An Automata-theoretic Approach to Interprocedural Data-flow Analysis

Javier Esparza[1] and Jens Knoop[2]

[1] Technische Universität München, Arcisstr. 21, D-80290 München, Germany
e-mail: esparza@in.tum.de
[2] Universität Dortmund, Baroper Str. 301, D-44221 Dortmund, Germany
e-mail: knoop@ls5.cs.uni-dortmund.de

Abstract. We show that recent progress in extending the automata-theoretic approach to model-checking beyond the class of finite-state processes finds a natural application in the area of interprocedural data-flow analysis.

Keywords: Interprocedural data-flow analysis, model-checking, automata theory, program optimisation.

1 Introduction

Recent work [15, 24] has shown that model-checking algorithms for abstract classes of infinite-state systems, like context-free processes [1, 5] and pushdown processes [6], find a natural application in the area of data-flow analysis (DFA) for programming languages with procedures [16], usually called interprocedural DFA. A large variety of DFA problems, whose solution is required by optimising compilers in order to apply performance improving transformations, can be solved by means of a unique model-checking technique.

The techniques of [5, 6] are based on what could be called the fixpoint approach to model-checking [24], in which the set of states satisfying a temporal property is defined and computed as a fixpoint in an adequate lattice. Some years ago, Vardi and Wolper presented in a seminal paper [25] an alternative automata-theoretic approach in which—loosely speaking—verification problems are reduced to the emptyness problem for different classes of automata. This approach has had considerable success for finite-state systems, and constitutes the theoretical basis of verification algorithms implemented in tools like SPIN [13], PROD [26], or PEP [27]. Recently, the approach has also been extended to context-free processes and pushdown processes [4, 10], and to other infinite-state classes able to model parallelism [18].

The goal of this paper is to show that the techniques derived from these recent developments can also be applied to DFA. We provide solutions for the interprocedural versions of a number of important DFA problems, starting with the class of so-called bitvector problems. On the one hand, the structural simplicity of these problems allows us a gentle way of introducing our approach. On the other hand, these problems are quite important as they are the prerequisite of numerous optimisations like *partially redundant expression elimination*, *partially redundant assignment elimination*, *partially dead code elimination*, and *strength reduction* [23], which are widely used in practice. In detail, we investigate:

(a) the four problems of Hecht's taxonomy of bitvector problems [12],
(b) the computation of faint variables, and
(c) the problems of (a) for parallel languages.

In contrast to (a), for which there exist several solutions in the literature, (b) and (c) have—to the best of our knowledge—not been considered yet in an interprocedural setting; solutions for the intraprocedural case can be found in [11, 14] for (b), and in [17] for (c).

The paper is organised as follows. Section 2 contains an informal introduction to DFA, recalls the DFA problems mentioned above, and in particular presents the flow graph model. Section 3 gives flow graphs a structured operational semantics. Sections 4, 5, and 6 present the solutions to the problems (a), (b) and (c) above, respectively, and Section 7 contains our conclusions.

2 Data-flow Analysis

Intuitively, *data-flow analysis (DFA)* is concerned with deciding run-time properties of programs without actually executing them, i.e., at compile time. Basically, the properties considered can be split into two major groups (cf. [12]). Properties whose validity at a program point depends on the program's *history*, i.e., on the program paths reaching it, and properties whose validity depends on the program's *future*, i.e., on the suffixes of program paths passing it. Both groups can further be split into the subgroups of *universally* and *existentially* quantified properties, i.e., whose validity depends on *all* or on *some* paths, respectively.

Background. Using the standard machinery of DFA (cf. [12]), the validity of a property at a program point n is deduced from a data-flow fact computed for n. This fact reflects the meaning of the program at n with respect to an abstract, simpler version of the "full" program semantics, which is tailored for the property under consideration. The theory of *abstract interpretation* provides here the formal foundation (cf. [7–9, 19]). In this approach the data-flow facts are given by elements of an appropriate lattice, and the abstract semantics of statements by transformations on this lattice. The *meet-over-all-paths* (*MOP*) semantics defines then the reference solution, i.e., the data-flow fact desired for a program point n: It is the "meet" (intersection) of all data-flow facts contributed by all program paths reaching n. The *MOP*-semantics is conceptually quite close to the program property of interest, but since there can be infinitely many program paths reaching a program point it does not directly lead to algorithms for the computation of data-flow facts. Therefore, in the traditional DFA-setting the *MOP*-semantics is approximated by the so-called *maximal-fixed-point* (*MFP*) semantics. It is defined as the greatest solution of a system of equations imposing consistency constraints on an annotation of the program points with data-flow facts. The *MFP*-semantics coincides with its *MOP*-counterpart when the functions specifying the abstract semantics of statements are distributive, a result known as the Coincidence Theorem.

Note that the *MOP*-semantics is defined in terms of possible program executions. From the point of view of temporal logic this means in a pathwise, hence *linear time* fashion. In contrast, the computation of the *MFP*-semantics proceeds by consistency checks taking the immediate neighbours of a program point simultanenously into account. From the point of view of temporal logic this means in a tree-like, hence *branching time* fashion. Thus, in the traditional DFA-setting there is a gap between the reference semantics defining the data-flow facts of the program annotation desired (the *MOP*-semantics), and the semantics the computation of the program annotation with data-flow facts relies on (the *MFP*-semantics). An important feature of the automata-theoretic approach to DFA we are going to present here is the absence of a similar separation of concerns providing in this respect a more natural and conceptually simpler access to DFA.

Flow graphs. In DFA programs are commonly represented by systems of *flow graphs*, where every flow graph represents a procedure of the underlying program. Flow graphs are directed graphs, whose nodes and edges represent the statements and the intraprocedural control flow of the procedure represented. Usually, control flow is nondeterministically interpreted in order to avoid undecidabilities. As illustrated in Figure 1(a) and (b), which show the flow graph and the flow graph system representing a single procedure and a program with procedures, we consider edge-labelled flow graphs, i.e., the edges represent both the statements and the control flow, while nodes represent program points. We assume that statements are assignments of the form $v := t$ including the empty statement, call statements of the form *call* $\Pi(t_1, \ldots, t_n)$, or output operations of the form $out(t)$, where v is a variable, Π a procedure identifier, and $t, t_1, \ldots, t_n$ are terms.

Bitvector properties and faint variables. Bitvector properties correspond to structurally particularly simple DFA-problems, which, simultaneously, are most important in practice because of the broad variety of optimisations based on them (cf. Section 1). Their most prominent representatives, the *availability* and *very busyness* of terms, the *reachability* of program points by definitions, and the *liveness* of variables, span the complete space of the taxonomy recalled above as was shown by Hecht [12]. Intuitively, a term t is *available* at a program point n, if on all program paths reaching n term t is computed without that any of its operands is assigned a new value afterwards. Thus, availability is a universally quantified history-dependent property. Very busyness is its dual counterpart. A term t is *very busy* at a program point n, if it is computed on all program paths passing n and reaching the end node before any of its operands is assigned a new value after leaving n. Hence, very busyness is a universally quantified future-dependent property. For illustration consider Figure 1(a), in which the program points where $a + b$ is very busy are greyly highlighted.

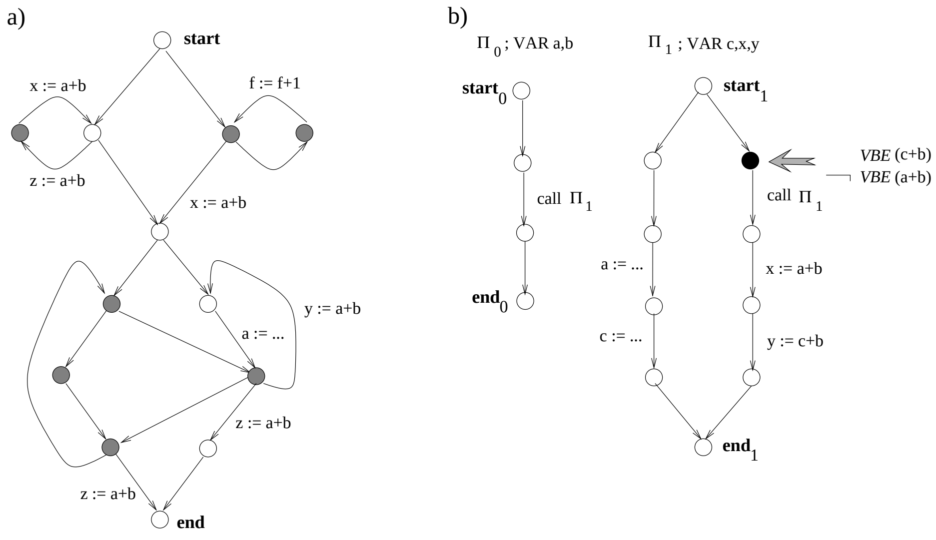

Fig. 1. Flow graphs and flow graph systems.

Reaching definitions (for convenience referred to as reachability later) and live variables are existentially quantified history- and future-dependent properties. Intuitively, a program point n is *reached* by the definition of a particular edge e if there is a program path across e reaching n which after passing e is free of definitions of the left-hand side variable of the definition of e. A variable v is *live* at a program point n, if on some program path passing n there is a reference to v, which after leaving n is not preceded by a redefinition of v. Conversely, a variable is *dead* at a program point, if it is not live at it.

This latter property is well-suited in order to illustrate how bitvector properties can be used for optimisation. Every assignment whose left-hand side variable is dead is "useless," and can be eliminated because there is no program continuation on which its left-hand side variable is referenced without a preceding redefinition of it. This is known as *dead-code elimination*.

Like the bitvector problems recalled above, DFA-problems are often concerned with sets of program items like terms, variables, or definitions. Characteristic for bitvector problems, however, is that they are "separable (decomposable):" The validity of a bitvector property for a specific item is independent of that of any other item. This leads to particularly simple formulations of bitvector problems on sets of items (and implementations in terms of bitvectors).

Faintness is an example of a program property which lacks the decomposability property. Intuitively, faintness generalizes the notion of dead variables. A variable f is faint at a program point if on all program continuations any right-hand side occurrence of f is preceded by a modification of f, or occurs in an assignment whose left-hand side variable is faint, too. A simple example of a faint but not dead variable is the variable f in the assignment $f := f + 1$, assuming that the assignment occurs in a loop without any other occurrence of f elsewhere in the program (cf. Figure 1(a)). Assignments to faint variables can be eliminated as useless like those to dead ones. Whereas deadness, however, is a bitvector property, faintness is not. It is not separable preventing the computation of faintness for a variable in isolation.

DFA in the interprocedural and parallel setting. DFA is particularly challenging in the presence of recursive procedures—because of the existence of potentially infinitely many copies of local variables of recursive procedures at run-time—and parallel statements—because of the phenomena of interference and synchronisation. In the following we illustrate this by means of the programs of Figure 1(b), 2, and 3 using very busyness of the terms $a + b$ and $c + b$ as example.

In the example of Figure 1(b), the term $c + b$ is very busy at the program point preceding the recursive call of Π_1 in procedure Π_1, while $a + b$ is not. The difference lies in the fact that in the case of $a + b$ a global operand is modified within the recursive call, while it is a local one in the case of $c + b$. Thus, very busyness of $c + b$ is not affected because the assignment $c := \ldots$ modifies a new incarnation of c. In fact, after returning from the procedure call, the incarnation of c which has been valid when calling Π_1 is valid again, and, of course, it has not been modified. In contrast, the modification of a affects a global variable, and hence, the modification survives the return from the call. Thus, computing $a + b$ after the recursive call will yield a different value than computing it before this call. Similar phenomena can be observed for the other bitvector problems. This is illustrated in Figure 2 and Table 1, which summarizes for specific pairs of program points and program items the availability, reachability and liveness information. In the framework of [16] these obstacles of interprocedural DFA are overcome by mimicking the behaviour of the run-time stack by a corresponding DFA-stack, and additionally, by keeping track if modifications of operands affect a global or a local variable. So-called return functions, which are additionally introduced in this setting enlarging the specification of an abstract semantics, extract this information from the DFA-informations valid at call time and valid immediately before returning from the called procedure, which allows a proper

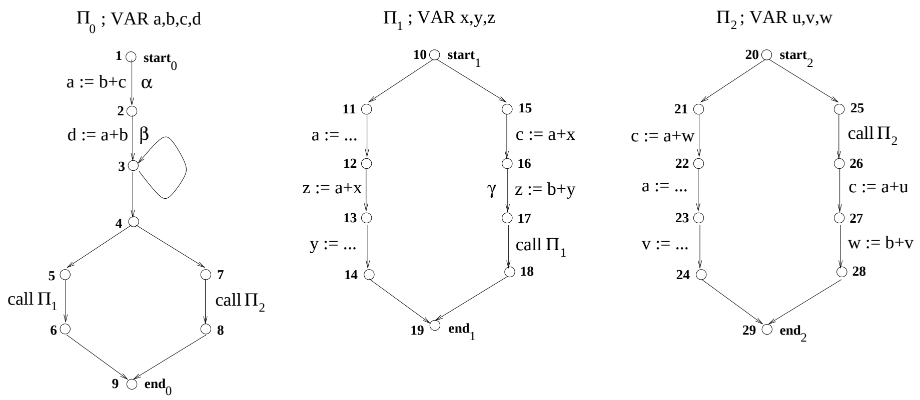

Fig. 2. The sequential interprocedural setting.

Program Point	Availability				Reaching Definitions		
	$a+b$	$b+c$	$a+x$	$b+y$	$\alpha: a :=$	$\beta: d :=$	$\gamma: z :=$
5	tt	tt	—	—	tt	tt	—
6	ff	ff	—	—	ff	tt	—
17	tt	ff	tt	tt	tt	tt	tt
18	ff	ff	ff	tt	ff	tt	tt

Program Point	Liveness			
	b	c	v	w
7	tt	ff	—	—
8	ff	ff	—	—
25	tt	ff	tt	ff
26	tt	ff	tt	ff

Table 1. Values of some bitvector problems.

treatment of programs with both local and global variables. In this paper we present a different approach to this problem.

Consider now Figure 3, and imagine that the three procedures shown in (a) are embedded either into a sequential (b) or parallel (c) program context, respectively. The pattern of very busy program points is different because of the effects of interference and synchronisation. While in (b) the term $a+b$ is very busy at nodes **1**, **7**, and **8**, in (c) it is very busy at nodes **1** and **10**. DFA of programs with explicit parallelism have attracted so far little attention, possibly because naive adaptations of the sequential techniques typically fail [21], and the costs of rigorous straightforward adaptations are prohibitive because of the number of interleavings expressing the possible executions of a parallel program. Though for an intraprocedural setting it could recently be shown that bitvector problems are an exception, which can be solved as easily and as efficiently as their sequential counterparts [17], a corresponding result for a setting with procedures has not yet been presented.

Conventions. Without loss of generality we make the following assumptions on flow graphs, which allow us a simpler notation during the presentation of the automata-theoretic approach. Flow graphs have a unique start node and end node without incoming and outgoing edges, respectively. Each node of a flow graph lies on a path connecting its start node and end node. The main procedure of a program cannot be called by any procedure of the program. Procedures are not statically nested. Edges leaving (approaching) a node with more than one successor (predecessor) are labelled by the empty statement. And, finally, the left-hand-side variable of an assignment does not occur in its right-hand-side term.

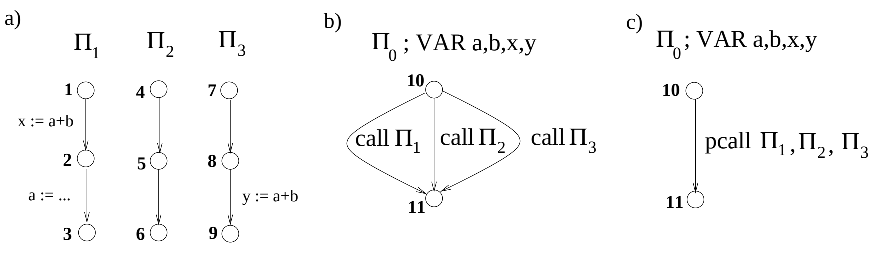

Fig. 3. The parallel interprocedural setting.

3 A Structured Operational Semantics for Flow Graph Systems

In this section we give flow graph systems a formal semantics very much in the style of the operational semantics of process algebras (see for instance [3]). Intuitively, we interpret a node of a flow graph as an *agent* that can execute some *actions*, namely the labels of the edges leaving it. The execution of an action transforms an agent into a new one. The actions that an agent can execute and the result of their execution are determined by *transition rules*. So, for instance, for a flow graph edge of the form $n \xrightarrow{v:=3} n'$, interpreted as "the agent n can execute the action $v := 3$, and become the agent n'," we introduce the rule $N \xrightarrow{v:=3} N'$ (uppercase letters are used to avoid confusion). In order to model procedure calls we introduce a sequential composition operator on agents with the following intended meaning: The sequential composition of N_1 and N_2, denoted by the concatenation $N_1 \cdot N_2$, is the agent that behaves like N_1 until it terminates, and then behaves like N_2. It is now natural to assign to an edge $n \xrightarrow{call\ \Pi_i(T)} n'$, where T is a vector $(t_1, \ldots, t_k)$ of terms, the rule $N \xrightarrow{\Pi(T)} START_i \cdot N'$ (the name of the action is shortened for readability)[1].

Let us now formally define the semantics. We associate to a flow graph system a triple (Con, Act, Δ), called a *process system*[2], where Con is a set of *agent constants*, Act is a set of *actions*, and $\Delta \subseteq Con^+ \times Act \times Con^*$ is a set of *transition rules*. An *agent* over Con is a sequential composition of agent constants, seen as an element of Con^*. In particular, the empty sequence ϵ is an agent. The set Δ induces a reachability relation $\xrightarrow{a} \subseteq Con^* \times Con^*$ for each $a \in Act$, defined as the smallest relation satisfying the following inference rules:

- if $(P_1, a, P_2) \in \Delta$, then $P_1 \xrightarrow{a} P_2$;
- if $P_1 \xrightarrow{a} P_1'$ then $P_1 \cdot P_2 \xrightarrow{a} P_1' \cdot P_2$ for every $P_2 \in Con^*$.

The second rule captures the essence of sequential composition: Only the first constant of a sequence can perform an action. Since the left-hand side of a rule cannot be empty, the agent ϵ cannot execute any action, and so it corresponds to the terminated agent.

In the sequel we overload the $\xrightarrow{a}$ symbol and write $P_1 \xrightarrow{a} P_2$ instead of $(P_1, a, P_2) \in \Delta$.

We associate to a flow graph system the process system (Con, Act, Δ) where Con is the set of program nodes plus a special agent constant $START$, Act is the set of edge labels plus special actions τ, *start*, *end*, and end_i for each procedure Π_i, and Δ contains the rules shown in Table 2. Observe that the left-hand sides of the rules of Δ have length 1, and that all terminating executions of the flow graph system begin with the action $start_0$ and end with end_0.

---

[1] Recall that $start_i$ is the start node of the flow graph Π_i.

[2] Process systems are very close the Basic Process Algebra of [2] or the context-free processes of [5]. We use another name due to small syntactic differences.

Flow graph	Process rule
$n \longrightarrow n'$	$N \xrightarrow{\tau} N'$
$n \xrightarrow{v := t} n'$	$N \xrightarrow{v := t} N'$
$n \xrightarrow{out(t)} n'$	$N \xrightarrow{out(t)} N'$
$n \xrightarrow{call\ \Pi_i(T)} n'$	$N \xrightarrow{\Pi_i(T)} START_i \cdot N'$
start node $start_0$	$START \xrightarrow{start_0} START_0$
end nodes end_i	$END_i \xrightarrow{end_i} \epsilon$

Table 2. Rules of the process system.

4 Interprocedural Bitvector Problems

In this section we provide solutions to the basic four bitvector problems using a language-theoretic formalism. We show how to compute the set of program points satisfying the existentially quantified properties. For universally quantified properties we first compute the set of points satisfying the *negation* of the property, which is existentially quantified, and then take the complement with respect to the set of all program points.

We first consider the case in which all variables are global, and subsequently move to the general case with both local and global variables.

4.1 Global Variables

We introduce some notations:

- Def_v denotes the set of actions of the form $v := t$;
- Ref_v denotes the set of actions of the form $u := t$ such that v appears in t;
- $Comp_t$ denotes the set of actions of the form $v := t'$ such that t' contains t as subterm;
- Mod_t denotes the set of actions of the form $v := t'$ such that v appears in t.[3]

Let us start by formalising the liveness problem. A global variable v is live at a program point n if there exists a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying the following constraints:

1. $P_1 = N \cdot P_1'$ (so σ_1 corresponds to a program path ending at the program point n);
2. $\sigma_2 \in (Act - Def_v)^* Ref_v$ (so in σ_2 the variable v is referenced before it is defined).

The other problems (or their negations) can be formalised following the same pattern. In fact, in the case of very busyness and availability what we directly compute in our approach, as mentioned above, is the set of program points at which v is *not* very busy or t is *not* available, respectively. Table 3 lists the constraints on P_1, σ_2, P_2 that must be satisfied in each case (there are no constraints on σ_1).

[3] Notice that, due to the conventions at the end of Section 2, the sets $Comp_t$ and Mod_t are disjoint.

Property	P_1	σ_2	P_2
v is live at n	$N \cdot P_1'$	$LI_v = (Act - Def_v)^* Ref_v$	Con^*
$m \xrightarrow{v:=t} m'$ reaches n	$M \cdot P_1'$	$RE_v = (v := t)(Act - Def_v)^*$	$N \cdot P_2'$
t is *not* very busy at n	$N \cdot P_1'$	$NVB_t = (Act - Comp_t)^*(Mod_t + end_0)$	Con^*
t is *not* available at n	Con^*	$NA_t = (start_0 + Mod_t)(Act - Comp_t)^*$	$N \cdot P_2'$

Table 3. Constraints on P_1, σ_2, and P_2.

Solving the Problems. Given a process system (Con, Act, Δ), we make the following straightforward but crucial observation: A set of agents is just a language over the alphabet Con, and so it makes sense to speak of a regular set of agents. Automata can be used to finitely represent infinite regular sets of agents.

This observation has been exploited by Finkel, Williams and Wolper in [10] and by Bouajjani, Maler and the first author in [4] to develop efficient algorithms for reachability problems in process systems. We present some of the results of [10, 4], and apply them to the bitvector problems for flow graph systems.

Let $L \in Con^*$ and $C \in Act^*$ be regular languages. We call C a *constraint*, and define

$$post^*[C](L) = \{P \in Con^* \mid P' \xrightarrow{\sigma} P \text{ for some } P' \in L \text{ and some } \sigma \in C\}$$

In words, $post^*[C](L)$ is the set of agents that can be reached from L by means of sequences satisfying the constraint C. Analogously, we define

$$pre^*[C](L) = \{P \in Con^* \mid P \xrightarrow{\sigma} P' \text{ for some } P' \in L \text{ and some } \sigma \in C\}$$

So $pre^*[C](L)$ is the set of agents from which it is possible to reach an agent in L by means of a sequence in C. We abbreviate $post^*[Act^*](L)$ and $pre^*[Act^*](L)$ to $post^*(L)$ and $pre^*(L)$, respectively. We have the following result:

Theorem 1 ([10, 4]). *Let (Con, Act, Δ) be a process system such that each rule $P_1 \xrightarrow{a} P_2$ satisfies $|P_1| \leq 2$, and let $L \subseteq Con^*$ and $C \subseteq Act^*$ be regular sets. Then $pre^*[C](L)$ and $post^*[C](L)$ are regular sets of agents. Moreover, automata accepting these sets can be computed in $O(n_\Delta \cdot n_L^2 \cdot n_C)$ time, where n_Δ is the size of Δ, and n_L, n_C are the sizes of two automata accepting L and C, respectively.*

Let us use this result to compute the set of program points at which the variable v is live. This is by definition the set of program points n for which there is a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying $P_1 = N \cdot P_1'$ and $\sigma_2 \in LI_v$. Observe that $pre^*[LI_v](Con^*)$ is the set of agents from which a sequence $\sigma_2 \in LI_v$ can be executed. Notice however that not all these agents are necessarily reachable from $START$. Since the set of agents reachable from $START$ is $post^*(START)$, we compute an automaton accepting

$$pre^*[LI_v](Con^*) \cap post^*(START)$$

Now, in order to know if v is live at n it suffices to check if this automaton accepts some word starting by N.

Problem	Set of agents
liveness	$pre^*[LI_v](Con^*) \ \cap \ post^*(START)$
reachability	$post^*[RE_v](post^*(START) \cap M \cdot Con^*)$
very busyness	$pre^*[NVB_t](Con^*) \ \cap \ post^*(START)$
availability	$post^*[Act^* NA_t](START)$

Table 4. Agents corresponding to the four bitvector problems.

Table 4 shows the sets of agents that have to be computed to solve all four bitvector problems. For the complexity, notice that the number of states of the automata for L and C depends only on the bitvector problem, and not on the process system. So the liveness and reaching definition problems for a given variable v and the very busyness and availability problems for a given term t can be solved in $O(n_\Delta)$ time.

4.2 Local and Global Variables

The reader has possibly noticed that we have not exploited all the power of Theorem 1 so far. While the theorem holds for rules with a left-hand side of length 1 or 2, we have only applied it to rules with left-hand sides of length 1. We use now full power in order to solve the bitvector problems in a setting with global and local variables.

A local variable v is live at a program point n if and only if there exists a sequence $START \xrightarrow{\sigma_1} N \cdot P \xrightarrow{\sigma_2 d} N' \cdot P$ for some agent P such that

(1) $d \in Ref_v$,
(2) all the agents reached along the execution of $\sigma_2 d$ are of the form $P' \cdot P$, and
(3) for every transition $P_1 \cdot P \xrightarrow{a} P_2 \cdot P$ of σ_2, if $a \in Def_v$, then $|P_1| \geq 2$.

Here, condition (2) guarantees that the incarnation of v referenced by d and the incarnation of $N \cdot P$ are the same. Condition (3) guarantees that this incarnation is not modified along the execution of σ_2.

We now apply a general strategy of our automata approach, which will be used again in the next section: Instead of checking conditions (1) to (3) on the simple process system coresponding to the flow graph, we check a simpler condition on a more complicated process system. Intuitively, this new system behaves like the old one, but at any procedure call in a computation (or at the beginning of the program) it can nondeterministically decide to push a new variable M onto the stack—used to Mark the procedure call—and enter a new mode of operation, called the *local mode.* In local mode the process distinguishes between actions occurring at the current procedure call (the *marked call* in the sequel), and actions occurring outside it, i.e., actions occurring after encountering further procedure calls before finishing the marked call, or actions occurring after finishing the marked call.

We extend the process system with new agents, actions, and rules. The additional new agents are M (the Marker) and O (used to signal that we are Outside the marked call). There is also a new action a_m for each old action a, plus extra actions *mark*, *return*, *exit*.[4] In the new process system a_m can only occur at the marked call, and a only at other levels. For each old rule we add one or more new ones as shown in Table 5. Notice that once the marker is introduced, all

Old rule	New additional rule(s)	
$N \xrightarrow{a} N'$	$M \cdot N \xrightarrow{a_m} M \cdot N'$	(a_m in the marked call)
	$O \cdot N \xrightarrow{a} O \cdot N'$	(a outside the marked call)
$START \xrightarrow{start} START_0$	$START \xrightarrow{mark} M \cdot START_0$	
$N \xrightarrow{\Pi_i(T)} START_i \cdot N'$	$N \xrightarrow{\Pi_i(T)} M \cdot START_i \cdot N'$	(marking the current call)
	$M \cdot N \xrightarrow{\Pi_i(T)} O \cdot START_i \cdot M \cdot N'$	(entering a deeper level)
	$O \cdot N \xrightarrow{\Pi_i(T)} O \cdot START_i \cdot N'$	
$N \xrightarrow{end_i} \epsilon$	$M \cdot N \xrightarrow{exit} O$	(end of the marked call)
	$O \cdot N \xrightarrow{end_i} O$	
	$O \cdot M \xrightarrow{return} M$	(return to the marked call)

Table 5. Rules of the extended process system.

reachable agents have either M or O in front, and that once the marked call terminates no agent ever has M in front again.

Given a local variable v, we define $Rel\_Def_v = \{a_m \mid a \in Def_v\}$ and $Rel\_Ref_v = \{a_m \mid a \in Ref_v\}$, where *Rel* stands for "relevant." If an agent moves into local mode at a procedure call in which a local variable v is incarnated, then Rel_Def_v and Rel_Ref_v are the actions concerning this same incarnation of v.

If we let Ext_Act be the extended set of actions of the new process system, then a local variable v is live at a program point n if and only if there exists a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying the following constraints:

- $P_1 = M \cdot N \cdot P_1'$, and
- $\sigma_2 \in (Ext\_Act - Rel\_Def_v)^* Rel\_Ref_v$.

So the constraint on σ_2 when the program has both local and global variables is obtained from the constraint for global variables by substituting Ext_Act for Act, Rel_Def_v for Def_v, and Rel_Ref_v for Ref_v. The reaching definitions problem can be solved analogously.

For the very busyness and the availability problems we have to take into account that the term t may contain local and global variables. Let $LocId(t)$ and $GlobId(t)$ be the set of local and global variables that appear in t. We define

$$Rel\_Mod_t = \bigcup_{v \in GlobId(t)} Def_v \cup \bigcup_{v \in LocId(t)} Rel\_Def_v$$

$$Rel\_Comp_t = \{a_m \mid a \in Comp_t\}$$

A term is not very busy at a program point n if and only if there exists a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying the following constraints:

[4] These actions are not strictly necessary, they are only included for clarity.

– $P_1 = M \cdot N \cdot P_1'$, and
– $\sigma_2 \in (Ext\_Act - Rel\_Comp_t)^*(Rel\_Mod_t + end_0)$.

Since the number of transition rules of the new process system increases only by a constant factor, the algorithm still runs in $O(n_\Delta)$ time.

5 Interprocedural Faint Variables

Recall that a variable v is *faint* at a program point n if on every program path from n every right-hand side occurrence of v is either preceded by a modification of v or is in an assignment whose left-hand side variable is faint as well. We show how to compute the set of program points at which a variable is faint in an interprocedural setting with global and local variables. This requires to split the set of references of a variable into the subset of references occurring in an output statement, and its complement set. To this end we introduce the notation:

– $RefOut_v$ denotes the set of actions of the form $out(t)$, such that v appears in t.

Faintness is a universal property, i.e., one that holds only if *all* program paths from a point n satisfy some condition. We formalise its negation, which is an existential property. A variable v is not faint at a program point n if there exists a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying the following constraints:

– $P_1 = N \cdot P_1'$ (so σ_1 corresponds to a program path ending at the program point n), and
– v is not faint at $P_1 \xrightarrow{\sigma_2} P_2$.

It remains to define the set of paths at which v is not faint. In comparison to the related bitvector property of deadness, the only new difficulty is to deal adequately with the recursive nature of the definition of faintness. The set is recursively defined as the smallest set of finite paths $P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} P_3 \cdots P_n$, where $P_1 = N \cdot P_1'$, such that

(1) $a_1 \in RefOut_v$, or
(2) v is global, $a_1 \notin Def_v$ and v is not faint at $P_2 \xrightarrow{a_2} P_3 \cdots P_n$, or
(3) v is local for N, a_1 is an assignment not in Def_v, and v is not faint at $P_2 \xrightarrow{a_2} P_3 \cdots P_n$, or
(4) $a_1 \equiv u := t$, where v appears in t, and u is not faint at $P_2 \xrightarrow{a_2} P_3 \cdots P_n$.

Notice the difference between (2) and (3). If v is local and a_1 is a call action, then after the execution of a_1 the new program point is out of the scope of v, and so v is faint at $P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} P_3 \cdots P_n$.[5] If v is global, then we remain within the scope of v, and so we do not know yet whether v is faint or not.

[5] With our definition v may be faint at $N \cdot P_1' \xrightarrow{a_1} P_2 \xrightarrow{a_2} P_3 \ldots P_n$ but not faint at some other path $N \cdot P_1'' \xrightarrow{a_1'} P_2' \xrightarrow{a_2'} P_3' \cdots P_{n'}'$, and so not faint at N.

As we did in the last section, we do not check the complicated property "v is faint at n" on a simple process system, but a simpler property on a more complicated process system obtained by adding new agents and rules (although this time no actions) to the old one. Intuitively, the new process system behaves like the old one, but at any point in a computation it can nondeterministically decide to push a program variable v into the stack—represented by an agent constant V—and enter a *faint mode* of operation. From this moment on, the system updates this variable according to the definition of non-faintness. The (updated) variable stays in the stack as long as the path executed by the system can be possibly extended to a path at which the variable is not faint. So the variable is removed from the stack only when the process system knows that (a) v is not faint for the path executed so far, or (b) v is faint at all paths extending the path executed so far. In case (a) the process system pushes a new agent $\perp$ into the stack, and in case (b) it pushes an agent $\top$.

Formally, the extended process system is obtained by adding the two agents $\top$ and $\perp$, and new rules as shown in Table 6.

Old rule	New additional rule(s)
$N \xrightarrow{a} N'$	$N \xrightarrow{a} V \cdot N'$ for each variable v (the faint mode can be entered anytime) $V \cdot N \xrightarrow{a} \perp$ if $a \in \mathit{RefOut}_v$ (condition (1), $\perp$ signals that v is not faint) $V \cdot N \xrightarrow{a} V \cdot N'$ if v global and $a \notin \mathit{Def}_v$ (condition (2)) $V \cdot N \xrightarrow{a} U \cdot N'$ if $a \equiv u := t$ and v appears in t (condition (4))
$N \xrightarrow{\Pi_i(T)} START_i \cdot N'$	$V \cdot N \xrightarrow{\Pi_i(T)} V \cdot START_i \cdot N'$ if v global (condition (2)) $V \cdot N \xrightarrow{\Pi_i(T)} START_i \cdot V \cdot N'$ if v local at n (out of the scope of v)
$N \xrightarrow{end_i} \epsilon$	$V \cdot N \xrightarrow{end_i} V$ if v global (condition (2)) $V \cdot N \xrightarrow{end_i} \top$ if v local at n (condition (3), this incarnation of v can no longer be defined or referenced)

Table 6. Rules of the extended process system.

In order to obtain the set of program points at which the variable v is faint, we compute the set of agents N for which there is a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying $P_1 = V \cdot N \cdot P_1'$ and $P_2 = \perp \cdot P_2'$. It suffices to compute an automaton for

$$(pre^*(\perp \cdot Con^*) \cap V \cdot Con^*) \cap post^*(START)$$

6 Interprocedural Bitvector Problems with Parallelism

In flow graph systems with parallelism, which we call in the sequel *parallel flow graph systems*, we allow edge-labels of the form $n \xrightarrow{pcall\ \Pi_{i_1}(T_1),\ldots,\Pi_{i_k}(T_k)} n'$. The procedures $\Pi_{i_1}, \ldots, \Pi_{i_k}$ are called in parallel; if and when they all terminate, execution is continued at n'. Notice that parallel calls can be nested, and so the number of procedures running in parallel is unbounded. Notice also that flow graphs without parallelism are the special case in which $k = 1$ for all *pcall* instructions.

We show that the four bitvector problems can be solved for parallel flow graph systems in polynomial time when all variables are global. For this it will suffice to apply a beautiful extension of Theorem 1 recently proved by Lugiez and Schnoebelen in [18].

In order to model parallel flow graph systems by process systems we need to extend these to *parallel process systems* (also called process rewrite systems in [20]). An *agent* of a parallel process system is a tree whose root and internal nodes are labelled with either $\cdot$ or $\|$, representing sequential and parallel composition, and whose leaves are labelled with agent variables. So, for instance, the intended meaning of the tree $(X\|Y) \cdot Z$ is that X and Y are first executed in parallel, and if and when they terminate Z is executed. The empty tree γ, which satisfies

$$\gamma \cdot P = P \cdot \gamma = \gamma \| P = P \| \gamma = P$$

plays now the rôle of the terminated process. We denote the set of agents by $T(Con)$ (trees over Con). Rules are now elements of $(T(Con) \setminus \{\gamma\}) \times Act \times T(Con)$. A set Δ of rules induces a reachability relation $\xrightarrow{a} \subseteq T(Con) \times T(Con)$ for each $a \in Act$, defined as the smallest relation satisfying the following inference rules:

- if $(P_1, a, P_2) \in \Delta$, then $P_1 \xrightarrow{a} P_2$;
- if $P_1 \xrightarrow{a} P_1'$ then $P_1 \cdot P_2 \xrightarrow{a} P_1' \cdot P_2$ for every $P_2 \in T(Con)$;
- if $P_1 \xrightarrow{a} P_1'$ then $P_1 \| P_2 \xrightarrow{a} P_1' \| P_2$ and $P_2 \| P_1 \xrightarrow{a} P_2 \| P_1'$ for every $P_2 \in T(Con)$.

We make free use of the fact that parallel composition is associative and commutative with respect to any reasonable behavioural equivalence between agents, such as bisimulation equivalence [22].

We associate to a parallel flow graph system the process system (Con, Act, Δ) as in the sequential case, the only difference being the rule corresponding to parallel calls, which is shown in Table 7. Observe that the left-hand side of all

Parallel flow graph	Process rule
$n \xrightarrow{pcall\ \Pi_{i_1}(T_1),\ldots,\Pi_{i_k}(T_k)} n'$	$N \xrightarrow{\Pi_1(T_1),\ldots,\Pi_k(T_k)} (START_{i_1} \| \ldots \| START_{i_k}) \cdot N'$

Table 7. Rules of the parallel process system

rules consists of just one variable. Parallel process systems with this property are closely related to the PA-algebra studied in [18].

The four bitvector problems are defined almost as in the non-parallel case. The only difference is that in the parallel setting a program path can begin or end at *many* nodes due to the existence of parallel computation threads. In the non-parallel case, the agents corresponding to "being at node n" are those of the form $N \cdot P$. In the parallel case, they are the agents (trees) satisfying the following property: there is a leaf N which does not belong to the right subtree of any node labelled by $\cdot$. We call the set of these agents At_n.

Solving the Problems. An agent is no longer a word, but a tree—and so a set of agents is now a tree language. Tree automata can be used to finitely represent infinite regular sets of agents. We briefly introduce the tree automata we need. They are tuples (Q, A, δ, F) where:

- Q is a finite set of states and $F \subseteq Q$ is a set of final states.
- A is a finite alphabet containing the set *Con* and two binary infix operators $\cdot$ and $\|$. The automaton accepts terms over this alphabet, which we just call trees.
- δ is a finite set of transition rules of the form $N \to q$, $q_1 \cdot q_2 \to q$, or $q_1 \| q_2 \to q$. The rules define a rewrite relation on terms over $A \cup Q$.

The automaton accepts the trees that can be rewritten into a final state using the transition rules. As an example, we present a tree automaton accepting the set At_n. It has two states $\{q_1, q_2\}$, with q_1 as final state, rules $N \to q_1$ and $N' \to q_2$ for every program point $n' \neq n$, and rules

$$q_i \cdot q_j \to \begin{cases} q_1 & \text{if } i = 1 \\ q_2 & \text{otherwise} \end{cases} \qquad q_i \| q_j \to \begin{cases} q_1 & \text{if } i = 1 \text{ or } j = 1 \\ q_2 & \text{otherwise} \end{cases}$$

for $i, j \in \{1, 2\}$.

The question arises whether Theorem 1 can be extended to the tree case, i.e., the case in which Con^* is replaced by $T(Con)$. The answer is unfortunately negative. For instance, it is not difficult to see that the problem of deciding whether $post^*[C](L)$ is nonempty is undecidable even for the special case in which each rule $P_1 \xrightarrow{a} P_2$ satisfies $P_1 \in Con$ and L contains only one agent [18]. However, Lugiez and Schnoebelen show in [18] that it is possible to save part of the theorem. In particular, they prove the following result:

Theorem 2 ([18]). *Let (Con, Act, Δ) be a parallel process system such that each rule $P_1 \xrightarrow{a} P_2$ satisfies $P_1 \in Con$, let $L \in T(Con)$ be a regular set of agents, and let $A \subseteq Act$. Then $pre^*[A](L)$, $post^*[A](L)$, $pre^*[A^*](L)$, and $post^*[A^*](L)$ are regular sets of agents, and tree automata accepting them can be effectively computed in polynomial time.*

In order to check if the variable v is live at a program point n, we have to decide if there is a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying $P_1 \in At_n$ and $\sigma_2 \in LI_v$. Fortunately, LI_v is the concatenation of two languages of the form

A and A^* for which Theorem 2 holds, namely $(Act - Def_v)^*$ and Ref_v. So it suffices to compute a tree automaton accepting

$$pre^*[(Act - Def_v)^*](pre^*[Ref_v](T(Con))) \cap post^*(START) \cap At_n$$

and check if it is empty. The other three bitvector problems are solved analogously. If we now wish to extend this result to the case with both local and global variables, we can proceed as in Section 4.2. However, the process system so obtained contains rules whose left-hand side is a sequential composition of two agents. Since Theorem 2 has only been proved for the case $P_1 \in Con$, we cannot directly apply it. The question whether the bitvector problems can also be efficiently computed for local and global variables is still open.

7 Conclusions

We have shown that recent progress in extending the automata-theoretic approach to classes of processes with an infinite state space finds interesting applications in interprocedural data-flow analysis. Even though research in this area is at its very beginning, it is already possible to envisage some advantages of automata techniques. First of all, data-flow problems are expressed in terms of the possible executions of a program, and so it is very natural to formalise them in language terms; from the point of view of temporal logic, data-flow problems correspond to linear-time properties, and so the automata-theoretic approach, which is particularly suitable for linear-time logics, seems to be very adequate. Secondly, the approach profits from the very well studied area of automata theory. For instance, Lugiez and Schnoebelen obtained their results [18] by generalising constructions of [4, 10] for word automata to tree automata, and we could immediately apply them to bitvector problems in the interprocedural parallel case.

References

1. J. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *Journal of the ACM*, 40(3):653–682, 1993.
2. J. C. M. Baeten and W. P. Weijland. Process Algebra. *Cambridge Tracts in Theoretical Computer Science*, 1990.
3. B. Bloom. Structured operational semantics as an specification language. In *Proceedings of POPL '95*, pages 107–117, 1995.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of CONCUR '97*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, 1997.
5. O. Burkart and B. Steffen. Model checking for context-free processes. In *Proceedings of CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137, 1992.
6. O. Burkart and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic Journal of Computing*, 2(2):89–125, 1995.

7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. 4th Symp. Principles of Prog. Lang.* (*POPL'77*), pages 238 – 252. ACM, NY, 1977.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conf. Rec. 4th Symp. Principles of Prog. Lang.* (*POPL'79*), pages 269 – 282. ACM, New York, 1979.
9. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. of Logic and Computation*, 2(4):511 – 547, 1992.
10. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.
11. R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations. In *Proc. 3rd Conf. Europ. Co-operation in Informatics*, Informatik-Fachberichte 50, pages 1 – 10. Springer-V., 1981.
12. M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.
13. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
14. S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data flow analysis. *Acta Informatica*, 24:679 – 694, 1987.
15. M. Klein, J. Knoop, D. Koschützki, and B. Steffen. DFA&OPT-MetaFrame: A tool kit for program analysis and optimization. In *Proc. 2nd Int. Workshop on Tools and Algorithms for Constr. and Analysis of Syst.* (*TACAS'96*), LNCS 1055, pages 422 – 426. Springer-V., 1996.
16. J. Knoop. *Optimal Interprocedural Program Optimization: A new Framework and its Application.* PhD thesis, Univ. of Kiel, Germany, 1993. LNCS Tutorial 1428, Springer-V., 1998.
17. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Prog. Lang. Syst.*, 18(3):268 – 299, 1996.
18. D. Lugiez and P. Schnoebelen. The regular viewpoint on PA-processes. In *Proceedings of CONCUR '98*, volume 1466 of *Lecture Notes in Computer Science*, pages 50–66, 1998.
19. K. Marriot. Frameworks for abstract interpretation. *Acta Informatica*, 30:103 – 129, 1993.
20. R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems.* Ph.D. thesis, Technische Universität München, 1998.
21. S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proc. Int. Conf. on Parallel Processing, Volume II*, pages 105 – 113, 1990.
22. R. Milner. *Communication and Concurrency.* Prentice Hall, 1990.
23. S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, CA, 1997.
24. B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In *Proc. 6th Int. Conf. on Concurrency Theory* (*CONCUR'95*), LNCS 962, pages 72 – 87. Springer-V., 1995. Invited contribution.
25. M. Y. Vardi and P. Wolper. Automata Theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.
26. K. Varpaaniemi. PROD 3.3.02. An advanced tool for efficient reachability analysis. Technical report, Department of Computer Science and Engineering, Helsinky University of Technology, 1998. Available at http://www.tcs.hut.fi/pub/prod/.
27. F. Wallner. Model checking LTL using net unfoldings. In *Proceedings of CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 207–218, 1998.