Demand-Driven Analysis of Explicitly Parallel Programs: An Approach Based on Reverse Data-Flow Analysis

Jens Knoop

Universität Dortmund, D-44221 Dortmund, Germany
knoop@ls5.cs.uni-dortmund.de
http://sunshine.cs.uni-dortmund.de/~knoop

Abstract. *Demand-driven data-flow analysis* (*DD-DFA*) has been pioneered in the field of interprocedural analysis of sequential programs to accommodate the specific needs of analysis tools like debuggers and slicers, but also to compensate for the continuous growth of programs and program complexity making static program analyses more and more intricate and time consuming. Particularly successful in practice turned out to be a DD-DFA approach based on *reverse* DFA. Recently, we have shown how to generalize this approach for the class of GEN/KILL-problems to explicitly parallel programs. Here, we focus on the theoretical foundation of this approach.

Keywords: Explicit parallelism, shared memory, (reverse) data-flow analysis, demand-driven data-flow analysis, parallel data-flow analysis.

1 Motivation

Data-flow analysis (*DFA*) is an indispensable prerequisite for compilers for high performance providing them with the necessary information on the applicability of performance improving optimizations. However, growing programs and growing program complexity due to powerful new language features offered by advanced programming paradigms like e.g. parallelism make these analyses more and more intricate and time consuming. In order to cope with these problems, but also to accommodate the specific needs of program analysis tools like debuggers and slicers, approaches for *demand-driven* (*DD*) program analysis have been developed (cf. [6–8, 11, 24]). Intuitively, these approaches restrict the computation of information to the program points, where it is needed, and the point of time when it is needed. Particularly successful in practice turned out to be an approach based on *reverse DFA*, which was pioneered in the field of interprocedural DFA of sequential programs (cf. [7, 8]). In [17], we have recently shown how to extend this approach for the class of so-called GEN/KILL-problems (cf. [9]) to explicitly parallel programs with shared memory. For this problem class the challenges posed by interference and synchronization, which are a major obstacle of analyzing parallel programs, can elegantly and efficiently be solved. Moreover, GEN/KILL-problems are particularly important in practice because of the multitude of optimizations based on them (cf. [23]). Figure 1 illustrates the benefit of the demand-driven approach by means of the *availability* of terms, a classical DFA-problem (cf. [9]).[2] Using conventional DFA, the program of Figure 1 must completely be investigated in order to detect the availability and

[2] Intuitively, a term is *available* at a program point, if it has been computed on every program path reaching this point without a subsequent modification of any of its operands.

the inavailability of the terms $a + b$ and $c + b$ at the destination nodes of edges **64** and **32**. In contrast, the demand-driven approach based on reverse DFA succeeds here to answer these data-flow queries by considering only small portions of the program. In this article, we reconsider this approach focusing on its theoretical foundation.

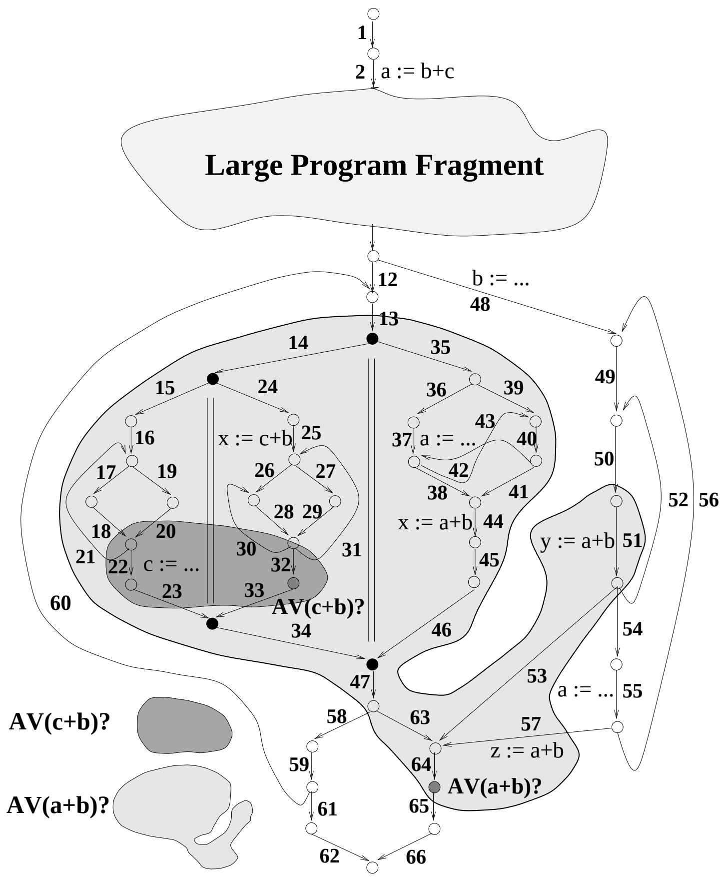

Fig. 1. Conventionally, the complete program must be investigated. Demand-drivenly, the investigation of minor fragments of the program is generally sufficient.

2 The Parallel Setting

We consider explicitly parallel programs with shared memory and interleaving semantics. Parallelism is syntactically expressed by means of a `par` statement. As usual, we assume that there are neither jumps leading into a component of a parallel statement from outside nor vice versa. As shown in Figure 1, we represent a parallel program by an edge-labeled *parallel flow-graph* $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N, edge set E, and a unique start node $\mathbf{s}$ and end node $\mathbf{e}$ without incoming and outgoing edges, respectively. By $\mathcal{G}_{\mathcal{P}}(G)$ we denote the set of subgraphs of G representing a parallel statement. They are encapsulated by a `ParBegin` node and a ParEnd node. The sets of these nodes are denoted by $N^N_{\mathcal{P}}$ and $N^X_{\mathcal{P}}$. If $G' \in \mathcal{G}_{\mathcal{P}}(G)$, then $\mathcal{G}_{\mathcal{C}}(G')$ denotes the set of subgraphs constituting the parallel components of G'. Like G, each component is assumed to have a unique start node and end node, whose only predecessor and successor is a node of $N^N_{\mathcal{P}}$ and $N^X_{\mathcal{P}}$, respectively. The function *Nodes* maps a graph (or subgraph) to the set of its nodes, the functions *start* and *end* map a graph to its start node and end node, and the functions *pfg* and *cfg* map a node n occurring in the parallel part of a program to the smallest parallel statement and component graph of a parallel statement containing n. A *parallel* (*program*) *path* is an edge sequence representing a valid interleaving sequence of G. Most conveniently, parallel paths can be defined by considering explicitly parallel programs as "abbreviations" of (usually much larger nondeterministic) programs, which result from a product construction between parallel components (cf. [3]). The product programs represent all the possible interleaving sequences explicitly (cf. [10]). A parallel path is then an edge sequence of the product program. By $\mathbf{PP}[m,n]$, $m,n \in N$, we denote the set of all finite parallel paths from m to n. Finally, for each node $n \in N$, we denote by $\mathit{InterfEdges}(n) \subseteq E$ the set of its *interfering edges*. Intuitively, this is the set of edges whose execution can be interleaved with the execution of the statements of n's incoming edge(s). They are the source of interference among the statements of different components of a parallel statement.

3 Background: Data-Flow Analysis & Reverse Data-Flow Analysis

Intuitively, DFA aims at computing (specific aspects of) the run-time behaviour of a program without actually executing it. In order to avoid undecidabilities and to allow fully automatic analyses, the "full" semantics of a program is usually replaced by a simpler more abstract version, which is tailored for the problem under consideration (cf. [4, 5]). Usually, the abstract semantics is given by a *semantic functional* $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$, also called a *data-flow functional*, which gives abstract meaning to the elementary statements of a program in terms of transfer functions on a complete lattice $\mathcal{C}$ with least element $\bot$ and greatest element $\top$. Its elements represent the data-flow facts of interest. Without loss of generality we here assume that $\top \in \mathcal{C}$ represents "unsatisfiable (inconsistent)" data-flow information, and that it is invariant under all semantic functions, i.e., $[\![\, e \,]\!](\top) = \top$ for all $e \in E$.[3]

3.1 Data-Flow Analysis

Given a data-flow functional $[\![\ ]\!]$, and a data-flow fact $c_{\mathbf{s}} \in \mathcal{C}$, which is assumed to be valid at the program entry $\mathbf{s}$, DFA intuitively aims at computing for each program point the "strongest" data-flow fact valid there with respect to $[\![\ ]\!]$ and $c_{\mathbf{s}}$. Formally, this is captured by the *meet-over-all-paths* (*MOP*) solution, which is induced by the *MOP*-approach (cf. [14, 15]). For each program point, the *MOP*-solution is the result of the meet of all data-flow facts yielded by a program path reaching it. Fundamental for this is the extensibility of a data-flow functional to paths. If $p = \langle e_1, \ldots, e_q \rangle$, then $[\![\, p \,]\!]$ is the identity $Id_{\mathcal{C}}$ on $\mathcal{C}$, if $q < 1$, and $[\![\, \langle e_2, \ldots, e_q \rangle \,]\!] \circ [\![\, e_1 \,]\!]$, otherwise. In the parallel setting, we have to consider the set of parallel paths. To emphasize this, we speak of the *PMOP*-approach and the *PMOP*-solution in the following, where the first "*P*" reminds to "parallel."

[3] In case of need a lattice can always be extended by a new top element satisfying this requirement.

$$\boxed{\textbf{The } \textit{PMOP}\textbf{-Solution } \forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ \textit{PMOP}_{c_{\mathbf{s}}}(n) = \bigsqcap \{\, [\![\, p \,]\!](c_{\mathbf{s}}) \mid p \in \mathbf{PP}[\mathbf{s}, n] \,\}}$$

Note that the interleaving semantics underlying our language is generally an indication for an exponential effort of computing the *PMOP*-solution. GEN/KILL-problems, however, are an exception. In [21], we have shown how to compute for GEN/KILL-problems the *PMOP*-solution efficiently by means of an extension of the well-known *maximal-fixed-point* (*MFP*) approach to explicitly parallel programs (cf. [14]). In the remainder of this article, we will complement this approach by an approach allowing us for GEN/KILL-problems to compute the *PMOP*-solution demand-drivenly based on reverse data-flow analysis.

3.2 Reverse Data-Flow Analysis

Fundamental for reverse DFA is the *reverse semantic functional* $[\![\ ]\!]_R : E \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ induced by an (ordinary) data-flow functional $[\![\ ]\!]$. It maps a data-flow fact $c \in \mathcal{C}$ to the greatest lower bound of the set of data-flow facts guaranteeing the validity of c after executing e, i.e., $\forall\, e \in E\ \forall\, c \in \mathcal{C}.\ [\![\, e \,]\!]_R(c) =_{df} \bigsqcap \{\, c' \mid [\![\, e \,]\!](c') \sqsupseteq c \,\}$. An immediate consequence of this definition is (cf. [12, 13]):[4]

Lemma 1. *For each edge $e \in E$ we have: (1) $[\![\, e \,]\!]_R$ is well-defined and monotonic. (2) If $[\![\, e \,]\!]$ is distributive, then $[\![\, e \,]\!]_R$ is additive.*

Intuitively, the image of $[\![\, e \,]\!]_R$ for an argument c is expected to be the "weakest" data-flow fact ensuring the validity of c after executing e. For distributive data-flow functionals, which imply the additivity of their reverse, this intuition is indeed valid (cf. Lemma 2 and Theorem 1). In fact, calling a data-flow fact attached to the program entry a *context information*, and otherwise a *component information*, conventional DFA aims at computing strongest component informations with respect to a given context information, while reverse DFA aims at computing the weakest context information guaranteeing the validity of a specific component information at a program point. The latter is often called a *data-flow query*, and the program point the *query node*. Conceptually, this is modeled by the *reverse join-over-all-paths* (*R-JOP*) approach, the reverse counterpart of the meet-over-all-paths approach.

To define this approach formally, we introduce the following convention. If the query node $q \in N$ is different from the program entry $\mathbf{s}$, then G is extended by a copy $\mathbf{q}$ of q. This is a fresh node, which has the same predecessors as q but no successors. Note that the addition of $\mathbf{q}$ does not affect the *PMOP*-solution of any of the original nodes of G. Moreover, the *PMOP*-solutions of $\mathbf{q}$ and q coincide. Though inconspicuous, this observation is important for proving the correctness of the overall approach, in particular, for establishing the relationship between the *PMOP*-approach and the fixed-point approach introduced in Section 4 complementing the *R-PJOP*-approach, and thus for proving the correctness of mimicking conventional DFA by reverse DFA. The definition of the *R-PJOP*-approach relies on the extensibility of a reverse data-flow functional to (finite) parallel paths p. If $p = \langle e_1, \ldots, e_q \rangle$, then $[\![\, p \,]\!]_R$ is the identity $Id_{\mathcal{C}}$ on $\mathcal{C}$, if $q < 1$, and $[\![\, \langle e_1, \ldots, e_{q-1} \rangle \,]\!]_R \circ [\![\, e_q \,]\!]_R$, otherwise. Note that this means a "backward" traversal of path p.

$$\boxed{\textbf{The } \textit{R-PJOP}\textbf{-Solution: } \forall\, c_q \in \mathcal{C}\ \forall\, n \in N.\ \textit{R-PJOP}_{c_q}(n) = \bigsqcup \{\, [\![\, p \,]\!]_R(c_q) \mid p \in \mathbf{PP}[n, \mathbf{q}] \,\}}$$

[4] A function $f : \mathcal{C} \rightarrow \mathcal{C}$ on a complete lattice $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$ is *distributive* (*additive*) iff $\forall\, C' \subseteq \mathcal{C}.\ f(\bigsqcap C') = \bigsqcap \{ f(c) \mid c \in C' \}$ $(f(\bigsqcup C') = \bigsqcup \{ f(c) \mid c \in C' \})$.

Lemma 2 (Weakest Context Informations).
Let $[\![\ ]\!]$ *be a distributive data-flow functional, let* $c_q \in \mathcal{C}$ *be a data-flow query, and let* $\hat{c} =_{df} \bigsqcap \{ c_{\mathbf{s}} \in \mathcal{C} \mid \mathit{PMOP}_{c_{\mathbf{s}}}(q) \sqsupseteq c_q \}$. *Then we have:* $\mathit{PMOP}_{\hat{c}}(q) \sqsupseteq c_q$.

Intuitively, $\{ c_{\mathbf{s}} \in \mathcal{C} \mid \mathit{PMOP}_{c_{\mathbf{s}}}(q) \sqsupseteq c_q \}$ is the set of context informations guaranteeing the validity of the data-flow query c_q at q. Lemma 2 ensures that for distributive data-flow functionals the greatest lower bound of this set enjoys this property, too. It is thus the weakest context information ensuring this. For distributive data-flow functionals, on which we focus in the following, we denote this element by wci_{c_q}, i.e., $wci_{c_q} =_{df} \bigsqcap \{ c_{\mathbf{s}} \in \mathcal{C} \mid \mathit{PMOP}_{c_{\mathbf{s}}}(q) \sqsupseteq c_q \}$.

3.3 Conventional vs. Reverse DFA: The Connecting Link

In terms of program verification, Theorem 1 provides the link between the "strongest post-condition" and the "weakest precondition" view of DFA (cf. [1, 2]). Given a distributive data-flow functional, it opposes the *PMOP*-solution and the *R-PJOP*-solution.

Theorem 1 (Link Theorem).
Let $[\![\ ]\!]$ *be a distributive data-flow functional, and let* $c_{\mathbf{s}}, c_q \in \mathcal{C}$. *Then we have:*

1. $\mathit{R\text{-}PJOP}_{c_q}(\mathbf{s}) = wci_{c_q}$
2. $\mathit{R\text{-}PJOP}_{c_q}(\mathbf{s}) \sqsubseteq c_{\mathbf{s}} \iff \mathit{PMOP}_{c_{\mathbf{s}}}(q) \sqsupseteq c_q$

The first part of Theorem 1 ensures that for distributive DFA-problems the *R-PJOP*-solution at the program entry **s** coincides with the weakest context information of the considered data-flow query. The second part guarantees that if it is a lower bound of some context information at **s**, then this context information guarantees the validity of the data-flow query, i.e., the *PMOP*-solution of this context information is an upper bound of the data-flow query at the query node, and vice versa. Intuitively, this equivalence, which holds for distributive data-flow functionals, is the key to demand-driven DFA based on reverse DFA.

4 Reverse Data-Flow Analysis of Explicitly Parallel Programs

In this section we focus on GEN/KILL-problems. As mentioned before, for this problem class, the problems of interference and synchronization can elegantly and efficiently be captured (cf. [21]). This applies to reverse DFA, too. We demonstrate this by complementing the approach of [21] for conventional DFA of explicitly parallel programs with its reverse counterpart. In fact, the presentation in this section parallels the one of [21].

4.1 GEN/KILL-Problems and Their Reverse Counterparts

GEN/KILL-problems, also known as *bitvector* problems (cf. [9]), are characterized by the simplicity of their data-flow functional $[\![\ ]\!] : E \to (\mathcal{B} \to \mathcal{B})$. It specifies the effect of an edge e on a particular component of the bitvector, where $\mathcal{B}$ denotes the lattice of Boolean truth values $(\{\mathit{false}, \mathit{true}\}, \sqcap, \sqcup, \sqsubseteq)$ with $\mathit{false} \sqsubseteq \mathit{true}$ and the logical "and" as meet operation $\sqcap$, and the logical "or" as join operation $\sqcup$. Despite their simplicity, GEN/KILL-problems are most relevant in practice (cf. [23]) because of their broad scope of applications ranging from simple analyses like liveness of a variable or availability of a term (cf. [9]) to powerful program optimizations like code motion (cf. [22, 18, 20]) or partial dead-code elimination (cf. [19, 16]).

Here, we consider the availability of a term as an example. Intuitively, a term t is available at a program point n, if it is computed on each program path from $\mathbf{s}$ to n without a subsequent modification of any of the operands of t. Table 1 summarizes the definition of the abstract semantics for availability (and of its reverse counterpart), where *Comp* and *Transp* are two local predicates of edges indicating if the considered term is computed and its value is transparent for an edge. Note that we consider the lattice $\mathcal{B}_X$ extending $\mathcal{B}$ by an artificial top element *failure* in order to be compliant with our assumption that the top element of a data-flow lattice represents unsatisfiable information. The semantic functions $Id_{\mathcal{B}_X}$, Cst^X_{true} and Cst^X_{false} occurring in the definition are the identity on $\mathcal{B}_X$, and the extensions of the constant functions Cst_{true} and Cst_{false} from $\mathcal{B}$ to $\mathcal{B}_X$ leaving the argument *failure* invariant.

In fact, these are the only functions which are involved in a GEN/KILL-problem. Their simplicity allows us to specify their reverse counterparts directly as shown below.

$$\forall\, b \in \mathcal{B}_X.\ R\text{-}Cst^X_{true}(b) \ = \begin{cases} \mathit{false} & \text{if } b \in \mathcal{B} \\ \mathit{failure} & \text{otherwise (i.e., if } b = \mathit{failure}) \end{cases}$$

$$\forall\, b \in \mathcal{B}_X.\ R\text{-}Cst^X_{false}(b) \ = \begin{cases} \mathit{false} & \text{if } b = \mathit{false} \\ \mathit{failure} & \text{otherwise} \end{cases}$$

$$R\text{-}Id_{\mathcal{B}_X} = Id_{\mathcal{B}_X}$$

Abstract semantics for *availability*:

1. *Data-flow lattice*: $(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathcal{B}_X, \wedge, \vee, \leq, \mathit{false}, \mathit{failure})$
2. *Data-flow functional*: $[\![\ ]\!]_{av} : E \rightarrow (\mathcal{B}_X \rightarrow \mathcal{B}_X)$ defined by

$$\forall\, e \in E.\ [\![\, e \,]\!]_{av} =_{df} \begin{cases} Cst^X_{true} & \text{if } Comp_{\,e} \wedge Transp_{\,e} \\ Id_{\mathcal{B}_X} & \text{if } \neg Comp_{\,e} \wedge Transp_{\,e} \\ Cst^X_{false} & \text{otherwise} \end{cases}$$

Reverse abstract semantics for *availability*:

1. *Data-flow lattice*: $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top) =_{df} (\mathcal{B}_X, \wedge, \vee, \leq, \mathit{false}, \mathit{failure})$
2. *Reverse data-flow functional*: $[\![\ ]\!]_{av_R} : E \rightarrow (\mathcal{B}_X \rightarrow \mathcal{B}_X)$ defined by

$$\forall\, e \in E.\ [\![\, e \,]\!]_{av_R} =_{df} \begin{cases} R\text{-}Cst^X_{true} & \text{if } [\![\, e \,]\!]_{av} = Cst^X_{true} \\ R\text{-}Id_{\mathcal{B}_X} & \text{if } [\![\, e \,]\!]_{av} = Id_{\mathcal{B}_X} \\ R\text{-}Cst^X_{false} & \text{if } [\![\, e \,]\!]_{av} = Cst^X_{false} \end{cases}$$

Table 1. Abstract and reverse abstract semantics for availability.

Obviously, constant functions and the identity on a lattice are distributive. This holds for the extended functions Cst^X_{true} and Cst^X_{false} on $\mathcal{B}_X$, too. We have (cf. Lemma 1):

Lemma 3. *$[\![\ ]\!]_{av}$ is distributive, and $[\![\ ]\!]_{av_R}$ is additive.*

Denoting the set of functions $\{R\text{-}Id_{\mathcal{B}_X}, R\text{-}Cst^X_{true}, R\text{-}Cst^X_{false}\}$ by $\mathcal{F}^{\mathcal{B}_X}_{\mathcal{R}}$, we introduce next two orders on $\mathcal{F}^{\mathcal{B}_X}_{\mathcal{R}}$ denoted by $\sqsubseteq_{seq}$ and $\sqsubseteq_{par}$. Their subscripts remind to *seq*uential and *par*allel, respectively. Intuitively, this is because the join operations $\sqcup_{seq}$ and $\sqcup_{par}$, which are implicitly defined by these orders, model the merge of information at "sequential" and "parallel" synchronization points. This will be important for the reverse fixed-point analysis complementing the *R-PJOP*-approach (cf. Section 4.3).

(1) $R\text{-}Cst^X_{true} \sqsubseteq_{seq} R\text{-}Id_{\mathcal{B}_X} \sqsubseteq_{seq} R\text{-}Cst^X_{false}$ (2) $R\text{-}Id_{\mathcal{B}_X} \sqsubseteq_{par} R\text{-}Cst^X_{true} \sqsubseteq_{par} R\text{-}Cst^X_{false}$

Note that $\sqsubseteq_{seq}$ is the pointwise order on functions. For convenience, we will sometimes drop the index *seq*. This means $\sqcup$ extends to $\sqcup_{seq}$.

4.2 The Reverse Functional Approach for Sequential Programs

Fundamental for capturing the reverse semantics of parallel statements is Equation System 1. It can be considered the reverse counterpart of the equation system characterizing the functional approach of Sharir and Pnueli for interprocedural DFA of sequential programs (cf. [25]). It allows us to capture the reverse effect of a sequential program G. Note that it is formulated for arbitrary data-flow problems, not just for GEN/KILL-problems.

Equation System 1 (Reverse Functional *R-MinFP*-Approach for Seq. Programs).

$$[\![\, n \,]\!]^q_R = \begin{cases} Id_{\mathcal{C}} & \text{if } n = \mathbf{q} \\ \bigsqcup\{[\![\, (n,m) \,]\!]_R \circ [\![\, m \,]\!]^q_R \mid m \in succ(n)\} & \text{otherwise} \end{cases}$$

Denoting the least solution of this equation system by $[\![\ ]\!]^q_R : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$, too, we can prove the following coincidence result. It is the reverse analogue of a theorem introduced by Sharir and Pnueli in the field of conventional interprocedural DFA (cf. [25]).

Theorem 2 (Reverse Functional Coincidence).
Let $[\![\ ]\!]$ be a distributive data-flow functional. Then we have:

$$\forall\, c_q \in \mathcal{C}\ \forall\, n \in N.\ R\text{-}JOP_{c_q}(n) = [\![\, n \,]\!]^q_R(c_q)$$

In the following, we identify the reverse semantics of a sequential program G, in symbols $[\![\, G \,]\!]_R$, with the global semantics of its start node, i.e., $[\![\, start(G) \,]\!]^{end(G)}_R$, considering the end node of G as query node. This will be important to compute the reverse effect of (possibly nested) parallel statements (cf. Section 4.3).

4.3 The *R-PMinFP$_{BV}$*-Approach

In this section we restrict again our attention to GEN/KILL-problems, and show how to capture for these problems the challenges posed by interference and synchronization elegantly and efficiently. As for its conventional counterpart, the key for this is the observation that for a GEN/KILL-problem the reverse effect of a complete path is given by a single statement: the first statement, whose semantics is different from the identity. The Reverse Main Lemma 4, which is the counterpart of the Main Lemma of [21] for conventional DFA, provides the formal foundation of this observation.

Lemma 4 (Reverse Main Lemma).
Let $f_i : \mathcal{B}_X \rightarrow \mathcal{B}_X$, $1 \leq i \leq q$, $q \in I\!N$, be functions of $\mathcal{F}^{\mathcal{B}_X}_{\mathcal{R}}$. Then we have:

$$\exists\, k \in \{1, \ldots, q\}.\ f_q \circ \ldots \circ f_2 \circ f_1 = f_k\ \wedge\ \forall\, j \in \{1, \ldots, k-1\}.\ f_j = Id_{\mathcal{B}_X}.$$

Interference Considering interference, the relevance of the Reverse Main Lemma 4 is that it restricts the way of possible interference within a parallel program: each possible interference is due to a single statement within a parallel component. Combining this observation with the fact that for $e \in \mathit{InterfEdges}(n)$, there is a parallel path leading to n whose last step requires the execution of e, this yields that the potential of interference, which in general would be given in terms of paths, is fully characterized by the set of interference edges. In fact, considering the computation of the reverse of universal properties which are described by minimal fixed points (the computation of the reverse of existential properties requires the dual argument), the obvious existence of a path to n that does not require the execution of any edge of $\mathit{InterfEdges}(n)$ implies that the only effect of interference is "destruction." As in [21], this is captured by a predicate *DestructInference*, whose denomination reminds to "Destructive Interference." It is defined for each node $n \in N$ by

$$\mathit{DestructInference}(n) \Longleftrightarrow_{df} \exists\, e \in \mathit{InterfEdges}(n).\ [\![\, e \,]\!] = \mathit{Cst}^X_{\mathit{false}}.$$

Intuitively, this predicate indicates if there is an edge of a parallel component destroying the property under consideration. In Equation System 2 presented later only the constant function of its negation denoted by *NoDestructInterf* is used to model interference. Theorem 4 ensures that this is sufficient to capture the effects of interference.

Synchronization Intuitively, synchronization reflects the fact that control can leave a parallel statement only after the termination of all its parallel components. Considering the reverse counterpart of a forward analysis (as we are doing), synchronization takes place at nodes of $N^N_{\mathcal{P}}$. The data-flow fact describing the situation after synchronization can be computed in a hierarchical fashion, where only purely sequential programs need to be considered. This is the reason for the relevance of the functional approach presented in Section 4.2. The central idea underlying this process coincides with that of interprocedural analysis (cf. [25]): we need to compute the reverse effect of complete subgraphs. In this case, the reverse effect of complete parallel components. Straightforwardly, this information is computed in an "innermost" fashion and then propagated to the next surrounding parallel statement. The following definition describes the complete three-step procedure for a maximal parallel statement in G. The predicate *Harmful* computed in the course of this process allows us to conclude on the value of the predicate *DestructInference*. Note, if the query node q is contained in the considered parallel statement, then the reverse effect of its enclosing component for this node dominates the synchronization process.

Reverse Hierarchical Preprocess

1. If q is a node of the considered maximal parallel statement, initialize $\hat{q}$ with q.
2. Terminate, if $\hat{G}$ does not contain any parallel statement. Otherwise, select successively all maximal flow graphs G' occurring in a graph of $\mathcal{G}_{\mathcal{P}}(\hat{G})$ that do not contain any parallel statement, and determine the effect $[\![\![\, G' \,]\!]\!]_R$ of this (purely sequential) graph according to the Reverse Equation System 1. If $\hat{q} \in \mathit{Nodes}(G')$ then compute $[\![\![\, \mathit{start}(G') \,]\!]\!]^{\hat{q}}_R$, and set $\tilde{q} := \hat{q}$ and $\hat{q} := \mathit{end}(\mathit{pfg}(G'))$. Moreover, keep track if G' contains a destroying instruction, i.e., if it is harmful to its parallel relatives

$$\mathit{Harmful}(\mathit{start}(G')) = \bigvee_{e \in \mathit{Edges}(G')} [\![\, e \,]\!] = \mathit{Cst}^X_{\mathit{false}} \ \vee \bigvee_{n \in \mathit{Nodes}(G')} \mathit{Harmful}(n)$$

3. Compute the effect $[\![\,\bar{G}\,]\!]_R^*$ of the innermost parallel statements $\bar{G}$ of $\hat{G}$ by

$$[\![\,\bar{G}\,]\!]_R^* = \begin{cases} \bigsqcup_{par}\{[\![\,G'\,]\!]_R \,|\, G' \in \mathcal{G}_{\mathcal{C}}(\bar{G})\} & \text{if } \tilde{q} \notin \mathit{Nodes}(\bar{G}) \\ \mathit{R\text{-}Cst}^X_{\mathit{false}} \quad \text{if } \tilde{q} \in \mathit{Nodes}(\bar{G}) \wedge \exists e \in \mathit{InterfEdges}(\tilde{q}).\ [\![\,e\,]\!] = \mathit{Cst}^X_{\mathit{false}} \\ [\![\,\mathit{start}(G')\,]\!]_R^{\hat{q}} \quad \text{if } \tilde{q} \in \mathit{Nodes}(\bar{G}) \wedge \forall e \in \mathit{InterfEdges}(\tilde{q}).\ [\![\,e\,]\!] \neq \mathit{Cst}^X_{\mathit{false}} \end{cases}$$

4. Replace all innermost parallel statements $\bar{G} = (\bar{N}, \bar{E}, \bar{\mathbf{s}}, \bar{\mathbf{e}})$ in $\hat{G}$ by $(\{\bar{\mathbf{s}}, \bar{\mathbf{e}}\}, \{(\bar{\mathbf{s}}, \bar{\mathbf{e}})\}, \bar{\mathbf{s}}, \bar{\mathbf{e}})$, and define the local semantics of $(\bar{\mathbf{s}}, \bar{\mathbf{e}})$ by $[\![\,\bar{G}\,]\!]_R^*$. Continue with step 2.

Intuitively, this three-step algorithm is a hierarchical adaptation of the algorithm for computing the reverse functional version of the *R-MinFP*-solution for the sequential case (cf. [17]). In the context of the complete approach it is applied to maximal parallel statements of a program, a step which can be done in parallel as there is no overlap among them. The first and the third step of this process are rather straightforward. The correctness of the second step realizing the synchronization at nodes in $N_{\mathcal{P}}^N$ is essentially a consequence of Lemma 5 and Theorem 3 given in Section 4.4. Additional care is only required, if the query node lies in a parallel part of the program. Next, we present the parallel version of the reverse minimal-fixed-point approach for GEN/KILL-problems.

The *R-PMinFP*$_{BV}$-Solution Fundamental for the (full) *R-PMinFP*$_{BV}$-approach is Equation System 2, which relies on the results of the hierarchical preprocess. For the purpose of demand-driven analysis, however, this premise can be weakened and the computation process more economically be organized. Note that the join occurring in the defining equation for the copy of the query node is only relevant, if the query node is located inside of a parallel statement. In this case, it reflects the potential impact the parallel relatives may have on it. If it is located in the sequential part of the program, the join is always trivial yielding $Id_{\mathcal{B}_X}$.

Equation System 2 (The Reverse Functional *R-PMinFP*$_{BV}$-EQS).

$$[\![\,n\,]\!]_R = \begin{cases} Id_{\mathcal{B}_X} \sqcup \mathit{R\text{-}Cst}^X_{\mathit{NoDestructInterf}(q)} & \mathit{if}\ n = \mathbf{q} \\ [\![\,\mathit{pfg}(n)\,]\!]_R^* \circ [\![\,\mathit{end}(\mathit{pfg}(n))\,]\!]_R \sqcup \mathit{R\text{-}Cst}^X_{\mathit{NoDestructInterf}(n)} & \mathit{if}\ n \in N_{\mathcal{P}}^N \\ \bigsqcup\{\,[\![\,(n,m)\,]\!]_R \circ [\![\,m\,]\!]_R \,|\, m \in \mathit{succ}(n)\} \sqcup \mathit{R\text{-}Cst}^X_{\mathit{NoDestructInterf}(n)} & \mathit{otherwise} \end{cases}$$

Denoting by $[\![\ ]\!]_R^*$ the least solution of this equation system, the *R-PMinFP*$_{BV}$-solution for GEN/KILL-problems is defined as follows:

The *R-PMinFP*$_{BV}$-Solution: *R-PMinFP*$_{BV}$ $: N \to \mathcal{F}^{\mathcal{B}_X}$ defined by

$$\forall\, c_q \in \mathcal{B}_X\ \forall\, n \in N.\ \mathit{R\text{-}PMinFP}_{BV\,c_q}(n) = [\![\,n\,]\!]_R^*(c_q).$$

It is worth noting that an algorithm computing the *R-PMinFP*$_{BV}$-solution could be implemented in the same fashion as its counterpart for conventional DFA of [21], i.e., as a two stage process computing first the reverse effect of parallel statements, and subsequently the *R-PMinFP*$_{BV}$-solution using the results of the first stage. Note, however, that for GEN/KILL-problems, *true* is the only nontrivial query for reverse DFA. The outcomes for the queries *false* and *failure* are immediately obvious: (1) $\textit{R-PMinFP}_{BV\,\textit{false}}(n) = \textit{false}$ for each $n \in N$, and (2) $\textit{R-PMinFP}_{BV\,\textit{failure}}(n) = \textit{failure}$, if n occurs on a path from $\mathbf{s}$ to $\mathbf{q}$, and *false*, otherwise. Together with Corollary 1, this will be the basis of a more efficient algorithm for demand-driven DFA.

4.4 The Reverse Coincidence Theorem

The Reverse Coincidence Theorem 4 presented in this section allows us to bridge the gap to Theorem 1. It states that for GEN/KILL-problems the $\textit{R-PMinFP}_{BV}$-solution coincides with the *R-PJOP*-solution. Important for its proof are Lemma 5 and Theorem 3.

Lemma 5. *Let $\bar{G}$ be a parallel flow graph, which only consists of purely sequential parallel components. Then the R-PJOP-solution of $\bar{G}$ is given by:*

$$\textit{R-PJOP}_{c_{end(\bar{G})}}(start(\bar{G})) = \bigsqcup_{par}\{ [\![\, G' \,]\!]_R \mid G' \in \mathcal{G}_{\mathcal{C}}(\bar{G})\}$$

Theorem 3 (Reverse Hierarchical Bitvector Coincidence).
Let $G' \in \mathcal{G}_{\mathcal{P}}(G)$ be a parallel flow graph, and let $[\![\ ]\!] : E \rightarrow \mathcal{F}_{\mathcal{R}}^{\mathcal{B}_X}$ be the reverse data-flow functional of a GEN/KILL-problem. Then we have: $\textit{R-PJOP}(start(G')) = [\![\, G' \,]\!]_R^*$.

Theorem 4 (Reverse Parallel Bitvector Coincidence).
Let $[\![\ ]\!]_R : E \rightarrow \mathcal{F}_{\mathcal{R}}^{\mathcal{B}_X}$ be the reverse data-flow functional of a GEN/KILL-problem. Then the $\textit{R-PMinFP}_{BV}$-solution and the R-PJOP-solution coincide, i.e.,

$$\forall\, n \in N.\ \textit{R-PMinFP}_{BV}(n) = \textit{R-PJOP}(n)$$

Together with Theorem 4 and Theorem 1, we obtain:

Corollary 1. *Let $[\![\ ]\!] : E \rightarrow \mathcal{F}^{\mathcal{B}_X}$ be the data-flow functional of a GEN/KILL-problem, and let $c_{\mathbf{s}}, c_q \in \mathcal{B}_X$. Then we have: $\textit{R-PMinFP}_{BV\,c_q}(\mathbf{s}) = wci_{c_q} \sqsubseteq c_{\mathbf{s}} \iff \textit{PMOP}_{c_{\mathbf{s}}}(q) \sqsupseteq c_q$*

In the following section, we show how to use reverse DFA to construct an algorithm for demand-driven DFA.

4.5 Demand-Driven DFA based on Reverse DFA

Considering GEN/KILL-problems, we are interested in the set of program points satisfying the property under consideration with respect to a context information assured to be valid at the program entry $\mathbf{s}$. For a specific program point, Corollary 1 suggests to solve this problem by computing the $\textit{R-PMinFP}_{BV}$-solution for this query node and comparing its value at the program entry with the context information there assured. The property holds at the query node if and only if it is a lower bound of this context information. Following this approach, the computation of the strongest component information for a program point is reduced to a verification problem. In fact, this is the essence of DD-DFA based on reverse DFA. Moreover, it is the key allowing us a more economic organization of the computation process. Above all, the computation of the reverse effects of parallel statements need not be computed in advance, but by need only. Moreover, as soon as the invalidity of a data-flow query is detected, i.e., essentially, whenever "failure" needs to be ensured at some program point, the computation of the $R\text{-}PMinFP_{BV}$-solution can be stopped.

Pragmatically, it is important that this approach can easily be parallelized according to two orthogonal principles. First, on the level of queries: different queries can be processed in parallel. Second, on the level of the fixed-point computation: it can be organized such that the entries of the workset controling the fixed-point computation are processed in parallel. This is particularly beneficial in a setting providing parallel processes. It offers the opportunity to parallel DFA of parallel programs. In the setting here, the ease of sharing answers to data-flow queries computed by other processes results in a further speed-up of the analysis. Though demand-driven approaches have generally the same worst case time complexity as their conventional counterparts, in practice they are typically much more efficient (cf. [8]). In the border case, it here allows us to compute the information provided by conventional DFA at the usually much smaller costs of answering a data-flow query. In practice, this can lead to a substantial speed-up of analysing and optimizing explicitly parallel programs.

5 Conclusions

In this article, we focused on the theoretical foundation underlying the approach of [17] for demand-driven data-flow analysis of explicitly parallel programs. Basically, it relies on complementing the "strongest postcondition" view underlying conventional DFA with the "weakest precondition" view underlying reverse DFA. For GEN/KILL-problems this allows us to reduce the computation of strongest component informations conventional DFA is aiming at to a (set of) verification problem(s) efficiently solvable by reverse DFA, and thus to compute the information provided by conventional DFA demand-drivenly. Pragmatically, it is important that this approach can easily be parallelized. In the border case, this allows us to compute the full information provided by conventional DFA at the usually much smaller costs of answering a data-flow query. This is important in practice, since optimizations rely typically on the complete information. Apart from this, approaches for demand-driven program analysis like the one presented here are particularly well-suited for applications requiring information only for selected program points. Following [8], this includes (1) *interactive tools* handling user queries, as e.g., debugging tools, (2) *"hot spot" optimizers* focusing on specific program parts like e.g. loops, and (3) *incremental tools* selectively updating data-flow information after incremental program modifications. Demand-driven approaches are thus a valuable means for a variety of tools supporting the software development process.

References

1. K. R. Apt. Ten years of Hoare's logic: A survey – Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431 – 483, 1981.
2. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. 2nd edition, Springer-Verlag, Heidelberg, Germany, 1997.
3. J.-H. Chow and W. L. Harrison. Compile time analysis of parallel programs that share memory. In *Conference Record of the 19th Annual Symposium on Principles of Programming Languages* (*POPL'92*), pages 130 – 142. ACM, New York, 1992.
4. P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324 – 328, 1996.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual*

Symposium on Principles of Programming Languages (*POPL'77*) (*Los Angeles, California*), pages 238 – 252. ACM, New York, 1977.
6. E. Duesterwald. *A Demand-driven Approach for Efficient Interprocedural Data Flow Analysis*. PhD thesis, University of Pittsburgh, Department of Computer Science, Pittsburgh, Pennsylvania, 1996.
7. E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of the 22nd Annual Symposium on Principles of Programming Languages* (*POPL'95*) (*San Francisco, California*), pages 37 – 48. ACM, New York, 1995.
8. E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992 – 1030, 1997.
9. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
10. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.
11. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*FSE-3*), pages 104 – 115, 1995.
12. J. Hughes and J. Launchbury. Reversing abstract interpretations. In *Proceedings of the 4th European Symposium on Programming* (*ESOP'92*) (*Rennes, France*), Lecture Notes in Computer Science, vol. 582, pages 269 – 286. Springer-Verlag, Heidelberg, Germany, 1992.
13. J. Hughes and J. Launchbury. Reversing abstract interpretations. *Science of Computer Programming*, 22:307 – 326, 1994.
14. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305 – 317, 1977.
15. G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the 1st Annual Symposium on Principles of Programming Languages* (*POPL'73*) (*Boston, Massachusetts*), pages 194 – 206. ACM, New York, 1973.
16. J. Knoop. Eliminating partially dead code in explicitly parallel programs. *Theoretical Computer Science*, 196(1-2):365 – 393, 1998. (Special issue devoted to *Euro-Par'96*).
17. J. Knoop. Parallel data-flow analysis of explicitly parallel programs. In *Proceedings of the 5th European Conference on Parallel Processing* (*Euro-Par'99*) (*Toulouse, France*), Lecture Notes in Computer Science, vol. 1685, pages 391 – 400. Springer-Verlag, Heidelberg, Germany, 1999.
18. J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.
19. J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'94*) (*Orlando, Florida*), volume *29*,6 of *ACM SIGPLAN Notices*, pages 147 – 158, 1994.
20. J. Knoop and B. Steffen. Code motion for explicitly parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP'99*) (*Atlanta, Georgia*), volume *34*,8 of *ACM SIGPLAN Notices*, pages 13 – 24, 1999.
21. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268 – 299, 1996.
22. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96 – 103, 1979.
23. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, California, 1997.
24. T. Reps. Solving demand versions of interprocedural analysis problems. In *Proceedings of the 5th International Conference on Compiler Construction* (*CC'94*) (*Edinburgh, UK*), Lecture Notes in Computer Science, vol. 786, pages 389 – 403. Springer-V., Heidelberg, Germany, 1994.
25. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189 – 233. Prentice Hall, Englewood Cliffs, New Jersey, 1981.