From DFA-Frameworks to DFA-Generators: A Unifying Multiparadigm Approach

Jens Knoop

Universität Dortmund, D-44221 Dortmund, Germany
Phone: ++49-231-755-5803 Fax: ++49-231-755-5802
E-mail: knoop@ls5.cs.uni-dortmund.de
http://sunshine.cs.uni-dortmund.de/~knoop

Abstract. Program analysis is still characterized by paradigm-specific approaches, which are developed to accommodate to the diversities of the different programming paradigms as e.g. the imperative, object-oriented, or parallel one. Switching between paradigms or transferring analyses across paradigm boundaries requires usually detailed knowledge of the peculiarities of the various approaches. This complicates both the reuse of analyses and the proofs of their correctness. On the other hand, abstract interpretation provides a unifying access to program analysis. In this article we exploit this for the construction of program analysis generators based on a uniform design principle. Basically, we proceed by extracting the abstract kernel from the standard analysis framework, which we then consider under a generic perspective. We show that there are concrete instances in such different paradigms as those above. As a by-product their decomposition into a "theoretical" and "practical" part which are *specificational* and *computational* in nature, reveals the aforementioned design principle. The frameworks and their respective generators, which can be fed by concise specifications, can thereby be considered black-boxes: analysis designers only need to know of the (quite similar) interfaces. The proof of correctness or even precision of a generated algorithm with respect to a specific property reduces to checking the premises of a few theorems. This considerably eases the construction of analyses within a specific paradigm as well as the switch between and the transfer of analyses to other paradigms.

Keywords: Program optimization, abstract interpretation, data-flow analysis (DFA), DFA-frameworks, DFA-generators, coincidence theorems, intraprocedural, interprocedural, parallel, object-oriented, conditional DFA.

1 Motivation

Static program analysis — in the context of optimizing compilers usually called data-flow analysis (DFA) — is an almost indispensable prerequisite for the application of performance improving transformations by optimizing compilers (cf. [1,11,33,34,35]). Typical questions to be answered by DFA in order to enable classical optimizations like *code motion* (cf. [32]), *constant propagation* (cf. [13]), or

W.R. Cleaveland (Ed.): TACAS/ETAPS'99, LNCS 1579, pp. 360–374, 1999.

dead code elimination (cf. [1]) are, if a program term t has always been computed when reaching a specific program point ("Is t *available*?"), if its evaluation always yields the same constant value there ("Is the value of t a *constant*?"), or if a variable v will not be used on any program continuation leaving this point without a preceding redefinition ("Is v *dead*?").

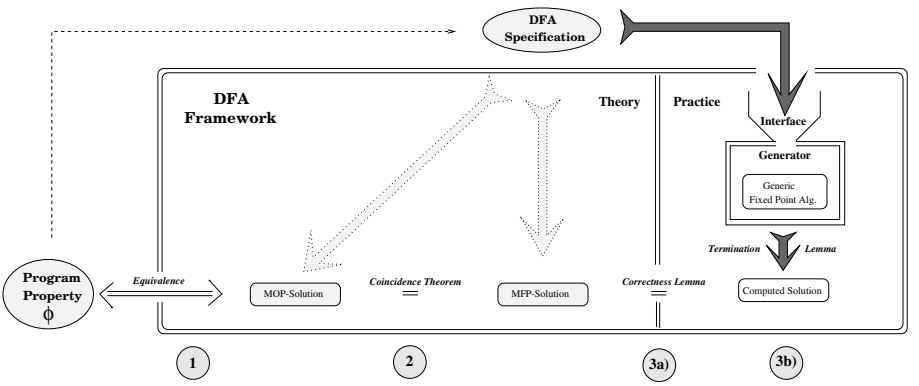

Fig. 1. The general structure of a DFA-framework.

After fixing the relevant program property DFA-designers are thus typically faced with two problems: first, *inventing* an algorithm for computing the set of program points enjoying the property under consideration; second, *proving* that it computes this set precisely. A common observation here is that the more powerful and expressive the features are the underlying programming language provides (procedures, parallelism, objects, polymorphism, etc.), the more constraints must be respected by a DFA-designer which are imposed by the features of the programing language rather than the property considered. This is quite important because these constraints tremendously influence the technical complexity of the algorithms and the proofs of their correctness and precision. Consequently, it is the less adequate to construct DFA-algorithms by means of ad-hoc techniques the more powerful and expressive the considered programming language is because this usually amounts to inventing an individual solution for the considered analysis problem both on the algorithmical and the proof side.

In this article we reconsider the design process of DFA-algorithms under the unifying view of abstract interpretation. This leads us to a uniform multiparadigm approach, which in addition suggests a principle for the automatic generation of DFA-algorithms. To this end we first reduce the standard framework for intraprocedural DFA to its abstract kernel. Considering it then from a generic point of view, we demonstrate that it has instances in *intraprocedural*, *interprocedural*, (*data-*) *parallel*, *explicitly parallel*, *object-oriented*, and *condi-*

tional DFA; hence, it applies to quite different paradigms. Moreover, this approach provides DFA-designers with strong support both on the practical and the theoretical side. On the *theoretical* side because precise guidelines can be set up, which structure and simplify the development of DFA-algorithms as well as the proofs of their correctness or even precision. On the *practical* side because DFA-generators can be distilled from the frameworks allowing the automatic generation of DFA-algorithms from concise specifications. Though the concrete DFA-frameworks and their respective DFA-generators differ in their details for such different programming paradigms, from the perspective of a DFA-designer, they can be considered black-boxes sharing almost the same interface. In fact, knowing their (quite similar) interfaces is sufficient for the successful and rapid development of proven correct DFA-algorithms.

Overview. Figure 1 illustrates the essence of our approach from the point of view of a DFA-designer. In this figure ϕ is assumed to denote the property of interest. Fundamental for computing the set of program points enjoying ϕ is the theory of *abstract interpretation* (cf. [5,6,7,30,36]). It provides a well-founded basis for DFA. The point here is to replace the "full" semantics of a program by a simpler more abstract version, which is tailored for the problem under consideration. Usually, the abstract semantics is defined by a (*local*) *semantic functional*, which gives abstract meaning to the (elementary) statements of a program in terms of transformations on a complete lattice. Its elements represent the data-flow facts of interest. A local abstract semantics induces two notions of a solution of the respective DFA-problem, which result from two different globalization approaches: the *MOP*-solution of the "operational" *meet-over-all-paths* (*MOP*) approach, and the *MFP*-solution of the "denotational" *maximal-fixed-point* (*MFP*) approach.

The *MOP*-solution mimics the effect of possible program executions: for every program point it is the "meet" (intersection) of all data-flow facts contributed by program paths reaching it. Usually, the *MOP*-solution is conceptually quite close to the program property of interest, but in general the underlying *MOP*-approach is not effective. It is thus specifying in nature. In distinction, the *MFP*-solution is defined as the greatest solution of a system of equations imposing consistency constraints on an annotation of the program with data-flow facts: in essence, the data-flow fact attached to a program point must be implied by the results of transforming the informations attached to its predecessors according to their abstract meaning.

In contrast to the *MOP*-approach, the *MFP*-approach is practically relevant. It induces a generic fixed point algorithm, which (under specific side-conditions) terminates with the *MFP*-solution. As shown in Figure 1, this algorithm can directly be fed with a local abstract semantics: the concrete DFA-algorithm results automatically from instantiating the generic algorithm by the DFA-specification under consideration, and need not be implemented by the DFA-designer. The DFA-designer is only left with proving that the generated algorithm precisely computes the set of program points enjoying ϕ. This can be proved in only three independent steps, which are based on properties of the specification only.

1. *Equivalence:* prove that the program property ϕ under consideration is equivalent to the *MOP*-solution of the DFA-problem specified (1).
2. *Coincidence:* prove that the *MOP*-solution and the *MFP*-solution of the DFA-problem considered coincide (2).
3. *Effectivity:* prove that the automatically generated DFA-algorithm terminates (3b)) with the *MFP*-solution (3a)).

The gap which must be bridged in the first step is usually considerably small because typically both ϕ and the *MOP*-solution are defined in terms of the effect of program paths over the same basic properties. This gap can be bridged by a usually straightforward induction on the length of program paths.

The second step is the central one of this proof sequence. It has to bridge the gap between the theoretical part of the DFA-framework and its respective DFA-generator. The glue combining them is a coincidence theorem establishing the coincidence of the conceptually quite different *MOP*- and *MFP*-solution. The *coincidence theorem* gives a sufficient condition for the coincidence of the *MOP*-solution, which constitutes the requested reference solution of a DFA-problem, and the *MFP*-solution, which is computed by the generated DFA-algorithm. In fact, the coincidence theorem can be considered the theoretical backbone of a DFA-framework.

In the third step, finally, it must be verified that the generated DFA-algorithm terminates with the *MFP*-solution. Similar to the coincidence theorem, an *effectivity theorem* gives a sufficient condition for this. It can be checked knowing the DFA-specification only.

Benefits. The proof obligations of the second and third step require to check the premises of a coincidence theorem and an effectivity theorem only. In general, this is much simpler than establishing the corresponding results for an algorithm invented afresh for a problem. Moreover, the concrete DFA-algorithm, which decides the program property of interest, comes for free in this approach. It is the algorithm automatically resulting from instantiating the generic algorithm of the framework with the considered DFA-specification. This is particularly important because the specification interface and the proof obligations remain essentially the same though the internal structure of the DFA-frameworks becomes more complex when the features of the programming language are enriched. Thus, the benefits of applying the DFA-framework are the greater the more powerful the programming language is the framework is designed for. In fact, though the details of the frameworks for intraprocedural, interprocedural, parallel or conditional DFA are quite different, the DFA-designer can think of and apply them as black boxes: a framework and its respective DFA-generator is a black box, which accepts in a specific format a DFA-specification which is tuned to the program property of interest. It returns an algorithm, which computes the set of program points enjoying this property, provided that the three proof obligations labeled *equivalence*, *coincidence*, and *effectivity* are supplied as illustrated in Figure 2.

Summarizing, for a DFA-designer the major benefits are as follows:

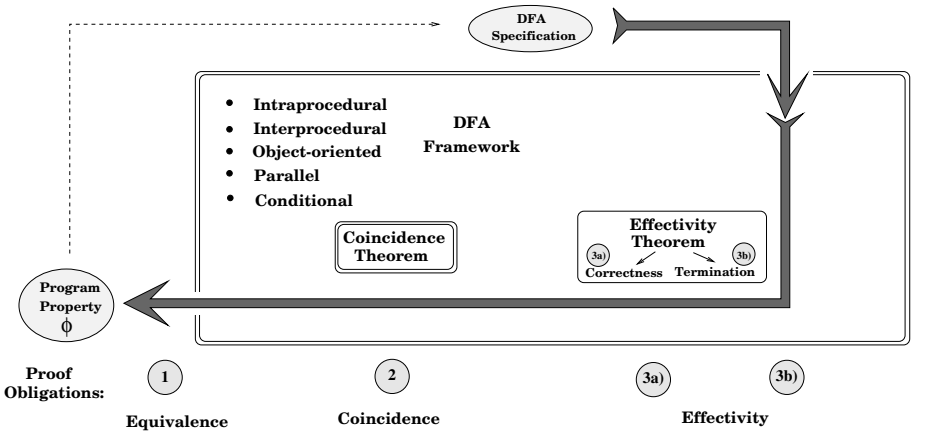

Fig. 2. The black-box view of DFA-frameworks.

– *Information hiding:* all details which are not relevant for a particular application are hidden.
– *Automatic generation of DFA-algorithms:* the concrete DFA-algorithm for a DFA-problem results automatically from a concise specification in terms of an abstract interpretation.
– *Precise proof obligations:* proving the generated algorithm to be precise for the property of interest requires only knowledge of the specification.

And last but not least:

– *Uniformity:* applicability of the overall approach to a broad range of programming paradigms.

In the remainder of this article we focus on the last point. We demonstrate that the benefits summarized above, which have previously been demonstrated for *intraprocedural* and *interprocedural* DFA (cf. [14,15,24]), can be realized for further paradigms, too, including *object-oriented*, *parallel*, and *conditional* DFA.

Structure of the Article. In Section 2 we reconsider the intraprocedural base case and illustrate the essence of our approach by discussing the internal structure of the standard intraprocedural DFA-framework. Subsequently, we demonstrate that the pattern of the intraprocedural case carries over to interprocedural, data-parallel and object-oriented, explicitly parallel, and conditional DFA. In fact, demonstrating the generality of the underlying pattern is a central concern of this article, rather than applying the frameworks to concrete DFA-problems. Thus, the presentation remains on purpose on a conceptual level

taking a user's point of view in order to demonstrate that the DFA-designer is offered almost the same interface independently of the paradigm and the specific setting considered. Additionally, we put emphasis on the coincidence theorems underlying the concrete DFA-frameworks. They are the theoretical backbone providing the key to the overall approach.

2 Intraprocedural Data-flow Analysis

In this section we illustrate the essence of our approach by reconsidering the standard framework for intraprocedural DFA of imperative programs (cf. [12,13]). Figure 3 shows the intraprocedural instance of the "abstract" framework of Figure 1.

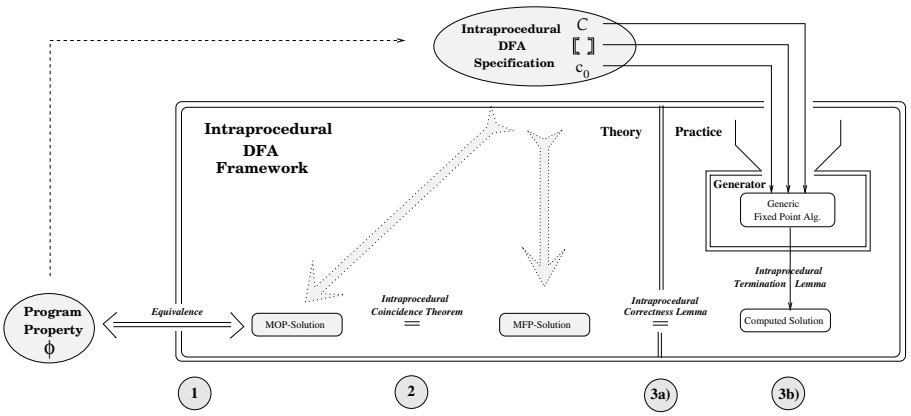

Fig. 3. The intraprocedural DFA-framework.

Intraprocedural DFA is characterized by a separate and independent investigation of the procedures of a program. Following [19] we assume that procedures are represented as directed edge-labeled flow graphs $G = (N, E, \mathbf{s}, \mathbf{e})$, whose nodes $n \in N$ represent program points, and whose edges $e \in E$ represent the statements and the nondeterministic control flow of the underlying procedure, and where $\mathbf{s}$ and $\mathbf{e}$ denote two distinct program points, the so-called start and end node of G. In this setting a local abstract semantics specifying a DFA-problem is a functional $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$ which gives abstract meaning to the statements of the procedure in terms of transformation functions on a set of data-flow facts, usually a complete lattice of finite height $\mathcal{C}$.[1] The following

[1] In [31] it is thus called a *lattice framework*.

straightforward extension of $[\![\ ]\!]$ to finite program paths $p \equiv (e_1, \ldots, e_q)$, where $Id_{\mathcal{C}}$ denotes the identity on $\mathcal{C}$, is the key to the meet-over-all-paths globalization:

$$[\![\, p \,]\!] =_{df} \begin{cases} Id_{\mathcal{C}} & \text{if } p \text{ is the "empty" path} \\ [\![\, (e_2, \ldots, e_q) \,]\!] \circ [\![\, e_1 \,]\!] & \text{otherwise} \end{cases}$$

Denoting the set of all program paths reaching a program point n by $\mathbf{P}[\mathbf{s}, n]$, the *MOP*-solution with respect to a local abstract semantics $[\![\ ]\!]$ is defined by:

$$\textbf{The } \textit{MOP}\textbf{-Solution: } \forall\, c_0 \in \mathcal{C}\ \forall\, n \in N.\ \textit{MOP}(n) =_{df} \bigsqcap \{\, [\![\, p \,]\!](c_0) \mid p \in \mathbf{P}[\mathbf{s}, n] \,\}$$

In contrast, the *MFP*-solution is defined as the greatest solution of the following equation system:

$$\mathbf{dfi}(n) \;=\; \begin{cases} c_0 & \text{if } n = \mathbf{s} \\ \bigsqcap \{\, [\![\, (m, n) \,]\!](\mathbf{dfi}(m)) \mid m \text{ is a predecessor of } n \,\} & \text{otherwise} \end{cases}$$

Let $\mathtt{dfi}_{c_0}$ denote the greatest solution of this equation system with respect to the start information c_0. Then the *MFP*-solution is defined by:

$$\textbf{The } \textit{MFP}\textbf{-Solution: } \quad \forall\, c_0 \in \mathcal{C}\ \forall\, n \in N.\ \ \textit{MFP}(n) \;=\; \mathtt{dfi}_{c_0}(n)$$

The well-known (intraprocedural) Coincidence Theorem of Kildall [13], and Kam and Ullman [12] gives a sufficient condition for the coincidence of the *MOP*-solution and the *MFP*-solution in this setting.

Theorem 1 (Intraprocedural Coincidence Theorem).
The (intraprocedural) MFP- and MOP-solution coincide, if the local semantic functions of the abstract interpretation are all distributive.[2]

As an example we consider the availability of a program term t. This is a classical (cf. [11]) and practically relevant (cf. [22,32]) DFA-problem, where the set of data-flow facts is given by the lattice of Boolean truth values *tt* and *ff* with $ff \sqsubseteq tt$. Intuitively, t is *available* at a program point n, if on every program path reaching n the last modification of one of t's operands is followed by a computation of t. This is illustrated by the program of Figure 4(a). Figure 4(b) highlights the program points where $a + b$ is available, and Figure 4(c) shows the program points where $c + b$ is available.

The DFA solving the availability problem is specified by the local abstract semantics

$$[\![\, e \,]\!]_{av} =_{df} \begin{cases} Const_{tt} & \text{if } Transp_t(e) \wedge Comp_t(e) \\ Id_{\mathcal{B}} & \text{if } Transp_t(e) \wedge \neg Comp_t(e) \\ Const_{ff} & \text{otherwise} \end{cases}$$

[2] A function $f : \mathcal{C} \to \mathcal{C}$ is called *distributive* iff $\forall\, C' \subseteq \mathcal{C}.\ f(\bigsqcap C') = \bigsqcap \{f(c) \mid c \in C'\}$. It is called *monotonic* iff $\forall\, C' \subseteq \mathcal{C}.\ f(\bigsqcap C') \sqsubseteq \bigsqcap \{f(c) \mid c \in C'\}$. Hence, monotonicity is a weaker requirement than distributivity. For monotonic semantic functions the *MFP*-solution is a safe approximation of its *MOP*-counterpart, i.e., $\textit{MFP} \sqsubseteq \textit{MOP}$; a fact, which holds for the other coincidence theorems given in the course of this article, too.

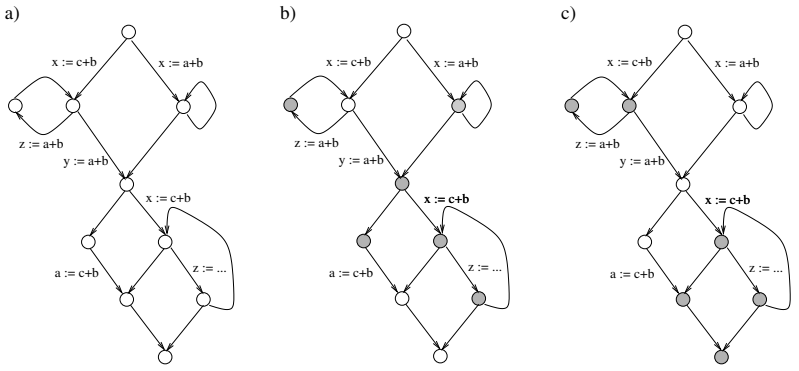

Fig. 4. Illustrating availability in the intraprocedural setting.

where $Id_{\mathcal{B}}$ denotes the identity, and $Const_{tt}$ and $Const_{ff}$ the constant functions on $\{tt, ff\}$, respectively. Moreover, *Comp* and *Transp* are two local predicates defined for the statements of the procedure under consideration. They are true if t is computed along edge e, and if no operand of t is modified along e, respectively. Obviously, all semantic functions are distributive. Hence, the *MOP*-solution and the *MFP*-solution coincide, which proves the central step of verifying that the DFA-algorithm, which is automatically generated from this specification, precisely computes the set of program points, where term t is available.

Remark 1. Note that in abstract interpretation correctness has both a "horizontal" and a "vertical" aspect. The coincidence theorem above (and those considered in the following sections) reflect the horizontal aspect: they are concerned with the precision of a fixed point solution ("MFP") computed with respect to a reference solution ("MOP") desired, which both refer to the same level of abstraction, which is fixed by the local abstract semantics they are sharing. In contrast, the vertical aspect concerns the correctness of the abstract semantics induced by its underlying local abstract semantics with respect to a reference semantics, usually the "concrete" program semantics or some other abstract semantics, e.g. the so-called *static semantics* (cf. [5]). We do not consider the vertical aspect here. It is an orthogonal issue, which cannot meaningfully be considered on the level of abstraction of the current presentation.

3 Interprocedural Data-flow Analysis

Figure 6 shows the interprocedural instance of the DFA-framework of Figure 1. Note that the internal structure of the framework and its corresponding generator is more complicated. However, the user interface is almost the same: in comparison to the intraprocedural setting only a single component has been added. In the following we discuss the differences in more detail.

Interprocedural DFA takes the semantics of procedure calls into account. For the interprocedural version of the *MOP*-approach this requires that only

interprocedurally *valid* paths are taken into account, i.e., paths respecting the call/return-behaviour of procedure calls (cf. [37]). Whereas the respective extension of the intraprocedural *MOP*-approach is rather straightforward, the interprocedural extension of the *MFP*-approach requires additional care. The key of this extension is a preprocess, which computes the semantics of procedure calls according to the local abstract semantics of the considered DFA-problem. This preprocess is realized by a second generic algorithm. As a consequence (cf. Figure 6), the internal structure of the DFA-framework is more complicated than its intraprocedural counterpart. However, the specification interface and the proof obligations remain essentially the same. Only a *return functional* $\mathcal{R}$ is additionally required. It is the handle to properly deal with local variables of recursive procedures. Intuitively, the point here is that effects on global variables must be maintained after returning from a recursive call, whereas local variables must be reset to their values at call time. This is illustrated in the example of Figure 5(a) using availability of program terms as example. While $c + b$ is available at the program point following the recursive call of π_1 in procedure π_1, $a + b$ is not. The difference lies in the fact that in case of $a + b$ a global operand is modified within the recursive call, while it is a local one for $c + b$. Return functions extract this information from the data-flow informations valid at call time and valid immediately before leaving the called procedure, which are stored in a DFA-stack mimicing the run-time stack. This is discussed in detail in [15,24,26].

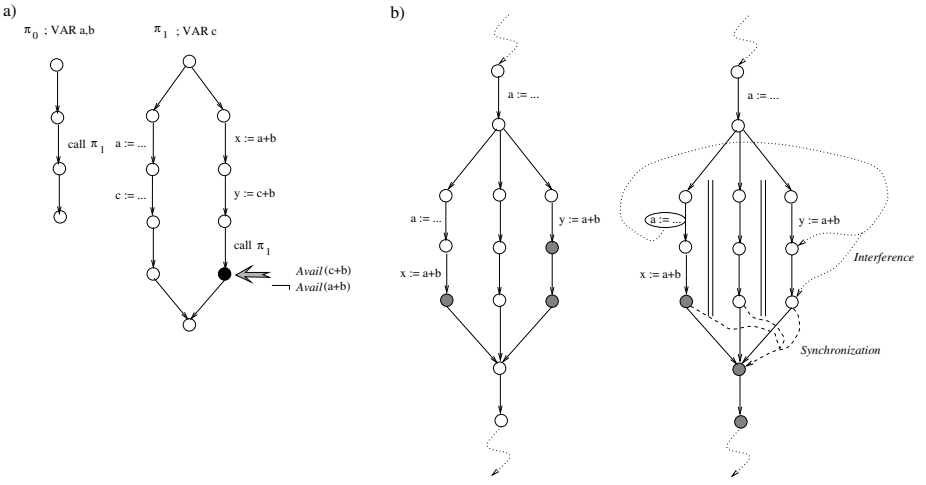

Fig. 5. Illustrating availability in the interprocedural and parallel setting.

The interprocedural variant of the intraprocedural coincidence theorem presented next captures programs with mutually recursive procedures, global and local variables, and value and reference parameters [24,26].[3]

[3] Sharir and Pnueli were the first who presented an interprocedural extension of the intraprocedural coincidence theorem (cf. [37]). Their version, however, did not cap-

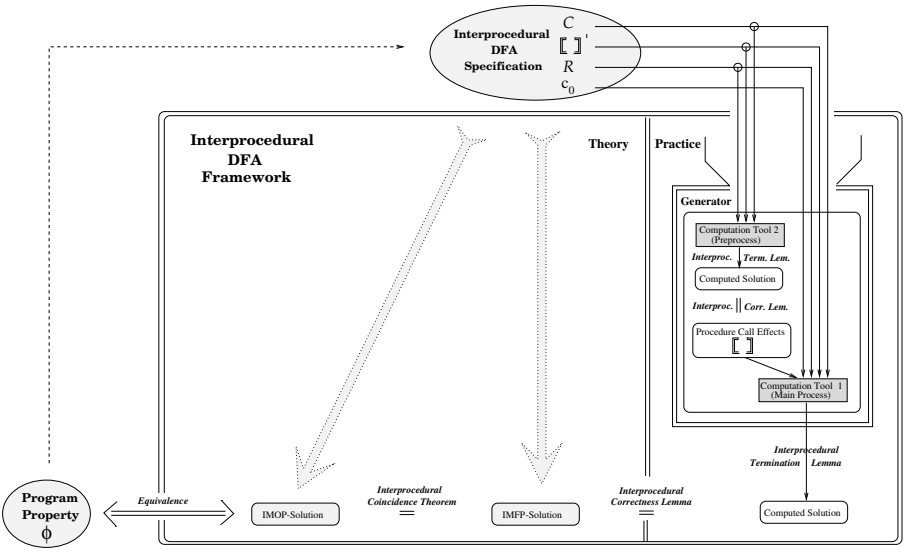

Fig. 6. The interprocedural DFA-framework.

Theorem 2 (Interprocedural Coincidence Theorem).
The interprocedural *MFP- and MOP-solution coincide, if the local semantic functions and the return functions of the abstract interpretation are all distributive.*

A collection of applications of the framework of Figure 6 including the interprocedural counterpart of the availability problem considered in Section 2 can be found for example in [15] and [24].

4 Data-parallel and Object-oriented Data-flow Analysis

The interprocedural machinery considered in the previous section can rather straightforwardly be enhanced to *data-parallel* languages like High Performance Fortran (HPF) [8], Fortran D [9], or Vienna Fortran [39], and to *object-oriented* languages like Smalltalk [10] or Oberon [38]. In fact, Figure 6 can be considered an illustration of both the data-parallel and object-oriented situation, too, and thus we do not present a separate figure here. In [20] this has been exploited for the data-parallel setting of HPF considering *distribution assignment placement* (*DAP*) as application. This is a new aggressive optimization which reduces communication costs in HPF-programs by eliminating partially redundant and

ture local variables and parameters of recursive procedures. The version presented in [15] captures even procedural parameters.

partial dead (re-)distributions.[4] The DFAs required for DAP, which resemble the one for availability, were specified according to the pattern of Figure 6 of Section 3. In [18] and [25] the approach has been extended to an object-oriented setting set up by a Smalltalk- and an Oberon-like language, considering type analysis as application. In all these cases a coincidence theorem relating the computational and the specificational part of the framework is crucial. For illustration we here recall the coincidence theorem of the DFA-framework fitting to the object-oriented setting considered in [18].

Theorem 3 (The Object-oriented Coincidence Theorem).
The object-oriented *MFP- and MOP-solution coincide, if the local semantic functions* (*including the filter functions*)[5] *of the abstract interpretation are all distributive.*

5 Parallel Data-flow Analysis

In this section we consider explicitly parallel programs with interleaving semantics and shared memory. In this setting one is faced with the phenomena of *interference* and *synchronization*. Figure 5(b) illustrates their impact by opposing a sequential and a parallel program using the availability of $a+b$ for demonstration. Of course, parallel programs can equivalently be expressed by a sequential "product" program which make all the interleavings explicit. Though this would allow us to directly apply the results of intraprocedural DFA, it would not be of much practical use as the size of the product program is exponential in the number of parallel components: a dilemma often condensed to the catch-phrase "state explosion problem." For the large and practically most important class of bitvector problems (cf. [11]), however, it has been shown that interleavings need not be considered at all to capture the effects of interference and synchronization (cf. [27,29]). This allows us a two-step approach similar as in the interprocedural case. The key of the *MFP*-approach of the parallel setting is a preprocess, which in an innermost fashion computes the semantics of parallel statements. As interprocedurally, the designer of a (bitvector) DFA need not to know any details of this process when applying the framework. The treatment of capturing interference and synchronization can be encapsulated inside the framework and its corresponding generator. For the parallel setting we have the following version of the coincidence theorem [27,29]. It applies to bitvector problems. However, extensions to specific non-bitvector problems are possible (cf. [17]).

Theorem 4 (Parallel Bitvector Coincidence Theorem).
The parallel *MFP- and MOP-solution for bitvector problems coincide.*

Following the pattern of Figure 7, all the bitvector analyses (e.g. availability and very busyness of terms, reaching definitions or liveness of variables), which

[4] In first practical measurements this optimization proved to be most powerful: often a speed-up of several hundred per cent have been observed (cf. [20,21]).
[5] See Section 6.

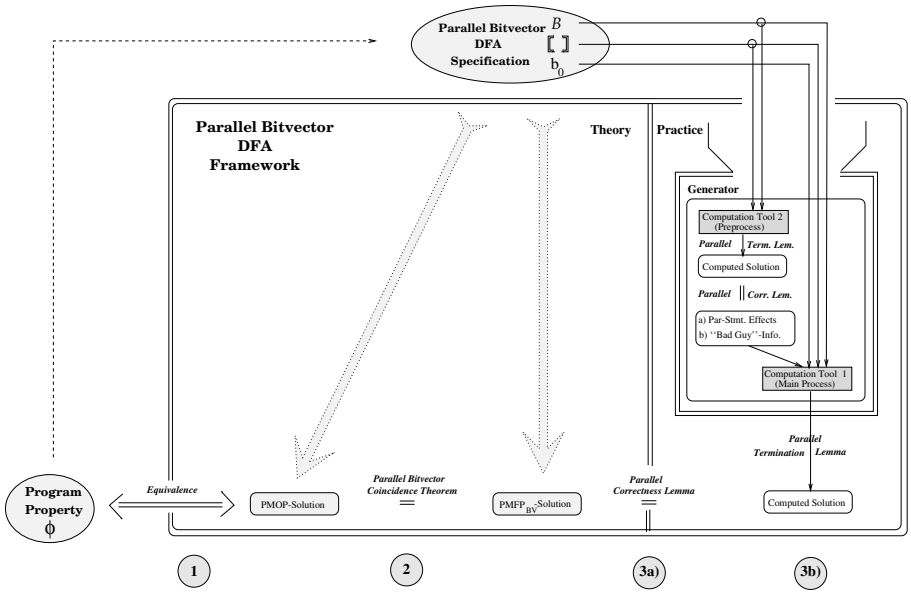

Fig. 7. The parallel DFA-framework.

have originally been developed in the sequential imperative paradigm (cf. [11]), can now be transferred to the parallel setting together with the optimizations based thereon. In [28] and [16] this has been demonstrated for *code motion* (cf. [22,32]) and *partial dead-code elimination* (cf. [23]), respectively.

6 Conditional Data-flow Analysis

Conditional branches are usually nondeterministically interpreted in DFA in order to avoid undecidabilities. The framework of abstract interpretation, however, is inherently powerful enough in order to also properly deal with conditional branching. Here, we demonstrate this for the intraprocedural setting. Technically, this can be achieved by introducing *filter functions* of the form

$$f_c : \mathcal{C} \to \mathcal{C} \quad \text{defined by} \quad \forall\, c' \in \mathcal{C}.\ f_c(c') =_{df} c \sqcup c'$$

where $\sqcup$ is the lattice operator dual to the one for modelling the "merge" of data-flow information at join nodes of the control flow. Intuitively, a filter function matching the pattern above enriches the current data-flow information by the data-flow facts, which are guaranteed by the particular program branch taken. After introducing filter functions the DFA-process proceeds as in the intrapro-

cedural case. However, due to the special nature of the filter functions we need here the following version of the coincidence theorem.

Theorem 5 (Conditional Coincidence Theorem).
The conditional *MFP- and MOP-solution coincide, if the lattice, the local semantic functions, and the filter functions of the abstract interpretation are all distributive.*

In [18] and [25], an alternative variant of filter functions have been introduced, aiming at achieving an "almost" deterministic treatment of program branches and method calls. Basically, the filters introduced there propagate data-flow information (i.e., type information) only along program branches they are qualified for. Thus, in contrast to the filter functions sketched above, they do not enrich data-flow information according to the branching condition, but act like a sieve letting pass only those parts of the information satisfying the (abstractly interpreted) branching condition. This is discussed in detail in [18,25]. For the purpose of this article it suffices that these filters fit to the general pattern of Figure 1, which can be considered an abstraction of the corresponding instance of the object-oriented setting.

7 Conclusions

The origins of DFA-frameworks based on abstract interpretation lie in the imperative programming paradigm with the main focus on intraprocedural and interprocedural DFA. In this article we reconsidered this approach from a generic point of view. We showed that the resulting generic framework has instances in quite different programming paradigms ranging from the classical imperative over the parallel and data-parallel one to the object-oriented paradigm, which are becoming more and more important in practice. From the perspective of a DFA-designer this unifying approach simplifies to switch between paradigms as well as to transfer analyses beyond paradigm boundaries. Moreover, as a by-product of our approach, we obtained a natural decomposition of the DFA-frameworks into a "theoretical" and "practical" part suggesting a uniform principle for the construction of DFA-generators. In each case the backbone of the decomposition is a specific coincidence theorem relating the solution computed by a DFA-algorithm to the solution specified. According to this principle DFA-generators (or tool kits) have already successfully been realized for intra- and interprocedural DFA, e.g., in terms of the DFA&OPT-MetaFrame tool kit [14], and in similar form in the DFA-generator systems PAG (cf. [2]) and OPTIMIX (cf. [3,4]). As demonstrated here, these approaches can uniformly be extended to further paradigms and settings. An extension to parallel programs is integrated in the tool kit of [14], further extensions are in progress.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

2. M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Proc. 2nd Int. Static Analysis Symp.* (*SAS'95*), LNCS 983, pages 33 – 50. Springer-V., 1995.
3. U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proc. 6th Int. Conf. on Compiler Const.* (*CC'96*), LNCS 1060, pages 121 – 135. Springer-V., 1996.
4. U. Aßmann. OPTIMIXing. In *Proc. Poster Session 7th Int. Conf. on Compiler Const.* (*CC'98*), pages 28 – 35. Departamento de Informática, Univ. Lisboa, 1998.
5. 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.
6. 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.
7. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. of Logic and Computation*, 2(4):511 – 547, 1992.
8. High Performance Fortran Forum. High Performance Fortran language specification version 2.0. Technical report, Rice University, Houston,TX, January 1997. Available via HPFF home page: http://www.crpc.rice.edu/HPFF.
9. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. FORTRAN D language specification. Technical report, Rice University, Houston,TX, January 1992.
10. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.
11. M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.
12. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305 – 317, 1977.
13. G. A. Kildall. A unified approach to global program optimization. In *Conf. Rec. 1st Symp. Principles of Prog. Lang.* (*POPL'73*), pages 194 – 206. ACM, NY, 1973.
14. 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.
15. 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.
16. J. Knoop. Eliminating partially dead code in explicitly parallel programs. *TCS*, 196(1-2):365 – 393, 1998. (Special issue devoted to *Euro-Par'96*).
17. J. Knoop. Parallel constant propagation. In *Proc. 4th Europ. Conf. on Parallel Processing* (*Euro-Par'98*), LNCS 1470, pages 445 – 455. Springer-V., 1998.
18. J. Knoop and W. Golubski. Abstract interpretation: A uniform framework for type analysis and classical optimization of object-oriented programs. In *Proc. 1st Int. Symp. on Object-Oriented Technology* (*WOON'96*), pages 126 – 142, 1996.
19. J. Knoop, D. Koschützki, and B. Steffen. Basic-block graphs: Living dinosaurs? In *Proc. 7th Int. Conf. on Compiler Construction* (*CC'98*), LNCS 1383, pages 65 – 79. Springer-V., 1998.
20. J. Knoop and E. Mehofer. Interprocedural distribution assignment placement: More than just enhancing intraprocedural placing techniques. In *Proc. 5th Int. Conf. on Parallel Architectures and Compilation Techniques* (*PACT'97*), pages 26 – 37, 1997.

21. J. Knoop and E. Mehofer. Optimal distribution assignment placement. In *Proc. 3rd Europ. Conf. on Parallel Processing* (*Euro-Par'97*), LNCS 1300, pages 364 – 373. Springer-V., 1997.
22. J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. Prog. Lang. Syst.*, 16(4):1117–1155, 1994.
23. J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.* (*PLDI'94*), volume *29*,6 of *ACM SIGPLAN Not.*, pages 147 – 158, 1994.
24. J. Knoop, O. Rüthing, and B. Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *J. Prog. Lang.*, 4(4):211–246, 1996.
25. J. Knoop and F. Schreiber. Analysing and optimizing strongly typed object-oriented languages: A generic approach and its application to Oberon-2. In *Proc. 2nd Int. Symp. on Object-Oriented Technology* (*WOON'97*), pages 252 – 266, 1997.
26. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proc. 4th Int. Conf. on Compiler Construction* (*CC'92*), LNCS 641, pages 125 – 140. Springer-V., 1992.
27. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Bitvector analyses → No state explosion! In *Proc. 1st Int. Workshop on Tools and Algorithms for Constr. and Analysis of Syst.* (*TACAS'95*), LNCS 1019, pages 264 – 289. Springer-V., 1995.
28. J. Knoop, B. Steffen, and J. Vollmer. Code motion for parallel programs. In *Proc. of the Poster Session of the 6th Int. Conf. on Comp. Constr.* (*CC'96*), pages 81 – 88. TR LiTH-IDA-R-96-12, 1996.
29. 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.
30. K. Marriot. Frameworks for abstract interpretation. *Acta Informatica*, 30:103 – 129, 1993.
31. P. M. Masticola, T. J. Marlowe, and B. G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Trans. Prog. Lang. Syst.*, 17(5):777 – 802, 1995.
32. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Comm. ACM*, 22(2):96 – 103, 1979.
33. R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.
34. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.
35. S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1981.
36. F. Nielson. A bibliography on abstract interpretations. *ACM SIGPLAN Not.*, 21:31 – 38, 1986.
37. 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.
38. Niklaus Wirth. The programming language Oberon. In *Software-Practice and Experience*, volume 18, pages 671–690. John Wiley and Sons, 1988.
39. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - A language specification version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992.

10.1007/b107031130025