Generating Data Flow Analysis Algorithms from Modal Specifications

Bernhard Steffen
Lehrstuhl für Informatik II
RWTH – Aachen
D-5100 Aachen

Aachener Informatikberichte
Nr. 91 – 31

Abstract

The paper develops a framework that is based on 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 implementations of DFA-algorithms from modal specifications by partially evaluating a specific model checker with respect to 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. – The framework is illustrated by means of a real life example: the problem of determining optimal computation points within flow graphs.

Keywords: bit-vector algorithms, data flow analysis, data flow analysis generator, modal logic, model checking, mu-calculus, partial evaluation, transition systems.

1 Introduction

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

$$\text{DFA-algorithm for a property}: \ \text{programs} \ \rightarrow \ \text{program points enjoying the property}$$

Model checking is concerned with the automatic identification of those states of a finite system satisfying a specific modal (or temporal) formula that expresses, for example, properties like deadlock, divergence or liveness. Typically, model checkers are parameterized on the formula of interest and therefore have the following functionality:

$$\text{model checker}: \ \text{modal formulas} \times \text{model} \ \rightarrow \ \text{states satisfying the argument formula}$$

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 a parameter.

In this paper, we exploit this observation in order to develop an algorithm that automatically generates efficient implementations of DFA-algorithms from specifications written in a modal logic. In essence, this DFA-*generator* works by *partially evaluating* an appropriate model checker with respect to its modal formula parameter; the result is a standard iterative DFA-algorithm (cf. [Kil]), which directly runs on the machine the model checker is implemented on. For simplicity, we will refer to these implementations just as DFA-algorithms.

Our framework covers the standard bit-vector DFA algorithms[1] 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. Moreover it supports the specification of and reasoning about DFA-algorithms. Both can be done structurally by reasoning within the modal logic serving as specification language.

The framework 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 from a one line specification[2] consists of a sequence of a backward and a forward analyses. Thus our algorithm is as efficient as the standard uni-directional algorithms, because the decomposition of the usual bi-directional structure into a sequence of two uni-directional analyses guarantees the efficiency of the well-known bit-vector techniques. Moreover, the results of the generated algorithm are provably optimal. 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 two specification languages: a low level specification language with a general fixpoint operator, and a high level specification language, where the general fixpoint

[1] Typically, such algorithms determine things like life variables, (very) busy expressions or use-definition chains[He].

[2] Usually, this type of algorithm is given by means of a complex equation system.

operator is replaced by intuitively easy to understand derived operators. The section closes with logical characterizations of structural constraints of certain program models.

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. The development in this section is dramatically simplified by the assumption that the argument programs are transformed into a specific format (transparent placement models). This demonstrates the power of combining simple program transformations with general purpose analysis methods.

Section 5 provides a correct and complete model checker, which is linear in the size of the program model being investigated[3]. Our DFA-generator works by partially evaluating this model checker with respect to the modal formula used to specify the DFA-property of interest. This partial evaluation process is illustrated by generating 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, which essentially works on syntax trees and generates DFA-algorithms from specifications given as computation functions for attribute values [Wil1,Wil2]. Thus its 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 (eg. [Gie,Nie2,LPRS,SFRW,Ven]). In contrast, we specify DFA-algorithms just by means of the program property 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* $(\mathcal{S}, \mathcal{A}, \rightarrow, \mathcal{B}, \lambda)$ *where*

1. $\mathcal{S}$ *is a finite set of* nodes *or* program states.
2. $\mathcal{A}$ *is a set of* atoms. *The subsets of* $\mathcal{A}$ *play the role of what is usually called* actions. *We will denote this set of actions by Act.*
3. $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ *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* $\lambda : \mathcal{S} \rightarrow 2^{\mathcal{B}}$ *that labels states with subsets of* $\mathcal{B}$.

We will write $p \xrightarrow{A} q$ *instead of* $(p, A, q) \in \rightarrow$, *and given* $\alpha \subseteq Act$, *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*[4].

[3] This algorithm is a variant of the algorithm presented in [CES], modified to simplify partial evaluation.

[4] Note that the *Act*-predecessors and *Act*-successors are just the predecessors and successors in the usual sense.

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 set structure of the actions. This is necessary, because we want to deal with *abstract interpretations*, which treat concrete statements as sets of properties (cf. [CC]). However, one should abstract from the internal structure of actions, where it is unimportant, like e.g. in Section 3.1 and most of Section 3.2.

Definition 2.2 *A* DFA-model *is a program model with two distinct states* $\mathbf{s}$ *(the* start *state) and* $\mathbf{e}$ *(the* end *state) satisfying:*

1. $\mathbf{s}$ *and* $\mathbf{e}$ *do not possess any predecessor and successor, respectively.*
2. *Every program state is reachable from* $\mathbf{s}$.
3. $\mathbf{e}$ *is reachable from every program state.*

The additional requirements for DFA-models are standard in data flow analysis. In fact, they do not impose any restrictions there, because one can always reach this format by eliminating unreachable parts and adding nodes $\mathbf{s}$ and $\mathbf{e}$ appropriately.

Modelling Nondeterministic Flow Graphs

We will transform *nondeterministic flow graphs* into DFA-models. Nondeterministic flow graphs are directed graphs whose nodes represent statements (as usual, we will concentrate on assignments here) and whose 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 represent 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*.[5] In the discussion of our example we will deal with precondition 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 atomic propositions, and λ_c the labeling function, which associates the empty set of properties with each node of the model except for $\lambda_c(\mathbf{s}) = \{\texttt{start}\}$ and $\lambda_c(\mathbf{e}) = \{\texttt{end}\}$. Then the precondition model (or equivalently the postcondition model) of a flow graph forms a DFA-model $(\mathcal{S}, A_c, \rightarrow, \mathcal{B}, \lambda_c)$, where the state labeling just identifies the start and the end state, and the transition labeling the statement corresponding to a state transition.

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:

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

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

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* 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 slight extensions. However, in general, the appropriate abstract interpretation of the statements must be chosen problem dependently.

The following notion of computation DAG captures the essences of the computational behaviour of a program.[6]

Definition 2.3 *A* computation DAG *is a totally expanded DFA-model, i.e. every backward edge is transformed into a forward edge by copying its destiny.*

In the following, we will use computation DAGs whenever we formally deal with the computational behaviour of a program, because they define the appropriate level of abstraction. In particular, they allow proofs by induction on the depth of a node, which corresponds to the number of computation steps that have been executed.

3 The Specification Languages

We present two specification languages, a *low level* language, which is primary, and a derived *high 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 by denumerable sets *Var*, $\mathcal{B}$ and *Act* of propositional variables, atomic propositions and actions, respectively, where X ranges over *Var*, β over $\mathcal{B}$ and α over subsets of *Act*:

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

The semantics of *closed* formulas, where all occurrences of variables are bound by some fixpoint operator ν, is defined with respect to a program model $\mathcal{P}$ according to the following intuition: 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, it satisfies β if it is labeled by a set containing β and it satisfies $[\,\alpha\,]\,\Phi$ if every of its α-successors 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 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

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

[6]Note that the (similar) notion of computation tree would *not* be sufficient, because we are dealing with the forward as well as the backward flow of programs here.

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

Figure 1: The semantics of formulas.

and the substitution $\Phi[\Gamma/X]$ is defined in the standard way

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 actions here, rather than just for single actions. This is a convenient generalization, which simplifies the representation of certain properties enormously (cf. [BS]).

The formal semantic definition of the logic is given in Figure 1. It maps closed formulas to sets of program states — intuitively, the program states for which the formula is "true". Note that the semantics of $\nu X.\ \Phi$ is based on Tarski's fixpoint theorem [Tar]: its meaning is defined as the greatest fixpoint of a continuous function over the powerset of the set of states. The continuity of this function follows from the syntactic restriction mentioned above, which deals with the problems of negations, and the continuity of the semantic interpretations of the other propositional constructors.

In the following we will write $[\,.\,]$ or $\overline{[\,.\,]}$ instead of $[\,\mathit{Act}\,]$ or $\overline{[\,\mathit{Act}\,]}$. Moreover, as usual, we can define the following duals to the operators of our language and the implication operator $\Rightarrow$ by:

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

Our low level specification language consists of all *closed* and *guarded* formulas, where no variables occur free inside the scope of a fixpoint expression.[7] Closed means that all variables are bound by a fixpoint operator, and guarded that they all 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.

[7]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.

Figure 2: The Weak Until Operator

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, general fixpoint formulas are in general unintuitive and difficult to understand. Therefore, we will consider the intuitively easy to understand derived "Henceforth" and "Until" operators of the temporal logic CTL [CES][8]:

$$\begin{aligned}\mathbf{AG}\,\Phi &= \nu X.\,(\Phi \wedge [\,.\,]X)\\ \Phi\ \mathbf{U}\ \Psi &= \nu X.\,(\,\Psi \vee (\Phi \wedge [\,.\,]X)\,)\end{aligned}$$

and their past-time counterparts:

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

$\Phi\ \mathbf{U}\ \Psi$ holds of p if Φ holds for every state of every path that starts in p until a state is reached where Ψ is true, while $\Phi\ \overline{\mathbf{U}}\ \Psi$ holds of p, if Φ holds for every state of every path that ends in p after Ψ has been true the last time. This is illustrated in Figure 2, which sketches the essence of these definitions on a computation DAG. $\mathbf{AG}\,\Phi$ is simply an abbreviation for $\Phi\ \mathbf{U}\ f\!f$.

Our high level specification language, which will be denoted by $\mathcal{L}$, arises from the low level specification language by replacing the general fixpoint operator with the operators established above. This language is quite expressive and it allows concise specifications of the standard bit-vector algorithms. In addition to this, our specification language 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. For example, 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 universally satisfying the following three properties:

1. $(\texttt{start} \Rightarrow \overline{[\,.\,]}f\!f\,) \wedge (\texttt{end} \Rightarrow [\,.\,]f\!f\,)$
2. $\neg(\,\overline{\mathbf{AG}}\,\neg\texttt{start}\,)$
3. $\neg(\,\mathbf{AG}\,\neg\texttt{end}\,)$

[8] In [Ste1] more complex operators have been introduced. In this exposition, I will concentrate on a specific format of program models, which can easily be obtained by transformation (see Section 4.1), and simplifies the specification of the data flow analysis algorithms enormously.

Whereas part (1) is obvious, parts (2) and (3) express the corresponding properties of a DFA-model indirectly: part (3) says: "for every state of the model it is not the case that all the path never reach the end state", which is equivalent to: "from every state there exists a path reaching the end state". Part (2) is expressed analogously.

Also a property of precondition models, which is important for the proof of the Correctness Theorem 4.5 and the Optimality Theorem 4.7, can be stated within our specification language. This requires the introduction of the set complement operator "^{c}", which will also be used in the sequel.

Proposition 3.2 (Precondition Models)
A precondition model satisfies universally for every $\alpha \subseteq Act$: $\quad \langle\, \alpha \,\rangle tt \;\Rightarrow\; [\, \alpha^c \,] ff$.

Intuitively, this property means that all the transitions of a state (program point) in a precondition model are labelled identically. This property is valid in precondition models, because there, the labels of all the transitions leaving a specific state are derived from the same statement.

4 Example: Optimal Placement of Computations

In this section, we will develop a specification of a bi-directional DFA-algorithm that determines optimal computation points for a given term t within a DFA-model (flow graph). During this development we will fix t, which simplifies the notation, because we can drop t from the argument list of the predicates.

4.1 Transparent Placement Models

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: (for details see e.g. (cf. [RWZ,SKR1]). This problem can be solved by means of the following transformation: insert an artificial state into each original transition that ends at a state with more than one predecessor, whose (unique) ingoing transition maintains the label of the splitted original transition and whose (unique) outgoing transition is labeled with the empty statement "skip" (cf. [Dha,SKR1,SKR2][9]). In fact, this transformation preserves the central property of precondition models in the following sense: the result after this transformation is the same as if the transition splitting would have been done for the original flow graph by inserting "skip nodes", and subsequently constructing the precondition model for this transformed flow graph.

For DFA-models, the essence of this transformation can be characterized logically as follows:

Characterization of Placement Models

A placement model is characterized by universally satisfying the following three properties:

1. $(\Phi \Rightarrow \overline{[\,.\,]}[\,.\,]\Phi) \vee (\overline{\langle\,.\,\rangle}\Phi \Rightarrow \overline{[\,.\,]}\Phi)$
2. $(\Phi \Rightarrow [\,.\,]\overline{[\,.\,]}\Phi) \vee (\langle\,.\,\rangle\Phi \Rightarrow [\,.\,]\Phi)$ and
3. $\forall \alpha \subseteq Act.\ \overline{\langle\, \alpha \,\rangle} tt \;\Rightarrow \overline{[\, \alpha^c \,]} ff$

Intuitively the first property means that there are two classes of states in a placement model:

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

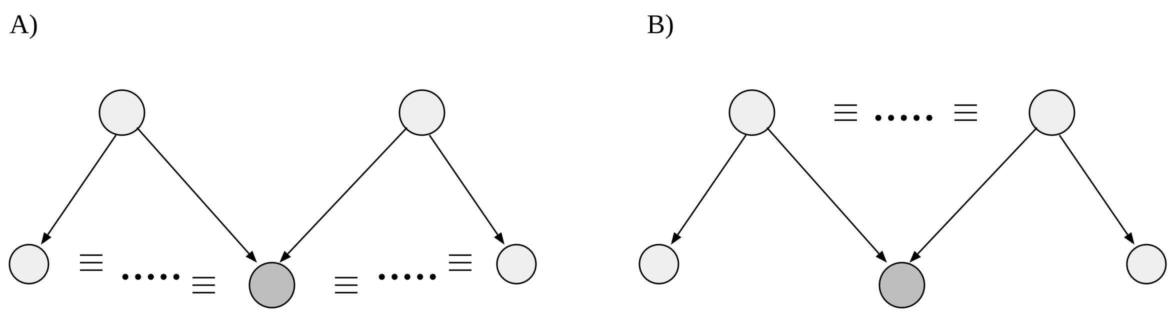

Figure 3: Placement Models

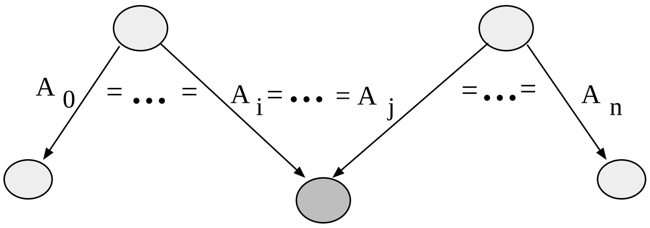

Figure 4: Transparent Models

- the ones that are "similar" to (have exactly the same properties as) all their brothers, as it is illustrated in Figure 3 (A), and
- the ones whose predecessors are all "similar", as it is illustrated in Figure 3 (B).

The second property is the "forward" analogon of the first property, and the third property guarantees that all the transitions that lead to the same state have identical labels.

In fact, it is possible to obtain an optimal placement algorithm, as soon as we restrict ourselves to precondition models with this property (cf. Optimality Theorem 4.7). From the specification point of view these models are particularly interesting, because they are *transparent* in the following sense: they allow to control the transition properties by means of state properties, which simplifies our specification framework enormously[10]. This control is due to the fact that the label of the last and the next transition are determined by the current state. Figure 4 illustrates the essence of the following corollary, which implicitly formalizes the notion of transparency.

Corollary 4.1 (Transparency)
A precondition placement model satisfies universally for every $\alpha \subseteq Act$*:*

1. $\langle\, \alpha \,\rangle tt \;\Rightarrow [\, \alpha^c \,] ff$
2. $\overline{\langle\, \alpha \,\rangle} tt \;\Rightarrow \overline{[\, \alpha^c \,]} ff$

The corollary is simply the conjunction of Proposition 3.2 and the third characterizing property for placement models.

[10] In [Ste1] much more complicated specifications needed to be introduced in order to deal with slightly more general models.

4.2 Correctness and Optimality of a Placement

In this section we are going to prepare the proof of our claim that the placement of computations specified in this paper is *optimal*, i.e. *better* than every other placement which is guaranteed to be *correct*. In order to establish this result within our logical framework, we need to formalize in the logic, what it means to be "correct" and "better". The following conventions, which are justified by Corollary 4.1, support a concise formalization:

Conventions:

- $p \models \langle\, \alpha \,\rangle tt$ is abbreviated by $p \models \alpha$
- $p \models \overline{\langle\, \alpha \,\rangle} tt$ is abbreviated by $p \models \overline{\alpha}$

Moreover, we define $M =_{df} \{\, A \,|\, \mathtt{mod} \in A \,\}$ and $U =_{df} \{\, A \,|\, \mathtt{use} \in A \,\}$, and we will allow a set of states to be regarded as an atomic proposition that is true exactly for its members. – The latter propositions do not appear in the specification of the algorithm. They are only necessary when reasoning about the properties of the specified algorithm.

4.2.1 Correctness

A correct placement of the computations of a term t is a set S of states, which guarantees that the following program transformation does not effect the program semantics:

- Initialize a distinct auxiliary variable (or a register) h at every state of S.
- Replace every original computation of t by h.

In particular, a correct placement of computations must guarantee that the auxiliary variable contains the required value at each of its usages, and that the introduction of the computations of t at S does not introduce any new values on any paths. Whereas the need for the first requirement, the *definedness*, is immediately clear, the second requirement, the *safety*, is necessary, because the introduction of new computations may lead to new run-time errors, e.g. "division by 0" or "overflow".

Definedness is logically expressed by requiring that each backward path from a state satisfying U will not meet the start state or a modifying transition before a member of S is reached[11]:

Definition 4.2 *A subset set $S \subseteq \mathcal{S}$ (here regarded as a set of computation points)* guarantees definedness *if every state of the program model satisfies*[12]*:*

$$\mathtt{definedness}\,(S) =_{df}\; U \;\Rightarrow\; (\neg(\overline{M} \vee \mathtt{start}))\, \overline{\mathbf{U}}\, S$$

More intuitively, this property could be read as follows: on each path to a usage of t there is a member of S (a computation point) after which the value of t is guaranteed to be preserved.

The definition of safety follows a similar pattern:

Definition 4.3 *A subset set $S \subseteq \mathcal{S}$ (again viewed as a set of computation points) is* safe *if the following predicate holds of all states:*

$$\mathtt{safety}\,(S) =_{df}\; S \Rightarrow\; (\,(\neg(\overline{M} \vee \mathtt{start}))\, \overline{\mathbf{U}}\, \overline{U}\;) \;\vee\; (\,(\neg(M \vee \mathtt{end}))\, \mathbf{U}\; U\;)$$

This formalization is based on the observation that for a given computation point, safety is equivalent to the requirement that the computations must either occur on each path ending at this point[13], or they must occur on each path starting in it and ending in the end state, meaning

[11] This pattern is typical for specifications in our framework.

[12] Remember the definition of $\overline{M}$, which simply says that the transitions entering the currently considered state are labeled by a member of M, i.e. they modify t.

[13] This means that the computation is *redundant* at this point.

that the computation is *necessary* at this point. This equivalence is due to the fact that there are now "mixed" situations, because every path reaching a certain program point can be continued by every path leaving this point.

Given two arbitrary subsets S and S' of $\mathcal{S}$, we consider S' as being better than S if on every path through the program model the number of occurrences of states of S is as least as large as the number of occurrences of states of S'. This reflects the intuition that a set of computation points is better than another one if it causes less computations during the program execution. The following logical formulation is more restrictive, because it additionally requires that the members of the better set of computation points must also occur earlier. This slightly stronger requirement will be satisfied by the placement specified in the next section.

Definition 4.4 *Let $S, S' \subseteq \mathcal{S}$ be arbitrary sets of states. Then S' is* better *than S if the following predicate holds universally:*

$$(S' \wedge S^c) \;\Rightarrow\; [\,.\,]\,((\neg(S' \vee \mathtt{end}))\;\mathbf{U}\;S)$$

Intuitively, the formula guarantees that all the S'-occurrences and the end state are pairwisely separated by S-occurrences.

4.3 The Specification of an Optimal Placement

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. These can be characterized in two steps.

As mentioned in the previous section the placement must be *safe*. This property is satisfied, if all the inserted computations are *necessary*, i.e. satisfy the second disjunct in the definition of safety:

Guaranteeing Safety $\mathtt{NEC} =_{df} (\neg(M \vee \mathtt{end}))\;\mathbf{U}\;U$

The definedness will be guaranteed by the requirement that computations should be placed as "early" as possible. Note that a computation can only be moved upwards, if it can be moved upwards along every ingoing transition. Thus earliestness means intuitively that such a movement is not possible without violating correctness. In particular, the start state is earliest, because there it is not possible to move the computation up at all.

Earliestness $\mathtt{EAR} =_{df} \mathtt{start} \;\vee\; \neg\,(\overline{[\,.\,]}((\neg(M \vee \mathtt{start}))\;\overline{\mathbf{U}}\;(\mathtt{NEC} \wedge \neg M)))$

Figure 5 illustrates the negation of earliestness: on each path to the currently considered state there exists an "earlier" state, which could safely be used for the initialization of the auxiliary variable. Technically this is expressed by these "earlier" states satisfying NEC, and the connecting paths being modification-free.

The combination of necessity and earliestness yields the desired specification:

The Computation Points $\mathtt{OCP} =_{df} \mathtt{EAR} \;\wedge\; \mathtt{NEC}$

As will be established in the next subsection, OCP is already the complete specification of the DFA-algorithm that determines the optimal computation points (for t). Thus, a subsequent program transformation, which

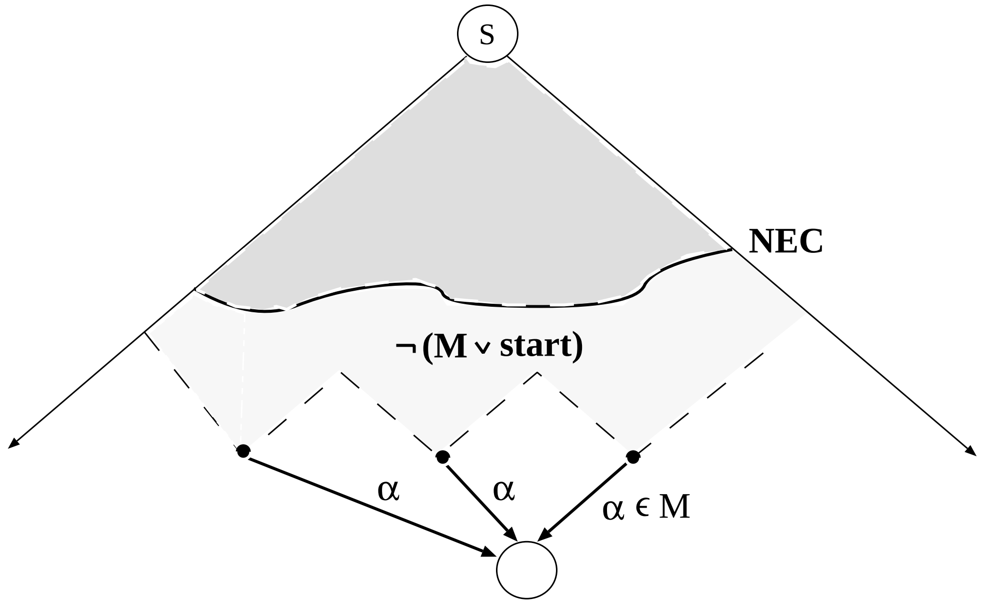

Figure 5: Earliestness

- initializes an auxiliary variable at each program point satisfying `OCP` and
- replaces all the original computations (of t) by this auxiliary variable

places the computations of t optimally within a precondition placement model.

During the rest of this section, let $S_I =_{df} \{\, s \mid s \models \texttt{OCP} \,\}$. Then the following theorem yields that `OCP` specifies a correct placement (set of program points or states):

Theorem 4.5 (Correctness)
In precondition placement models we have:

- $\texttt{safety}\,(S_I)$
- $\texttt{definedness}\,(S_I)$

Proof: Whereas, the proof of the first part is immediate from the definitions, the second part needs an inductive argument on the depth of the computation DAG. Because U implies `NEC`, it is enough to show:

$$(*) \quad \texttt{NEC} \quad \text{implies} \quad \neg(\overline{M} \vee \texttt{start}))\, \overline{\mathbf{U}}\, S_I$$

for every state of a precondition placement model. We proceed by induction on the depth in the corresponding computation DAG. $(*)$ is trivially true for the start state $\mathbf{s}$, where the earliestness predicate is always satisfied. Thus let $p \neq \mathbf{s}$ be a state of the computation DAG such that all states q with smaller depth in the computation DAG satisfy $(*)$. Then the following holds at state p:

$$\begin{array}{lll}
\texttt{NEC} & \text{implies} & \mathrm{S}_I \vee \neg\texttt{EAR} \\
 & \text{implies} & \mathrm{S}_I \vee ((\boxed{.}((\neg(M \vee \texttt{start}))\, \overline{\mathbf{U}}\, (\texttt{NEC} \wedge \neg M))) \wedge \neg\texttt{start}) \\
 & \text{implies} & \mathrm{S}_I \vee ((\boxed{.}((\neg(M \vee \texttt{start}))\, \overline{\mathbf{U}}\, (((\neg(\overline{M} \vee \texttt{start}))\, \overline{\mathbf{U}}\, S_I) \wedge \neg M))) \wedge \neg\texttt{start}) \\
 & \text{implies} & \mathrm{S}_I \vee ((\neg\overline{M} \wedge (\boxed{.}((\neg(\overline{M} \vee \texttt{start}))\, \overline{\mathbf{U}}\, ((\neg(\overline{M} \vee \texttt{start}))\, \overline{\mathbf{U}}\, S_I)))) \wedge \neg\texttt{start}) \\
 & \text{implies} & \mathrm{S}_I \vee (\neg(\overline{M} \vee \texttt{start}) \wedge (\boxed{.}((\neg(\overline{M} \vee \texttt{start}))\, \overline{\mathbf{U}}\, S_I))) \\
 & \text{implies} & \neg(\overline{M} \vee \texttt{start}))\, \overline{\mathbf{U}}\, S_I
\end{array}$$

where the third implication follows by induction, the fourth is a rearrangement of the formalization that guarantees the absence of intermediate modifications[14], the fifth implication is a consequence of the logical equivalence between $\Phi\,\overline{\mathbf{U}}\,(\,\Phi\,\overline{\mathbf{U}}\,\Psi\,)$ and $\Phi\,\overline{\mathbf{U}}\,\Psi$, and all the other implications are straightforward. □

Finally, let us turn to establishing the optimality of the placement. This requires some preparation:

Lemma 4.6 *The following holds universally in every precondition placement model:*

1. $\neg\mathtt{NEC} \;\Rightarrow\; (\neg\mathtt{NEC})\,\overline{\mathbf{U}}\,U$
2. $\neg\mathtt{EAR} \;\Rightarrow\; (\neg\mathtt{EAR})\,\mathbf{U}\,\overline{M}$ *and*
3. $(S_I \wedge \neg U) \;\Rightarrow\; [\,.\,]\,(\,(\neg(S_I \vee \mathtt{end})\;\mathbf{U}\;U\,)$

Proof: Obviously we have $\neg\mathtt{NEC} \;\Rightarrow\; \overline{[\,.\,]}(\neg\mathtt{NEC}\;\vee U)$ and therefore, by a simple inductive extension, the first part of the proof. The proof of the second part is similar.

For the proof of the third part, we first observe that the necessity requirement for the members of S_I reduces our goal to the statement that every $p \in S_I$ with $p \models \neg U$ satisfies:

$p \models [\,.\,]\,(\,(\neg S_I\;\mathbf{U}\;U\,)$

because it is not possible to reach the end state, before meeting a state satisfying U. This means that every path between p and another member q of S_I has at least one U-occurrence. Thus let π be such a path. Then we can assume without loss of generality that

(a) p is not the predecessor of q in π, because this would only be possible in case of $p \models M \wedge U$

(b) $\mathtt{NEC}$ holds along π, because part (1) of this lemma would otherwise require a U-occurrence in order to retain $\mathtt{NEC}$ for p, and that

(c) every intermediate state q' lies in ${S_I}^c$.

Now (c) and the definition of S_I yields that none of the intermediate states q' of π satisfy $\mathtt{EAR}$. On the other hand, $q \in S_I$ yields $q \models \mathtt{EAR}$. Thus applying part (2) of this lemma, we obtain that q must satisfy $\overline{M}$. Now (b) implies that the predecessor of q in π satisfies U, and therefore the existence of the required U-occurrence in π. □

After this preparation, we are able to prove our optimality result:

Theorem 4.7 (Optimality)
The placement of computations specified by OCP *is* better *than every other correct placement.*

Proof: Let $S \in \mathcal{S}$ be an arbitrary correct placement of computations. Then we must show

$(*)\;\; (S_I \wedge S^c) \;\Rightarrow\; [\,.\,]\,(\,(\neg(S_I \vee \mathtt{end}))\;\mathbf{U}\;S)$

As a first step we show that this goal is implied by:

$(**)\;\; S_I \;\Rightarrow\; (\neg U)\;\mathbf{U}\;S$

The correctness of this reduction is a consequence of the following sequence of implications which derives $(*)$ from $(**)$ by means of Lemma 4.6(3):

$$
\begin{array}{lll}
S_I & \text{implies} & (\,(\neg U)\;\mathbf{U}\;S\,) \;\wedge\; (\,U \vee [\,.\,]\,(\,(\neg(S_I \vee \mathtt{end}))\;\mathbf{U}\;U\,)\,) \\
 & \text{implies} & (\,U \wedge S\,) \;\vee\; (\,\neg U \wedge [\,.\,]\,(\,(\neg U)\;\mathbf{U}\;S\,) \;\wedge\; [\,.\,]\,(\,(\neg(S_I \vee \mathtt{end}))\;\mathbf{U}\;U\,)\,) \\
 & \text{implies} & S \;\vee\; (\,\neg U \;\wedge\; [\,.\,]\,(\,(\,(\neg U)\;\mathbf{U}\;S\,) \wedge (\,(\neg(S_I \vee \mathtt{end}))\;\mathbf{U}\;U\,)\,)\,) \\
 & \text{implies} & S \;\vee\; [\,.\,]\,(\,(\neg(S_I \vee \mathtt{end}))\;\mathbf{U}\;S\,)
\end{array}
$$

[14] Note that we are only considering transparent models here.

Thus it remains to verify $(**)$. Let therefore $\pi' = (p_1, \cdots, p_n)$ be a path in the model under consideration, satisfying:[15]

- $p_1 \models S_I$
- $\forall i \in \{1, \cdots n-1\}.\ p_i \models \neg U$ and
- $p_n \models U$.

Then it is obviously enough to show:

$(***) \quad \exists i \in \{1, \cdots n\}.\ p_i \models S$

First, we obtain because of $p_1 \models S_I$ that

$p_1 \models \mathtt{EAR} = \mathtt{start} \vee \neg(\overline{[\,.\,]}((\neg(M \vee \mathtt{start}))\ \overline{\mathbf{U}}\ (\mathtt{NEC} \wedge \neg M)))$

In the case of $p_1 \models \mathtt{start}$, the definedness property for S directly implies $(***)$. Thus we can assume that there exists a path $\pi'' = (q_1, \cdots, q_m)$ satisfying:

- $q_1 \models M \vee \mathtt{start}$ and
- $\forall i \in \{1, \cdots m\}.\ q_i \models \neg\mathtt{NEC}$.
- q_m is a predecessor of p_1.

Now let $\pi = \pi''; \pi' = (r_1, \cdots, r_{n+m})$ be the concatenation of π'' and π'. Obviously, π is a path from a state satisfying $M \vee \mathtt{start}$ to a state satisfying U, and therefore because of the definedness property for S possesses a component r_i with $r_i \models S$. Thus it remains to show that $i \geq m+1$. This, however, is a consequence of the fact that

$\forall i \in \{1, \cdots m\}.\ r_i = q_i \models \neg\mathtt{NEC}$

excludes the validity of the safety property for $i \in \{1, \cdots m\}$: whereas the first disjunct in the definition of safety is explicitly excluded, the second disjunct can also not be valid, because it would require

$\exists i \in \{1, \cdots m\}.\ r_i \models U$

which, in conjunction with the fact that U implies NEC, would lead to a contradiction. $\Box$

5 DFA-Generation

5.1 The Principle

The principle of the DFA-generator proposed here is *partial evaluation* of an appropriate model checker with respect to 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 presented in [CES], modified to support partial evaluation with respect 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:

[15] Note that n may well be 1.

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 program state of every potential program model 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 having one component corresponding to each subformula that appears as an operand of a modality. 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 (see Figure 7 for illustration).

   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 with respect to 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*[16] 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*. Note that the usual frame conditions are implicit in the atomic propositions the states are labeled with, e.g. `start` and `end` (see 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 Φ_s is determined using the fixpoint values computed in the third step. We will write $p \vdash \Phi_s$ if this yields *tt* for 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 covering CTL can be obtained straightforwardly. – 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 $\mathcal{P}$.

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

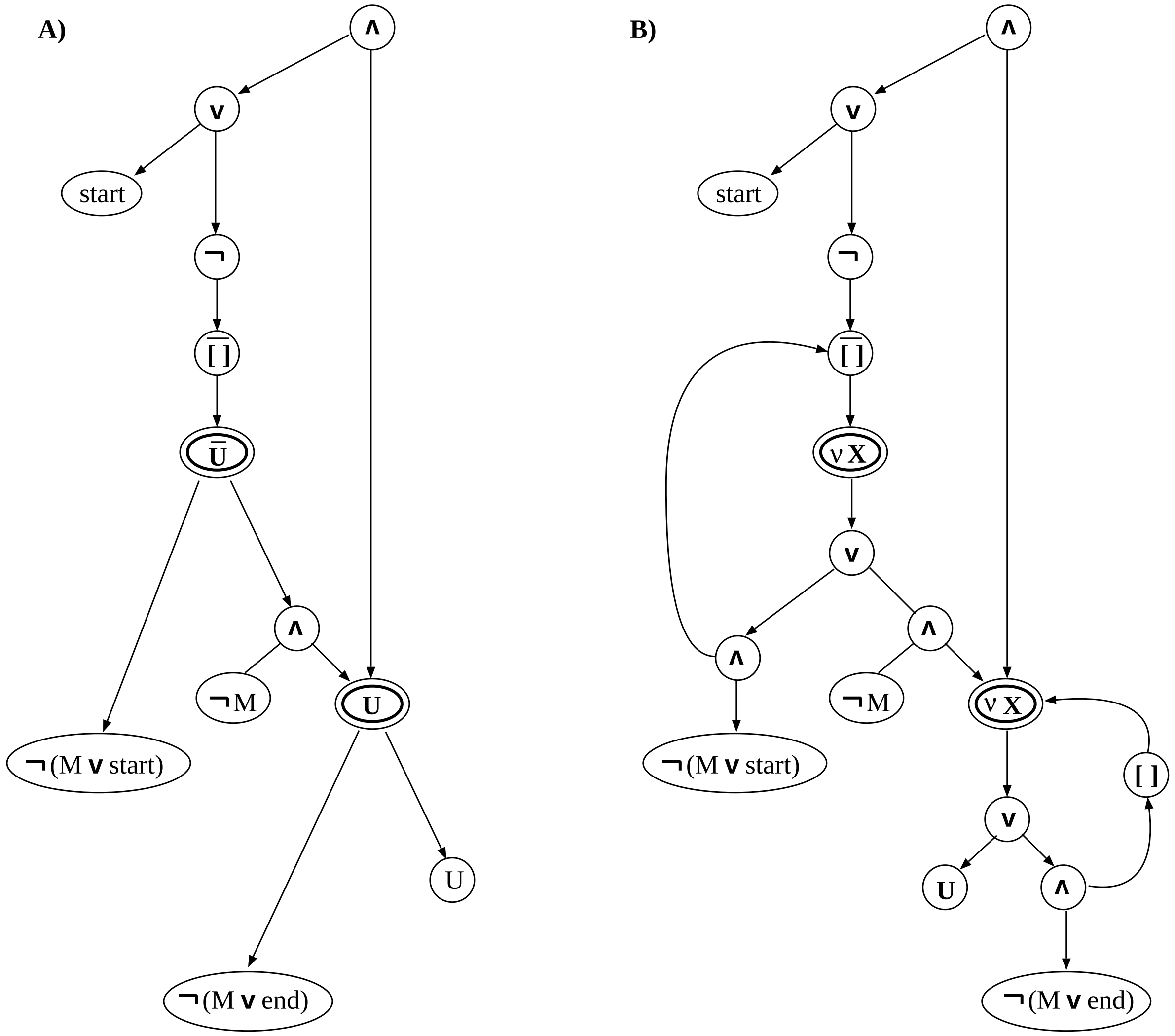

Figure 6: Construction of the Low Level Specification Representation

Partial evaluation of this model checker with respect to a high level specification consists of executing the first two (program independent) steps. This can be automated, yielding a generator that produces nested iterative data flow analysis algorithms from modal specifications, whose worst case run time complexity is proportional to the size of $\mathcal{P}$.

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 `OCP` $=$ `EAR` $\wedge$ `NEC` stepwisely by means of macro substitution into the graph representations shown in Figure 6. 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[17]. In our example, the predicates are represented by means of bit-vectors

[17]If required, it is also possible to straightforwardly generate traditional equational specifications of data flow

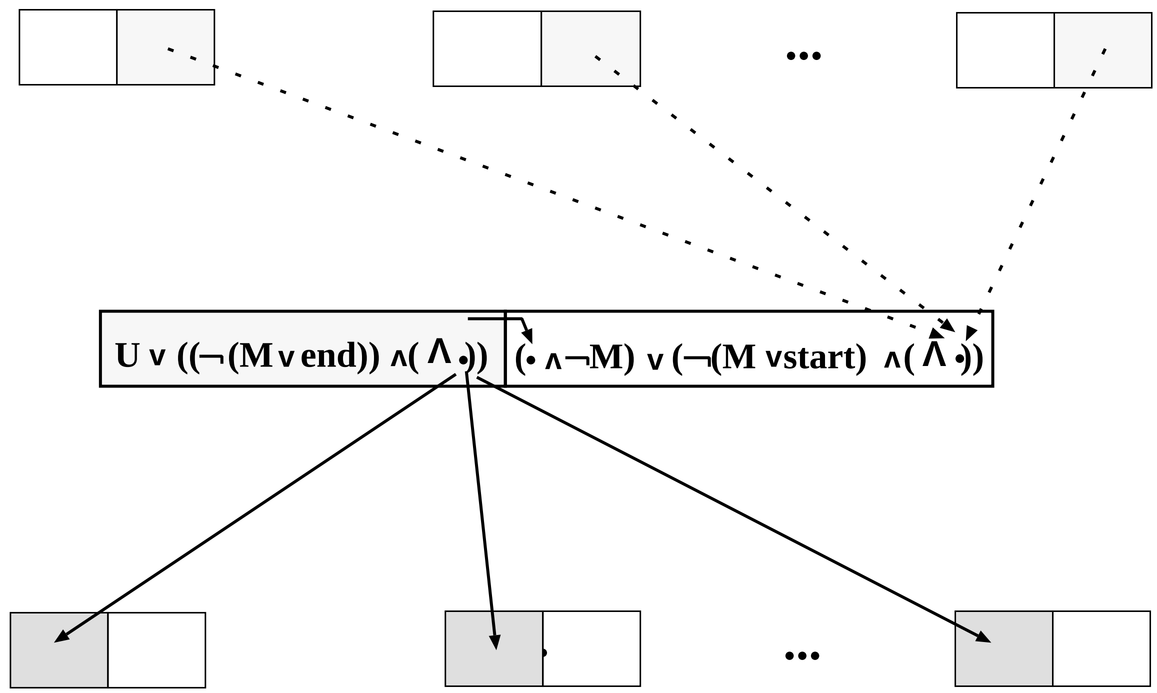

Figure 7: The Predicate Transformer

of length two, whose components hold the current approximations of the values of the subformulas that are marked with an extra circle in Figure 6. This choice reflects the fact that during the global iteration process, the only relevant information is given by the approximate values of the subformulas that appear as an operand of a modality. The values of all the other subformulas can be computed locally.

Figure 7 shows how the approximate bit-vector values are updated in step 3(b): here, the next approximation for the first bit of the bit-vector in the center of the picture is computed with respect to the conjunction of the approximate solutions for the first bits of the successor states. Analogously, the value of the second component of this bit-vector is updated with respect to its own first bit and the conjunction of the approximate values of the second components of its predecessor states.

The two (automatic) evaluation steps described above generate an iterative DFA-algorithm for the placement problem, which directly runs on the machine the model checker is implemented on.

It is worth noting that we only need to iterate according to two Boolean values here, one for the forward flow and one for the backward flow. This flow structure is much cleaner and simpler than in previous algorithms, where backward and forward flow are interwoven (cf. [Dha,MR]). Besides yielding clarity, this simplicity also improves the run time behaviour of the algorithm, because it guarantees the efficiency of the well-known bit vector techniques for reducible flow graphs.

---

analyses from our modal specifications.

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 works by partial evaluation of a specific model checker, and the compositionality 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. This example is particularly suitable for illustration, because it is complex enough to require a nontrivial specification and verification phase. Moreover, the automatically generated algorithm improved on previously know algorithm for this problem.

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 automatically 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.

Current research (cf. [CS1,CS2]) has shown that one can analyze formulas containing mutually recursive equation systems in linear time, as long as there is no alternated nesting [EL]. This suggests to use these more general formulas for low level specification, which, in particular, would allow to express the shared representation used in Section 5.2 directly on this level. We plan to investigate this change in detail, in particular with respect to its impact on the partial evaluation process.

Acknowledgement

I am very grateful to Olaf Burkart, Rance Cleaveland, Marion Dreimüller, Hardi Hungar, Jens Knoop, Jens Palsberg and Oliver Rüthing for their careful proof reading and their helpful comments, and to Karla Kerres for drawing the pictures.

References

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

[CC] 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

[CS1] R. Cleaveland, B. Steffen. *Computing Behavioural Relations, Logically.* In Proceedings ICALP'91, LNCS, 1991

[CS2] R. Cleaveland, B. Steffen. *A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus.* In Proceedings CAV'91

[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

[La] K. Larsen. *Proof Systems for Hennessy–Milner Logic with Recursion*, in Proceedings CAAP 1988.

[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".* 15^{th} 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.* In Proceedings TAPSOFT'91, LNCS, 1991

[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

[Ste1] B. Steffen. *Data Flow Analysis as Model Checking.* In Proceedings TACS'91, LNCS 526, 1991

[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

Aachener Informatik-Berichte

Please write to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, W-5100 Aachen

87-1	Jahresbericht 1986
87-2	D. de Frutos Escrig / K. Indermark: Equivalence relations of non-deterministic Ianov-schemes
87-3	M. Nagl: A Software Development Environment Based on Graph Technology
87-4	C. Lewerentz / M. Nagl / B. Westfechtel: On Integration Mechanisms within a Graph-based Software Development Environment
87-5	R. Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
87-6	W. Damm / G. Döhmen: Specifying Distributed Computer Architectures in AADL*
87-7	G. Engels / C. Lewerentz / W. Schäfer: Graph Grammar Engineering: A Software Specification Method
87-8	M. Nagl: Set Theoretic Approaches to Graph Grammars
87-9	C. Lewerentz / A. Schürr: Experiences with a Database System for Software Documents
87-10	H. Klaeren / K. Indermark: A New Implementation Technique for Recursive Function Definitions
87-11	R. Loogen: Design of a parallel programmable graph reduction machine with distributed memory
87-12	J. Börstler / U. Möncke / R. Wilhelm: Table Compression for Tree Automata
87-13	C. Lewerentz: Extended Programming in the Large in a Software Development Environment
87-14	G. Engels / Th. Janning / W. Schäfer: A Highly Integrated Tool Set for Program Development Support
87-15	R. Loogen / U. Goltz: A non-interleaving semantic model for nondeterministic concurrent processes
87-16	Th. Hafer: On the Boolean Closure of Büchi Tree Automaton Definable Sets of w-Trees.
87-17	U. Heuter: First-Order Properties of Trees, Star-Free Expressions and Aperiodicity
88-1	G. Esser / J. Rückert / F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik
88-2	P. Martini / O. Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
88-3	Th. Welzel: Simulation of a Multiple Token Ring Backbone
88-4	P. Martini: Performance Comparison for HSLAN Media Access Protocols
88-5	P. Martini: Performance Analysis of Multiple Token Rings
88-6	A. Mann / J. Rückert / O. Spaniol: Datenfunknetze
88-7	A. Mann / J. Rückert: Packet Radio Networks for Data Exchange

88-8 A. Mann / J. Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks

88-9 W. Kremer / F. Reichert / J. Rückert / A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs

88-10 K. Jakobs: Towards User-Friendly Networking

88-11 K. Jakobs: The Directory - Evolution of a Standard

88-12 K. Jakobs: Directory Services in Distributed Systems - A Survey

88-13 M. Schümmer: RS-511, a Protocol for the Plant Floor

88-14 U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing

88-15 P. Martini / O. Spaniol / Th. Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation

88-16 Jahresbericht 1987

88-17 W. Thomas: Automata on Infinite Objects

88-18 M. Sonnenschein: On Petri Nets and Data Flow Graphs

88-19 H. Vogler: Functional Description of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers

88-20 Th. Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen

88-21 Th. Janning / C. Lewerentz: Integrated Project Team Management in a Software Development Environment

88-22 J. Engelfriet / H. Vogler: Modular Tree Transducers

88-23 W. Thomas: Automata and Quantifier Hierarchies

88-24 U. Heuter: Generalized Definite Tree Languages

89-1 Jahresbericht 1988

89-2 G. Esser / J. Rückert / F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik

89-3 H. Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions

89-4 A. Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language

89-5 J. Börstler: Wiederverwendbarkeit und Softwareentwicklung - Probleme, Lösungsansätze und Bibliographie

89-6 K. Jakobs: OSI - An Appropriate Basis for Group Communication?

89-7 K. Jakobs: ISO's Directory Proposal Evolution, Current Status and Future Problems

89-8 B. Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo / Redo and Revision Control

89-9 P. Martini: High Speed Local Area Networks - A Tutorial

89-00 Jozef Filser: Bolidische Gedangen (Gabidel 5): "Ieber die Kombjuderkunzt"

89-10 P. Davids / Th. Welzel: Performance Analysis of DQDB Based on Simulation

89-11 M. Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science, Rolduc (NL) near Aachen (D), June 14-16, 1989.

89-12 P. Martini: The DQDB Protocol - Is it Playing the Game?

89-13 M. Schümmer: CNC / DNC Communication with MAP

89-14 M. Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements

89-15 M. Schümmer / Th. Welzel / P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

89-16 G. Vossen / K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

89-17 J. Derissen / P. Hruschka / M.v.d.Beeck / Th. Janning / M. Nagl: Integrating Structured Analysis and Information Modelling

89-18 A. Maaßen: Programming with Higher Order Functions

89-19 M. Rodriguez-Artalejo / H. Vogler: Note on A Narrowing Machine for Syntax Directed BABEL

89-20 H. Kuchen / R. Loogen / J. J. Moreno-Navarro / M. Rodriguez-Artalejo Graph-based Implementation of a Functional Logic Language

89-i4 O. Spaniol: Natürliche Intelligenz oder: Der Parallel-Igel

89-21 G. Vossen / S. Wang: Correct and Efficient Deadlock Detection and Resolution in Distributed Database Systems

89-22 G. Vossen / R. Schwarz: Dynamic Scheduling of Transactions in Design Databases based on Version Consistency

90-1 Jahresbericht 1989

90-2 V. Jansen / A. Potthoff / W. Thomas / U. Wermuth: A Short Guide to the AMORE System (computing Automata, MOnoids, and Regular Expressions)

90-3 J. Skurczynski: On Three Hierarchies of Weak SkS Formulas

90-4 R. Loogen: Stack-based Implementation of Narrowing

90-5 H. Kuchen / A. Wagener: Comparison of Dynamic Load Balancing Strategies

90-6 K. Jakobs / F. Reichert: Directory Services for Mobile Communication

90-7 K. Jakobs: What's Beyond the Interface-OSI Networks to Support CooperativeWork

90-8 K. Jakobs: Directory Names and Schema - An Evaluation

90-9 U. Quernheim / D. Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

90-10 U. Quernheim / St. Jacobs / D. Kreuer: Adaptive ARQ-Fehlersicherungsverfahren für die Point-to-Multipoint-Satellitenkommunikation

90-11 J.J.Moreno-Navarro / H. Kuchen / R. Loogen / M. Rodriguez-Artalejo: Lazy Narrowing in a Graph Machine

90-12 K. Jakobs / J. Kaltwasser / F. Reichert / O. Spaniol: Der Computer fährt mit

90-13 R. Mathar / A. Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

90-14 A. Maaßen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung

90-15 M. Nagl / A. Schürr: A Specification Environment for Graph Grammars

90-16 A. Schürr: PROGRESS: a VHL-Language Based on Graph Grammars

90-17 M. Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

90-18 E. Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

90-19 F. Strauß: Erklärungsfähigkeit eines ebenenorientierten Expertensystemmodells

90-20 Y. Ortega-Mallen / D. de Frutos-Escrig: A Complete Proof System for Timed Observations

90-21 M. Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

90-22 H. Faßbender / H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

90-23 Th. Janning / M. Lefering: A Transformation from Requirements Engineering into Design - The Method and the Tool

90-24 J. Noack: Natürlichsprachliche Interaktion mit Datenbanken

90-25 R. Mathar: Concurrence Probabilities for a Locally Slotted Packet Radio Network by Combinatorial Methods

90-26 F. Herrmann / Heiko Vogler: Primitive Recursive Programs for Defining Tree Functions

90-27 Z. Fülöp / F. Herrmann / S. Vagvölgyi / H. Vogler: Tree Transducers with External Functions

90-28 P. Bachmann / W. Oberschelp / B. Thalheim / G. Vossen: The Design of RAD: Towards an Interactive Toolbox for Database Design

91-1 Jahresbericht 1990

91-2 F. Beuting / F. Piron / K. Rausch: Ein neuronales Modell zur Feststellung von für die visuelle Aufmerksamkeit relevanten Figureneigenschaften

91-3 B. Steffen / A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

91-4 M. Portz: A new class of cryptosystems based on interconnection networks

91-5 H. Kuchen / G. Geiler: Distributed Applicative Arrays

91-6 L. Staiger: Kolmogorov Complexity and Hausdorff Dimension

91-7 L. Staiger: Syntactic congruences for ω-languages

91-8 K. Jakobs / K. D. Kleefeld: Using Public Communication Services for Archaeological Applications

91-9 E. Kuikka: A Proposal for a Syntax-Directed Text Processing System

91-10 K. Gladitz / H. Faßbender / H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming

91-11 R. Loogen / S. Winkler: Dynamic Detection of Determinism in Functional Logic Languages

91-12 K. Indermark / M. Rodríguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming

91-13 R. Hager / W. Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline.

91-14 A. Fasbender / W. Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes.

91-15 J. Börstler / A. Zündorf: Revisiting extensions to Modula-2 to improve reusability.

91-16 J. Börstler / T. Janning: Bridging the gap between Requirements Analysis and Design

91-17 A. Zündorf / A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems

91-18 M. Jarke / J. Mylopoulos / J. W. Schmidt / Y. Vassiliou: DAIDA: An Environment for Evolving Information Systems

91-19 M. Jeusfeld / M. Jarke: From Relational to Object-Oriented Integrity Simplification

91-20 G. Hogen / A. Kindler / R. Loogen: Automatic Parallelization of Lazy Functional Programs

91-21 O. Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint

91-22 H. Kuchen / F. Lücking / H. Stoltze: The Topology Description Language TDL

91-23 S. Graf / B. Steffen: Compositional Minimization of Finite State Systems

91-24 R. Cleaveland / J. Parrow / B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems

91-25 R. Mathar/J. Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks

91-26 M. Jeusfeld/ M. Staudt: Query Optimization in Deduktive Object Bases

91-27 J. Knoop/B. Steffen: The Interprocedural Coincidence Theorem

91-28 J. Knoop/B. Steffen: Unifying Strength Reduction and Semantic Code Motion

91-29 R. Cleaveland/M. Dreim"uller/B. Steffen: Faster Model Checking in the Model Mu-Calculus

91-30 T. Margaria: First-Order Theories for the Verification of Complex FSMs

91-31 B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications