection in concurrent programs. In *[Gehani et al 88]*, chapter 23, pages 567–585. Addison-Wesley, 1988.

[Chow *et al* 92a] Jyh-Herng Chow and Williams Ludwell III Harrison. A general framework for analyzing shared-memory parallel programs. In *1992 International Conference on Parallel Processing (ICPP), Conference Proceedings*, pages II192–II199, August 1992.

[Chow *et al* 92b] Jyh-Herng Chow and Williams Ludwell Harrison III. Compile-time analysis of parallel programs that share memory. In *[ACM92]*, pages 130–141, 1992.

[Chow *et al* 94] Jyh-Herng Chow and Williams Ludwell III Harrison. State space reduction in abstract interpretation of parallel programs. In *Proceedings of the 1994 International Conference on Computer Languages ICCL'94, Toulouse, France*, pages 277–288. IEEE, IEEE Computer Society Press, May 1994.

[Cousot *et al* 90] Patrick Cousot and Radhia Cousot. Semantic analysis of communicating sequential processes. In *[Bakker et al 80]*, pages 119–133, 1990.

[Cytron *et al* 89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *[ACM89]*, pages 25–35, 1989.

[DEC 92] Digital Equipment Corperation DEC. *Alpha Architecture Handbook*. DEC, USA, February 1992. Preliminary.

[Gehani *et al* 88] Narain Gehani and Andrew D McGettrick, editors. *Concurrent Programming*. Addison-Wesley, 1988.

[Grunwald *et al* 93] Dirk Grunwald and Harini Srinivasan. Data flow equations of explicitly parallel programs. In *PPoPP 93*. ACM SIGPLAN NOTICES, 1993.

[Hecht 77] Matthew S. Hecht. *Flow analysis of computer programs*. North Holland, 1977.

[Hecht *et al* 75] Matthew S. Hecht and Jeffery D. Ullmann. A simple algorithm for global data flow analysis problems. *SIAM*, 4(4):519–532, December 1975.

[Hoare 78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoare 85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, Inc., 1985.

[Horwitz *et al* 87] S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data-flow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[Kam *et al* 77] John B. Kam and Jeffery D. Ullmann. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Kennedy 81] Ken Kennedy. A survey of data flow analysis techniques. In *[Muchnick et al 81]*, pages 5–54. , 1981.

[Kildall 73] Gary A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

[Knoop *et al* 94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1177–1155, July 1994.

[Lamport 79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multi process programs. *IEEE Transactions on Computers*, c-28(9):690–691, September 1979.

[Midkiff *et al* 90] S.P. Midkiff and David A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990* International Conference on Parallel Processing, pages 105–113, Volume II, August 1990.

[Muchnick *et al* 81] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis*. Prentice-Hall, Inc., 1981.

[Netzer *et al* 92] Robert H.B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[Philippsen *et al* 91] Michael Philippsen and Walter F. Tichy. Modula-2* and its compilation. In *[Zima 91]*, pages 169–183, 1991.

[Reif 84] John H. Reif. Data flow analysis of distributed communicating processes. Technical Report TR-12-83, Harvard University, Center for Research in Computing Technology, September 1984.

[Rozenberg 90] G Rozenberg, editor. *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*. Springer Verlag, Heidelberg, New York, 1990.

[Sharir *et al* 81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *[Muchnick et al 81]*, pages 189–234. , 1981.

[Srinivasan *et al* 91a] Harini Srinivasan and Dirk Grunwald. An efficient construction of parallel static single assignment form for structured parallel programs. Technical Report CU-CS-564-91, University of Colorado at Boulder, Department of Computer Science, December 1991.

[Srinivasan *et al* 91b] Harini Srinivasan and Michael Wolfe. Analysing programs with explicit parallelism. In *[Banerjee et al 91]*, 1991.

[Valmari 90] Antti Valmari. Stubborn sets for reduced state space generation. In *[Rozenberg 90]*, pages 491–515, 1990.

[Vollmer 89] Jürgen Vollmer. Modula-P, a language for parallel programming. *Proceedings of the First International Modula-2 Conference October 11-13, 1989, Bled, Yugoslavia*, pages 75–79, 1989.

[Vollmer 94] Jürgen Vollmer. Dataflow equations for parallel programs that share memory. Technical Report GMD-1101-dfepp, Release 1.1, Universiät Karlsruhe, January 1994. Deliverable 2.11.1 of the ESPRIT Project COMPARE # 5933.

[Vollmer 95] Jürgen Vollmer. Data flow analysis of parallel programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 95*. IEEE/ACM; to be published, June 1995.

[Vollmer *et al* 92] Jürgen Vollmer and Ralf Hoffart. Modula-P, a language for parallel programming: Definition and implementation on a transputer network. In *Proceedings of the 1992 International Conference on Computer Languages ICCL'92, Oakland, California*, pages 54–64. IEEE, IEEE Computer Society Press, April 1992.

[Vollmer *et al* 94] Jürgen Vollmer, Jens Knoop, and Bernhard Steffen. Parallelism for free: Efficient and optimal bitvector analysis for parallel programs. Technical Report MIP-9409, Universität Passau, Fakultät für Mathematik und Informatik, August 1994.

[Wirth 85] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, Heidelberg, New York, third, corrected edition, 1985.

[Wolfe *et al* 91] Michael Wolfe and Harini Srinivasan. Data structures for optimizing programs with explicit parallelism. In *[Zima 91]*, pages 139–156, 1991.

[Zima 91] Hans Zima, editor. *Parallel Computing, 1. Int. ACPC Conference Salzburg, Austria*, volume 591 of *Lecture Notes in Computer Science*. Springer Verlag, Heidelberg, New York, September 1991.

[Zima *et al* 90] Hans Zima and Barbara Chapman. *Super Compilers for Parallel and Vector Computers.* Addison-Wesley, 1990. ACM Press.

[Zimmermann *et al* 94] Wolf Zimmermann and Welf Löwe. An approach to machine-independent parallel programming. In *Parallel Programming, CONPAR 94–VAPP VI, Lecture Notes in Computer Science, 854*, pages 277–288. Springer Verlag, Heidelberg, New York, 1994.

Appendix A: Additional Proofs

Proof (Theorem 8)
Using equation 5, we see:

$$
\begin{array}{ll}
\prod_{\tau \in s\_perm(1,n)} f_{S_{i_1};\ldots;S_{i_n}}(x) & \\
= & f_{S_1;S_2;\ldots;S_{n-1};S_n}(x) \sqcap f_{S_n;S_2;\ldots;S_{n-1};S_1}(x) \sqcap \ldots \sqcap f_{S_1;S_2;\ldots;S_n;\ldots;S_{n-1};S_i}(x) \sqcap \ldots \sqcap f_{S_1;S_2;\ldots;S_n;S_{n-1}}(x) \\
= & (\mathrm{gen}_n \cup \mathrm{gen}_{n-1} - \mathrm{kill}_n \cup \mathrm{gen}_{n-2} - (\mathrm{kill}_{n-1} \cup \mathrm{kill}_n) \cup \ldots (\mathrm{gen}_1 - \bigcup_{j=2}^{n} \mathrm{kill}_j) \cup (\mathrm{in}_s - \bigcup_{j=1}^{n} \mathrm{kill}_j)) \\
& \sqcap \\
& (\mathrm{gen}_1 \cup \mathrm{gen}_{n-1} - \mathrm{kill}_1 \cup \mathrm{gen}_{n-2} - (\mathrm{kill}_1 \cup \mathrm{kill}_{n-1}) \cup \ldots (\mathrm{gen}_n - \bigcup_{j=1}^{n-1} \mathrm{kill}_j) \cup (\mathrm{in}_s - \bigcup_{j=1}^{n} \mathrm{kill}_j)) \\
& \sqcap \\
\ldots & \\
& (\mathrm{gen}_i \cup \mathrm{gen}_{n-1} - \mathrm{kill}_i \cup \mathrm{gen}_{n-2} - (\mathrm{kill}_i \cup \mathrm{kill}_{n-1}) \cup \ldots (\mathrm{gen}_1 - \bigcup_{j=2}^{n} \mathrm{kill}_j) \cup (\mathrm{in}_s - \bigcup_{j=1}^{n} \mathrm{kill}_j) \\
& \sqcap \\
\ldots & \\
& (\mathrm{gen}_{n-1} \cup \mathrm{gen}_n - \mathrm{kill}_{n-1} \cup \mathrm{gen}_{n-2} - (\mathrm{kill}_{n-1} \cup \mathrm{kill}_n) \cup \ldots (\mathrm{gen}_1 - \bigcup_{j=2}^{n} \mathrm{kill}_j) \cup (\mathrm{in}_s - \bigcup_{j=1}^{n} \mathrm{kill}_j)
\end{array}
\tag{8}
$$

For $\sqcap = \cup$ we can conclude: $\bigcup_{i=1}^{n} \mathrm{gen}_i \cup \mathrm{in}_s \Leftrightarrow \bigcup_{i=1}^{n} \mathrm{kill}_i$.
Now let's consider $\sqcap = \cap$. Generally for arbitrary sets $a_{i,j}$ it holds:

$$
\begin{array}{ll}
(a_{1,1} \cup a_{1,2} \cup \ldots \cup a_{1,n}) \cap & \\
(a_{2,1} \cup a_{2,2} \cup \ldots \cup a_{2,n}) \cap & (a_{1,1} \cap a_{2,1} \cap \ldots \cap a_{n,1}) \cup \\
\ldots & = \quad \ldots \\
(a_{n,1} \cup a_{n,2} \cup \ldots \cup a_{n,n}) \cap & (a_{1,k_1} \cap a_{2,k_2} \cap \ldots \cap a_{n,k_n}) \cup \\
 & \ldots
\end{array}
\tag{9}
$$

where the n-tuples $(k_1, \ldots, k_n)$ take all values in the range $\{1, \ldots, n\}^n$.
In equation 8 the "isolated" term gen_i may be replaced without any changes by $\mathrm{gen}_i \Leftrightarrow \mathrm{kill}_i$, since $\mathrm{gen}_i \cap \mathrm{kill}_i = \emptyset$. Hence all elements of a row of equation 8 have the form: $g_j \Leftrightarrow (\bigcup_{l \in L} \mathrm{kill}_l)$, where $L \subseteq \{1, \ldots, n\}$.
Since the resulting union the elements of $\mathrm{in} \Leftrightarrow \bigcup_{i=1}^{n} \mathrm{kill}_i$ are contained, all intersection terms which contain also this term $\mathrm{in} \Leftrightarrow \bigcup_{i=1}^{n} \mathrm{kill}_i$ may be removed. According to equation 9 for one row we have (where $j_i \in \{1, \ldots, n\}$):

$$
\begin{array}{ll}
a_{1,k_1} \cap a_{2,k_2} \cap \ldots \cap a_{n,k_n} & \\
= & (\mathrm{gen}_{j_1} \Leftrightarrow (\mathrm{kill}_1 \cup \ldots)) \cap (\mathrm{gen}_{j_2} \Leftrightarrow (\mathrm{kill}_2 \cup \ldots)) \cap \ldots \cap (\mathrm{gen}_{j_n} \Leftrightarrow (\mathrm{kill}_n \cup \ldots)) \\
= & \mathrm{gen}_{j_1} \cap \mathrm{gen}_{j_2} \cap \ldots \cap \mathrm{gen}_{j_n} \cap \overline{\mathrm{kill}}_1 \cap \overline{\mathrm{kill}}_2 \cap \ldots \cap \overline{\mathrm{kill}}_n
\end{array}
$$

From equation 9 it follows that there are rows, in which all j_i are equal.
Since $\mathrm{gen}_{j_1} \cap \mathrm{gen}_{j_2} \cap \ldots \cap \mathrm{gen}_{j_n} \cap \overline{\mathrm{kill}}_1 \cap \overline{\mathrm{kill}}_2 \cap \ldots \cap \overline{\mathrm{kill}}_n \subseteq \mathrm{gen}_{j_1} \cap \overline{\mathrm{kill}}_1 \cap \overline{\mathrm{kill}}_2 \cap \ldots \cap \overline{\mathrm{kill}}_n$ it follows that equation 8 evaluates to

$$
\begin{array}{ll}
\bigcup_{i=1}^{n} (\mathrm{gen}_i \cap \bigcap_{j=1}^{n} \overline{\mathrm{kill}}_k) \cup \mathrm{in} \Leftrightarrow (\bigcup_{i=1}^{n} \mathrm{kill}_i) & \\
= & (\bigcup_{i=1}^{n} \mathrm{gen}_i) \cap (\bigcap_{j=1}^{n} \overline{\mathrm{kill}}_j) \cup \mathrm{in} \Leftrightarrow \bigcup_{i=1}^{n} \mathrm{kill}_i \\
= & (\bigcup_{i=1}^{n} \mathrm{gen}_i) \Leftrightarrow (\bigcup_{j=1}^{n} \mathrm{kill}_j) \cup \mathrm{in} \Leftrightarrow \bigcup_{i=1}^{n} \mathrm{kill}_i
\end{array}
$$

■