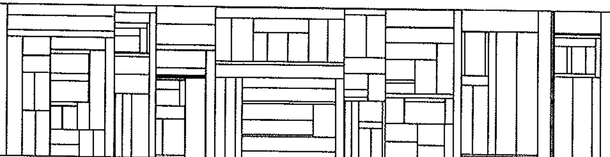
ISSN 0105-8517

Data Flow Analysis as Model Checking

Bernhard Steffen

DAIMI PB – 325
July 1990

COMPUTER SCIENCE DEPARTMENT
AARHUS UNIVERSITY
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark



ISSN 0105-8517

PB – 325 B. Steffen: Data Flow Analysis as Model Checking

Data Flow Analysis as Model Checking

Bernhard Steffen *

Abstract

The paper develops the idea that modal logic provides an appropriate framework for the specification of data flow analysis (DFA) algorithms as soon as programs are represented as models of the logic. This can be exploited to construct a DFA-*generator* that generates efficient DFA-algorithms from modal specifications by partially evaluating a specific model checker wrt the specifying modal formula. Moreover, the use of a modal logic as specification language for DFA-algorithms supports the compositional development of specifications and structured proofs of properties of DFA-algorithms. These ideas are applied to the problem of determining optimal computation points within flow graphs.

1 Introduction

Data flow analysis (DFA) is concerned with the automatic identification of program points enjoying specific properties — for example lifeness of variables, equivalence of program terms, etc. Typically, data flow analysis algorithms are constructed for a given program property Φ of interest and therefore have the following functionality:

$$\text{DFA-algorithm}: \text{ programs } \rightarrow \text{ program points enjoying property } \Phi$$

Model checking is concerned with the automatic identification of those states of a finite state model[1] satisfying a specific modal (or temporal) formula. Typically, such formulas express deadlock, divergence, liveness, etc. Model checkers are parameterized on the formula of interest and therefore have the following functionality:

$$\text{model checker}: \text{ modal formulas } \Phi \times \text{ model } \rightarrow \text{ states satisfying } \Phi$$

*Department of Computer Science, University of Aarhus, DK-8000 Aarhus C

[1] Essentially these models are finite automata.

Identifying programs with models, program points with states and program properties with modal formulas, model checkers can be seen as DFA algorithms that have the program property of interest as parameter.

In this paper we exploit this observation in order to develop an algorithm that generates DFA-algorithms from specifications written in a modal logic. In essence, this DFA-*generator* works by *partially evaluating* an appropriate model checker wrt its modal formula parameter; the result is a bit-vector DFA-algorithm. Our framework covers the standard bit-vector DFA algorithms [He] in an efficient manner: it allows concise high level specifications, and the generated algorithms are guaranteed to be linear in the size of the program being analysed.

Another benefit of the approach proposed here concerns the specification and investigation of DFA-algorithms. Both can be done modularly by reasoning within the modal logic serving as specification language. All this is illustrated by means of an example of practical relevance: an improved version of Morel/Renvoise's algorithm for eliminating partial redundancies [MR]. The algorithm generated here is linear in the size of the argument program and its results are optimal. This improves on the complexity estimation $O(n \log(n))$ given in [Dha][2]. Moreover, to our knowledge, the only comparable optimality results that have been proved before are the ones in [SKR1, SKR2], which concern more complex placement algorithms.

Summary of Technical Results

Section 2 presents the program representation, which consists of transition systems, where nodes and states are labeled. This representation is very close to the standard models for modal logics [Sti1], and it allows a simple adaptation of the program representations used in DFA. For example, nondeterministic flow graphs, a standard program representation in DFA, can be easily transformed into this format.

Section 3 develops the specification languages: a low level specification language with a general fixpoint operator, and a high level specification language, where the general fixpoint operator is replaced by intuitively easy to understand derived operators. The section closes with logical characterizations of structural constraints of certain program models.

[2]Here n stands for the number of edges in the flow graph. Dhamdhere also mentions that his algorithm is $O(n)$ for specific graph structures. In contrast, the estimation here does not need any structural requirements.

Section 4 deals with a "real life" example. A bidirectional DFA-algorithm determining the optimal placement of computations within programs is specified and its correctness and optimality is established. All the reasoning is done purely within the high level specification language.

Section 5 provides a correct and complete model checker, which is linear in the size of the program model being investigated. The DFA-generator works by partially evaluating this model checker wrt to the modal formula used to specify the DFA-property of interest. The partial evaluation process is illustrated by means of the generation of a DFA-algorithm from the specification developed in Section 4.

Finally, Section 6 contains conclusions and directions for future work.

Related Work

Already in the seventies a DFA-generator has been developed that essentially works on syntax trees and generates DFA-algorithms from specifications given as computation functions for attribute values [Wil1, Wil2]. Thus the specifications explicitly describe the way in which the program properties of interest are determined. To our knowledge, this principle has been maintained in all later developments (cf. [Gie, Nie2, LPRS, SFRW, Ven]). In contrast, we specify DFA-algorithms just by means of the program properties under consideration. All the details about the corresponding analysis algorithm are hidden in the model checker our approach is based upon. This yields concise high level specifications, simplifies the specification development and supports the reasoning about features, such as correctness and optimality, of the corresponding DFA-algorithms.

2 Models for Programs

We model programs as transition systems whose states and transitions are labeled with sets of atomic propositions and actions, respectively. Intuitively, atomic propositions describe properties of states, while actions describe (properties of) statements. As usual, the control flow is modelled by the graph structure of the transition system.

Definition 2.1 *A* program model $\mathcal{P}$ *is a quintuple* $(S, \mathcal{A}, \rightarrow, \mathcal{B}, \lambda)$ *where*

1. *S is a finite set of* nodes *or* program states.

2. $\mathcal{A}$ *is a set of* actions.

3. $\rightarrow \subseteq S \times 2^{\mathcal{A}} \times S$ *is a set of* labeled transitions, *which define the control flow of* $\mathcal{P}$.

4. $\mathcal{B}$ *is a set of* atomic propositions.

5. λ *is a function* $\lambda : S \rightarrow \mathcal{B}$ *that label states with subsets of* $\mathcal{B}$.

We will write $p \xrightarrow{A} q$ *instead of* $(p, A, q) \in \rightarrow$, *and given* $\alpha \subseteq 2^{\mathcal{A}}$ *we will call* p *an* α-predecessor *of* q *and* q *an* α-successor *of* p *if* $A \in \alpha$. *The set of all* α*-predecessors and* α*-successors will be abbreviated by* $Pred_\alpha$ *and* $Succ_\alpha$, *respectively*[3].

Essentially, a program model is a combination of a standard labeled transition system and a Kripke structure, which allows us to speak about state and statement properties explicitly and separately without using any complicated encodings. New is only the fact that transitions are labeled with *sets* of actions, rather than just single actions, here, which is necessary when dealing with *abstract interpretations*, where a concrete statement is treated as a set of properties (cf. [CC1]).

Definition 2.2 *A* DFA-model *is a program model with two distinct states* **s** *and* **e** *satisfying:*

- **s** *and* **e** *do not possess any predecessor and successor, respectively.*
- *Every program state is reachable from* **s**.
- **e** *is reachable from every program state.*

We call **s** *and* **e** start state *and* end state, *respectively.*

The additional constraints for DFA-models are standard in data flow analysis, and in fact, they do not impose any restrictions there, because flow graphs can be modified accordingly without any harm.

Modelling Nondeterministic Flow Graphs

We will consider *nondeterministic flow graphs* as DFA-models. Nondeterministic flow graphs are directed graphs whose nodes represent statements (as usual, we will concentrate on assignments here) and whose

[3] Note that the $2^{\mathcal{A}}$-predecessors and $2^{\mathcal{A}}$-successors are just the predecessors and successors in the usual sense.

edges represent the flow of control. As mentioned above, we can additionally assume that they possess unique start and end nodes. There are two straightforward ways to transform flow graphs into DFA-models. First, by pushing the statements from the nodes into the outgoing edges. In this case, we arrive at a *precondition model*, because here the nodes will characterize preconditions to the statements that have been originally associated with the nodes. Second, and dually, by pushing the statements upwards into the ingoing edges. Here one arrives at a *postcondition model*[4]. In the discussion of our example we will deal with postcondition models. In general, the appropriate choice of model depends on the particular application.

In order to establish the setup for our "real life" example (see Section 4), let $\mathbf{V}$ and $\mathbf{T}$ be sets of program variables and program terms, respectively, and A_c be the set of all assignments of the form $v := t$, where $v \in \mathbf{V}$ and $t \in \mathbf{T}$. Furthermore, let $\mathcal{B} =_{df} \{\texttt{start},\ \texttt{end}\}$ be a set of state labels with the labeling function given by $\lambda_c(\mathbf{s}) = \{\texttt{start}\}$ and $\lambda_c(\mathbf{e}) = \{\texttt{end}\}$. Then the postcondition models (or equivalently the precondition models) of a flow graph form a DFA-model $(\mathcal{S}, A_c, \rightarrow, \mathcal{B}, \lambda_c)$, where the state labeling just identifies the start state $\mathbf{s}$ and the end state $\mathbf{e}$ and the transition labeling the corresponding statements.

We will refer to such DFA-models as *concrete* DFA-models. However, the DFA-models we want to deal with, and which allow an automatic analysis, arise as abstractions from concrete DFA-models. A typical abstraction is given by choosing $A_a = \{\texttt{mod}(t) \mid t \in \mathbf{T}\} \cup \{\texttt{use}(t) \mid t \in \mathbf{T}\}$ as the set of transition labels together with the abstraction function $abstr : A_c \rightarrow A_a$ which is defined by:

$$abstr(\{v := t\}) =_{df}$$
$$\{\texttt{mod}\,(t') \mid v \text{ is subterm of } t'\} \cup \{use\,(t') \mid t' \text{ is a subterm of } t\}$$

and the additivity property:

$$abstr(A) = \bigcup \{\, abstr(v := t) \mid v := t \in A\,\}$$

Transitions labeled with $\texttt{mod}(t)$ or $\texttt{use}(t)$ represent statements that *modify* or *use* the term t. Our illustrating example will work with this abstraction, which is tailored to address problems dealing with *invariance*

[4] In order to make these transformations work in general, we assume that start and end nodes of flow graphs represent skip statements.

and *usage* of program terms, and in particular, program variables. In fact, many DFA-problems can be dealt with by means of this abstraction or a slight extension. However, in general, the appropriate abstract interpretation of the statements must be chosen problem dependently.

3 The Specification Languages

We have two specification languages, a *low level* language, which is primary, and a *high level* language consisting of derived operators of the low level language. Whereas the low level language is used to formally define the semantics of formulas, the high level language is easier to understand and should be used for specification.

3.1 Low Level Specifications

Our low-level specification language is essentially a sublanguage of the modal mu-calculus [Koz], which is characterized by a restricted use of fixpoint constructions. The syntax of our low level specification language is parameterized wrt denumerable sets *Var*, $\mathcal{B}$ and $\mathcal{A}$ of propositional variables, atomic propositions and actions, respectively. Let X range over *Var*, β over $\mathcal{B}$ and α over subsets of $2^{\mathcal{A}}$. Then the formulas of the logic are given by the following grammar.

$$\Phi ::= X \mid tt \mid \Phi \wedge \Phi \mid \neg\Phi \mid \beta \mid [\,\alpha\,]\,\Phi \mid \overline{[\,\alpha\,]}\,\Phi \mid \nu X.\Phi$$

In formula $\nu X.\Phi$, the free occurrences of X in Φ are bound by ν in the usual fashion, and the substitution $\Phi[\Gamma/X]$ is also defined in the standard way.

The semantics of closed formulas is defined wrt a program model $\mathcal{P}$. Let us now provide the intuition behind closed formulas. Every program state satisfies the formula *tt*, while program state p satisfies $\Phi_1 \wedge \Phi_2$ if it satisfies both Φ_1 and Φ_2. Moreover, a program state satisfies $\neg\Phi$ if it does not satisfy Φ, and it satisfies β if it is labeled by a set containing β. A program state p satisfies $[\,\alpha\,]\,\Phi$ if every α-successor satisfies Φ. Note that this implies that a program state p satisfies $[\,\alpha\,]\,ff$ exactly when p has no α-successors. Analogously, a program state p satisfies $\overline{[\,\alpha\,]}\,\Phi$ if every α-predecessor satisfies Φ. Thus in analogy, a program state p satisfies $\overline{[\,\alpha\,]}\,ff$ exactly when p has no α-predecessors. The formula $\nu X.\Phi$ is a

$$
\begin{array}{rcl}
[\![tt]\!] & = & S \\
[\![\Phi_1 \wedge \Phi_2]\!] & = & [\![\Phi_1]\!] \cap [\![\Phi_2]\!] \\
[\![\neg\Phi\,]\!] & = & S \setminus [\![\Phi]\!] \\
[\![\beta]\!] & = & \{p \in S \,|\, \beta \in \lambda(p)\} \\
[\![[\,\alpha\,]\,\Phi]\!] & = & \{\,p \in S \mid \forall q \in \mathit{Succ}_{\alpha}(p).\; q \in [\![\Phi]\!]\,\} \\
[\![\overline{[\,\alpha\,]}\,\Phi]\!] & = & \{\,q \in S \mid \forall p \in \mathit{Pred}_{\alpha}(q).\; p \in [\![\Phi]\!]\,\} \\
[\![\nu X.\Phi]\!] & = & \bigcup\{\,S' \subseteq S \,|\, S' \subseteq [\![\Phi_{S'}]\!], \text{ where } \Phi_{S'} \text{ is } \Phi \text{ with } X \\
 & & \qquad\qquad \text{interpreted as } S'\,\}
\end{array}
$$

Figure 1: The semantics of formulas.

recursive formula and should be thought of as the "largest" solution to the "equation" $X = \Phi$. Since $\mathcal{P}$ is finite-state, this formula is equivalent to the infinite conjunction $\bigwedge_{i=0}^{\infty} \Phi_i$, where

$$
\begin{array}{rcl}
\Phi_0 & = & tt \\
\Phi_{i+1} & = & \Phi[\Phi_i/X]
\end{array}
$$

As usual, there is a syntactic restriction on expressions of the form $\nu X.\Phi$, which is necessary to ensure the continuity of the fixpoint operator: X is required to appear within the range of an even number of negations in Φ. All this is completely standard, except for the meaning of modalities, which is defined for sets of transition labels, i.e. sets of sets of actions here, rather than just for single actions. This double powerset construction arises naturally in our setting: the first level is necessary, because we want to model statements abstractly by means of sets of properties, and the second level corresponds to the convenient generalization of modalities, which allows to speak about sets of transition labels, rather than just singletons, which simplifies the representation of certain properties enormously (cf. [BS]).

The formal semantic definition of the logic maps closed formulas to sets of program states—intuitively, the program states for which the formula is "true". Figure 1 describes the formal definition. Note that the semantics of $\nu X.\Phi$ uses the Tarski fixpoint theorem [Tar] to define the meaning of this formula as the greatest fixpoint of a monotonic function

over the powerset of the set of states. The monotonicity of this function follows from the monotonicity of the semantic interpretation of the other propositional constructors.

In the following we will write $[\,.\,]$ or $\overline{[\,.\,]}$ instead of $[\,2^{\mathcal{A}}\,]$ or $\overline{[\,2^{\mathcal{A}}\,]}$. Moreover, we will use $\mathcal{P} \models \Phi$ (or just $\models \Phi$ if $\mathcal{P}$ is understood) to indicate that every state of the program model $\mathcal{P}$ satisfies Φ. As usual, we can define the following duals to the operators of our language and the implication operator $\Rightarrow$ by:

$$\begin{aligned} ff &= \neg tt \\ \Phi_1 \vee \Phi_2 &= \neg(\neg\Phi_1 \wedge \neg\Phi_2) \\ \langle \alpha \rangle\, \Phi &= \neg[\alpha]\,(\neg\Phi) \\ \overline{\langle \alpha \rangle}\, \Phi &= \neg\overline{[\alpha]}\,(\neg\Phi) \\ \mu X.\Phi &= \neg\,\nu X.\neg(\Phi[\neg X/X]) \\ \Phi \Rightarrow \Psi &= \neg\Phi \vee \Psi \end{aligned}$$

Our low level specification language consists of all *closed* and *guarded* formulas, where no variables occur free inside the scope of a fixpoint expression[5]. Closed means that all variables are bound by a fixpoint operator, and guarded that all variables occur inside the range of a modality.

In future, we will also use the derived operators except for the minimal fixpoint operator, which we avoid here in order to keep the presentation of the model checker (Section 5.1) as simple as possible.

3.2 High Level Specifications

The recursive proposition constructors add a tremendous amount of expressive power to the logic (cf. [EL, Ste1]). For example, they allow the description of invariance (or *safety*) and eventuality (or *liveness*) properties. However, the formulas are in general unintuitive and difficult to understand. We will therefore define a collection of intuitively easy to understand derived operators that are based on the following "Henceforth"-operator of the temporal logic CTL [CES]:

$$\mathbf{AG}\,\Phi = \nu X.(\Phi \wedge [\,.\,]X)$$

[5]The point of this condition is to avoid the possibility of *alternated nesting* [EL]. Of course, there are weaker conditions to guarantee this, but they are unnecessarily complicated for our purpose.

and its past-time counterpart:

$$\overline{\mathbf{AG}}\ \Phi \;=\; \nu X.(\Phi \wedge \overline{[\,.\,]}\,X)$$

AG Φ holds of p if Φ holds for every state of every path that starts in p, while $\overline{\mathbf{AG}}$ Φ holds of p, if Φ holds for every state of every path that ends in p.

DFA is concerned with a specific kind of *eventuality* properties: certain program transformations are only admissible if a specific value must be computed before the program terminates, i.e. before the end state is reached (cf. Section 4). This cannot be expressed with the standard CTL operators. However, the following parameterized version of the Henceforth-operator suites this purpose:

$$\begin{aligned}\mathbf{AG}_\alpha\ \Phi &= \nu X.(\Phi \wedge [\,\alpha\,]\,X)\\ \overline{\mathbf{AG}}_\alpha\ \Phi &= \nu X.(\Phi \wedge \overline{[\,\alpha\,]}\,X)\end{aligned}$$

Intuitively, the parameter reduces the set of relevant paths to those being labeled with elements of α. Thus in order to express that a computation must happen before the end state is reached, one may equivalently express that one never reaches the end state on a path, whose transitions do not perform this computation. Formally, this is expressed by setting $\Phi = \neg\texttt{end}$ and $\alpha = 2^{\mathcal{A}\setminus\{\texttt{comp}\}}$, where comp represents the computation of interest. This pattern is typical for specifications in our framework.

Our high level specification language arises from the low level specification language by replacing the general fixpoint operator with the operators established above. This language is already quite expressive, and it suffices for the specification of the standard bit-vector algorithms. However, other operators, like for example the strong "Until"-operator or the existential path quantifier of CTL [CES], may be added later to enhance the expressive power of the language.

The three structural restrictions for DFA-models given in Definition 2.2 are characterized by:

Proposition 3.1 (DFA-Models)
A DFA-model is characterized by the following three properties:

1. $\models (\texttt{start} \Rightarrow \overline{[\,.\,]}\,\mathit{ff}) \wedge (\texttt{end} \Rightarrow [\,.\,]\,\mathit{ff})$
2. $\models \neg(\,\mathbf{AG}\ \neg\texttt{end}\,)$

3. $\models \neg(\overline{\mathbf{AG}}\ \neg\mathtt{start})$

Also, a property of postcondition models, which is important for the proof of the Correctness Theorem 4.1 and the Optimality Theorem 4.2, can be stated within our specification language:

Proposition 3.2 (Postcondition Models)
A postcondition model satisfies: $\models \overline{\langle . \rangle}\,\langle \alpha \rangle tt \Rightarrow \overline{[.]}\,\langle \alpha \rangle tt$

This demonstrates that our specification language is not limited to specify DFA-algorithms. It is expressive enough to cover structural properties of program models as well. This allows us to prove properties of DFA-algorithms within our logical framework, even if these properties depend on structural restrictions of the program models under consideration. We will illustrate this in the next section.

4 Example: Optimal Placement of Computations

In this section we will develop a specification for a bi-directional DFA-algorithm that determines optimal computation points for a given term t within a DFA-model (flow graph). The development improves on the original work by Morel/Renvoise [MR] in that it also establishes the optimality of the placement. We will fix t from now on, in order to be able to drop t from the argument list of some predicates and therefore simplify the notation.

4.1 Specifying the DFA-Algorithm

It is well-known that in completely arbitrary graph structures the placement process may deliver unsatisfactory results, because specific patterns may cause that the code motion process gets blocked. This problem can be solved by means of the following transformation: insert an artificial state in each transition that starts at a state with more than one successor and ends at a state with more than one predecessor (cf. [Dha, SKR1, SKR2][6]). For postcondition models, the essence of this transformation can be characterized logically as follows:

[6] In [Dha] this is done implicitly by placing the computations in the edges of the flow graph under consideration.

Placement Models $\models (\Phi \Rightarrow \overline{[\,.\,]}[\,.\,]\Phi) \vee (\overline{\langle\,.\,\rangle}\Phi \Rightarrow \overline{[\,.\,]}\Phi)$

Intuitively, this means that there are two classes of states in a placement model:

- the ones that are "similar" to all their brothers, and
- the ones whose predecessors are all "similar".

In fact, it is possible to obtain an optimal placement algorithm, as soon as we restrict ourselves to postcondition models with this property, which we call *postcondition placement models* (cf. Optimality Theorem 4.2).

In our framework DFA-algorithms are specified by means of the program property they are checking for. Thus the DFA-algorithm to determine the optimal placement of computations is specified by means of the specification of the optimal computation points. They can be characterized in two steps.

First, the placement of a computation at a computation point must be *safe*, i.e. it must not introduce computations of new values on paths. This requirement is necessary in order to guarantee that no run time errors (e.g. division by 0) are introduced: a safe placement does not change the potential for run time errors. This property is satisfied if the inserted computations are *necessary*, i.e. if their values will be computed on every continuation of a program execution that terminates in **e**. Logically, this property of a program point can be characterized by:

Guaranteeing Safety $\mathtt{NEC} =_{df} \mathbf{AG}_{U^c}(\neg \mathsf{end} \wedge [M \cap U^c]\mathit{ff})$

where $M =_{df} \{A \mid \mathtt{mod} \in A\}$, $U =_{df} \{A \mid \mathtt{use} \in A\}$ and "^{c}" is the set complement operator.

Second, to achieve optimality, we require that computations should be placed as "early" as possible. This can be logically characterized as follows: $\overline{[M^c]}(\overline{\mathbf{AG}}_{M^c} \neg\mathtt{NEC})$, meaning that an "earlier" placement would either be unsafe for t or an evaluation of t there would not always yield the required value. Together we obtain the following characterization of the (optimal) computation points:

The Computation Points $\mathtt{INIT} =_{df} \mathtt{NEC} \wedge \overline{[M^c]}(\overline{\mathbf{AG}}_{M^c} \neg\mathtt{NEC})$

In fact, INIT is already the complete specification of the DFA-algorithm to determine the optimal placement of computations.

4.2 Correctness and Optimality of the Placement

Correctness of a placement of computations means that all computations of the original flow graph are covered by computation points, i.e. the initialization of an auxiliary variable at the computation points allows to replace all original computations of the flow graph by the auxiliary variable without changing the program semantics. In order to express this property logically, we equip transitions ending in a state being labeled by INIT with init. This reflects the idea of inserting computations at the computation points in a postcondition model. The logical formulation of correctness is the heart of the following theorem

Theorem 4.1 (Correctness)
For every postcondition placement model being equipped with init *we have:*

$$\models \langle U \rangle tt \Rightarrow \overline{\mathbf{AG}}_{I^c}(\neg\mathtt{start} \wedge [M] ff)$$

where $I =_{df} \{ A \mid \mathtt{INIT} \in A \}$.

In the following we are going to show that there does not exist any placement of computations that is safe, covers all original computations and improves on the placement specified by init. Let us therefore first define the set of all *admissible placements*, i.e. those satisfying the first two properties. As before with init, we describe a placement by means of a transition labeling place. Admissibility is now logically characterized by:

Admissible Placements

$$\models \mathtt{NEC} \wedge ([U] \Rightarrow (\overline{\mathbf{AG}}_{P^c}(\neg\mathtt{start} \wedge [M] ff)))$$

where $P =_{df} \{ A \mid \mathtt{place} \in A \}$.

Intuitively this means that place specifies an admissible placement if the placement is safe at the place-marked transitions and each original computation of the program is covered by place-marked transitions (cf. definition of correctness).

Now, optimality means that there does not exist any admissible placement (here given by place) that improves on the placement specified by

`init`, i.e. on every computation path the number of transitions marked by `place` is as least as large as the number of transitions marked by `init`. The optimality of the placement can now be stated using the logical formulation of optimality. This theorem is actually new for Morel/Renvoise-like placement algorithms:

Theorem 4.2 (Optimality)
Every postcondition placement model and every admissible placement (here given by `place`*) satisfy:*

$$\models \ \mathbf{AG}\ \ (\,[\,I\,]\ (\,\mathbf{AG}_{P^c}\,(\neg\mathsf{end} \wedge [\,M \cap U^c\,]\,f\!f\,)))$$

The proofs of the Correctness Theorem 4.1 and the Optimality Theorem 4.2 can be derived from Proposition 3.2 and the defining property for placement models, purely by reasoning within the modal logic. — More details about placement algorithms can be found in [Dha, MR, RWZ, SKR1, SKR2].

5 DFA-Generation

5.1 The Principle

The principle of the DFA-generator we propose is *partial evaluation* of an appropriate model checker wrt the modal formula that serves as the specification of the DFA-algorithm. The model checker we are going to use is a variant of the algorithm proposed in [CES], modified to support the partial evaluation wrt to the specifying formula. It iteratively determines the set of all states of the program model under consideration that satisfy the argument formula. This is done by computing a maximal fixpoint over a node labeling consisting of bit-vectors that represent approximate truth values of certain (low level) modal formulas. Our model checker works in four steps:

The Model Checker

1. Translate the high level specification into the corresponding low level specification. This can be done in a straightforward manner. However, one can also add "optimizations" here that yield an equivalent, but more compact representation, as for example the *shared representation* used in Section 5.2.

2. Construct a (higher order) function from the low level formula that associates every potential program state with its corresponding *predicate transformer*, i.e. with a function that computes the next approximate bit-vector labeling for a given program state from the current approximate solutions (bit-vector values) of its predecessors and successors. The resulting predicate transformers operate on bit-vectors that have one component corresponding to each subformula that appears as an operand of a modality[7]. We will refer to these subformulas as *critical* subformulas. Given a program point p and a critical subformula Φ the predicate transformers update the corresponding bit-vector component with the truth value of the formula that arises from Φ by replacing all subformulas having a modality as the top most operator by their truth value under the current approximation.

   All this is rather straightforward, except for the following fact: approximations for critical subformulas that are separated by negation operations must be computed in a hierarchical manner, in order to guarantee the monotonicity of the iteration process. Therefore, the low level formula is also partitioned into *monotonicity levels*, the iteration mechanism of the third step depends upon. Note however that all this can be done *independently* of the particular program model under consideration.

3. Compute the greatest fixpoint over the bit-vector labeling of the program model under consideration wrt the predicate transformers that have been generated by the algorithm of the second step. This can be done by means of a standard *work list algorithm*[8] modified to determine the fixpoints monotonicity-level-wise: a level gets executed, if all its sub-levels are already dealt with.

   The computation of the greatest fixpoint for a given monotonicity level proceeds in two steps:

   (a) Initialize the bit-vectors of all program states to tt[9].

[7]In fact, it would be enough to consider only those subexpressions here that contain a variable, which is not guarded by a modality.

[8]A work list is a mean to obtain a fair and therefore terminating computation.

[9]Note that the usual frame conditions are implicit in the atomic propositions the states are labeled with, e.g. `start` and `end` of Section 4.

(b) Process the elements of the work list by applying their corresponding predicate transformer, until the maximal fixpoint is reached.

4. Finally, for each state the value of the complete specifying formula is determined using the fixpoint values computed in the third step. We will write $p \vdash \Phi$ if the complete formula holds of p.

This model checker is quite general: it can easily be extended to larger high level languages just by extending the first step. In particular an extension to CTL is straightforward. In analogy to [CES] we have:

Theorem 5.1 (Correctness, Completeness and Efficiency)
Let $\mathcal{P}$ be a program model and p be a state of $\mathcal{P}$. Then we have: $p \vdash \Phi$ iff $p \in [\![\Phi]\!]$. Moreover the effort to determine the set of all states satisfying Φ is proportional to the size of $\mathcal{P}$.

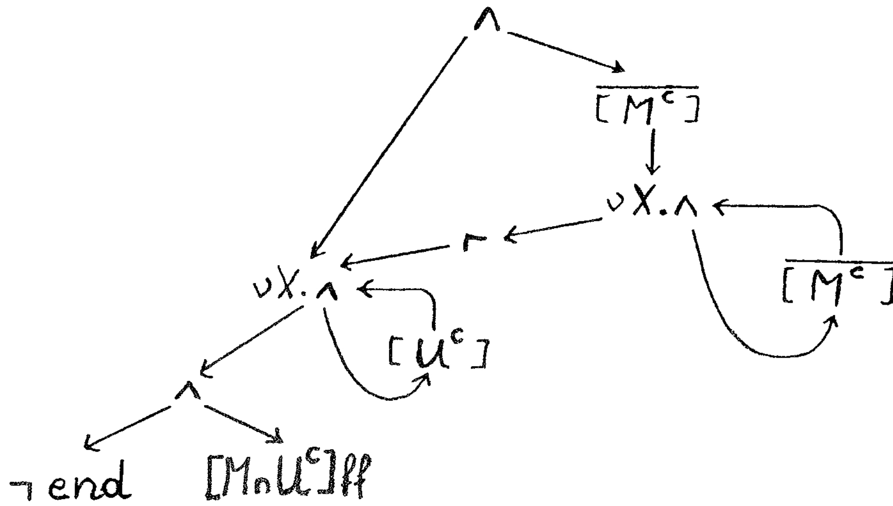
Partial evaluation of this model checker wrt a high level specification consists of executing the first two (program independent) steps, which results in a nested bit-vector algorithm. Instead of going into formal detail here, we will rather continue our "real life" example for illustration:

5.2 Continuation of the Example: Generating the DFA-Algorithm

The evaluation of the first step transforms the specifying formula

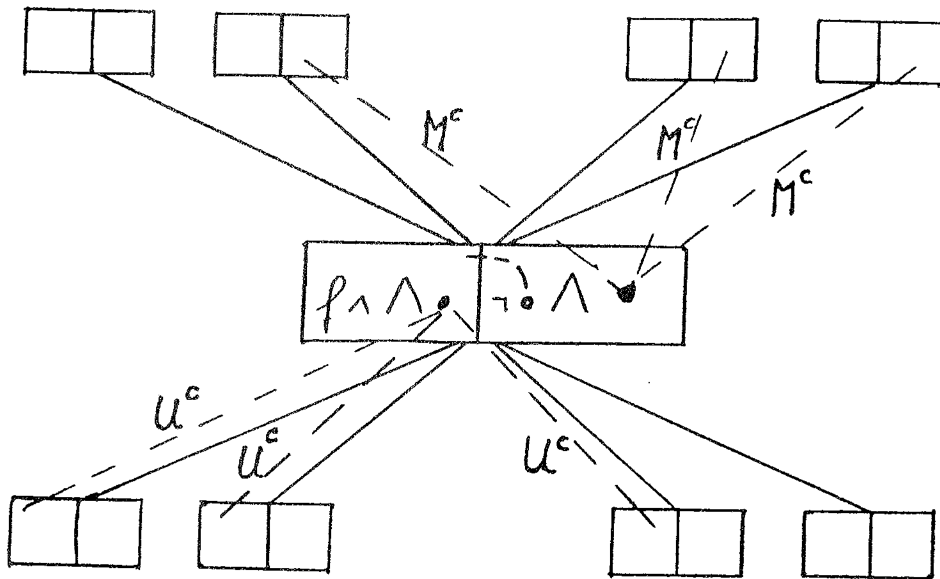
$$\texttt{INIT} \;=\; \texttt{NEC} \wedge \overline{[M^c]}(\overline{\mathbf{AG}}_{M^c}\neg\texttt{NEC})$$

by means of macro substitution into the following diagram:



Such shared representations take care of common subexpressions and therefore improve the efficiency of the DFA-algorithm generated.

The second step constructs a higher order function that associates every state of a program model with a predicate transformer, which realizes the evaluation of the formula above in the current approximation:



Here, the next approximation for the first bit of the bit-vector in the center of the picture is computed as the conjunction of $f =_{df} \neg\mathrm{end} \wedge [\,M \cap U^c\,]\,ff$ and the approximate values of the first bits of the bit-vectors of its U^c-successors. Analogously, the value of the second component of this bit-vector is updated by means of the approximate values of its own first bit and the ones of the second components of its M^c-predecessors. Note that the fixpoint computations for these two bits must be computed

hierarchically, because they belong to different monotonicity levels. — The result of these two evaluation steps is essentially a standard iterative DFA-algorithm for the predicate transformers specified above.

It is worth noting that we only need to iterate according to two Boolean values here; one for each fixpoint expression, or, one for the forward flow and one for the backward flow. Thus the generated algorithm only uses a bit-vector of length two, which improves on the algorithms proposed previously, see for example [Dha, MR].

6 Conclusion and Future Work

A framework has been developed, using a modal logic for the specification of DFA-algorithms. Main achievements of this development are the DFA-generator, which naturally arose by means of partially evaluating a specific model checker, and the modularity of both the development of DFA-specifications and the structure of proofs of properties of the DFA-algorithms generated. All these features have been illustrated by means of the problem of optimally placing computations within a program.

Currently, we focus on imperative languages and plan, as a first step, to implement a generator for intraprocedural DFA-algorithms as an extension of the Edinburgh Concurrency Workbench [CPS1, CPS2]. Subsequently, this generator will be extended to generate interprocedural algorithms as well. This can be done along the lines indicated in [SP].

Ultimately, it is planned to achieve language independency by using Mosses' *action notation* (cf. [Mos, MW]) as a common intermediate language. The program models for our DFA-generator will then be given by abstractly interpreted transitions systems, which arise from the corresponding structured operational semantics (cf. [Plo]). This will allow us to uniformly deal with imperative and functional languages and distributed systems.

Acknowledgement

I am very grateful to Rance Cleaveland, Hardi Hungar, Jens Knoop, Jens Palsberg and Oliver Rüthing for their careful proof reading and their helpful comments.

References

[BS] J. Bradfield, C.Stirling. *Local Model Checking for Finite State Spaces.* LFCS Report Series ECS-LFCS-90-115, June 1990

[CC1] P. Cousot, R. Cousot. *Abstract interpretation: A unified Lattice Model for static Analysis of Programs by Construction or Approximation of Fixpoints.* In Proceedings 4th POPL, Los Angeles, California, January, 1977

[CES] E. Clarke, E.A. Emerson, A.P. Sistla. *Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications: A Practical Approach.* In Proceedings 10th POPL'83, 1983

[CPS1] R. Cleaveland, J.G. Parrow, B. Steffen. *A Semantic-Based Verification Tool for Finite-State-Systems.* Protocol Specification, Testing and Verification, IX, Elsevier Science Publications B.V. (North Holland), 287-302, 1990

[CPS2] R. Cleaveland, J.G. Parrow, B. Steffen. *The Concurrency Workbench.* Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407, 1989

[Dha] D. Dhamdhere. *A Fast Algorithm for Code Movement Optimization.* SIGPLAN Notices, Vol. 23, 1988

[EL] E. Emerson, J. Lei, *Efficient model checking in fragments of the propositional mu-calculus.* In Proceedings LICS'86, 267-278, 1986

[Gie] R. Giegerich. *Automatic Generation of Machine Specific Code Generation.* In Proceedings 9th POPL, Albuquerque, New Mexico, January, 1982

[He] S.M. Hecht *Flow Analysis of Computer Programs.* Elsevier, North Holland, 1977

[Kil] G.A. Kildall. *A Unified Approach to Global Program Optimization.* In Proceedings 1st POPL, Boston, Massachusetts, 194-206, 1973

[Koz] D. Kozen. *Results on the Propositional mu-Calculus.* TCS 27, 333-354, 1983

[Mos] P.D. Mosses. *Action Semantics.* To appear 1991

[MR] E. Morel, C. Renvoise. *Global Optimization by Suppression of Partial Redundancies.* CACM 22, 96-103, 1979

[MW] P.D. Mosses, A.A. Watt. *The Use of Action Semantics.* In Formal Description of Programming Concepts - III, 1986

[Nie1] F. Nielson. *A Bibliography on Abstract Interpretations.* ACM SIGPLAN Notices 21, 31-38, 1986

[Nie2] F. Nielson. *A Denotational Framework for Data Flow Analysis.* Acta Informatica 18, 265-287, 1982

[Plo] G. Plotkin. *A Structural Approach to Operational Semantics.* University of Aarhus, DAIMI FN-19, 1981

[LPRS] Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. *The Ergo Support System: An Integrated Set of Tools for Prototyping Integrated Environments.* SIGPLAN Notices, Vol. 24, No. 2, 25-34, February 1989

[RWZ] B. K. Rosen, M. N. Wegman and F. K. Zadeck. *"Global Value Numbers and Redundant Computations".* 15th POPL, San Diego, California, 12 - 27, 1988

[SFRW] S. Sagiv, N. Francez, M. Rodeh, R. Wilhelm. *A Logic-Based Approach to Data Flow Analysis Problems.* To appear 1990

[SKR1] B. Steffen, J. Knoop, O. Rüthing. *The Value Flow Graph: A Program Representation for Optimal Program Transformations.* In Proceedings ESOP'90, LNCS 432, 1990

[SKR2] B. Steffen, J. Knoop, O. Rüthing. *Optimal Placement of Computations within Flow Graphs: A Practical Approach.* Submitted for POPL'91

[SP] M. Sharir, A. Pnueli. *Two Approaches to Interprocedural Data Flow Analysis. In: S.S. Muchnick, N.D. Jones. Program Flow Analysis: Theory and Applications.* Prentice-Hall, Englewood Cliffs , N.J., 1981

[Ste1] B. Steffen. *Characteristic Formulae.* In Proceedings ICALP'89, LNCS 372, 1989

[Sti1] C. Stirling. *Modal and Temporal Logics.* In *Handbook of Logics in Computer Science*, Vol. 1, Oxford University Press, to appear 1990.

[Tar] Tarski, A. "A Lattice-Theoretical Fixpoint Theorem and its Applications." *Pacific Journal of Mathematics*, v. 5, 1955.

[Ven] G.A. Venkatesh. *A framework for construction and evaluation of high-level specifications for program analysis techniques.* In Proceedings SIGPLAN'89, 1989

[Wil1] R. Wilhelm. *Global Flow Analysis and Optimization in the MUG2 Compiler Generating System. In: S.S. Muchnick, N.D. Jones. Program Flow Analysis: Theory and Applications.* Prentice-Hall, Englewood Cliffs, N.J., 1981

[Wil2] R. Wilhelm. *Codeoptimierung mittels attributierter Transformationsgrammatiken.* LNCS 26, 257-266, 1974