Efficient FixPoint Computation

B. Vergauwen, J. Wauman, J. Lewi

Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract. Most of the algorithms presented for computing fixpoints have been developed for very specific application areas, e.g. dataflow analysis, abstract interpretation, denotational semantics, system verification, to mention but a few. Surprisingly little attention has been devoted to the construction of general purpose, application independent fixpoint algorithms (one notable exception being [17]). The goal of this paper is to put known ideas and algorithms into a more general and abstract setting. More precisely we discuss a variety of efficient general purpose algorithms for computing (part of) the least solution of a monotone system of equations over a complete partial order structure. The advantage of having general purpose fixpoint algorithms is twofold. Firstly, once proven correct, they can be instantiated to a wide variety of application domains. Secondly, they separate the essentials of fixpoint computation from concrete application domain aspects. We consider algorithms based on (chaotic) fixpoint approximation, as well as algorithms based on fixpoint induction techniques. The algorithms are constructed in a stepwise fashion: First a basic schema, capturing the essence of the algorithm, is discussed, which is then subsequently refined using a number of optimisation steps. As a sample application, we sketch how an algorithm for computing the prebisimulation preorder is obtained, matching the complexity of the so far best known 'ad hoc' constructed algorithm for this preorder.

1 Monotone Equation Systems

Fix a set $\mathcal{V}$ of values and an order relation $\sqsubseteq$ on $\mathcal{V}$ such that $\langle \mathcal{V}, \sqsubseteq \rangle$ forms a complete partial order (cpo) structure of finite height with bottom element $\bot$. A *monotone equation system* E over the cpo $\langle \mathcal{V}, \sqsubseteq \rangle$ is of the form

$$\begin{cases} x_1 = f_1 \\ \quad \vdots \\ x_n = f_n \end{cases}$$

where each left hand side x_i is a variable (an unknown), and each right hand side f_i is a function of the form $f_i : \Theta \to \mathcal{V}$, where $\Theta = [X \to \mathcal{V}]$ with $X = \{x_1, \ldots, x_n\}$. Left hand side variables are assumed to be distinct. Furthermore, right hand side functions f_i are required to be *monotone*. Recall that a function $f : \Theta \to \mathcal{V}$ is monotone iff $f(\theta) \sqsubseteq f(\theta')$ whenever $\theta \sqsubseteq \theta'$, where the order relation on Θ is just the pointwise extension of $\sqsubseteq$, i.e., $\theta \sqsubseteq \theta'$ iff $\theta(x) \sqsubseteq \theta'(x)$ for every $x \in X$. An element $\theta \in \Theta$ is a *solution* of E iff $\theta(x) = f_x(\theta)$ for every $x \in X$, where f_x denotes the right hand side function of the equation having variable x as its left hand side. As right hand sides are monotone, it immediately follows from basic fixpoint theory that E has a *unique least* solution w.r.t. $\sqsubseteq$, noted $[\![E]\!]$. It is this least solution $[\![E]\!]$ that we are interested in computing.

2 Global FixPoint Approximation (Basic Schema)

The least solution $[\![E]\!]$ of E can be computed using chaotic [6] fixpoint approximation:

$val := \lambda x \in X.\bot$
do not $(\forall x \in X : val(x) = f_x(val))$ $\longrightarrow$
  $x := choose(X)$
  $val(x) := \mathbf{EvalRhs}(x, val)$
od
-- $val = [\![E]\!]$

Data-structure $val : X \to \mathcal{V}$ encodes the current value of variables. $[\![E]\!]$ is approximated from below (i.e. $val \sqsubseteq [\![E]\!]$) by repeatedly selecting a variable, evaluating the associated right hand side function, and updating the value of the selected variable accordingly. To abstract away the details of right hand side evaluation, an evaluation function **EvalRhs** is assumed such that $\mathbf{EvalRhs}(x, \theta)$ returns $f_x(\theta)$ (for $x \in X$ and $\theta \in \Theta$). The updating process terminates when all variables are 'stable'.

The above schema captures the essentials of fixpoint approximation. It is easily seen to be partially correct. Assuming that the *choose*-operation is fair (i.e. variables that are unstable are eventually selected for evaluation), termination is also guaranteed. To obtain an efficient practical algorithm, the *choose*-operation should be implemented in a way that minimizes the number of iteration (evaluation) steps needed for reaching the least solution $[\![E]\!]$. The key idea of our variable selection strategy is the following: Assume that a variable x has been selected for evaluation. If it now turns out that $f_x(val)$ evaluates to $val(x)$, then x was certainly a 'bad' choice: Time has been wasted computing $f_x(val)$ without there being any progress made. Hence in order to minimize the number of evaluations needed for reaching $[\![E]\!]$, care should be taken in order to avoid selecting variables that are currently stable. How do we know, without computing $f_x(val)$, whether or not x is stable? To find out, an additional workset $ws \subseteq X$ is used for keeping track of variables that are potentially unstable. In other words, variables outside ws are known to be stable. Formally the meaning of ws is captured by the following assertion

$$(I_{ws}) \qquad \forall x \in \overline{ws} : \; f_x(val) = val(x)$$

which will be kept invariant ($\overline{ws}$ stands for $X \setminus ws$). Assuming that I_{ws} is kept invariant, then obviously only variables belonging to ws should be selected for re-evaluation. Furthermore, as a byproduct, a simple stability detection mechanism is obtained by checking emptyness of ws: If $ws = \emptyset$ then it immediately follows from I_{ws} that val is stable. The whole point now is, of course, to keep I_{ws} invariant, while at the same time keeping the size $|ws|$ as small as possible! The challenging point here is to restore I_{ws} when components of val change value as a result of re-evaluating right hand sides. More exactly, in order to restore I_{ws} upon modifying $val(x)$, a code fragment $Restore(x, ws)$ has to be constructed satisfying the following Hoare-like specification:

$$\{ \; I_{ws} \; \wedge \; x \in ws \; \wedge \; val(x) \neq f_x(val) \; \}$$
$$ws := ws \setminus \{x\} \quad val(x) := f_x(val) \quad \text{Restore(x,ws)}$$
$$\{ \; I_{ws} \; \}$$

(It is of course understood that $Restore(x, ws)$ may only modify variable ws). Assuming a correct implementation for $Restore(x, ws)$, we obtain:

Global fixpoint approximation : basic schema, parameterised by $Restore(x, ws)$

$val := \lambda x \in X.\bot \quad ws := X$
while $ws \neq \emptyset \longrightarrow$
  $x := choose(ws) \quad ws := ws \setminus \{x\}$
  $v_{old} := val(x) \quad val(x) := \mathbf{EvalRhs}(x, val)$
  if $v_{old} \neq val(x) \longrightarrow Restore(x, ws)$ **fi**
od
-- $val = [\![E]\!]$

Plugging in any correct implementation for $Restore(x, ws)$ yields a correct schema for computing $[\![E]\!]$. Furthermore the number of evaluations is in the worst case $\mathcal{O}(|X|^2.\mathtt{H})$, where H is the height of the cpo $\langle \mathcal{V}, \sqsubseteq \rangle$. It now remains to implement $Restore(x, ws)$.

3 Implementing *Restore*(x, ws)

A (naive) straightforward implementation for $Restore(x, ws)$ is to simply re-initialize ws. I.e. take $Restore(x, ws) \equiv ws := X$. Hence whenever a variable changes value, *all* variables are scheduled for re-evaluation, thereby trivially restoring I_{ws}. Re-initialising ws is a simple but rather crude way to restore I_{ws}. Below we discuss two more sophisticated implementations. The key idea is to keep track of *dependencies* between variables and right hand sides. Whenever a variable then changes value, this additional dependency information is consulted to find out which variables are affected and hence might become unstable as a result of the change. Those variables are then scheduled for re-evaluation by including them in ws. Both schema's differ in the *granularity* of the dependency information that is carried around.

3.1 Chasing Static (Syntactic) Dependencies

The first schema for $Restore(x, ws)$ uses syntactic dependency information. Static dependencies are well-known, and their use has been discussed by several researchers in various application domains, e.g. by [12] in the context of dataflow analysis, and more recently, by [7] in the context of abstract interpretation, and by [1,5,22] in the context of model-checking. They are also at the heart of O'Keefe's general algorithm [11].
The key idea underpinning static dependencies is a follows: For a function $f : \Theta \to \mathcal{V}$ define $\mathbf{Arg}(f)$ as the set of argument variables of f, i.e. only variables from $\mathbf{Arg}(f)$ are needed in order to evaluate f. Formally $\mathbf{Arg}(f)$ is defined as the least set $A \subseteq X$ such that the following holds[1]:

$$\forall \theta, \theta' \in \Theta : \quad \theta|_A = \theta'|_A \;\Rightarrow\; f(\theta) = f(\theta')$$

Note that in most practical applications $\mathbf{Arg}(f_x)$, or at least a small superset of it, can easily be computed by scanning the defining expression of f_x and recording all variables occurring in it. Clearly upon modifying $val(x)$, only variables y that depend upon x (i.e. $x \in \mathbf{Arg}(f_y)$) are affected and hence may become unstable. Hence:

$$Restore(x, ws) \;\equiv\; ws := ws \cup \{y \in X \mid x \in \mathbf{Arg}(f_y)\}$$

[1] $\theta|_A$ denotes the restriction of θ to A. I.e., $\theta|_A$ is the mapping $m : A \to \mathcal{V}$ such that $m(x) = \theta(x)$ for every $x \in A$.

To efficiently execute the above updating-statement for ws, an additional data-structure $Dep : X \to 2^X$ is used, such that $Dep(x) = \{y \in X | x \in \mathtt{Arg}(f_y)\}$. Note that, as Dep does not depend upon val, it can be computed prior to fixpoint iteration. Putting everything together yields:

Global fixpoint approximation guided by syntactic dependencies

---

$val := \lambda x \in X.\bot \quad ws := X$
$Dep := \lambda x \in X.\{y \in X \mid x \in \mathtt{Arg}(f_y)\}$
while $ws \neq \emptyset \longrightarrow$
  $x := choose(ws) \quad ws := ws \setminus \{x\}$
  $v_{old} := val(x) \quad val(x) := \mathtt{EvalRhs}(x, val)$
  if $v_{old} \neq val(x) \longrightarrow ws := ws \cup Dep(x)$ **fi**
od
-- $val = [\![E]\!]$

As a variable x is only re-evaluated when one of the variables from $\mathtt{Arg}(f_x)$ changes value, it is clear that x is evaluated at most $1 + \mathtt{H}.|\mathtt{Arg}(f_x)|$ times, where H is the height of the cpo $\langle \mathcal{V}, \sqsubseteq \rangle$. Hence the number of evaluations for x only depends on its right hand side function, and is independent of the rest of the equation system. Summing up over all variables, the total number of evaluations is, in the worst case, $\mathcal{O}(\mathtt{H}.|E|)$, where $|E| = |X| + \sum_{x \in X} |\mathtt{Arg}(f_x)|$.

3.2 Chasing Dynamic (Semantic) Dependencies

In the syntactic approach a right hand side function f is re-evaluated anew whenever one of its argument variables changes value. The set $\mathtt{Arg}(f)$, however, is *static* in the sense that it does not take into account the specific context val in which f is to be evaluated. By properly taking this context into account, some of these reevaluations may be further avoided. Let's illustrate this by means of an example: Consider the boolean function $f \equiv x_1 \wedge x_2 \wedge x_3$. Clearly, $\mathtt{Arg}(f) = \{x_1, x_2, x_3\}$. Now assume that f is evaluated in a context where $val(x_1) = \mathtt{false}$. Hence $f(val) = \mathtt{false}$. Furthermore, as long as $val(x_1)$ stays `false`, so does $f(val)$. Hence there is no need to re-evaluate f when x_2 or x_3 changes value. Only when the value of x_1 is modified should f be reevaluated. In order to keep track of semantic dependency information, an additional data-structure $arg : X \to 2^X$ is used. Informally $arg(x)$ records the set of variables that x *currently* depends on (for $x \in \overline{ws}$). Formally the meaning of arg is given by the following assertion

$$(I_{dep}) \qquad \forall x \in \overline{ws} : \ \mathtt{Suff}(f_x, val, arg(x))$$

where predicate $\mathtt{Suff}(f, \theta, A)$ holds iff the values of variables from A suffice to evaluate f in context θ. I.e.:

$$\mathtt{Suff}(f, \theta, A) \overset{def}{\iff} \forall \theta' \in \Theta : \quad \theta|_A = \theta'|_A \ \Rightarrow \ f(\theta) = f(\theta')$$

To compute semantic dependency information (and hence to keep assertion I_{dep} invariant), we slightly extend function $\mathtt{EvalRhs}(x, \theta)$ such that the additional set A of variables that were needed (used) in order to evaluate $f_x(val)$ is also returned. The refined specification for EvalRhs now reads as follows:

func EvalRhs(x, θ) **return** (v, A)
-- postcondition: $v = f_x(\theta)$, $A \subseteq \mathtt{Arg}(f_x)$, and $\mathtt{Suff}(f_x, \theta, A)$

Assuming invariance of I_{dep}, $Restore(x, ws)$ can be implemented as follows:

$$Restore(x, ws) \equiv ws := ws \cup \{y \in \overline{ws} \mid x \in arg(y)\}$$

To efficiently execute the above updating-statement for ws, an additional data-structure $dep : X \to 2^X$ is used, such that $dep(x) = \{y \in \overline{ws} \mid x \in arg(y)\}$. Note that, unlike Dep, variables arg and dep are dynamic in that they depend upon the current context val. Putting everything together yields:

Global fixpoint approximation guided by semantic dependencies

---

$val := \lambda x \in X.\bot \quad ws := X$
$arg := \lambda x \in X.\emptyset \quad dep := \lambda x \in X.\emptyset$
while $ws \neq \emptyset \longrightarrow$
  $x := choose(ws) \quad ws := ws \setminus \{x\}$
  $v_{old} := val(x) \quad (val(x), arg(x)) := \mathtt{EvalRhs}(x, val)$
  for $y \in arg(x)$ **do** $dep(y) := dep(y) \cup \{x\}$ **od**
  if $v_{old} \neq val(x) \longrightarrow$
    for $y \in dep(x)$ **do**
      for $z \in arg(y)$ **do** $dep(z) := dep(z) \setminus \{y\}$ **od**
      $ws := ws \cup \{y\} \quad arg(y) := \emptyset$
    od
  fi
od
-- $val = [\![E]\!]$

As $arg(x) \subseteq \mathtt{Arg}(f_x)$, the number of evaluations using semantic dependencies is always less (or equal) than when syntactic dependencies are used. (There is, however, also slightly more overhead in the semantic case).
The use of semantic dependencies for guiding fixpoint iteration is rather novel. It has been discussed in a general setting by the authors in [26], and, independently, by [10,16] in the context of abstract interpretation for Prolog. Semantic dependencies gain interest as right hand sides become 'sparse'. Sparseness is a rather qualitative notion. A function f is said to be sparse if, for most environments θ, only a 'small' fragment of θ is needed in order to evaluate $f(\theta)$, although the actual fragment needed might greatly depend on θ. Boolean equation systems e.g. usually exhibit a high degree of sparseness ($true \vee \_ = true$, and $false \wedge \_ = false$).

4 Local FixPoint Approximation

The algorithms discussed so far are *global*, in that they compute the complete least solution $[\![E]\!]$. In practice, however, one is often only interested in the value of one particular variable, say $\hat{x}$. This is e.g. the case in model-checking [1,2,5,8,15,20,23,25] where one is only interested in whether the *initial* system state satisfies a given property. Of course, one can always first compute $[\![E]\!]$, and then pick the component $[\![E]\!](\hat{x})$ of interest. It seems, however, overwhelming to compute the complete least solution $[\![E]\!]$ just in order

to decide the value $[\![E]\!](\hat{x})$ of interest. This observation is central to the development of *local* algorithms. As opposed to global algorithms, local algorithms aim at computing the desired component $[\![E]\!](\hat{x})$ by investigating only a 'necessary' fragment of the equation system E. In this section we discuss how to derive local algorithms from the global algorithms discussed in sections 2 and 3.
The basic local fixpoint approximation schema for computing $[\![E]\!](\hat{x})$ is depicted below. The essential (only) difference with the basic global schema of section 2 is the stability detection mechanism: Instead of a priori stabilizing *all* variables as is done in the global schema, the local schema tries to construct only a *partial* fixpoint. I.e., it tries to construct a *subset* $\mathcal{S}$ of variables (called the *search space*) that is stable and complete (closed, self-contained) in the sense that values of variables from $\mathcal{S}$ do not depend upon values of variables outside $\mathcal{S}$. More formally, the goal is to construct an $\mathcal{S} \supseteq \{\hat{x}\}$ such that the following holds:

$$Stable(\mathcal{S}, val) \quad and \quad Complete(\mathcal{S}, val)$$

where $Stable(\mathcal{S}, val) \equiv \forall x \in \mathcal{S} : val(x) = f_x(val)$, and
$Complete(\mathcal{S}, val) \equiv \forall x \in \mathcal{S} : \mathtt{Suff}(f_x, val, \mathcal{S})$.

Two basic actions are needed in order to construct such a stable and complete search space:

- `Update` : Variables of $\mathcal{S}$ that are in ws are potentially unstable. Hence they are reevaluated and updated, just as in the global schema.
- `Expand` : If a variable of $\mathcal{S}$ is stable but currently depends upon variables outside $\mathcal{S}$, then the search space is expanded as a step towards completeness.

The local schema terminates when the search space $\mathcal{S}$ is both stable and complete. We then have that val agrees with $[\![E]\!]$ on $\mathcal{S}$. Hence, as $\hat{x} \in \mathcal{S}$, $[\![E]\!](\hat{x}) = val(\hat{x})$.

Local fixpoint approximation : basic schema, parameterised by $Restore(x, ws)$

$val := \lambda x \in X.\bot \quad ws := X \quad \mathcal{S} := \{\hat{x}\}$
loop
  if $\mathcal{S} \cap ws \neq \emptyset \longrightarrow$
    -- `Update`
    $x := choose(\mathcal{S} \cap ws) \quad ws := ws \setminus \{x\}$
    $v_{old} := val(x) \quad val(x) := \mathtt{EvalRhs}(x, val)$
    if $v_{old} \neq val(x) \longrightarrow Restore(x, ws)$ **fi**
  $[\!]$ $\{x \in \mathcal{S} \cap \overline{ws} \mid not\ \mathtt{Suff}(f_x, val, \mathcal{S})\} \neq \emptyset \longrightarrow$
    -- `Expand`
    $x := choose(\{x \in \mathcal{S} \cap \overline{ws} \mid not\ \mathtt{Suff}(f_x, val, \mathcal{S})\})$
    $y := choose(\mathtt{Arg}(f_x) \setminus \mathcal{S})$
    $\mathcal{S} := \mathcal{S} \cup \{y\}$
  $[\!]$ **otherwise** $\longrightarrow$ **exit loop**
  fi
endloop
-- $\hat{x} \in \mathcal{S}$ and $val|_{\mathcal{S}} = [\![E]\!]|_{\mathcal{S}}$

Code fragment $Restore(x, ws)$ serves the same purpose as in the global schema and can be implemented as discussed in section 3. Implementing rule `Expand` is also easy: In case

semantic dependencies are used, the information in arg can be used to keep track of $\{x \in \mathcal{S} \cap \overline{ws} \mid not\ \mathtt{Suff}(f_x, val, \mathcal{S})\}$. For y (the variable to be added to $\mathcal{S}$), one of the variables from $arg(x) \setminus \mathcal{S}$ is then chosen. In case of syntactic dependencies, the extended evaluation function of section 3.2 can be exploited in following way: Whenever a variable of $\mathcal{S}$ is re-evaluated, we use this extended evaluation function to find out whether its new value currently depends on variables outside $\mathcal{S}$. In this way we can easily keep track of (a small superset of) $\{x \in \mathcal{S} \cap \overline{ws} \mid not\ \mathtt{Suff}(f_x, val, \mathcal{S})\}$.
Note that a practical implementation will probably give rule **Update** priority over rule **Expand**. I.e., in case the guards of **Update** and **Expand** are both enabled, rule **Update** is selected. In this way search space expansion only takes place when all variables of $\mathcal{S}$ are stable, thereby keeping the size of the search space as small as possible. For the same reason, additional search space reduction techniques can be integrated within the above local schema. Variables that were added to $\mathcal{S}$ at some point, may later on turn out not to be needed any longer, at least not as far as computing $[\![E]\!](\hat{x})$ is concerned. Such 'useless' variables may however cause a further (cumulative) useless expansion of the search space. The goal of (occasionally) running a reduction algorithm is precisely to remove useless variables from $\mathcal{S}$. Several reduction heuristics can be thought of. To mention one: In case semantic dependencies are used, then a simple, yet effective, reduction technique, exploiting the semantic dependency information contained in arg, would be to remove all variables from $\mathcal{S}$ that are no longer (directly or indirectly) 'reachable' from $\hat{x}$.
The idea underlying the local approximation schema, i.e. the construction of a stable and complete search space, is closely related to the "minimal function graphs" approach in refined denotational semantics [9]. It is also at the heart of the model-checking algorithms discussed in e.g. [1,23].

5 Local FixPoint Induction

In this section we present a local function **LocComp** for computing $[\![E]\!](x)$ based on fixpoint induction. An implicit use of fixpoint induction techniques can be found e.g. in [3,14,15,20,25], where algorithms, under the form of tableau systems, are discussed for model-checking the modal mu-calculus [13]. These tableau systems, however, essentially deal with boolean equations. The algorithm discussed in this section is based on a more explicit use of fixpoint induction. Furthermore it is not restricted to boolean variables.
Informally, the key idea is the following: In order to compute $[\![E]\!](x)$, first make a safe assumption about $[\![E]\!](x)$. I.e. take a value $v \sqsubseteq [\![E]\!](x)$ and *assume* that $[\![E]\!](x)$ equals v. Then check whether the latter assumption is correct. This is done by evaluating $f_x([\![E]\!])$ under the assumption that $[\![E]\!](x)$ equals v. If, under the latter assumption, it indeed turns out that $f_x([\![E]\!])$ evaluates to v, then the assumption was indeed correct, and hence $[\![E]\!](x) = v$. In order to compute $f_x([\![E]\!])$ under the assumption that v equals $[\![E]\!](x)$, many different components of $[\![E]\!]$ may be needed. For $[\![E]\!](x)$ there is no problem: According to the assumption, v can be used for $[\![E]\!](x)$. If, however, $[\![E]\!](x')$ is needed for some $x' \neq x$, then first compute $[\![E]\!](x')$ under the assumption that v equals $[\![E]\!](x)$. This can be done by a recursive application of the above schema. I.e. take a safe assumption v' for $[\![E]\!](x')$, and then try to prove that this assumption is correct by proving that $f_{x'}([\![E]\!])$ evaluates to v'. The difference with computing $f_x([\![E]\!])$ now is that $f_{x'}([\![E]\!])$ is computed under two assumptions: an assumption that v equals $[\![E]\!](x)$, and an assumption that v' equals $[\![E]\!](x')$. If, in order to compute $f_{x'}([\![E]\!])$, yet another component $[\![E]\!](x'')$ is needed ($x \neq x'' \neq x'$), then this component is again computed by recursively applying the above schema, etc. Termination is guaranteed as the list of assumptions grows with each recursive invocation.

To formally capture the above fixpoint induction technique, define $\rightsquigarrow$ as the smallest binary relation on $[X \rightharpoonup \mathcal{V}]$ generated by the following rules:[2]

$$(R_1) \quad \mathcal{A} \rightsquigarrow \epsilon$$

$$(R_2) \quad \frac{\mathcal{A} \rightsquigarrow \mathcal{C} \qquad \mathcal{A} \rightsquigarrow (x/v)}{\mathcal{A} \rightsquigarrow \mathcal{C}[x/v]}$$

$$(R_3) \quad \mathcal{A} \rightsquigarrow (x/\mathcal{A}(x)) \qquad x \in dom(\mathcal{A})$$

$$(R_4) \quad \frac{\mathcal{A}[x/v] \rightsquigarrow \mathcal{C}}{\mathcal{A} \rightsquigarrow (x/v)} \qquad x \notin dom(\mathcal{A}), v \sqsubseteq [\![E]\!](x), \text{ and } v = f_x(\theta[\mathcal{C}]) \text{ for all } \theta \in \Theta$$

Informally, $\mathcal{A} \rightsquigarrow \mathcal{C}$ may be read as follows: Under the assumption that $\mathcal{A}$ equals $[\![E]\!]$ on $dom(\mathcal{A})$, it follows that $\mathcal{C}$ equals $[\![E]\!]$ on $dom(\mathcal{C})$. Rules $(R_1), (R_2), (R_3)$ are straightforward. Rule (R_4) captures the induction principle: In order to derive (x/v) from $\mathcal{A}$ (with $x \notin dom(\mathcal{A})$ and $v \sqsubseteq [\![E]\!](x)$), a mapping $\mathcal{C}$ has to be derived from the extended assumption list $\mathcal{A}[x/v]$ such that the right hand side f_x associated with x evaluates to v when only information from $\mathcal{C}$ is used.

Theorem Let $\mathcal{A} \rightsquigarrow \mathcal{C}$. Then $\mathcal{A} = [\![E]\!]|_{dom(\mathcal{A})}$ implies $\mathcal{C} = [\![E]\!]|_{dom(\mathcal{C})}$.

Corollary Let $\epsilon \rightsquigarrow (x/v)$. Then $[\![E]\!](x) = v$.

Function `LocComp`$(x, \mathcal{A})$, based upon the above rules for $\rightsquigarrow$, is listed below. It computes $[\![E]\!](x)$ under the assumptions $\mathcal{A}$. Hence according to the above corollary, a toplevel call `LocComp`(x, ϵ) returns $[\![E]\!](x)$. Variable v is the current safe approximation for $[\![E]\!](x)$ (initially $v = \bot$). The goal of the inner while-loop is to check whether v equals $[\![E]\!](x)$ by evaluating $f_x([\![E]\!])$ under the extended assumption list $\mathcal{A}[x/v]$. This evaluation is done in a *lazy* (demand driven) manner: Only components of $[\![E]\!]$ that are really needed in order to evaluate $f_x([\![E]\!])$ are computed, one at a time. To implement lazy evaluation, the evaluation function `EvalRhs` is slightly extended in the following way: Its second parameter is allowed to be a *partial* mapping $\rho : X \rightharpoonup \mathcal{V}$, providing only values for variables belonging to its domain $dom(\rho)$. If the values of variables from $dom(\rho)$ suffice to evaluate f_x, then `EvalRhs`(x, ρ) returns this desired value, as before. Otherwise the symbol ? is returned, indicating that more argument values are needed in order to evaluate f_x. The latter can, of course, only occur in case $\mathtt{Arg}(f_x) \not\subseteq dom(\rho)$. The result of the lazy evaluation of $f_x([\![E]\!])$ is stored in v_{new}. If $v_{new} = v$, then it follows from rule (R_4) that, under the assumption $\mathcal{A}$, the assumption $v = [\![E]\!](x)$ was correct, and hence the value v is returned. If, on the other hand, $f_x([\![E]\!])$ evaluates to $v_{new} \neq v$, then it must be the case that $v \sqsubset v_{new} \sqsubseteq [\![E]\!](x)$. (This is easily proved using a simple inductive argument, exploiting the monotonicity of $E's$ right hand sides). Hence v is updated by v_{new}, and the whole procedure is repeated using the new value of v as a safe approximation for $[\![E]\!](x)$.

---

[2] The symbol $\rightharpoonup$ denotes *partial* mappings. The domain of $\mathcal{A} : X \rightharpoonup \mathcal{V}$, i.e. the set of variables $x \in X$ for which $\mathcal{A}(x)$ is defined, is denoted by $dom(\mathcal{A})$. ϵ denotes the 'empty' mapping, i.e. $dom(\epsilon) = \emptyset$. $\mathcal{A}[x/v]$ denotes $\mathcal{A}$ 'updated' by binding x to v, i.e., $dom(\mathcal{A}[x/v]) = dom(\mathcal{A}) \cup \{x\}$, $\mathcal{A}[x/v](x) = v$, and $\mathcal{A}[x/v](x') = \mathcal{A}(x')$ for $x' \neq x$. We also use (x/v) as an abbreviation for $\epsilon[x/v]$. Finally, $\theta[\mathcal{C}]$ denotes θ 'updated' by $\mathcal{C}$, i.e., $\theta[\mathcal{C}]$ is the mapping θ' such that $\theta'(x) = \mathcal{C}(x)$ if $x \in dom(\mathcal{C})$, and $\theta'(x) = \theta(x)$ otherwise.

Local fixpoint induction : basic schema

---

func LocComp($x, \mathcal{A}$) **return** (v) **is**
-- postcondition: $\mathcal{A} \rightsquigarrow (x/v)$
if $x \in dom(\mathcal{A}) \longrightarrow v := \mathcal{A}(x)$
[] $x \notin dom(\mathcal{A}) \longrightarrow$
$v := \bot$
loop
$\rho := \epsilon \quad v_{new} := \mathbf{EvalRhs}(f_x, \rho)$
while $v_{new} = ? \longrightarrow$
-- $\mathcal{A}[x/v] \rightsquigarrow \rho$
$x' := choose(\mathbf{Arg}(f_x) \setminus dom(\rho))$
$v' := \mathbf{LocComp}(x', \mathcal{A}[x/v])$
$\rho := \rho[x'/v'] \quad v_{new} := \mathbf{EvalRhs}(f_x, \rho)$
od
if $v = v_{new} \longrightarrow$ **exit loop fi**
-- $v \sqsubset v_{new} \sqsubseteq [\![E]\!](x)$
$v := v_{new}$
endloop
-- $\mathcal{A}[x/v] \rightsquigarrow \rho$, with $v \sqsubseteq [\![E]\!](x), x \notin dom(\mathcal{A}), f_x(\theta[\rho]) = v \quad \forall \theta \in \Theta$
-- Hence, using rule (R_4), $\mathcal{A} \rightsquigarrow (x/v)$
fi

Termination of LocComp is guaranteed as recursive invocations have more assumptions (i.e., $dom(\mathcal{A}) \subset dom(\mathcal{A}[x/v])$). Function LocComp, as it stands, is rather inefficient: The number of evaluations is in the worst case *exponential* in $|X|$. Hence its running time in practice may be catastrophic. It, however, only requires a strict minimum of storage space: Esentially, the current list of assumption $\mathcal{A}$ has to be stored, and for each variable of $dom(\mathcal{A})$ storage space for the local variable ρ is needed (usually $dom(\rho) << X$). In the local fixpoint approximation schema of section 4, on the other hand, storage space is needed to store the *complete* search space. LocComp only stores information concerning the current search path (i.e., for variables of $dom(\mathcal{A})$).

6 Optimising function LocComp

In this section we discuss two optimisations that, when combined, remove the exponential characteristic of LocComp. The result is an optimised function LocComp that runs in time *polynomial* (quadratic) in $|X|$.

Exploiting Lower Bounds In function LocComp($x, \mathcal{A}$) variable v is initialised to $\bot$. This is of course a safe approximation for $[\![E]\!](x)$. However, any value that is below $[\![E]\!](x)$ can be used to initialise v. Initialising v to a safe but somewhat bigger value than $\bot$ reduces the number of iteration steps needed for reaching local stability. Hence introduce an additional global data-structure $\mathcal{L} : X \to \mathcal{V}$ for keeping track of safe initialisation values, i.e. assertion $\mathcal{L} \sqsubseteq [\![E]\!]$ is maintained as an invariant. Hence $\mathcal{L}(x)$ can be used, instead of $\bot$, as an initialisation value for v. Keeping $\mathcal{L} \sqsubseteq [\![E]\!]$ invariant is easy: Initially $\mathcal{L} = \lambda x \in X.\bot$, thereby trivially satisfying $\mathcal{L} \sqsubseteq [\![E]\!]$. Upon return from LocComp($x, \mathcal{A}$) variable $\mathcal{L}(x)$ is updated by executing the assignment $\mathcal{L}(x) := v$, where v is the value returned by LocComp($x, \mathcal{A}$). It can easily be proved, exploiting monotocity, that v is still safe. Furthermore, the value of v upon return is always equal or greater than its initialisation value. Hence entries of $\mathcal{L}$ can only increase.

Reusing Information Consider again the while loop of LocComp. Its goal is to evaluate f_x by computing components (using a recursive call of LocComp), one at a time, until enough has been computed for evaluating f_x. The source of inefficiency here is that there is not the slightest re-use of information among the computation of different components. To remedy this, the specification of LocComp is extended as follows:

func LocComp($x, \mathcal{A}, \mathcal{C}$) **return** ($v, \Delta\mathcal{C}$)
-- precondition: $\mathcal{A} \rightsquigarrow \mathcal{C}$
-- postcondition: $\mathcal{A} \rightsquigarrow (x/v)$ and $\mathcal{A} \rightsquigarrow \Delta\mathcal{C}$

Hence there is an extra parameter $\mathcal{C} : X \twoheadrightarrow \mathcal{V}$, holding additional information (under the assumptions $\mathcal{A}$). Parameter $\mathcal{C}$ can be exploited to speed up LocComp by 'pruning' computations at the level of variables whose fixpoint value is known (under the assumptions $\mathcal{A}$). Furthermore, an additional component $\Delta\mathcal{C} : X \twoheadrightarrow \mathcal{V}$ is returned, holding the newly gathered information during execution of LocComp. This information can then be used by subsequent calls to LocComp (having the same assumption list). The computation of $\Delta\mathcal{C}$ is based on the following two additional rules:

$$(R_5) \quad \frac{\mathcal{A} \rightsquigarrow \mathcal{C}_1 \qquad \mathcal{A}[\mathcal{C}_1] \rightsquigarrow \mathcal{C}_2}{\mathcal{A} \rightsquigarrow \mathcal{C}_2}$$

$$(R_6) \quad \frac{\mathcal{A} \rightsquigarrow \mathcal{C}}{\mathcal{A}[x/v] \rightsquigarrow \mathcal{C}} \qquad x \notin dom(\mathcal{A})$$

Integrating the above two optimisations yields the following schema:

Local fixpoint induction : optimised schema

---

globvar $\mathcal{L} := \lambda x \in X.\bot$

func LocComp($x, \mathcal{A}, \mathcal{C}$) **return** ($v, \Delta\mathcal{C}$) **is**
  if $x \in dom(\mathcal{A}) \longrightarrow v := \mathcal{A}(x) \quad \Delta\mathcal{C} := \epsilon$
  [] $x \in dom(\mathcal{C}) \longrightarrow v := \mathcal{C}(x) \quad \Delta\mathcal{C} := \epsilon$
  [] **otherwise** $\longrightarrow$
    $v := \mathcal{L}(x) \quad \Delta\mathcal{C} := \epsilon$
    loop
      $\rho := \epsilon \quad v_{new} :=$ EvalRhs(x, ρ)
      while $v_{new} = ? \longrightarrow$
        $x' := choose(\mathbf{Arg}(f_x) \setminus dom(\rho))$
        ($v', \delta\mathcal{C}$) := LocComp($x', \mathcal{A}[x/v], \mathcal{C}[\Delta\mathcal{C}]$)
        $\Delta\mathcal{C} := \Delta\mathcal{C}[\delta\mathcal{C}]$
        $\rho := \rho[x'/v'] \quad v_{new} :=$ EvalRhs(x, ρ)
      od
      if $v_{new} = v \longrightarrow$ **exit loop fi**
      $v := v_{new} \quad \Delta\mathcal{C} := \epsilon$
    endloop
    $\mathcal{L}(x) := v \quad \Delta\mathcal{C} := \Delta\mathcal{C}[x/v]$
  fi

The exponential behaviour has been removed as a result of the above two optimisations. More precisely, it can be proved that for the above optimised function LocComp the

number of evaluations[3] is in the worst case $\mathcal{O}(\mathtt{H}.|X|^2)$. Intuitively this follows from the fact that $\mathcal{L}$-entries can only increase, and computations are pruned at the level of variables that are in $dom(\mathcal{C})$ (variables are only removed from $dom(\mathcal{C})$ when assumptions of $\mathcal{A}$ turn out to be invalid).
The price to pay for this reduction in time complexity is an additional global data-structure $\mathcal{L}$, and local structures $\mathcal{C}/\Delta\mathcal{C}$. Note however that $\mathcal{C}/\Delta\mathcal{C}$ were merely used to clearly illustrate how information is passed and reused between different LocComp calls. In a practical implementation, the latter variables can be replaced by one single global variable $g\mathcal{C} : X \rightharpoonup \mathcal{V}$, and a global stack *gstck* of variables.[4] Hence the increase in space complexity is rather small, and is usually not a problem. However, for applications where storage space is at premium (e.g. model-checking) the following intermediate method offers a good compromise between time and space complexity: Use the optimised schema, and, whenever storage space is needed (for $\mathcal{A}, \rho$) but none is available, then just clear some of the space occupied by $\mathcal{L}, g\mathcal{C}, gstck$. (This does not affect the correctness of the algorithm). If the erased information is needed later on, it will simply be recomputed. In this manner the given bounded storage space is optimally exploited, and the running time is close to optimal (w.r.t. the bounded memory).

7 Some Further Optimisations

Several further optimisations are possible. We briefly sketch two of them.

Collecting definitive values [18] : Upon return from LocComp$(x, \mathcal{A}, \mathcal{C})$ we have that $\mathcal{A} \rightsquigarrow \Delta\mathcal{C}$. Hence if the assumptions $\mathcal{A}$ are correct, i.e. if $\mathcal{A} = [\![E]\!]|_{dom(\mathcal{A})}$, then also $\Delta\mathcal{C} = [\![E]\!]|_{dom(\Delta\mathcal{C})}$. It is, however, at that point not known whether or not the assumptions $\mathcal{A}$ are correct. If, however, in order to compute $\Delta\mathcal{C}$, none of the assumptions of $\mathcal{A}$ have actually been used (consulted), either directly or indirectly via $\mathcal{C}$, then it really does not matter whether or not $\mathcal{A}$ is correct. For one might as well have taken $[\![E]\!]|_{dom(\mathcal{A})}$ instead of $\mathcal{A}$, and the same value for $\Delta\mathcal{C}$ would have been obtained. Hence, if $\mathcal{A}$ has not been used for computing $\Delta\mathcal{C}$, $\Delta\mathcal{C}$ agrees with $[\![E]\!]$ on $dom(\Delta\mathcal{C})$. In that case variables from $dom(\Delta\mathcal{C})$ have reached their final fixpoint value. An additional data-structure $\mathcal{D} : X \rightharpoonup \mathcal{V}$ can be used to collect variables that are known to have reached their final fixpoint value, i.e., $\mathcal{D} = [\![E]\!]|_{dom(\mathcal{D})}$ is maintained as an invariant. $\mathcal{D}$ can then be used to prune computations in a way similar to parameter $\mathcal{C}$. Hence it remains to detect, without too much overhead, whether or not $\Delta\mathcal{C}$ actually depends on $\mathcal{A}$. Note that $\Delta\mathcal{C}$ might have used $\mathcal{A}$ directly, but also indirectly via $\mathcal{C}$, which itself might depend upon $\mathcal{A}$. (cfr. precondition of LocComp). A straightforward approach would be to simply keep track of the set $used(x) \subseteq dom(\mathcal{A}) \cup dom(\mathcal{C})$ of variables that have been consulted during execution of LocComp$(x, \mathcal{A}, \mathcal{C})$. Computing $used(x)$ is straightforward (note that $used(x)$ can be reset to $\emptyset$ when $\Delta\mathcal{C}$ is reinitialised to ϵ). If $used(x)$ is empty upon return from LocComp$(x, \mathcal{A}, \mathcal{C})$, then $\Delta\mathcal{C}$ does not (in)directly depend on $\mathcal{A}$, and hence can be collected by $\mathcal{D}$. The above sketched approach roughly corresponds to the collection of definitive tuples as discussed in [18]. A disadvantage of this approach is that it requires set-manipulations. Below we sketch another approach, requiring less overhead (both in

[3] Only calls to EvalRhs that return a value different from ? are counted.

[4] $g\mathcal{C}$ is such that $g\mathcal{C} = \mathcal{C}$ upon entry of LocComp, and $g\mathcal{C} = \mathcal{C}[\Delta\mathcal{C}]$ upon exit. The stack *gstck* is used to implement the assignment $\Delta\mathcal{C} := \epsilon$ (following the assignment $v := v_{new}$), i.e. to restore $g\mathcal{C}$ to its entry-value $\mathcal{C}$. More precisely, upon entry of LocComp variable x is pushed onto *gstck* (in case $x \notin dom(\mathcal{A}), x \notin dom(\mathcal{C})$); Restoring $g\mathcal{C}$ to its entry value is done by popping all variables on top of x and then removing these variables from $dom(g\mathcal{C})$.

time and space). The idea is to use the algorithm from [21] for computing maximal strongly connected components (mscc) of directed graphs. Integrating [21] into LocComp is not too difficult, as both are based on depth-first search. By slightly modifying [21], the 'roots' of the mscc's of the 'dynamic dependency relation' can be detected. If, upon return from $\texttt{LocComp}(x, \mathcal{A}, \mathcal{C})$, it turns out that x is the 'root' of such a mscc, then $\Delta\mathcal{C}$ does not depend on $\mathcal{A}$ and hence can be collected. The main cost for detecting such roots, is an additional global variable $lowlink : X \to \texttt{Nat}$. No *used*-sets are needed. Informally, the reason that we can do without the $used(x)$ sets, is that we only have to check for emptyness of $used(x)$. By exploiting depth-first characteristics, this test can be implemented using *lowlink*, the set $used(x)$ itself is not needed to check emptyness of $used(x)$. (This is, btw, the reason why the algorithm in [21] is linear in the size of the graph, instead of quadratic).

Chasing dependencies : When v is updated by $v := v_{new}$, variable $\Delta\mathcal{C}$ is reset to ϵ. This is done in order to restore invariance of the assertion $\mathcal{A}[x/v] \quad \rightsquigarrow \quad \Delta\mathcal{C}$, which risks to be violated as a result of updating v. If, however, the old value of v has not been used in order to compute $\Delta\mathcal{C}$, then $\Delta\mathcal{C}$ need not be reinitialised but can stay as is. If the old value of v has been used, then $\Delta\mathcal{C}$ is re-initialised. Alternatively, one could try to find out (using e.g. the transitive closure of a semantic dependency graph as in [16]) by which fragment of $\Delta\mathcal{C}$ the old v-value has been used. Only that fragment then has to be reinitialised. Space limitations prevent us from discussing these optimisations in more detail (see [24] for more details). Just note that such optimisations do not touch upon the worst case behaviour of the algorithm: The number of evaluations remains, in the worst case, quadratic in $|X|$. They may however decrease the running time in practice.

8 Application : Preorder Checking

In this section we discuss one sample application for the generic algorithms discussed in previous sections. The application domain is the automatic verification of finite transition systems (finite-state machines). Verification is the process of comparing a transition system with a given system specification. One promising approach is *equivalence/preorder checking*. (Another complementary approach is model-checking). In this approach the system specification is also given under the form of second transition system. Verification then boils down to comparing two transition systems w.r.t. a given equivalence/preorder. As an example of a preorder relation we will consider the *prebisimulation preorder* [4], noted by $\preccurlyeq$.
Fix a finite transition system $\langle \mathcal{P}, \mathcal{A}ct, \rightarrow \rangle$, where $\mathcal{P}$ is the finite set of processes (states), $\mathcal{A}ct$ is the finite set of actions, and $\rightarrow \subseteq \mathcal{P} \times \mathcal{A}ct \times \mathcal{P}$ is the transition relation. We shall write $p \xrightarrow{a} q$ instead of $(p, a, q) \in \rightarrow$. Informally, $p \xrightarrow{a} q$ means that, when is state p, the system may move to state q by performing an action a.

Definition [4]. Let $\uparrow \subseteq \mathcal{P}$ be a set of divergent states. Define $\preccurlyeq$ (relative to $\uparrow$) as the largest relation $R \subseteq \mathcal{P} \times \mathcal{P}$ satisfying the following condition: Whenever $p\,R\,q$ and $a \in \mathcal{A}ct$ then

(i) $\forall p' \in \mathcal{P}$: $p \xrightarrow{a} p'$ implies ($\exists q' \in \mathcal{P}$: $q \xrightarrow{a} q'$ and $p'\,R\,q'$)
(ii) If $p \notin \uparrow$ then:
  (a) $q \notin \uparrow$, and
  (b) $\forall q' \in \mathcal{P}$: $q \xrightarrow{a} q'$ implies ($\exists p' \in \mathcal{P}$: $p \xrightarrow{a} p'$ and $p'\,R\,q'$)

Many other interesting behavioural relations can be obtained as special cases of $\preccurlyeq$ (see [4], e.g. by taking $\uparrow = \emptyset$ strong bisimulation equivalence [19] is obtained.) The preorder $\preccurlyeq$ is defined as the largest relation satisfying a number of conditions. These conditions can be rephrased in terms of a monotone *boolean* equation system. I.e., take $\mathcal{V} = \{false, true\}$, and let $\sqsubseteq$ be defined by $true \sqsubset false$.[5] For the set of left hand side variables take:

$$X = \{x_{p,q} \mid p \in \mathcal{P}, q \in \mathcal{P}\} \cup \{y_{p,q} \mid p \in \mathcal{P}, q \in \mathcal{P}\}$$
$$\cup \{z_{a,p',q} \mid a \in \mathcal{A}ct, p' \in \mathcal{P}, q \in \mathcal{P} \text{ such that } p \xrightarrow{a} p' \text{ for some } p \in \mathcal{P}\}$$

For each variable $x \in X$ there is one equation in E having x as its left hand side. The equations of E are of the following general form:

$$\begin{cases} x_{p,q} = y_{p,q} \wedge [\, p \notin \uparrow \;\Rightarrow\; (q \notin \uparrow \wedge y_{q,p})\,] \\ y_{p,q} = \bigwedge_{a,p' \mid p \xrightarrow{a} p'} z_{a,p',q} \\ z_{a,p',q} = \bigvee_{q' \mid q \xrightarrow{a} q'} x_{p',q'} \end{cases}$$

The following relationship holds between $\preccurlyeq$ and the equation system E defined above:

$$\forall p, q \in \mathcal{P}: \; p \preccurlyeq q \iff [\![E]\!](x_{p,q}) = true$$

Hence checking whether $p \preccurlyeq q$ is reduced to computing the least component $[\![E]\!](x_{p,q})$. Furthermore a simple calculation shows that $|E| = \mathcal{O}(|\mathcal{P}|.| \rightarrow |)$. Hence using the global algorithms of section 3, $[\![E]\!]$ can be computed using at most $\mathcal{O}(|E|) = \mathcal{O}(|\mathcal{P}|.| \rightarrow |)$ right hand side evaluations. The best algorithm [4] for computing $\preccurlyeq$ known today, runs in time $\mathcal{O}(| \rightarrow |^2)$. Note however that the latter complexity result includes the evaluation time of right hand sides. If we would take this time into account, our result $\mathcal{O}(|\mathcal{P}|.| \rightarrow |)$ would also become $\mathcal{O}(| \rightarrow |^2)$. Hence for this particular case, our generic global algorithm matches the so far best known 'ad hoc' constructed global algorithm. What we also have are local algorithms for checking $\preccurlyeq$, running in time $\mathcal{O}(| \rightarrow |^2)$.

9 Conclusions

In his paper we have discussed the systematic stepwise construction of efficient generic global and local fixpoint algorithms based on fixpoint approximation and fixpoint induction techniques. The proposed algorithms can be instantiated for a concrete equation system by providing a concrete implementation for the evaluation function `EvalRhs`. Most of the concepts used (e.g. semantic dependencies, local iteration strategies, etc.) are not new in se, but have already been discussed by various researchers in various application areas. However, putting these ideas into the more abstract and general setting of monotone equation systems is new. The only other generic fixpoint algorithm that we are aware of is [17], where a (more) general top-down fixpoint algorithm is proposed which only assumes that the transformation is given by an effective procedure satisfying some weak (monotonicity) properties. The algorithm in [17] seems to be similar to our local fixpoint induction algorithm combined with semantic dependencies for removing redundant computations.

For the global/local algorithms based on fixpoint approximation, the number of evaluations for x only depends upon the right hand side of x, and is independent of the rest of the program. This property does not hold for the local algorithms based on fixpoint

[5] We take $true \sqsubset false$ as $\preccurlyeq$ is defined as a largest fixpoint, whereas we compute least fixpoints.

induction, where a variable x is evaluated at most $\mathtt{H}.X$ times.
The fixpoint approximation schema's are well-suited to be implemented using parallel architectures. Imagine for example n processors, all selecting and evaluating variables in parallel. Because of the syntactic/semantic dependencies, the processing of a variable x only affects the immediate 'neighbourhood' of x. Hence little interference between the processors is to be expected. Such a parallel implementation might potentially substantially reduce the running in practice. The fixpoint induction algorithms, on the other hand, cannot benefit from parallel execution, as they are based on depth-first search which is inherently sequential.
Sparseness seems to be a necessary condition for successfully applying local algorithms. For if the equations system E is not sufficiently sparse, then a local algorithm will end up computing (almost) the complete least solution $[\![E]\!]$, which is precisely what a local algorithm attempts to avoid. Often a sufficiently high degree of sparseness can be obtained by first transforming E using some form of 'quotienting' [1,22] operation. To conclude, let us illustrate this important point by means of a simple example (rooted in model-checking). Consider a transition system $\langle \mathcal{P}, \mathcal{A}ct, \rightarrow \rangle$. Take $\mathcal{V} = 2^{\mathcal{P}}$, and for $\sqsubseteq$ take ordinary set-inclusion $\subseteq$. Let E be the following equation system:

$$E \;\equiv\; \{\; x = pre(\,a_1, x\,) \;\cup\; pre(\,a_2, \mathcal{P}\,)$$

where a_1 and a_2 are two actions from $\mathcal{A}ct$, and $pred$ is the predecessor function, i.e., $pred(\,a, Q\,)$ denotes the set of states from which a state of Q can be reached by performing an a-action, i.e., $pred(\,a, Q\,) = \{p \in \mathcal{P} \mid p \xrightarrow{a} p' \text{ for some } p' \in Q\}$. (Note: $[\![E]\!](x)$ is the set of states from which there is a path consisting of a finite number of a_1-action eventually followed by a a_2-action.) Suppose that we are given a state $\hat{p}$ (e.g. the intial system state), and we want to check whether $\hat{p} \in [\![E]\!](x)$. Instead of computing $[\![E]\!](x)$ and then checking whether $\hat{p}$ is in $[\![E]\!](x)$, we first construct a 'quotient' boolean equation system E' as follows: Left hand side variables of E' are of the form x_p with $p \in \mathcal{P}$. The equations of E' are of the form

$$E' \;\equiv\; \{\; x_p = (\; \bigvee_{p' \in \mathcal{P} \mid p \xrightarrow{a_1} p'} x_{p'}\;) \;\vee\; (\; \bigvee_{p' \in \mathcal{P} \mid p \xrightarrow{a_2} p'} true\;)$$

(There is one such equation for every $p \in \mathcal{P}$). E' is equivalent to E in the sense that $p \in [\![E]\!](x)$ iff $[\![E']\!](x_p) = true$. (Note that this time we take $false \sqsubset true$). As E' is sparse, it makes sense to apply local algorithms in order to compute $[\![E']\!](x_p)$. The above sketched technique can be generalised to e.g. equation systems used in abstract interpretation. There it is often the case that $\mathcal{V}$ is of the form $[D \rightarrow R]$ (in our example $\mathcal{V} = [\mathcal{P} \rightarrow \{false, true\}]$). And one is often interested, not in $[\![E]\!](\hat{x})$, but in $[\![E]\!](\hat{x})$ applied to some $\hat{d} \in D$. This situation occurs e.g. when taking into account the way predicates are called (call patterns) in the top-down semantics. Instead of first computing $[\![E]\!](\hat{x})$ and then computing $([\![E]\!](\hat{x}))(\hat{d})$, an equivalent 'quotient' equation system E' can first be constructed (at least conceptually). As E' often exhibits a high degree of sparseness, a local algorithm seems the proper choice for computing $[\![E']\!](\hat{x}_{\hat{d}})$.

References

1. Andersen, H. R.: *Model Checking and Boolean Graphs*, ESOP'92, LNCS 582, 1992
2. Clarke, E.M., Emerson, E.A., Sistla, A.P.: *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Progr. Languages and Systems, Vol.8, No. 2, pp. 244-263, April 1986

3. Cleaveland, R.: *Tableau-Based Model Checking in the Propositional Mu-Calculus*, Acta Informatica, 1990
4. Cleaveland, R., Steffen, B.: *Computing Behavioural Relations, Logically*, ICALP 91, pp. 127-138, LNCS 510
5. Cleaveland, R., Steffen, B.: *A Linear-Time Model Checking Algorithm for the Alternation-Free Modal Mu-Calculus*, CAV'91, LNCS 575, 1991
6. Cousot, P., Cousot, R.: *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, POPL, 1977.
7. Debray, S. K.: *Static inference of modes and data dependencies in logic programs*, TOPLAS, 11 (3), 1989.
8. Emerson, E.A., Lei, C.-L.: *Efficient model checking in fragments of the propositional μ-calculus*, LICS, 267-278, 1986
9. Jones, N.D., Mycroft, A.: *Data flow analysis of applicative programs using minimal function graphs*, POPL, 1986.
10. Jorgensen, N.: *Chaotic fixpoint iteration guided by dynamic dependency*, in Workshop on Static Analysis (WSA'93), LNCS.
11. O'Keefe, R.A.: *Finite fixed-point problems*, ICTL 1987.
12. Kildall, G.A.: *A unified approach to global program optimization*, POPL 1973.
13. Kozen, D.: *Results on the propositional μ-calculus*, TCS, 27, 1983.
14. Larsen, K.G.: *Proof systems for Hennessy-Milner logic with recursion*, CAAP, 1988, see also TCS, 72, 1990
15. Larsen, K.G.: *Efficient local correctness checking*, CAV'92, Forthcoming
16. Le Charlier, B., Van Hentenryck, P.: *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for PROLOG*, TOPLAS, Vol. 16, nr. 1, January 1994.
17. Le Charlier, B., Van Hentenryck, P.: *A universal top-down fixpoint algorithm*, Technical Report 92-22, Institute of Computer Science, University of Namur, Belgium, April 1992.
18. Le Charlier, B., Degimbe, O., Michel, L., Van Hentenryck, P.: *Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog*, in Workshop on Static Analysis (WSA'93), LNCS.
19. Milner, R.: *Communication and Concurrency*, Prentice-Hall International, 1989.
20. Stirling, C., Walker, D.: *Local model checking in the modal mu-calculus*, TCS, October 1991, see also LNCS 351, 369-383, CAAP 1989
21. Tarjan, R.E.: *Depth first search and linear graph algorithms*, SIAM J. Comput., 1 (2), 1972.
22. Vergauwen, B., Lewi, J.: *A linear algorithm for solving fixed points equations on transition systems*, CAAP'92, LNCS 581, 322-341
23. Vergauwen, B., Lewi, J.: *Efficient Local Correctness Checking for Single and Alternating Blocks*, ICALP'94, LNCS 820.
24. Vergauwen, B.: *Verification of Temporal Properties of Concurrent Systems*, Ph.D. thesis, in preparation.
25. Winskel, G.: *A note on model checking the modal ν-calculus*, ICALP, LNCS 372, 1989, see also TCS 83, 1991
26. Wauman, J.: *Ontwerp en implementatie van lokale verificatie-algoritmen voor reactieve systemen met behulp van temporele logic*, Ms. thesis, Dept. of Computer Science, K.U.Leuven, 1992-1993.