Vol. 48 No. SIG 10(PRO 33) IPSJ Transactions on Programming June 2007

Regular Paper

Sub-Computation Based Transition Predicate Abstraction

Carl Christian Frederiksen† and Masami Hagiya†

The transition predicate abstraction framework developed by Podelski, et al. (2005) captures size relations over state transitions which can be used to show infeasibility of certain program computations. In particular, general liveness properties (i.e., properties of infinite computations) can be verified by reducing the verification problem to one of fair termination and then proving that all (infinite) fair computations are infeasible. We present an extension of the algorithm by Podelski, et al. that can be used to improve the precision of transition predicate abstraction as well as speed up analysis time for programs with well-structured control-flow. The main key is to identify *sub-computations* that can be evaluated independently of their context. Efficiency is then readily improved by analyzing each sub-computation in turn, thus avoiding to reanalyze the effect of a given sub-computations for different contexts. Further, precision can be improved by using stronger methods for extracting summary information about a given sub-computation. We present two versions of the sub-computation based analysis: one for a non-parallel imperative language with loops and recursive procedures, serving as an introduction, and one for the extension of the non-parallel language to a parallel language with synchronous communication via statically named channels.

1. Introduction

Transition predicate abstraction[11] is in essence an adaption of size-change termination[8] to fair termination. By abstracting size changes of variables over program transitions the set of infinite computations can be approximated by computing the fix-point of composition of the abstract transitions for a suitable abstract domain. Termination can then be proven by demonstrating that all idempotent abstract transitions in the closure set cause some expression to progress towards a bound. The extension to fair termination requires the algorithm to only consider termination for computations that satisfy a given fairness requirement specified as a criterion over the program traces.

We develop the notion of a flow graph for infinite computations whose strongly connected components identify the proper sub-computations of a given program. By analyzing each sub-computation in turn (starting with those that do not contain other sub-computations) a number of advantages are gained: the closure set can be computed faster and the set of counter example traces is restricted to sub-computations that can cause a violation of the fairness requirement. Finally it is possible to uncover stronger abstract transitions for sub-computations than those generated by the closure set computation by using linear programming[6].

The analysis is further improved by computing compositions for "inter-recursion traces" (corresponding to basic blocks modulo conditionals) prior to the closure set computation, a technique commonly used for other types of program analyzes. This both improves on efficiency and precision: redundant abstractions for loops are avoided and non-monotone progress over inter-recursion traces can be captured by choosing appropriate abstraction functions.

We illustrate the methods with a small, but complete, running example and demonstrate the potential improvement in practice by analyzing the example with an implementation of the algorithm. Finally we present an extension of the sub-computation based verification algorithm to a parallel language with synchronous communication. We propose using state predicate abstraction to model the communication behavior and to guide the extraction of transition relations for a given program sub-computation. This allows the algorithm to only consider interleavings of sub-computations of processes that may affect each other's execution, thus potentially reducing the size of the resulting model.

Section 2 introduces the subject language for the analysis and Section 3 defines transition predicate abstraction based on linear inequal-

† Department of Computer Science, Graduate School of Information Science and Technology, the University of Tokyo

ities. Section 4 introduces a criteria for detecting progress towards a bound and formalizes the algorithm of Ref. 11). Diving into our contributions, Section 5 introduces the "*infinite flow graph*" which identifies the sub-computations, Section 6 improves analysis speed by abstraction over "inter-recursion sequences" and Section 7 defines a widening operator on transition predicates to ensure convergence of the closure set computation. Section 8 proves correctness of the verification algorithm and reports on experimental results.

Section 9 proceeds to build up to the parallel verification algorithm, Section 10 defines a parallel language with synchronous communication and the notion of sub-computations in the parallel setting is investigated in Section 11. Section 12 introduces state predicate abstraction which is used in conjunction with "progressive paths" of Section 13 to extract transition relations in Section 14. Section 15 extends the idea of "inter-recursion sequences" to the parallel setting and the parallel verification algorithm is given in Section 16. Finally Section 17 concludes.

2. Subject Language

The subject language is a simple imperative language with recursive procedures and a `choose` construct for guarded commands. The syntax is given in **Fig. 1**. In the subsequent analysis we will assume that each statement in the subject program has been labeled uniquely and for technical reasons two distinct labels in the case of `while` statements. The label $\mathtt{f}^E$ designates the first label in the procedure `f` and the label $\mathtt{f}^X$ designates the last label in the procedure `f`, i.e., the label preceding the `return` statement. The function *next* maps a label to the next label following the control flow. The set of all labels in program $\mathcal{P}$ is denoted *label*$(\mathcal{P})$. Finally, the notation $stmt_l$ designates the statement at label l, $\mathtt{f}^{(i)}$ designates the i^{th} parameter of the procedure `f` and $\mathtt{f}^{(R)}$ designates the return variable for procedure `f`.

(1) A *program store* $\rho \in Store$ is a mapping of variables into integer values and the special value `rtn`, which serves as a placeholder for the return value when evaluating a procedure, $\rho : Variables \rightarrow \mathbb{Z} \cup \{\mathtt{rtn}\}$ written $\rho = [\mathtt{x}_1 \mapsto v_1, \ldots, \mathtt{x}_n \mapsto v_n]$. Store updates are written: $\rho[\mathtt{x} \mapsto v]$.

(2) A *configuration* is a tuple $(l, \rho) \in Labels \times Store$.

(3) A *program stack* $s \in S$ is an ordered sequence of configurations: $s = (l_1, \rho_1), \ldots, (l_n, \rho_n)$. In the following we will sometimes use the shorthand notation $s = [(l_i, \rho_i)]_{i=1}^{n}$ and the operator ':' for pushing configurations onto a program stack: $[(l_i, \rho_i)]_{i=1}^{n-1} : (l_n, \rho_n) = [(l_i, \rho_i)]_{i=1}^{n}$. The label and store at the topmost configuration of the stack s above is given by $label(s) = l_n$ and $store(s) = \rho_n$.

(4) The one-step semantics of the language, given in **Fig. 2**, specifies a transition relation on program stacks, $s \rightarrow s'$. Note that `choose` statements nondeterministically selects any of the open guards for execution. Each operator op is associated with a function $[\![op]\!] : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$. For an expression e and a store ρ evaluation is denoted $\rho \models_{exp} e \rightarrow v$ where v is the resulting value.

(5) An *initial configuration* is a program stack $s_1 = (\mathtt{f}^E, \rho_1)$ where `f` is the first procedure definition in the program and ρ_1 specifies the program input.

(6) A *computation* is a finite or infinite sequence of program stacks $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ such that s_1 is an initial configuration and $s_i \rightarrow s_{i+1}$ for all $i \geq 1$.

(7) A *program trace* is a finite or infinite sequence of program labels $\tau = l_1 \rightarrow l_2 \rightarrow \ldots$ such that there exists a computation realizing the trace: $\pi = [(l_{(1,i)}, \rho_{(1,i)})]_{i=1}^{N_1} \rightarrow [(l_{(2,i)}, \rho_{(2,i)})]_{i=1}^{N_2} \rightarrow \ldots$ where l_j is the label of the configuration on top of the j^{th} stack: $l_j = l_{(j,N_j)}$ for all $j \geq 0$. Define the *trace* of π such that $trace(\pi) = \tau$.

(8) The *control-flow graph* $cfg(\mathcal{P})$ of a program $\mathcal{P}$ is a graph that contains precisely all the traces in $\mathcal{P}$. The restriction to the set of labels $L \subseteq Labels$ is the graph $cfg(\mathcal{P})|_L := (V \cap L, E \cap L \times L)$ where $(V, E) = cfg(\mathcal{P})$.

3. Extracting Transition Relations

The basis for proving fair termination is to extract "safe" relations between variables over transitions in the program that assert how the values of variables change for the different transitions in the program. The notion of transition relations is analogous to that of size-change graphs[8] which are used to prove termination for first order functional programs over well-

$$
\begin{array}{lcll}
\mathcal{P} & ::= & \mathit{def}_1 \ldots \mathit{def}_n & \text{(Program)}\\
\mathit{def} & ::= & \mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_k)\{\mathit{stmts};\ l:\ \mathtt{return}\ \mathtt{x}_r\} & \text{(Procedure)}\\
 & & & \\
\mathit{stmts} & ::= & \epsilon \mid \mathit{stmt};\ \mathit{stmts} & \text{(Statement Block)}\\
\mathit{stmt} & ::= & l:\mathit{stmt} \mid \mathtt{x} := e \mid \mathtt{x} := \mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_n) & \text{(Statement)}\\
 & \mid & \mathtt{if}(\mathit{test})\ \{\mathit{stmts}_t\}\ \mathtt{else}\ \{\mathit{stmts}_f\} & \\
 & \mid & \mathtt{while}(\mathit{test})\{\mathit{stmts}\} & \\
 & \mid & \mathtt{choose}\ \mathit{test}_1 \rightarrow \{\mathit{stmts}_1\}\ []\ \ldots\ []\ \mathit{test}_n \rightarrow \{\mathit{stmts}_n\} & \\
 & & & \\
e & ::= & v \mid \mathtt{x} \mid e_1\ \mathit{op}\ e_2 & \text{(Integer Expression)}\\
 & & & \\
\mathit{test} & ::= & \mathtt{T} \mid \mathtt{F} & \text{(Boolean Constant)}\\
 & \mid & \mathit{test}_1 \wedge \mathit{test}_2 \mid 0 = e \mid 0 \le e \mid 0 < e & \text{(Boolean Expression)}\\
 & & & \\
\mathtt{f} & : & \text{Procedure identifiers } (\mathtt{f},\ \mathtt{g},\ \ldots) & \\
\mathtt{x} & : & \text{Variable identifiers } (\mathtt{x},\ \mathtt{y},\ \ldots) & \\
l & : & \text{Labels identifiers } (1,\ 2,\ \ldots) & \\
v & : & \text{Integers } (\mathbb{Z}) & \\
\mathit{op} & : & \text{Binary Operators } (\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}) &
\end{array}
$$

Fig. 1 Program syntax.

$$
[\text{ASN}]\quad \frac{\rho \models_{exp} e \rightarrow v}{s:(l,\rho) \rightarrow s:(\mathit{next}(l), \rho[\mathtt{x} \mapsto v])}\quad (\text{if } \mathit{stmt}_l = \mathtt{x} := e)
$$

$$
[\text{CALL}]\quad \frac{}{s:(l,\rho) \rightarrow s:(\mathit{next}(l), \rho[\mathtt{x} \mapsto \mathtt{rtn}]) : (\mathtt{f}^E, [\mathtt{f}^{(\mathtt{j})} \mapsto \rho_{\mathtt{i}}(\mathtt{x}_{\mathtt{j}})]_{\mathtt{j}=1}^{\mathtt{k}})}\quad \left(\begin{array}{l}\text{if } \mathit{stmt}_l = \\ \mathtt{x} := \mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_k)\end{array}\right)
$$

$$
[\text{RTN}]\quad \frac{}{s:(l',\rho''):(l,\rho) \rightarrow s:(l', \rho''[\mathtt{y} \mapsto \rho(\mathtt{x})])}\quad \left(\begin{array}{l}\text{if } \mathit{stmt}_l = \mathtt{return}\ \mathtt{x},\\ \text{where } \rho''(\mathtt{y}) = \mathtt{rtn} \text{ for some } \mathtt{y}\end{array}\right)
$$

$$
[\text{IF}]\quad \frac{\rho \models_{exp} \mathit{test} \rightarrow b}{s:(l,\rho) \rightarrow s:(l_b,\rho)}\quad \left(\begin{array}{l}\text{if } \mathit{stmt}_l = \mathtt{if}(\mathit{test})\ \{l_{\mathtt{T}}:\ldots\}\ \mathtt{else}\ \{l_{\mathtt{F}}:\ldots\},\\ \text{where } b \in \{\mathtt{T},\mathtt{F}\}\end{array}\right)
$$

$$
[\text{WH TRUE}]\quad \frac{\rho \models_{exp} \mathit{test} \rightarrow \mathtt{T}}{s:(l,\rho) \rightarrow s:(l',\rho)}\quad (\text{if } \mathit{stmt}_l = \mathtt{while}(\mathit{test})\ \{l':\ldots\},\ )
$$

$$
[\text{WH FALSE}]\quad \frac{\rho \models_{exp} \mathit{test} \rightarrow \mathtt{F}}{s:(l,\rho) \rightarrow s:(\mathit{next}(l),\rho)}\quad (\text{if } \mathit{stmt}_l = \mathtt{while}(\mathit{test})\ \{l':\ldots\},\ )
$$

$$
[\text{CHOOSE}]\quad \frac{\rho \models_{exp} \mathit{test}_k \rightarrow \mathtt{T}}{s:(l,\rho) \rightarrow s:(l_k,\rho)}\quad \left(\begin{array}{l}\text{if } \mathit{stmt}_l = \left\{\begin{array}{ll}\mathtt{choose} & \mathit{test}_1 \rightarrow \{l_1:\ldots\}\\ \quad [] & \ldots\\ \quad [] & \mathit{test}_n \rightarrow \{l_n:\ldots\}\end{array}\right.\\ \text{for any } k\end{array}\right)
$$

$$
\frac{}{\rho \models_{exp} \mathtt{x} \rightarrow \rho(\mathtt{x})}\qquad \frac{}{\rho \models_{exp} v \rightarrow v}\ (\text{if } v \in \mathbb{Z})\qquad \frac{\rho \models_{exp} e_1 \rightarrow v_1,\ \rho \models_{exp} e_2 \rightarrow v_2}{\rho \models_{exp} e_1\ \mathit{op}\ e_2 \rightarrow [\![\mathit{op}]\!](v_1,v_2)}
$$

Fig. 2 Semantics.

founded data using the following principle: if all infinite computations strictly decrease the size of a parameter then no infinite computations are realizable because they would violate the well-foundedness assumption on the data domain. In the present setting the data domain is the set of integers which is not well-founded, so the method for proving infinite computations to be infeasible is to show progress towards a bound (in accordance with the transition relations). For this reason, the transition relations need to be augmented with a set of invariants (preconditions) that can be used as bounds.

Definition 3.1. (Transition Relation)*: A transition relation is a tuple:*
$(l \rightarrow l', P, R)$ *where* $l, l' \in$ *Labels,* P *is a set of preconditions and* R *is a set of transition predicates of form:*

$$
\begin{array}{lcll}
0 & \bowtie_p & k + \sum_{i=1}^{n} k_i \mathtt{x}_i & \textit{(precon.)}\\
\mathtt{x}' & \bowtie_t & k + \sum_{i=1}^{n} k_i \mathtt{x}_i & \textit{(trans. pred.)}
\end{array}
$$

```
dbAccess(    r,        // number of read locks held
             w,        // number of write locks held
             q ) {     // number of pending writers
 1:2: while(T) {
   3: choose  0 = r ∧ 0 = w ∧ 0 < q → {  4: w := 1;                            // acquire write lock
                                        5: q := updatePending(q,q);  }
        []           0 = w ∧ 0 = q → {  6: r := r + 1;  }                  // acquire read lock
        []           0 < w         → {  7: w := 0;  }                      // release write lock
        []    0 < r                → {  8: r := r − 1;  };  };             // release read lock
 9: return q  }
updatePending( q, i ) {
 10: i := i − 1;
 11: if(0 < i) {
   12: choose  T → { 13: q := q − 1;  } // request timed out
           []  T → { 14: };             // request still pending
   15: q := updatePending(q, i);  }
 16: else { q := q − 1;  }
 17: return q }
```

Fig. 3 Example program.

where $\bowtie_p$ is a relation over $\mathbb{Z}$ ($<$, $\leq$, $=$), $\bowtie_t$ is a relation over $\mathbb{Z}$ ($<$, $\leq$, $=$, $\geq$, $>$) and $k, k_1, \ldots, k_n \in \mathbb{Z}$ are integer constants. The primed variable $\mathtt{x}'$ on the left-hand side of a transition predicate denotes the value of x *after the transition. The set of all transition relations is denoted V_{inf}, and the set of all sets of preconditions and that of transition predicates are denoted $\mathcal{P}$ and $\mathcal{R}$ respectively.*

Example: *Consider the program in* **Fig. 3** *which will be used as a running example. The program models management of read/write access to a database: multiple read access is permitted, but write access is exclusive. The procedure* `dbAccess` *models a policy for granting read and write requests by non-deterministically choosing to execute any of the open guards. Read requests are granted only if no write requests are pending and no write lock is held. Write requests are assumed to have been enqueued and can be granted when no locks are held on the database. When a write request is granted, the queue of pending writers is updated by calling* `updatePending`. *Write requests can time out, which is modeled by non-deterministically dropping zero or more pending write requests. The procedure then finally "dequeues" the write request that was just granted.*

- $(8 \to 2, \emptyset, \{\mathtt{r}' = \mathtt{r} - 1, \mathtt{w}' = \mathtt{w}, \mathtt{q}' = \mathtt{q}\})$ *is a transition relation for the transition $8 \to 2$, corresponding to the statement* `r:=r-1`.
- $(3 \to 4, \{0 = \mathtt{r}, 0 = \mathtt{w}, 0 < \mathtt{q}\}, \{\mathtt{r}' = \mathtt{r}, \mathtt{w}' = \mathtt{w}, \mathtt{q}' = \mathtt{q}\})$ *is a transition relation for the transition $3 \to 4$ which selects the first guard in the* `choose` *statement.*

Transition relations assert a relation on program stacks over certain traces which are encoded by a non-deterministic trace automaton.

Definition 3.2. (Trace Automaton)*: A* trace automaton *A is a non-deterministic finite automaton over the set of program labels, Label. We write $\tau = l_1 \to \ldots \to l_n \in A$ if there exists a run $l_1, l_2, \ldots, l_n$ of the automaton A. In the following $A_1 \circ A_2$ denotes the concatenation of A_1 and A_2, such that $(w_1 l w_2) \in A_1 \circ A_2$ for all $(w_1 l) \in A_1, (l w_2) \in A_2$. When clear from the context we use the notation $l_1 \to \ldots \to l_n$ for the trace automaton that only accepts the trace $l_1 \to \ldots \to l_n$.*

Definition 3.3. (Safety)*: A transition relation v is* safe *for the trace automaton A iff*

$$v = (l \to l', \{0 \bowtie_i k_i + \textstyle\sum_{j=1}^{n} k_{(i,j)}\mathtt{x}_j\}, \{\mathtt{x}'_i \bowtie_i k_i + \textstyle\sum_{j=1}^{n} k_{(i,j)}\mathtt{x}_j\})$$

implies that for any computation $\pi = [(l_{(1,i)}, \rho_{(1,i)})]_{i=1}^{N_1} \to [(l_{(2,i)}, \rho_{(2,i)})]_{i=1}^{N_2} \to \ldots$ for which there exists indices $a < b$ such that $l_{(a,N_a)} \to \ldots \to l_{(b,N_b)} \in A$ and $l = l_{(a,N_a)}$, $l' = l_{(b,N_b)}$, the following holds:

$$\begin{aligned}\forall i: &\quad 0 \bowtie_i k_i + \textstyle\sum_{j=1}^{n} k_{(i,j)}\rho_{(a,N_a)}(\mathtt{x}_j),\\ \forall i: &\quad \rho_{(b,N_b)}(\mathtt{x}_i)\\ &\quad\quad \bowtie_i k_i + \textstyle\sum_{j=1}^{n} k_{(i,j)}\rho_{(a,N_a)}(\mathtt{x}_j).\end{aligned}$$

Example: *The two transition relations in the previous example are safe for the trace automata $8 \to 2$ and $3 \to 4$ respectively. The statement at label 8 is $\mathtt{r} := \mathtt{r} - 1$ so it is safe to assert that neither* w *or* q *change, i.e., $\mathtt{w}' = \mathtt{w}$ and $\mathtt{q}' = \mathtt{q}$. The value of* r *is decremented by 1, so it is also safe to assert $\mathtt{r}' = \mathtt{r} - 1$. For the transition from 3 to 4 the guard must necessarily evaluate to true, so it is safe to assert*

$$
\begin{array}{ll}
\mathcal{S}[\![l : \mathtt{x} := e]\!]\psi, l' & = (l \to l', \mathit{Id}(l,l')[\emptyset, \mathtt{x}' = \mathcal{S}_e[\![e]\!]]) \\
\mathcal{S}[\![l : \mathtt{if}(\mathit{test})\{l_{\mathtt{T}} \ldots\}\ \mathtt{else}\ \{l_{\mathtt{F}} \ldots\}]\!]\psi, l_{\mathtt{T}} & = (l \to l_{\mathtt{T}}, \mathit{Id}(l,l_{\mathtt{T}})[\mathit{test}, \emptyset]) \\
\mathcal{S}[\![l : \mathtt{if}(\mathit{test})\{l_{\mathtt{T}} \ldots\}\ \mathtt{else}\ \{l_{\mathtt{F}} \ldots\}]\!]\psi, l_{\mathtt{F}} & = (l \to l_{\mathtt{F}}, \mathit{Id}(l,l_{\mathtt{F}})[\neg\mathit{test}, \emptyset]) \\
\mathcal{S}[\![l : \mathtt{choose}\ \ \mathit{test}_1 \to \{l_1 \ldots\}\ []\ \ldots & \\
\qquad\qquad []\ \ \mathit{test}_n \to \{l_n \ldots\}\ ]\!]\psi, l_i & = (l \to l_i, \mathit{Id}(l,l_i)[\mathit{test}_i, \emptyset]) \\
\mathcal{S}[\![l : \mathtt{while}(\mathit{test})\{l_{\mathtt{T}} \ldots\}]\!]\psi, l' & =
\end{array}
$$

$$
\left\{\begin{array}{ll}
(A(\mathit{cfg}_{l_{\mathtt{T}}}^{l \to l'}), \mathit{Id}(l,l_{\mathtt{T}})[\psi(l)]) & \text{if } \psi(l) \text{ is defined.} \\
(l \to l_{\mathtt{T}}, \mathit{Id}(l,l_{\mathtt{T}})[\mathit{test}, \emptyset]) & \text{if } l' = l_{\mathtt{T}}. \\
(l \to l', (l \to l', \emptyset, \emptyset)) & \text{otherwise}
\end{array}\right.
$$

$$
\mathcal{S}[\![l : \mathtt{x} := \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)]\!]\psi, l' \quad =
$$

$$
\left\{\begin{array}{ll}
(l \to l', (l \to l', \emptyset, \{\mathtt{f}^{(1)'} = \mathtt{x}_1, \ldots, \mathtt{f}^{(n)'} = \mathtt{x}_n\})) & \text{if } l' = \mathtt{f}^E \\
(A(\mathit{cfg}_{\mathtt{f}^E}^{l \to l'}), \mathit{Id}(l,l')[\psi(\mathtt{f}^E)[\mathtt{x}'/\mathtt{f}^{(R)}, \mathtt{x}_1/\mathtt{f}^{(1)}, \ldots, \mathtt{x}_{\mathtt{n}}/\mathtt{f}^{(\mathtt{n})}]]) & \text{if } \psi(\mathtt{f}^E) \text{ is defined} \\
(A(\mathit{cfg}_{\mathtt{f}^E}^{l \to l'}), (l \to l', \emptyset, \{\mathtt{y}' = \mathtt{y} \mid \text{for all } \mathtt{y} \text{ except } \mathtt{x}\})) & \text{otherwise}
\end{array}\right.
$$

$$
\mathcal{S}_e[\![e]\!] = \left\{\begin{array}{ll}
e & \text{if } e \text{ is of form } c + \sum_{i=1}^{n} c_i \mathtt{x}_i \\
\top & \text{otherwise}
\end{array}\right.
$$

where

$$
\mathit{Id}(l,l')[P,R] := (l \to l', P, R \cup \{\mathtt{y}' = \mathtt{y} \mid \forall(\mathtt{x}' \bowtie k + \textstyle\sum_{i=1}^{n} k_i \mathtt{x}_i) \in R :\ \mathtt{y} \neq \mathtt{x}\})
$$

$$
A(\mathit{cfg}_{l_0}^{l \to l'}) := (L, L, l, \delta, \{l'\}) \text{ for } (L, \delta) = \mathit{cfg}(\mathcal{P})|_{\{l,l'\} \cup \{l'' \mid l_0 \to^* l''\}}
$$

Fig. 4 Transition relation analysis.

the guard as a precondition for the transition.

Definition 3.4. (Sub-Computation Bound Environment)*: A* sub-computation bound environment *is a mapping from labels to preconditions and transition predicates: ψ : Label $\to$ $\mathcal{P} \times \mathcal{R}$. The environment ψ is* safe *iff for all l:*

- *if l is the label at call $\mathtt{x} := \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ and l' is the following label then $(l \to l', P, R \cup \{\mathtt{y}' = \mathtt{y} \mid \mathtt{y} \neq \mathtt{x}\})$ is safe for $A(\mathit{cfg}_{\mathtt{f}^E}^{l \to l'})$ where $(P,R) = \psi(\mathtt{f}^E)[\mathtt{x}'/\mathtt{f}^{(R)}, \mathtt{x}_1/\mathtt{f}^{(1)}, \ldots, \mathtt{x}_{\mathtt{n}}/\mathtt{f}^{(\mathtt{n})}]$ and A is given in* **Fig. 4***,*
- *if l is the label at a loop $\mathtt{while}(\mathit{test})\{l_{\mathtt{T}} : \ldots\}$ then $(l \to l', \psi(l))$ is safe for $A(\mathit{cfg}_{l_{\mathtt{T}}}^{l \to l'})$.*

Safe bounds for procedures and loops can be computed by using linear programming as described in Ref. 6).

Example: *For now assume that we can prove that the overall effect of calling* `updatePending` *at label* 5 *is a strict decrease in the size of* `q`*, i.e., $\psi = [10 \mapsto \mathtt{q}' \leq \mathtt{q} - 1]$ is a safe sub-computation bound environment. Then $(5 \to 2, \emptyset, \{\mathtt{r}' = \mathtt{r}, \mathtt{w}' = \mathtt{w}, \mathtt{q}' \leq \mathtt{q} - 1\})$ is a safe transition relation for the transition $5 \to 2$, corresponding to the statement* `q:=updatePending(q,q)`*. Note that evaluation of the statement requires the procedure* `updatePending` *to be evaluated, so the trace automaton corresponding to the abstract transition must take the procedure body into account.*

The transition relation analysis given in Fig. 4 defines a function $\mathcal{S}[\![l : \mathit{stmt}]\!]\psi, l' = (A, v)$ such that v is a transition relation for the statement *stmt* for a given safe sub-computation bound environment ψ and v is safe for the trace automaton A.

Lemma 3.1. *For any one-step transition $l \to l'$ and any safe sub-computation bound environment ψ, the transition relation v is safe for the trace automaton A, where $(A, v) = \mathcal{S}[\![l : \mathit{stmt}_l]\!]\psi, l'$.*

Definition 3.5. (Composition)*: Given two transition relations v_1 and v_2, where $v_1 = (l_1 \to l'_1, P_1, R_1)$ and $v_2 = (l_2 \to l'_2, P_2, R_2)$, define the composition $v_1 \circ v_2 := (l_1 \to l'_2, P'', R'')$ if $l'_1 = l_2$, otherwise undefined, where P'' is a set of preconditions and R'' is a set of transition predicates. The sets P'' and R'' must be given such that P'' is a consequence of $P_1 \cup \sigma_1(R_1) \cup \sigma_2(P_2)$ and R'' is a consequence of $\sigma_1(R_1) \cup \sigma_2(R_2)$ for some function σ_1 that replaces all occurrences of primed variables with double primed variables and some function σ_2 that replaces unprimed variables with double primed variables. Suppose v_1, v_2 are safe for the trace automata A_1, A_2 respectively, then the composition $(A_1, v_1) \circ (A_2, v_2)$ is defined as $(A_1, v_1) \circ (A_2, v_2) := (A_1 \circ A_2, v_1 \circ v_2)$.*

Lemma 3.2. *The composition operator $\circ$ preserves safety: Suppose v_1, v_2 are safe for*

```
Input:  Set of trace automaton and transition relation pairs $W_0$
Output: Abstract transition program $(V, E)$
$Q := [Id]$                          // queue containing the identity transition relation
$V := \{Id\}$ ; $E := \emptyset$
while($Q \neq []$) { $v$:= dequeue($Q$)
                     foreach(($A_0, v_0) \in W_0$) { $v'$:=($v \circ v_0$)
                                                      if($v' \notin V$) { $V$:=$V \cup \{v'\}$
                                                                          enqueue($Q$, $v'$) }
                                                      $E := E \cup \{v \xrightarrow{A_0} v'\}$ } }
```

Fig. 5 Closure set computation.

the trace automata A_1, A_2 respectively and let $(A', v') = (A_1, v_1) \circ (A_2, v_2)$ be their composition, then v' is safe for A'.

Proof. Safety of the composition operator hinges on the correctness of the consequence relation $\vdash$. If P_1 is a set of preconditions for computations in A_1 then it must also be a set of preconditions for computations in $A_1 \circ A_2$. Further, if P_2 is a set of preconditions for A_2 and R_1 is a set of size relations for A_1 then any backwards extrapolation P_2'' of P_2 such that $\sigma_1(R_1) \cup \sigma_2(P_2) \vdash P_2''$ is also a precondition for $A_1 \circ A_2$. Finally, if R_1 and R_2 are sets of size relations for A_1 and A_2 respectively then any extrapolation R'' such that $\sigma_1(R_1) \cup \sigma_2(R_2) \vdash R''$ is a set of size relations for the composition $A_1 \circ A_2$. □

The composition operator can be used directly to prove fair termination by restricting the abstraction domain to a finite set of predicates and computing the least fix-point of the transition relations under the composition operator by the algorithm in **Fig. 5**, as in Ref. 11).

Theorem 3.1. *Given a composition operator $\circ$ which preserves safety, suppose W_0 is a set of pairs of trace automata and transition relations such that $(A, v) \in W_0$ implies v is safe for A and suppose (V, E) is the output of the algorithm in Fig. 5 where V is the set of transition relations and E is the set of edges between the transition relations, each labeled with a trace automaton corresponding to some transition in W_0. Then for the graph (V, E) any $v \in V$ is safe for the trace automaton: $A_v := (Labels, V \cup \bigcup_{e \in E} V_e, \bigcup_{e \in E} \delta'_e, Id, \{v\})$ where for each transition $e = v' \xrightarrow{A_e} v'' \in E$, such that $A_e = (Labels, V_e, \delta_e, v_e^0, F)$, the transfer function δ'_e is given by:*

$$\delta'_e = \{v' \xrightarrow{\epsilon} v_e^0\} \cup \{f \xrightarrow{\epsilon} v'' \mid f \in F\} \cup \delta_e.$$

Intuitively the automaton A_v is the graph (V, E) where each transition $v' \xrightarrow{A} v'' \in E$ is replaced with the automaton A plus ϵ-transitions leading from v' to the initial state in A and from all accepting states of A to v''; The initial state is the Identity transition relation (corresponding to zero length computations) and the single final state is the transition relation v.

Example: *The closure set for the program in Fig. 3 is given in* **Fig. 6**.

4. Decreasing Expressions

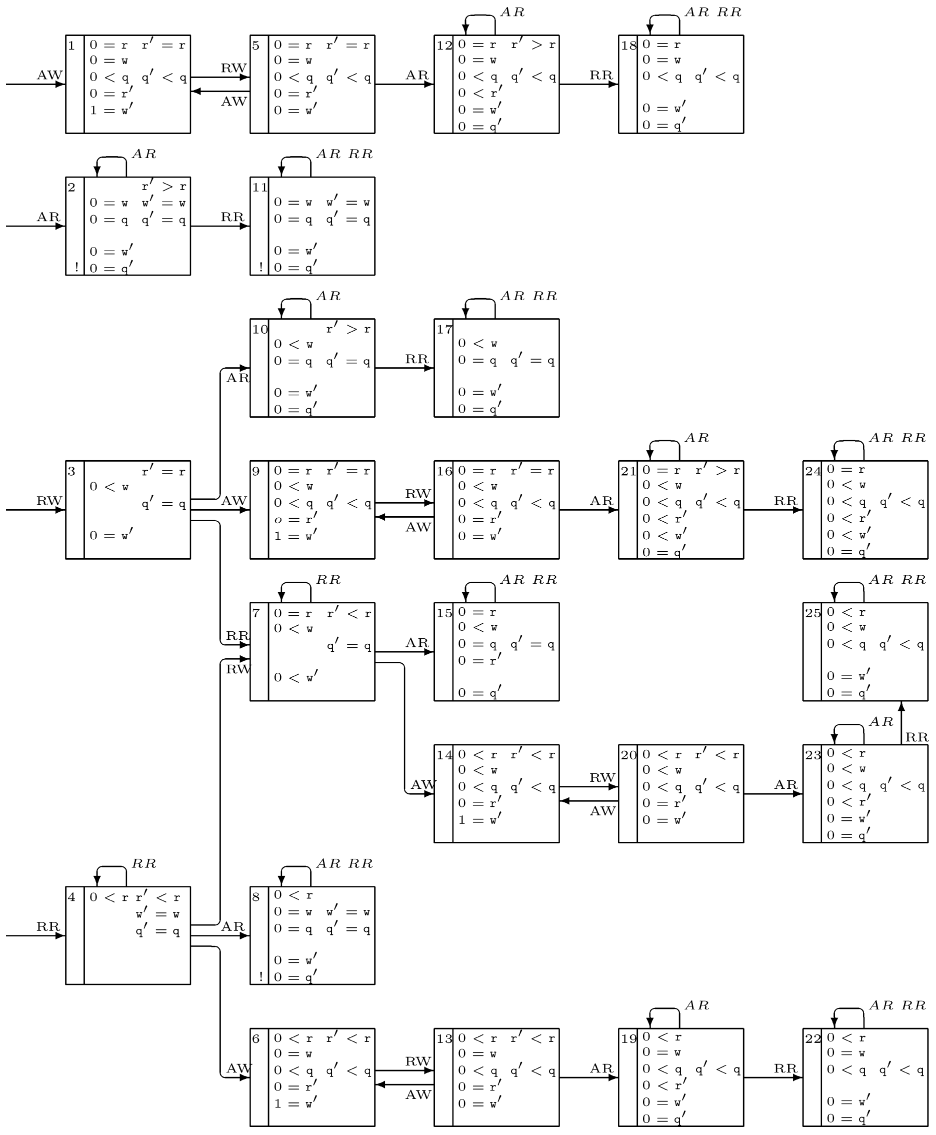
In order to prove infeasibility of infinite computations a notion of "progress" is required. In the following we use the idea of "Decreasing Expressions" of Ref. 1), in which Avery describes an automatic method for generating linear invariants for a small sub set of C that is orthogonal to our analysis. By using the same definition of well-foundedness, the method of Ref. 1) can be readily used to strengthen the preconditions.

Definition 4.1. (Decreasing Expressions)*: Given an expression $e = k + \sum_{i=1}^{n} k_i \mathtt{x}_i$ the transition relation $v = (l \rightarrow l', P, R)$ is said to* decrease *e iff*

$$\begin{array}{ll}
\forall i \in \{1 \ldots n\} : & \\
\quad k_i > 0 \textit{ implies} & (\mathtt{x}'_i < \mathtt{x}_i \in R \vee \\
 & \ \mathtt{x}'_i \le \mathtt{x}_i \in R), \\
\quad k_i < 0 \textit{ implies} & (\mathtt{x}'_i > \mathtt{x}_i \in R \vee \\
 & \ \mathtt{x}'_i \ge \mathtt{x}_i \in R), \\
\exists i \in \{1 \ldots n\} : & \\
\quad k_i > 0 \textit{ implies} & \mathtt{x}'_i < \mathtt{x}_i \in R, \\
\quad k_i < 0 \textit{ implies} & \mathtt{x}'_i > \mathtt{x}_i \in R.
\end{array}$$

A transition relation v is well-founded, *denoted well-founded(v), if there exists a precondition $0 < e \in P$ or $0 \le e \in P$ such that v decreases e.*

Theorem 4.1. *Suppose that the transition relation $v = (l \rightarrow l', P, R)$ is safe for the trace automaton A and suppose there exists a precondition $0 \bowtie e \in P$ such that v decreases $e = k + \sum_{j=1}^{n} k_j \mathtt{x}_j$, then for any computation $\pi =$*



$$
\begin{aligned}
W_0 = \{\ & (2 \to 3 \to 4 \to 5 \to 2,\ (2 \to 2, \{\ 0 = \mathtt{r},\ 0 = \mathtt{w},\ 0 < \mathtt{q}\ \}, \{\ \mathtt{r}' = \mathtt{r},\ \mathtt{q}' < \mathtt{q}\})),\\
& (2 \to 3 \to 6 \to 2,\ (2 \to 2, \{\ 0 = \mathtt{w},\ 0 = \mathtt{q}\ \}, \{\ \mathtt{r}' > \mathtt{r},\ \mathtt{w}' = \mathtt{w},\ \mathtt{q}' = \mathtt{q}\})),\\
& (2 \to 3 \to 7 \to 2,\ (2 \to 2, \{\ 0 < \mathtt{w}\ \}, \{\ \mathtt{r}' = \mathtt{r},\ \mathtt{q}' = \mathtt{q}\})),\\
& (2 \to 3 \to 8 \to 2,\ (2 \to 2, \{\ 0 < \mathtt{r}\ \}, \{\ \mathtt{r}' < \mathtt{r},\ \mathtt{w}' = \mathtt{w},\ \mathtt{q}' = \mathtt{q}\}))\ \}
\end{aligned}
$$

Fig. 6 Closure set for the example program.

$$[(l_{(1,i)}, \rho_{(1,i)})]_{i=1}^{N_1} \to \ldots \to [(l_{(k,i)}, \rho_{(k,i)})]_{i=1}^{N_k}$$

for which trace$(\pi) \in A$*:* $\rho_{(1,N_1)} \models_{exp} e \to v_1$ *and* $\rho_{(k,N_k)} \models_{exp} e \to v_k$ *imply* $v_1 > v_k$ *for some* v_1*,* v_k*.*

A liveness verification problem can be stated as a problem of fair termination, which in turn can be tested by checking that all fair graphs in the closure set cause progress towards a bound. Each abstract transition is checked for fairness by using the given fairness test.

Definition 4.2. (Fairness Test)*: Given a Büchi trace automaton* A_F *which encodes the set of fair traces a trace automaton* A *is* fair*, denoted* $\mathit{fair}_{A_F}(A)$*, iff* $A_F \cap A \neq \emptyset$*.*

Note that this definition of fairness implies that a trace automaton is fair if and only if it contains a fair trace. The approach taken in

```
Input:  Program $\mathcal{P}$ and fairness test $fair_{A_F}$.
$W_0 := \{\mathcal{S}[\![stmt_l]\!]\emptyset, l' \mid$ forall transitions $l \to l' \in Labels \times Labels\}$.
compute the closure set $(V, E)$ of $W_0$, as per Fig. 5
foreach idempotent $v \in V$:
    if( $fair_{A_F}((A_v)^\omega) \not\Rightarrow well\text{-}founded(v)$ )
        return "Requirement Potentially Violated for $A_v$"
return "Requirement is Satisfied"
```

Fig. 7 Podelski and Rybalchenko's verification algorithm.

Ref. 11) uses predicates based on *just* and *compassionate* transitions which avoids the need for intersecting Büchi automata, thus speeding up the analysis. The main algorithm of Ref. 11) can then be formulated as the algorithm in **Fig. 7**. However the analysis can be improved by analyzing while loops and mutually recursive procedures individually. In the following sections we extend the algorithm to take such sub-computations into account.

Definition 4.3. (Fair Termination)*: Given a fairness test, a program $\mathcal{P}$ is said to be fairly terminating if no infinite computation are fair.*

Theorem 4.2. *Given a fairness test $fair_{A_F}$ and a safety preserving composition operator $\circ$, suppose W_0 is a set of trace automaton and transition relation pairs such that $(A_v, v) \in W_0$ implies v is safe for A_v and suppose that any infinite trace $l_1 \to l_2 \to \ldots$ can be partitioned by a sequence $I_1, I_2, \ldots$ of indices such that for any i the subsequence $l_{I_i} \to \ldots \to l_{I_{i+1}} \in A_v$ for some v. Let (V, E) be the finite closure set generated by the algorithm in Fig. 5 using $\circ$. If all idempotent $v \in V$ are well-founded if $fair_{A_F}((A_v)^\omega)$ then the program is fairly terminating.*

Theorem 4.3. *Given a program $\mathcal{P}$ and a fairness test $fair_{A_F}$ on trace automata, if the algorithm in Fig. 7 returns "Requirement is Satisfied" then $\mathcal{P}$ is fairly terminating.*

Proof. The algorithm is almost a direct application of Theorem 4.2. Any transition $l \to l'$ in the program is abstracted by $\mathcal{S}[\![stmt_{l_i}]\!]\emptyset, l_{i+1}$ so any trace can trivially be partitioned into subsequences that are accepted by some trace automaton A where $(A, v) \in W_0$ by choosing $I = \mathbb{N}$. By Lemma 3.1 for any $(A, v) \in W_0$ is generated directly by the transition relation analysis so v is safe for A by Lemma 3.1. Since the algorithm was assumed to output "Requirement is Satisfied" the closure set computation of W_0 must terminate, so the closure set must be finite. By Lemma 3.2 the composition operator $\circ$ preserves safety, so by Theorem 4.2 the program is fairly terminating if for all idempotent $v \in V$: $fair_{A_F}((A_v)^\omega) \Rightarrow well\text{-}founded(v)$. Since this is precisely the condition tested, the algorithm outputs "Requirement is Satisfied" only if the program is fairly terminating. □

5. Infinite Computations

Any infinite trace eventually lies entirely within a subset of the program labels, i.e., the computation "gets stuck" in a sub-computation from some point onwards. Once a given sub-computation has been proven to satisfy fair termination, it need not be considered when analyzing other computations containing it for fair termination. In order to identify proper sub-computations in the program, we define a control flow graph that separates procedure calls from procedure returns.

Intuitively, the *infinite flow graph* models all possible infinite computation traces in the subject program. The graph contains a copy of the control flow-graph for each procedure definition in the program and edges from all call sites in the program to the initial statement in the called procedures. In order to model procedure returns, every statement that contains a procedure call also introduces a *summary edge* that "bypasses" the procedure call.

Definition 5.1. (Infinite Flow Graph)*: The* infinite flow graph *for the program $\mathcal{P}$ is defined as*

$$FLOW_{inf}(\mathcal{P}) = (label(\mathcal{P}), \mathcal{E}_{inf}[\![\mathcal{P}]\!])$$

where $\mathcal{E}_{inf}$ is given by **Fig. 9** *and $label(\mathcal{P})$ is the set of all labels in $\mathcal{P}$.*

The role of $l^{\mathcal{F}}$ and $l^{\mathcal{I}}$ in the case of `while` statements is to split up the loop into a choice between terminating and non-terminating computations in the loop by forcing the loop to become a strongly connected component in the infinite flow graph $FLOW_{inf}(\mathcal{P})$.

Example: *Consider the program in Fig. 3. The infinite flow graph $FLOW_{inf}(\mathcal{P})$ is given by* **Fig. 8**. *Note that the* `return` *statement at label* 17 *has no out-bound edge and that*

$$\begin{aligned}
&\mathcal{E}_{\mathit{inf}}\llbracket \mathit{def}_1,\ldots,\mathit{def}_n \rrbracket &&= \textstyle\bigcup_{i=1}^{n} \mathcal{E}_{\mathit{inf}}\llbracket \mathit{def}_i \rrbracket\\
&\mathcal{E}_{\mathit{inf}}\llbracket \mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_n)\ \{\mathit{stmts}\} \rrbracket &&= \mathcal{E}_{\mathit{inf}}\llbracket \mathit{stmts} \rrbracket \mathtt{f}^X\\
&\mathcal{E}_{\mathit{inf}}\llbracket l_1 : \mathit{stmt}_1;\ l_2{:}\mathit{stmt}_2 \rrbracket l' &&= \mathcal{E}_{\mathit{inf}}\llbracket l_1{:}\mathit{stmt}_1 \rrbracket l_2 \cup \mathcal{E}_{\mathit{inf}}\llbracket l_2{:}\mathit{stmt}_2 \rrbracket l'\\
&\mathcal{E}_{\mathit{inf}}\llbracket l{:}\ \mathtt{x} := \mathtt{e} \rrbracket l' &&= \{(l,l')\}\\
&\mathcal{E}_{\mathit{inf}}\llbracket l{:}\ \mathtt{x} := \mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_n) \rrbracket l' &&= \{(l,l'),(l,\mathtt{f}^E)\}\\
&\mathcal{E}_{\mathit{inf}}\llbracket l{:}\ \mathtt{return\ x} \rrbracket l' &&= \emptyset\\
&\mathcal{E}_{\mathit{inf}}\llbracket l^{\mathcal{F}}{:}l^{\mathcal{I}}{:}\ \mathtt{while}(\mathit{test})\ \{l_{\mathtt{T}}{:}\ \mathit{stmt}_{\mathtt{T}}\} \rrbracket l' &&= \{(l^{\mathcal{F}},l'),(l^{\mathcal{F}},l_{\mathtt{T}}),(l^{\mathcal{I}},l_{\mathtt{T}})\} \cup \mathcal{E}_{\mathit{inf}}\llbracket l_{\mathtt{T}}{:}\ \mathit{stmt}_{\mathtt{T}} \rrbracket l^{\mathcal{I}}\\
&\mathcal{E}_{\mathit{inf}}\llbracket l{:}\ \mathtt{if}(\mathit{test})\ \{l_{\mathtt{T}}{:}\ \mathit{stmt}_{\mathtt{T}}\}\ \mathtt{else}\ \{l_{\mathtt{F}}{:}\ \mathit{stmt}_{\mathtt{F}}\} \rrbracket l' =\\
&\qquad \{(l,l_{\mathtt{T}}),(l,l_{\mathtt{F}})\} \cup \mathcal{E}_{\mathit{inf}}\llbracket l_{\mathtt{T}}{:}\ \mathit{stmt}_{\mathtt{T}} \rrbracket l' \cup \mathcal{E}_{\mathit{inf}}\llbracket l_{\mathtt{F}}{:}\ \mathit{stmt}_{\mathtt{F}} \rrbracket l'\\
&\mathcal{E}_{\mathit{inf}}\llbracket l{:}\ \mathtt{choose}\ \mathit{test}_1 \rightarrow \{l_1{:}\ \mathit{stmt}_1\}\ [] \ldots []\ \mathit{test}_n \rightarrow \{l_n{:}\ \mathit{stmt}_n\} \rrbracket l' =\\
&\qquad \textstyle\bigcup_{i=1}^{n} (\{(l,l_i)\} \cup \mathcal{E}_{\mathit{inf}}\llbracket l_i{:}\ \mathit{stmt}_i \rrbracket l')
\end{aligned}$$

Fig. 9 Computing the flow-graph for infinite computation traces.

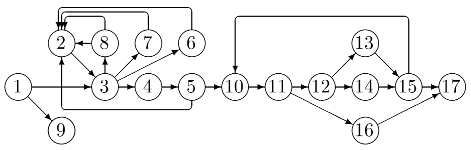


Fig. 8 $\mathit{FLOW}_{\mathit{inf}}(\mathcal{P})$ for the example program.

the while *loop at labels 1 and 2 is implemented as a choice between infinite computation in the while loop and computations that eventually reach label 9. The strongly connected components are thus* $\{2,3,4,5,6,7,8\}$ *and* $\{10,11,12,13,14,15\}$ *modeling infinite computations of the while loop in* `dbAccess` *and* `updatePending`*, respectively.*

The infinite flow graph captures all infinite computation traces in the following sense:

Lemma 5.1. *For any infinite computation* π *with* $\mathit{trace}(\pi) = \tau$ *there exists an infinite trace* τ' *in* $\mathit{FLOW}_{\mathit{inf}}(\mathcal{P})$ *such that* τ' *is a subsequence of* τ*.*

The strongly connected components (SCC) in the infinite flow graph identify precisely all possible infinite computations: For any infinite computation there exists an SCC such that the computation trace only visits labels in the SCC from some point onwards, modulo sub-computations. A strongly connected component in the infinite flow graph is either a single `while` loop or contains a set of mutually recursive procedures. A `while` loop that does not contain procedure calls cannot reach any other initial labels than that of the `while` loop (modulo sub-computations), since `while` loops are well nested.

6. Inter-Recursion Traces

Recursion is only possible using procedure calls or `while` loops, so the number of transition sequences that do not contain procedure calls or `while` loops is bounded and the composition of the corresponding transition relations can be computed without restrictions on the domain of the transition relations. Such traces correspond to basic blocks modulo branching for conditionals. The closure set computation can be faster if the composition of the transition relations over the "inter-recursion traces" have been computed prior to the closure set computation. Another advantage is that by working with a more precise abstraction domain (certain types of) non-monotone progress can be handled.

Definition 6.1. (Cut-Point Labels)*: Given a strongly connected component* c *in the infinite flow graph* $\mathit{FLOW}_{\mathit{inf}}(\mathcal{P})$*, define the set of* cut-point labels $L_{\mathit{cpt}}(c)$ *in* c *such that* $l \in L_{\mathit{cpt}}(c)$ *iff*

- l *is the label at a while loop, or*
- $l = \mathtt{f}^E$ *for some procedure* `f` *belonging to the set of mutually recursive procedures in* c*.*

The set $L_{\mathit{cpt}}(c)$ captures $\mathit{FLOW}_{\mathit{inf}}(\mathcal{P})$ over c in the following sense:

Lemma 6.1. *For any SCC* c *and any infinite computation* π *for which* $\mathit{trace}(\pi) = l_1 \rightarrow l_2 \rightarrow \ldots$ *and* $l_i \in c$ *from some point onwards* ($\exists i_0 \in I\!N : \forall i > i_0 : l_i \in c$)*, there exists an infinite set of indices* $I \subseteq I\!N$ *such that* $l_i \in L_{\mathit{cpt}}(c)$ *for all* $i \in I$*.*

The length of any sequence of labels $l_1, l_2, \ldots, l_n$ in c where l_1 and l_n are cut-point labels and l_i is non-cut-point for $1 < i < n$ is bounded for any program, since iteration is not possible without while loops or recursive procedure calls. Thus it is possible to compute the composition of the corresponding transition relations in finite time using the infinite abstract domain V_{inf}, since convergence is ensured by the fact that the length of the sequence is bounded.

Definition 6.2. (Inter-Recursion Trace)*: For a given set of cut-point labels* $L_{\mathit{cpt}}(c)$*, an* inter-recursion trace *is a trace* $l_1 \rightarrow \ldots \rightarrow l_n$ *in* c

where l_1 and l_n are cut-point labels and l_i is non-cut-point for all $2 \leq i \leq n-1$.

Definition 6.3. (Composition)*: Given a trace $l_1 \to \ldots \to l_n$ define:*

$$\begin{aligned} &\mathcal{S}[\![l_1 \to \ldots \to l_n]\!]\psi \\ &\quad := \quad (\mathcal{S}[\![l_1 : stmt_{l_1}]\!]\psi, l_2) \circ \ldots \\ &\qquad \circ (\mathcal{S}[\![l_{n-1} : stmt_{l_{n-1}}]\!]\psi, l_n) \end{aligned}$$

Example: *Consider again the program in Fig. 3: The set of cut-point labels for the SCC $c = \{2,3,4,5,6,7,8\}$ of $FLOW_{inf}(\mathcal{P})$ is given by $L_{cpt}(c) = \{2\}$ since label 2 occurs at a* `while` *loop. The set of inter-recursion traces is then:*

$$\begin{aligned} AW &:= 2 \to 3 \to 4 \to 5 \to 2, \\ AR &:= 2 \to 3 \to 6 \to 2, \\ RW &:= 2 \to 3 \to 7 \to 2, \\ RR &:= 2 \to 3 \to 8 \to 2. \end{aligned}$$

The corresponding transition relations are given by:

$$\begin{aligned} &(AW, (2 \to 2, \{0 = \mathtt{r}, 0 = \mathtt{w}, 0 < \mathtt{q}\}, \\ &\qquad\qquad \{\mathtt{r}' = \mathtt{r}, \mathtt{q}' \leq \mathtt{q} - 1\})), \\ &(AR, (2 \to 2, \{0 = \mathtt{w}, 0 = \mathtt{q}\}, \\ &\qquad\qquad \{\mathtt{r}' = \mathtt{r} + 1, \mathtt{w}' = \mathtt{w}, \mathtt{q}' = \mathtt{q}\})), \\ &(RW, (2 \to 2, \{0 < \mathtt{w}\}, \\ &\qquad\qquad \{\mathtt{r}' = \mathtt{r}, \mathtt{q}' = \mathtt{q}\})), \\ &(RR, (2 \to 2, \{0 < \mathtt{r}\}, \\ &\qquad\qquad \{\mathtt{r}' = \mathtt{r} - 1, \mathtt{w}' = \mathtt{w}, \mathtt{q}' = \mathtt{q}\})). \end{aligned}$$

7. Convergent Transition Relations

In order to ensure convergence of the closure set computation, it is necessary to restrict the domain of the transition predicates. In particular, the domain of transition predicates is restricted to only contain predicates over a single unprimed variable and single primed variable.

Definition 7.1. (Convergent Transition Relations)*: Let the* finite transition relation domain *V_{con} be defined as the set of transition relations of form:*

- $\mathtt{y}' < \mathtt{x}$ *or* $\mathtt{y}' \leq \mathtt{x}$ *for any* $\mathtt{y}'$ *and* $\mathtt{x}$ (*decrease, non-increase*),
- $\mathtt{y}' = \mathtt{x}$ *for any* $\mathtt{y}'$ *and* $\mathtt{x}$ (*equality*),
- $\mathtt{y}' > \mathtt{x}$ *or* $\mathtt{y}' \geq \mathtt{x}$ *for any* $\mathtt{y}'$ *and* $\mathtt{x}$ (*increase, non-decrease*),
- $0 < k + \sum_{i=1}^{n} k_i \mathtt{x}_i$ *for any* $k, k_1, \ldots, k_n$ (*precondition*),
- $0 = k + \sum_{i=1}^{n} k_i \mathtt{x}_i$ *for any* $k, k_1, \ldots, k_n$ (*precondition*),

let $\alpha_{con} : V_{inf} \to V_{con}$ be an abstraction operator such that $v \vdash \alpha_{con}(v)$ for any $v \in V_{inf}$ and let $\overline{\alpha}_{con}$ be the extension to trace automaton & transition relation pairs: $\overline{\alpha}_{con}((A, v)) = (A, \alpha_{con}(v))$. Finally let $\circ_{con}$ be a safety preserving composition operator which does not introduce new preconditions modulo permutations of variables.

We make the following observations on V_{con} and $\circ_{con}$:

Lemma 7.1. *If $V \subseteq V_{con}$ is a finite set of transition relations then the closure set $\{v_1 \circ_{con} \ldots \circ_{con} v_n \mid \forall i \in [1, n];\ v_i \in V\}$ under the composition operator is also finite.*

Lemma 7.2. *If the transition relation v is safe for the trace automaton A then $\alpha_{con}(v)$ is also safe for A.*

Proof. Follows trivially since $v \vdash \alpha_{con}(v)$ by definition. □

Lemma 7.1 ensures that the closure set will be finite (and thus computable) and Lemma 7.2 ensures safety of projection into the domain V_{con}. The closure set can be computed using the simple work list algorithm given in Fig. 5.

The above operators are not uniquely defined. In our experiments it suffices to use the following operators, though more accurate operators can be implemented for increased precision. The abstraction operator α_{con} is implemented by replacing transition predicates of form $\mathtt{y}' \bowtie k{+}\mathtt{x}$ with $\mathtt{y}' \bowtie' \mathtt{x}$ (if possible for some $\bowtie'$) and discarding any other transition predicates. Given two transition relations $v := (l \to l', P, R)$ and $v' := (l' \to l'', P', R)$ the convergent composition operator $\circ_{con}$ is implemented by computing $(l \to l'', P'', R'') := \alpha_{con}(v \circ v')$ and replacing P'' with P in order to avoid introducing any new preconditions.

8. Verification Algorithm

The overall structure of the sub-computation based algorithm, given in **Fig. 10**, is to first identify the sub-computations of the subject program and process each in bottom-up order. Due to the need for the closure set computation to converge, size-change termination can only handle non-monotone progress by tracking changes in size up to some arbitrary constant, or by using a widening operator. Non-monotone progress is handled in a natural way over inter-recursion traces by using two levels of abstraction: one for the inter-recursion traces and one more limited abstraction that ensures convergence of the closure set computation. A set of transition relations is extracted using the more precise abstractions which then are composed to yield abstractions over inter-recursion traces. The closure set is computed for the projection of the inter-recursion traces into the convergent abstract domain and the well-founded transition relations are identified. A potential

```
Input:   Program $\mathcal{P}$ and fairness test $fair$.
decompose $FLOW_{inf}(Pgm)$ into a set of strongly connected components $C$
$\psi := \emptyset$
foreach $c \in C$ in bottom-up order:
    $W_0 := \{\overline{\alpha}_{con}(\mathcal{S}[\![l_1 \to \ldots \to l_n]\!]\psi) \mid$ for all inter-recursion traces $(l_1 \to \ldots \to l_n) \in c\}$
    compute the closure set $(V, E)$ of $W_0$, as per Fig. 5 using $\circ_{con}$
    foreach idempotent $v \in V$:
        if( $fair_{A_F}((A_v)^\omega) \not\Rightarrow well\text{-}founded(v)$)
            return "Requirement Potentially Violated for $A_v$"
    compute safe bounds $\psi_0$ for $c$.
    $\psi := \psi \cup \psi_0$.
return "Requirement is Satisfied"
```

Fig. 10 Sub-computation based verification algorithm.

```
1: while(x>0) {
     2: if(a>0) {3:  a=a+1}
        else    {4:  a=a-1}
     5: if(b>0) {6:  b=b+1}
        else    {7:  b=b-1}
     8: if(c>0) {9:  c=c+1}
        else    {10:  c=c-1}
     11: x=x-1; }
```

Fig. 11 Program with exponentially many traces.

$\tau_1 = 1 \to 2 \to 3 \to 5 \to 6 \to 8 \to 9 \to 11 \to 1$
$\tau_2 = 2 \to 3 \to 5 \to 6 \to 8 \to 9 \to 11 \to 1 \to 2$
$\vdots$
$\tau_{11} = 11 \to 1 \to 2 \to 3 \to 5 \to 6 \to 8 \to 9 \to 11$

Fig. 12 Traces capturing the same loop.

violation is reported if any fair transition relation in the closure set is not well-founded. If no violations were detected, a safe abstraction is computed for the sub-computation just analyzed and added to the sub-computation bound environment for use when analyzing other computations.

Theorem 8.1. *Given a program $\mathcal{P}$ and a fairness test fair on trace automata, if the algorithm in Fig. 10 returns "Requirement is Satisfied" then $\mathcal{P}$ is fairly terminating.*

8.1 Expected Efficiency

The additional preprocessing steps of the sub-computation based algorithm can be implemented in polynomial time, with the exception of composition over inter-recursion traces. Consider the program in **Fig. 11**.

Since label 1 is a cut-point label, the algorithm computes the set of abstractions for different traces from label 1 to label 1, which is exponential in the number of transitions. However note that Podelski and Rybalchenko's algorithm in Fig. 7 would simply compute the composition of these traces in the closure set computation instead, so the sub-computation based algorithm is not less efficient. By computing the composition over inter-recursion traces prior to the closure set computation, the sub-computation based algorithm avoids computing abstractions for redundant loops from a label inside the `while`-loop back to itself. For instance the traces in **Fig. 12** all capture the same loop through the `while`-loop, so it suffices to consider just one of them.

Size-change termination has been proven to be PSPACE complete[8)] and since transition predicate abstraction uses the size-change termination principle to test infeasibility of infinite computations, the verification algorithm is also (at least) in PSPACE. The preprocessing steps can be computed in polynomial time (with the above caveat), so the preprocessing overhead becomes insignificant for sufficiently large programs.

In general significant improvements in efficiency can be expected for programs with long inter-recursion traces and highly hierarchical iteration/recursion. Consider the programs in **Fig. 13**. In both programs the simple algorithm would take a whole-program approach by extracting transition relations for all transitions in the program and computing the closure set once for a large number of transitions. In contrast the sub-computation based algorithm would isolate the sub-computations and analyze each in turn, thus computing many closure sets, each for a small number of transitions. Since the closure set computation scales poorly in the number of transition relations, the latter approach will be more efficient.

8.2 Experimental Results

The verification algorithm in Fig. 10 has been implemented in Haskell. **Table 1**

```
1: while(x>0) {                      f(x) { 1:  if(x<=0) {2:  } else {3:  f(x-1)+g(x) }
  2:  while(a>0) {3:  a=a-1 }               4:  return x; }
  4:  while(b>0) {5:  b=b-1 }        g(y) { 5:  if(y<=0) {6:  } else {7:  g(y-1)+h(y) }
  6:  while(c>0) {7:  c=c-1 }               8:  return y; }
  8:  x=x-1;                         h(z) { 9:  if(z<=0) {10:  } else {11:  h(z-1)+1 }
  }                                         12:  return z; }
```

Fig. 13 Nested `while`-loops and recursive procedures.

Table 1 Experimental results.

Algorithm	Transition Relations			
	Extracted	Inter-recursion	Closure set	¬well-founded
Simple	22	-	971	17
Sub-Comp.	20	6	27	3
{2, . . . , 8}	10	4	25	3
{10, . . . , 15}	10	2	2	0

shows the results running both Podelski and Rybalchenko's algorithm in Fig. 7, as well as the sub-computation based algorithm, on the running example in Fig. 3. The columns summarize the number of transition relations produced by the analyzer at each stage in verification algorithms.

- "Extracted": The number of transition relations extracted for single-step transitions in the program.
- "Inter-recursion": The number of transition relations generated by composing the "extracted" transitions corresponding to a given inter-recursion trace (only relevant for the sub-computation based analysis[☆]).
- "Closure set": The number of transition relations in the closure set.
- "¬well-founded": The number of transitions in the closure set that are not well-founded and thus must be tested for fairness.

While both algorithms initially extract about 20 transitions relations, the closure set computed by the simple algorithm is considerably larger than for the sub-computation based algorithm, which also leads to a larger number of transition relations that must be tested against the liveness requirement. This demonstrates the efficiency gain obtained by the sub-computation based algorithm. Note that verification of the running example requires the algorithm to prove that the `updatePending` procedure strictly decreases the size of the first argument — a case which only the sub-computation based algorithm is able to handle. Increased precision can relatively easily be demonstrated with simple programs, but unfortunately time constraints did not allow for additional experiments.

[☆] Denoted by "-" in the table.

9. Extension to Parallel Programs

One method for analyzing liveness properties of parallel programs is to compute the parallel composition of the processes in the program and then apply Podelski's algorithm to the resulting program. Unfortunately this approach is not well-suited for handling recursive procedures and can cause a combinatorial increase in size of the resulting model and thus the time needed for verification. In this section we outline an extension of the sub-computation based approach to a language with a fixed number of parallel processes and synchronous (blocking) communication via statically named channels. While the concept of "sub-computations" is clear in a sequential language, the interpretation is less clear for a parallel language since processes cannot in general be analyzed in isolation. In order to extend the methods of the previous sections to a parallel language, one must define "sub-computations" for parallel programs and provide a means of extracting transition relations for them. Our approach is to only analyze interleavings of sub-computations (in the sense of Section 5) that can communicate with each other and thus can potentially affect each other's execution. In this way unnecessary combinatorial increase of the size of the model can be avoided, e.g., sub-computations that do not communicate can be analyzed in isolation.

$$
\begin{array}{lcll}
\mathcal{P} & ::= & \mathit{prc}_1 \ldots \mathit{prc}_k,\ \mathit{def}_1 \ldots \mathit{def}_n & \text{(Program)} \\
\mathit{prc} & ::= & \texttt{process p} \ = \ \texttt{f}(\texttt{v}_1, \ldots, \texttt{v}_n); & \text{(Process)} \\
\mathit{def} & ::= & \texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_k)\{\mathit{stmts};\ \ l : \ \texttt{return x}_r\} & \text{(Procedure)} \\
 & & & \\
\mathit{stmts} & ::= & \epsilon \mid \mathit{stmt};\ \mathit{stmts} & \text{(Statement Block)} \\
\mathit{stmt} & ::= & \texttt{send c} \leftarrow e \mid \texttt{receive c} \rightarrow \texttt{x} & \text{(Statement)} \\
 & \mid & l : \mathit{stmt} \ \mid \ \texttt{x} := e \ \mid \ \texttt{x} := \texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_n) & \\
 & \mid & \texttt{if}(\mathit{test})\ \{\mathit{stmts}_t\}\ \texttt{else}\ \{\mathit{stmts}_f\} & \\
 & \mid & \texttt{while}(\mathit{test})\{\mathit{stmts}\} & \\
 & \mid & \texttt{choose}\ \mathit{test}_1 \rightarrow \{\mathit{stmts}_1\}\ [] \ \ldots \ []\ \mathit{test}_n \rightarrow \{\mathit{stmts}_n\} & \\
 & & & \\
e & ::= & v \ \mid \ \texttt{x} \ \mid \ e_1\ \mathit{op}\ e_2 & \text{(Integer Expression)} \\
 & & & \\
\mathit{test} & ::= & \texttt{T} \mid \texttt{F} & \text{(Boolean Constant)} \\
 & \mid & \mathit{test}_1 \wedge \mathit{test}_2 \mid 0 = e \mid 0 \leq e \mid 0 < e & \text{(Boolean Expression)} \\
 & & & \\
\texttt{p} & : & \text{Process identifiers } (\texttt{p}, \texttt{q}, \ldots) & \\
\texttt{c} & : & \text{Channel identifiers } (\texttt{c}, \texttt{d}, \ldots) & \\
\texttt{f} & : & \text{Procedure identifiers } (\texttt{f}, \texttt{g}, \ldots) & \\
\texttt{x} & : & \text{Variable identifiers } (\mathit{Variables} = \{\ \texttt{x}, \texttt{y}, \ldots\}) & \\
l & : & \text{Labels identifiers } (\mathit{Labels} = \{\ 1, 2, \ldots\}) & \\
v & : & \text{Integers } (\mathbb{Z}) & \\
\mathit{op} & : & \text{Binary Operators } (\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}) &
\end{array}
$$

Fig. 14 Parallel program syntax.

$$
\text{[EXEC]} \ \frac{s_i \rightarrow s'_i}{(s_1, \ldots, s_k) \Rrightarrow (s'_1, \ldots, s'_k)} \ \left(\begin{array}{l} \text{where } s'_m = s_m \text{ for all } m \neq i \\ \quad \text{for some } i \end{array} \right)
$$

$$
\text{[COM]} \ \frac{s''_j : (\texttt{x} = e; B_j, \rho_j) \rightarrow s'_j}{(s_1, \ldots, s_k) \Rrightarrow (s'_1, \ldots, s'_k)} \ \left(\begin{array}{l} \text{where for some } i, j \ s'_m = s_m \text{ for all } m \neq i, j \text{ and} \\ \quad s_i = s''_i : (\texttt{send c} \leftarrow e; B_i, \rho_i) \\ \quad s_j = s''_j : (\texttt{receive c} \rightarrow \texttt{x}; B_j, \rho_j) \\ \quad s'_i = s''_i : (B_i, \rho_i) \end{array} \right)
$$

Fig. 15 Parallel program semantics.

10. Parallel Language

The syntax for the parallel language given in **Fig. 14** is an extension of the syntax defined in Fig. 1. A program consists of a process definition for each process followed by a number of procedure definitions and the definition of a statement is extended with two communication primitives: `send` and `receive`, that communicate via statically named channels.

The semantics in **Fig. 15** is similarly extended to accommodate the parallelism and new statements. The extended semantics introduces a layer, on top of the non-parallel semantics, that governs the process scheduling. Any process not at `send` or `receive` statement can be scheduled for execution at any time. Processes at `send` or `receive` statements can only be scheduled for execution if another process can be scheduled at a corresponding `receive` or `send` statement on the same communications channel. In the following we will in general use vector notation for concepts in the parallel language that are analogous to those of the non-parallel language, e.g., $\vec{l}$ denotes a tuple of labels $\vec{l} = (l_1, \ldots, l_k)$ and $(\vec{l})_i$ denotes the i^{th} element $(\vec{l})_i = l_i$. The notation $[l_i]_{i=1}^k$ will be used as a shorthand notation for the tuple $(l_1, \ldots, l_k)$. In order to precisely state what some property refers to, properties of non-parallel programs are considered to be *process* properties as opposed to *program* properties, e.g., c denotes a *process* sub-computation while $\vec{c}$ denotes a *program* sub-computation.

(1) A *program state* $\vec{s} \in \vec{S}$ for a program with k processes is a tuple of program stacks $\vec{s} = (s_1, \ldots, s_k)$.

(2) The computational semantics of the language in Fig. 15 specifies a transition re-

```
process p = f(1000);
process q = g(1);
f(x) {
  1:  while(x ≥ 0) {
        2: send d ← x;
        3: x := updatePending(x, x) }
  4:  return x }
updatePending( q, i ) {
  5:  i := i − 1;
  6:  if(0 < i) {
        7: choose    T → { 8: q := q − 1; }      // request timed out
                 []  T → { 9: };               // request still pending
        10: q := updatePending(q, i); }
  11: else { q := q − 1; }
  12: return q }

g(y) {
  13: while(y ≠ 0) {
    14: y := recvPos() }
  15: return y }
recvPos() {
  16: receive d → z;
  17: if(z ≤ 0) { 18: z := −z }
      else      { 19: }
  20: return z }
```

Fig. 16 Parallel example program.

lation on program states, $\vec{s} \Rrightarrow \vec{s'}$.

(3) An *initial state* is a program state consisting only of initial stacks.

(4) A *computation* is a finite or infinite sequence of program states $\vec{\pi} = \vec{s_1} \to \vec{s_2} \to \ldots$ such that $\vec{s_1}$ is an initial state and $\vec{s_i} \Rrightarrow \vec{s_{i+1}}$ for all $i \geq 1$.

(5) A *program location* for a program with k processes is a k-tuple of labels: $\vec{l} = (l_1, \ldots, l_k)$. The location of a given state $\vec{s}$ is given by: $location(s) := (label(s_1), \ldots, label(s_k))$

(6) A *program trace* is a finite or infinite sequence of program locations $\vec{\tau} = \vec{l_1} \to \vec{l_2} \to \ldots$ such that there exists a computation realizing the trace: $\vec{\pi} = s_1 \to s_2 \to \ldots$ where $location(s_i) = \vec{l_i}$ for all $i \geq 0$. The program trace $\vec{\tau}$ of $\vec{\pi}$ is denoted $trace(\vec{\pi}) = \vec{\tau}$.

Example: *The parallel program in* **Fig. 16** *defines a "producer" process* p *which sends a decreasing sequence of numbers, terminated by a zero, on a channel* d*. The counter variable* x *for the* while *loop in* p *is decreased using the* updatePending *function from the running example for the non-parallel algorithm. The sequence is received by a "consumer" process* q *which loops until the terminating zero is encountered. A minor complication is that* q *uses the procedure* recvPos *to receive data on channel* d *and perform some conditioning of the data. Termination of procedure* g *thus depends both on the termination of process* p *and the (communication) side effects of the procedure* recvPos*. In the following we will use the example in Fig. 16 as a running example to illustrate the techniques used for verifying parallel programs.*

11. Program Sub-Computations

In order to identify sub-computations in a parallel program, a distinction must be made between *program sub-computations* and *process sub-computations*. Process sub-computations arise from analyzing processes in isolation while disregarding communication side effects. They can be computed similarly to the non-parallel case using the additional definitions in **Fig. 17**, the only difference being that we now have two additional statements `send` and `receive`. The infinite flow graph $FLOW_{inf}(\mathcal{P})$ is defined the same way as for the non-parallel analysis, except that we replace the definition of $\mathcal{E}_{inf}$.

Definition 11.1. (Infinite Flow Graph)*: The* infinite flow graph *for the program $\mathcal{P}$ is defined as $FLOW_{inf}(\mathcal{P}) = (label(\mathcal{P}), \mathcal{E}_{inf}[\![\mathcal{P}]\!])$ where $\mathcal{E}_{inf}$ is given by Fig. 17 and $label(\mathcal{P})$ is the set of all labels in $\mathcal{P}$.*

Example: *Consider again the program in Fig. 16. The infinite flow graph, which can be computed using the new definition of $FLOW_{inf}(\mathcal{P})$, contains three process sub-computations (strongly connected components):*

- *the* while *loop in* p*: $c_1 = \{1, 2, 3\}$,*
- *the recursive procedure* updatePending*: $c_2 = \{5, 6, 7, 8, 9, 10\}$ and*
- *the* while *loop in* q*: $c_3 = \{13, 14\}$.*

Lemma 11.1. *Given a program with k processes, for any infinite computation $\vec{\pi}$ for which $trace(\vec{\pi}) = \vec{\tau}$, for all $i \in \{1, \ldots, k\}$ there exists an infinite trace τ' in $FLOW_{inf}(\mathcal{P})$ such that τ' is a sub-sequence of $(\vec{\tau})_i$.*

We now extend the idea of *process* sub-

$$\begin{aligned}
\mathcal{E}_{inf}[\![l\!: \texttt{send c} \leftarrow e]\!]l' &= \{(l,l')\}\\
\mathcal{E}_{inf}[\![l\!: \texttt{receive c} \rightarrow \texttt{x}]\!]l' &= \{(l,l')\}
\end{aligned}$$

Fig. 17 Additional definition for computing the infinite flow graph.

computations, to one of *program* sub-computations. Intuitively, a program sub-computation is a set of process sub-computations from different processes that may affect each other's execution.

A given process sub-computation c becomes *blocked* if it tries to send or receive data on a given channel, but no other process executes a corresponding `receive` or `send` statement for the same channel. In order to identify such process sub-computations we define a set of channels that *may* cause a process sub-computation c to block for `send` and `receive` statements.

Definition 11.2. (Blocking Channels)*: The infinite flow transition relation $\mathcal{E}_{inf}$ on labels induces a relation on process sub-computations, so let $\rightsquigarrow^*$ denote its reflexive and transitive closure. For a given process sub-computation c define the* sender blocking channels*, $send(c)$, and the* receiver blocking channels*, $receive(c)$, by*

$$\begin{aligned}
send(c) &:= \{d \mid \exists c' :\ c \rightsquigarrow^* c' \ \wedge\ \exists l \in c' :\\
&\qquad (stmt_l = \texttt{send}\ d \leftarrow e)\},\\
receive(c) &:= \{d \mid \exists c' :\ c \rightsquigarrow^* c' \ \wedge\ \exists l \in c' :\\
&\qquad (stmt_l = \texttt{receive}\ d \rightarrow \texttt{x})\}.
\end{aligned}$$

The *communication flow* between process sub-computations approximate which process sub-computations might communicate with each other and thus might affects each other's execution. The set of program sub-computations can then be computed via the *communication flow graph.*

Definition 11.3. (Communication Flow)*: Suppose $p_1,\ldots,p_k$ are processes and for all i let $c_{(i,1)},\ldots,c_{(i,m_i)}$ be the sub-computations in process p_i then define the* communication relation *$c_{(i_1,j_1)} \rightleftharpoons c_{(i_2,j_2)}$ by*

$$\begin{aligned}
&\{\{c_{(i_1,j_1)}, c_{(i_2,j_2)}\} \mid i_1 \neq i_2 \wedge\\
&\quad (send(c_{(i_1,j_1)}) \cap receive(c_{(i_2,j_2)}) \neq \emptyset \vee\\
&\quad receive(c_{(i_1,j_1)}) \cap send(c_{(i_2,j_2)}) \neq \emptyset)\ \}
\end{aligned}$$

and let $\rightleftharpoons^*$ denote its reflexive, symmetric and transitive closure.

Note that communication is, due to the synchronous semantics, bidirectional in nature: process p sends data to process q and q unblocks process p.

A program sub-computation, is a set of process sub-computations, each representing a process in the program, such that any process sub-computation can reach any other via the communication relation $\rightleftharpoons^*$.

Definition 11.4. (Program Sub-Comp.)*: The set of* program sub-computations *for a program with processes $p_1,\ldots,p_k$ is given by*

$$\begin{aligned}
\{\ &\{c_1,\ldots,c_n\} \mid\\
&\forall i \leq n :\ (c_i \in C_{proc(c_i)}) \wedge\\
&\qquad (\forall j \leq n : i \neq j \Rightarrow (c_i \rightleftharpoons^* c_j \wedge\\
&\qquad\qquad proc(c_i) \neq proc(c_j)))\}
\end{aligned}$$

where C_j is the set of process sub-computations for process p_j for all $j = 1,\ldots,k$ and proc is a one-to-one mapping of process sub-computations to processes indices: $(proc(c_i)) \in \{1,\ldots,k\}$.

Example: *Consider the three process sub-computations $c_1 = \{1,2,3\}, c_2 = \{5,6,7,8,9,10\}, c_3 = \{13,14\}$ for the program in Fig. 16. The blocking channels for c_1 and c_3 are given by $send(c_1) = receive(c_3) = \{\texttt{d}\}$ since c_1 sends data on channel* `d` *at label 2 and c_3 calls the procedure* `recvPos` *which receives data on channel* `d` *at label 16. By the definition of the communications flow this implies that $c_1 \rightleftharpoons^* c_3$. The sub-computation c_2 does not contain any* `send` *or* `receive` *statements, so the two sets of blocking channels are the empty set: $send(c_2) = receive(c_2) = \emptyset$.*

Since $c_1 \rightleftharpoons^* c_3$ and $proc(c_1) \neq proc(c_3)$ we find that $\vec{c_1} = \{c_1, c_3\}$ is a program sub-computation. For the remaining process sub-computation, c_2, neither $send(c_2)$ nor $receive(c_2)$ can overlap with the blocking channels of another process sub-computation, so c_2 gives rise to a program sub-computation by itself: $\vec{c_2} = \{c_2\}$.

Lemma 11.2. *Let $\vec{C}$ be the set of program sub-computations for a given program with processes $p_1,\ldots,p_k$. If π is an infinite computation with trace $\tau = \vec{l_1} \rightarrow \vec{l_2} \rightarrow \ldots$ then for all processes $i \in proc(\vec{C})$ there exists a program sub-computation $\vec{c} \in \vec{C}$ such that the trace τ visits a label in $\vec{c}$ infinitely often.*

12. State Predicate Models

In order to extract transition relations for a program sub-computation, consisting of one or more process sub-computations, we use *state predicate abstraction* on the parallel program to obtain a *state predicate model* that can be used to guide the extraction of transition relations. A state predicate model is a graph where nodes approximate the state space of the program and the edges are derived from the semantics of the subject language. State predicate abstrac-

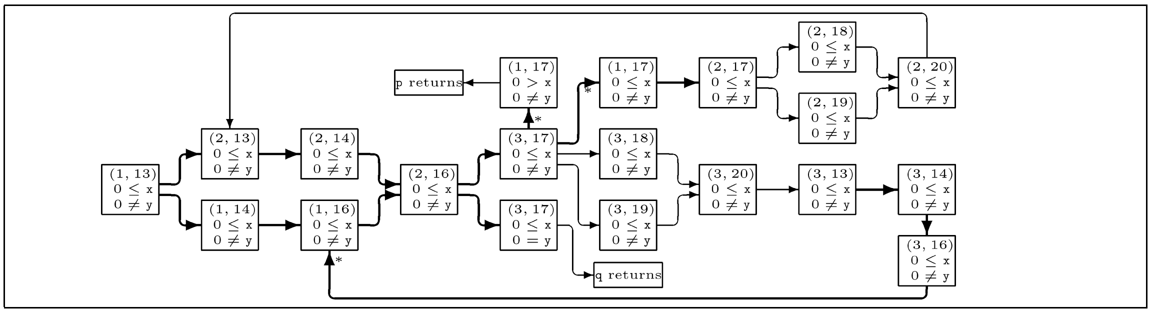


Fig. 18 State predicate model for the parallel program.

tion is a well-known tool in the model-checking community which can provide accurate models of communication protocols since they are useful for capturing communication side effects. Though transition predicate abstraction can describe even more complex behaviors (e.g., termination due to decrease in values passed between processes) we argue that state predicate abstraction is sufficient to describe the inter-process communication of most programs.

Definition 12.1. (State Predicate Model)*: For a given program $\mathcal{P}$ of k processes a* state predicate model *is a graph $M_{\mathcal{P}} = (V_{M_{\mathcal{P}}}, R_{M_{\mathcal{P}}})$ where the nodes $V_{M_{\mathcal{P}}} \in Labels^k \times \mathcal{P}(test)$*[☆] *approximate the reachable state of $\mathcal{P}$ in the following sense: Let γ be a concretization function mapping abstract states to concrete states*

$$\begin{aligned}\gamma(\vec{l}, P) := \{\vec{s} \mid\ & location(\vec{s}) = \vec{l} \wedge \\ & \forall p \in P:\ sto(\vec{s}) \models_{exp} p \to \mathtt{T}\}\end{aligned}$$

where $sto(\vec{s})$ is the store of the configuration at the top of $\vec{s}$. The graph $M_{\mathcal{P}}$ is a state predicate model for $\mathcal{P}$ under γ iff for every computations $\vec{\pi} = \vec{s_1} \to \vec{s_2} \to \ldots$ there exists a path $\vec{\pi'} = (\vec{l_1}, P_1) \to (\vec{l_2}, P_2) \to \ldots$ in $M_{\mathcal{P}}$ such that $\vec{s_i} \in \gamma(\vec{l_i}, P_i)$.

The state predicate model M' for program sub-computation $\vec{c}$ *is a projection of the state predicate model M onto the processes of $\vec{c}$: M' :=*

$$\begin{aligned}(\delta_{\vec{c}}(V_{M_{\mathcal{P}}}),\ \ \{&\delta_{\vec{c}}(v) \to \delta_{\vec{c}}(v') \mid \\ & v \to v' \in R_{M_{\mathcal{P}}}\})\end{aligned}$$

where

$$\begin{aligned}\delta_{\vec{c}}((\vec{l}, P)) := (&[(\vec{l})_i]_{i \in proc(\vec{c})}, \\ & \{p \mid p \in P, p \in exp(proc(\vec{c}))\}),\end{aligned}$$

$proc([c_i]_{i=1}^n) = \bigcup_{i=1}^n proc(c_i)$ and $p \in exp(I)$ iff all variables x *in p belong to process i for some $i \in I$.*

Example: *The state predicate model for the program sub-computation $\vec{c_1}$, in the program in Fig. 16, is given below in* **Fig. 18** *for input $0 < \mathtt{x}$ and $0 \neq \mathtt{y}$. In order to reduce the model size, scheduling is not performed after each statement, but only at the initial state, after communications and when a process becomes blocked. The abstract states that are guaranteed to lead to termination of either process have been replaced by "*p *returns" and "*q *returns" for the sake of brevity.*

Those marked with a star abbreviate calls to updatePending *and thus represent a sub graph modeling the behavior of* updatePending. *Note that since* updatePending *cannot cause any communication side effects, process* q *is not scheduled for execution and process* p *thus proceeds until* updatePending *eventually returns.*

In order to extract transition predicates for a given sub-computation, we need to define the set of relevant transitions that must be abstracted over, i.e., the notion of an infinite flow graph must be defined in the context of program transitions. Recall that, in the case of process transitions, $FLOW_{inf}$ defined summary edges to account for terminating computations, each representing some unknown number of concrete transitions. However for a parallel program we cannot guarantee that we can eventually reach a state where *all* processes have returned from their sub-computations such that a summary edge can be used.

Example: *Setting the running example aside for a moment, consider the program in* **Fig. 19***:*

The processes r *and* s *pass a decreasing value back and forth on the channel d and both eventually exit the loop. Both* while *loops in* a *and* b *cause the execution of a* send *and* receive *statement on channel d, so $\vec{c} = \{\{1, 2\}, \{5, 6\}\}$ is a program sub-computation. The state predicate model for $\vec{c}$ is given in* **Fig. 20**. *Again, processes are only scheduled at the initial state, af-*

[☆] Recall the notation *test* for the set of tests that can be used in a program.

```
process r = a(1);         process s = b(100);
a(x){                     b(y){                     c(z){
                            4 : send d ← y;           7 : receive d → z;
   1 : while(x > 0)         5 : while(y > 0)          8 : send d ← z − 1;
      {2 : x = c(0)}           {6 : y = c(0)}         9 : return z − 1
   3 : receive d → x;     }                         }
}
```

Fig. 19 Program cycle where at any point at least one sub-computation has not returned.

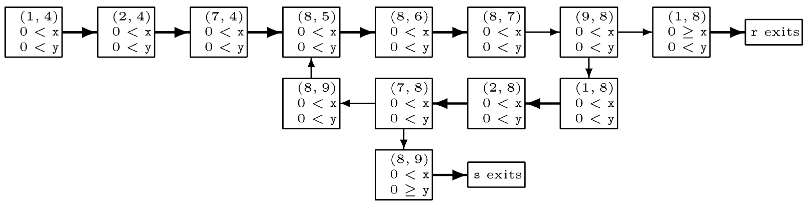


Fig. 20 State predicate model for the parallel composition of r and s.

ter communication and when a process becomes blocked. Note that the process s *starts out with a* send *command, so when either process iterates the* while *loop, the other is evaluating the procedure* c*. This is indicated by the cycle in the state predicate model where for all states either* r *or* s *is at a label in procedure* c*. It is thus not possible to use summary edges for evaluation of* c*, so steps inside* c *must be taken into account.*

Because of this problem the set of transition relations, modeling the behavior of the program, must take sub-computations explicitly into account by also extracting transitions relations for transitions in the sub-computations.

When abstracting transitions for non-parallel programs procedure calls are treated differently depending on the termination assumptions: terminating calls transition to the statement following the call using summary edges, so the sub-computation bound environment is used to obtain size relations. Non-terminating calls transition to a label in the called procedure, in which case the sub-computation bound environment is not used. For parallel programs procedure calls should also make use of the sub-computation bound environment depending on the termination assumptions, but since state predicate models do not have summary edges a new criterion is needed.

One approach is to explicitly annotate labels with termination assumptions and defining a new type of state predicate model that keeps track of termination assumptions for the individual procedures. However in the interest of clarity we will restrict the language to only allow linear recursion, in which case a simple criterion can be used. Suppose we wish to verify the program sub-computation $\vec{c}$. Since all procedures are linearly recursive, infinite computations can only continue if the recursive calls never return, so calls to procedures in $\vec{c}$ can be assumed not to terminate. Calls to procedures not in $\vec{c}$ can be assumed to already have been verified, so such calls must terminate. The criterion can thus be stated as whether the called procedure belongs to $\vec{c}$.

13. Progressive Paths

To verify a given program sub-computation $\vec{c}$, transition relations must be extracted for each possible "transition" in the sub-computation, i.e., transition relations must approximate all paths from a state where one or more processes are at a label in $\vec{c}$ to another such state. In order to exclude paths that eventually "get stuck" in a different sub-computation, we additionally require that at least one process must execute a statement in $\vec{c}$ on the first transition and no statements in $\vec{c}$ are executed for the remaining transitions. Paths that satisfy this criterion are called *progressive paths.*

Definition 13.1. (Progressive Path)*: Suppose M' is a state predicate model for a given program sub-computation $\vec{c}$, and let $L(\vec{c})$ denote the labels of $\vec{c}$: $L(\vec{c}) = \bigcup_{c \in \vec{c}} c$. A* progressive path *is a finite path $(\vec{l_1}, P_1) \rightarrow \ldots \rightarrow (\vec{l_n}, P_n)$ such that for some i: $(\vec{l_1})_i \in L(\vec{c})$ and $(\vec{l_1})_i \neq (\vec{l_2})_i$, for some i': $(\vec{l_n})_{i'} \in L(\vec{c})$ and for all j =*

$1,\ldots,k$ and $a = 3,\ldots,n-1$: $(\vec{l_a})_j \notin L(\vec{c})$ or $(\vec{l_a})_j = (\vec{l_{a-1}})_j$. The existence of a progressive path $(\vec{l_1}, P_1) \to \ldots \to (\vec{l_n}, P_n)$ in M' is denoted $(\vec{l_1}, P_1) \leadsto (\vec{l_n}, P_n)$.

Example: *Consider again the state predicate model in Fig. 18. Note the transitions that cause execution of a statement in $\vec{c_1}$ are marked with thick arrows, and those which do not are marked with a thin arrow. Progressive paths are thus maximal paths that begin with a single thick arrow transition followed by any number of thin arrow transitions.*

For instance $((2,14),P) \to ((2,16),P)$, where $P = \{0 \le \mathtt{x}, 0 \ne \mathtt{y}\}$, is a progressive path where process `q` *calls* `recvPos`*, enabling communication on channel* `d` *since the* `send` *statement at label 2 is matched by a* `receive` *statement at label 16.*

The progressive path given below executes the transaction on channel `d` *and then schedules* `q`*. Process* `q` *proceeds to evaluate the conditional, choosing the* `then` *branch, and then finally returns to label 13:*

$$\begin{array}{lll} ((2,16),P) & \to & ((3,17),P) \\ \to\ ((3,19),P) & \to & ((3,20),P) \\ \to\ ((3,13),P). & & \end{array}$$

Lemma 13.1. *Suppose $\vec{\pi}$ is an infinite computation which is confined to some program subcomputation $\vec{c}$ from some point onwards, then there exists a path $\tau = (\vec{l_1}, P_1) \to (\vec{l_2}, P_2) \to \ldots \in M_{\vec{c}}$ and a sequence of indices I such that $(\vec{l_{I_j}}, P_{I_j}) \leadsto (\vec{l_{I_{j+1}}}, P_{I_{j+1}})$ for any j.*

14. Extraction of Transition Relations

A transition relation in the context of a parallel program is simply a tuple of process transition relations, one for each process. Since program traces are now sequences of locations, trace automata need to be redefined to work with locations rather than labels.

Definition 14.1. (Program Trace Automaton)*: A* program trace automaton *$\vec{A}$ for a program with k processes is a non-deterministic finite automaton over the set of program locations, $Label^k$. We write $\vec{\tau} = (\vec{l_1} \to \ldots \to \vec{l_n}) \in \vec{A}$ if there exists a run $\vec{l_1}, \vec{l_2}, \ldots, \vec{l_n}$ of the automaton $\vec{A}$. In the following $\vec{A_1} \circ \vec{A_2}$ denotes the concatenation of $\vec{A_1}$ and $\vec{A_2}$, such that $(\vec{w_1}\vec{l}\vec{w_2}) \in \vec{A_1} \circ \vec{A_2}$ for all $(\vec{w_1}\vec{l}) \in \vec{A_1}, (\vec{l}\vec{w_2}) \in \vec{A_2}$.*

The state predicate model accounts for blocking and unblocking of `send` and `receive` statements, so when extracting a (program) transition relation for a given progressive path, we can extract a transition relation for each process and then merge them into a transition relation which describes the program sub-computation path by forming the process transition relations into a tuple. The extraction of process transition relations, defined in **Fig. 21** is similar to that of the non-parallel algorithm. One key difference is that the size relations track changes in size, not for the configuration at the top of the program stack, but rather for the last configuration that is at a label in the sub-computation being analyzed. Another difference is that we have `send` and `receive` statements, and that we must be able to abstract transitions where the source label does not belong to $\vec{c}$. Since the blocking and unblocking effects have already been accounted for, we can abstract `send` by the identity transition relation and `receive` by identity transition predicates for all variables but the one being updated. Transitions from a label *not belonging to* $\vec{c}$ represent execution of a statement that does not belong to $\vec{c}$, i.e., a statement in a terminating sub-computation which cannot affect the variables in $\vec{c}$, so such transitions can be abstracted by the identity transition relation. Conversely, transitions from a label *belonging to* $\vec{c}$ represent execution of a statement in $\vec{c}$ and can thus be abstracted using the new definition of $\mathcal{S}$. Note that transitions from a label belonging to $\vec{c}$ to a label *not* belonging to $\vec{c}$ represent the start of a terminating sub-computation and is thus abstracted by the sub-computation bound environment.

By construction, a progressive path can be abstracted by abstracting the first transition in the path since all other transitions are abstracted by the identity transition relation by definition. Note that even though the number of possible progressive paths is infinite, the set can be computed by abstracting the first transition, finding all possible endpoints and producing a transition relation for each endpoint. Any non-progressive loop, executing only statements not belonging to $\vec{c}$, is assumed to already have been analyzed and will thus either terminate, violate the liveness requirement or cause a deadlock.

Since the size relation analysis extracts size relations also for transitions in terminating sub-computations the use of size bound environments requires a modification of safety. The size relations stored in the size bound environ-

$$
\begin{aligned}
&\vec{\mathcal{S}}[\![(\vec{l},P) \to (\vec{l}',P')]\!]\psi && = (\vec{l} \to \vec{l}',\ [\ \mathcal{S}[\![(\vec{l})_i \to (\vec{l}')_i]\!]\psi\ ]_{i=1}^{k})\\
&\mathcal{S}[\![l \to l']\!]\psi && = \begin{cases} \mathcal{S}[\![l : \mathit{stmt}_l]\!]\psi, l' & \text{if } l \in L(\vec{c}) \text{ and } l \neq l' \\ \mathit{Id}(l \to l')[\emptyset,\emptyset] & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{S}[\![l : \texttt{send}\ c \leftarrow e]\!]\psi, l' && = \mathit{Id}(l \to l')[\emptyset,\emptyset]\\
&\mathcal{S}[\![l : \texttt{receive}\ c \to \mathtt{x}]\!]\psi, l' && = \mathit{Id}(l \to l')[\emptyset, \{\mathtt{y}' = \mathtt{y} \mid \text{for all } \mathtt{y} \neq \mathtt{x}\}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{S}[\![l : \mathtt{x} := e]\!]\psi, l' && = \mathit{Id}(l,l')[\emptyset, \mathtt{x}' = \mathcal{S}_e[\![e]\!]]\\
&\mathcal{S}[\![l : \texttt{if}(\mathit{test})\{l_{\mathtt{T}}\dots\}\ \texttt{else}\ \{l_{\mathtt{F}}\dots\}]\!]\psi, l_{\mathtt{T}} && = \mathit{Id}(l,l_{\mathtt{T}})[\mathit{test},\emptyset]\\
&\mathcal{S}[\![l : \texttt{if}(\mathit{test})\{l_{\mathtt{T}}\dots\}\ \texttt{else}\ \{l_{\mathtt{F}}\dots\}]\!]\psi, l_{\mathtt{F}} && = \mathit{Id}(l,l_{\mathtt{F}})[\neg\mathit{test},\emptyset]\\
&\mathcal{S}[\![l : \texttt{choose}\ \ \mathit{test}_1 \to \{l_1\dots\}\ []\ \dots && \\
&\qquad\qquad []\ \ \mathit{test}_n \to \{l_n\dots\}\ ]\!]\psi, l_i && = \mathit{Id}(l,l_i)[\mathit{test}_i,\emptyset]\\
&\mathcal{S}[\![l : \texttt{while}(\mathit{test})\{l_{\mathtt{T}}\dots\}]\!]\psi, l' && =\\
&\quad \begin{cases} \mathit{Id}(l,l')[\mathit{test},\emptyset] & \text{if } l' \in L(\vec{c}) \wedge l' = l_{\mathtt{T}}. \\ \mathit{Id}(l,l')[\psi(l)] & \text{otherwise} \end{cases} &&\\
&\mathcal{S}[\![l : \mathtt{x} := \mathtt{f}(\mathtt{x}_1,\dots,\mathtt{x}_n)]\!]\psi, l' && =\\
&\quad \begin{cases} (l \to l', \emptyset, \{\mathtt{f}^{(1)'} = \mathtt{x}_1, \dots, \mathtt{f}^{(n)'} = \mathtt{x}_n\}) & \text{if } l' \in L(\vec{c}) \\ \mathit{Id}(l,l')[\psi(\mathtt{f}^E[\mathtt{x}'/\mathtt{f}^{(R)}, \mathtt{x}_1/\mathtt{f}^{(1)}, \dots, \mathtt{x}_n/\mathtt{f}^{(n)}]]) & \text{otherwise} \end{cases} &&
\end{aligned}
$$

$$
\mathcal{S}_e[\![e]\!] = \begin{cases} e & \text{if } e \text{ is of form } c + \sum_{i=1}^{n} c_i \mathtt{x}_i \\ \top & \text{otherwise} \end{cases}
$$

Fig. 21 Transition relation analysis for parallel programs.

ment make assertions over the execution of a terminating sub-computation, from the point of entry to the point of exit/return. Thus size relations are not guaranteed to hold for a single step transition (which might end up inside a terminating sub-computation), but can be guaranteed to eventually hold after finitely many steps. We therefore allow process size-relations to be unsafe for terminating sub-computations if they are safe once the sub-computation eventually exits/returns.

Definition 14.2. (Safety)*: The program transition relation $\vec{v}$ is* safe *for the trace automaton $\vec{A}$ iff for each $i = 1,\dots,k$ the process transition relation $(\vec{v})_i$ is an identity transition relation or is safe for any path $l''_1 \to \dots \to l''_n$ where*

- *there exists a $j \in \{1,\dots,n\}$ such that $(l''_1 \to \dots \to l''_j) \in (\vec{A})_i$ and*
- *$l''_1, l''_n \in L(\vec{c})$, $l''_m \notin L(\vec{c})$ for all $m \in \{j,\dots,n-1\}$.*

Example: *Consider again the state predicate model in Fig. 18 containing the progressive paths ($P = \{0 \leq \mathtt{x}, 0 \neq \mathtt{y}\}$):*

$$
\begin{aligned}
\tau_1 = &\quad ((2,14),P) \to ((2,16),P)\\
\tau_2 = &\quad ((2,16),P) \to ((3,17),P)\\
&\quad \to ((3,19),P) \to ((3,20),P)\\
&\quad \to ((3,13),P)
\end{aligned}
$$

To extract transition relations for the two progressive paths, a safe sub-computation bound environment is needed: $\psi = [16 \to \emptyset]$. For τ_1 we must compute a process transition relation for each process transition: $2 \to 2$ and $14 \to 16$. The former is a identity transition, so $\mathcal{S}[\![2 \to 2]\!]\psi = (2 \to 2, \emptyset, \{\mathtt{y}' = \mathtt{y}\})$. The latter transition, $14 \to 16$, executes a call to `recvPos`*, so the transition relation is given by ψ: $\mathcal{S}[\![14 \to 16]\!]\psi = \mathit{Id}(14,16)[\psi(16)[\mathtt{y}'/\mathtt{z}]] = (14 \to 16, \emptyset, \emptyset)$. The transition relation for τ_1 is thus*

$$
\begin{aligned}
&\vec{\mathcal{S}}[\![((2,14),P) \rightsquigarrow ((2,16),P)]\!]\psi\\
&\ = (\quad (2,14) \to (2,16),\\
&\qquad\quad (\quad (2 \to 2, \emptyset, \{\mathtt{y}' = \mathtt{y}\}),\\
&\qquad\qquad (14 \to 16, \emptyset, \emptyset)))
\end{aligned}
$$

For the progressive path τ_2 we get

$$
\begin{aligned}
&\vec{\mathcal{S}}[\![((2,16),P) \rightsquigarrow ((3,13),P)]\!]\psi\\
&\ = (\quad (2,16) \to (3,13),\\
&\qquad\quad (\quad (2 \to 3, \emptyset, \{\mathtt{x}' = \mathtt{x}\}),\\
&\qquad\qquad (16 \to 13, \emptyset, \{\mathtt{y}' = \mathtt{y}\})))
\end{aligned}
$$

$$\begin{array}{lllllll}
(\vec{v'})_i: & \overbrace{l_1 \rightarrow \ \ldots \qquad\qquad \ldots \rightarrow l_{j_2}}^{\in(\vec{A'})_i=(\vec{A_1})_i\circ(\vec{A_2})_i} & \rightarrow & \ldots & \rightarrow l_{n'} \\
(\vec{v_1})_i: & l_1 \rightarrow \ \ldots \rightarrow l_{j_1} \rightarrow \ \ldots & & \ldots & \rightarrow l_{n_1} \\
(\vec{v_2})_i: & \qquad\qquad l_{j_1} \rightarrow \ \ldots \rightarrow l_{j_2} & \rightarrow & \ldots & \rightarrow l_{n'}
\end{array}$$

Fig. 22 Overview of the indices into the path τ (proof of Lemma 14.2).

since $(2 \rightarrow 3, \emptyset, \{\mathtt{x}' = \mathtt{x}\})$ *is safe for* $2 \rightarrow 3$ *and the identity transition relation is safe for all transitions that originate from labels not in* $\vec{c_1}$.

Lemma 14.1. *Let* M' *be a state predicate model for a program, and let* ψ *be a safe sub-computation bound environment. Then for any transition* $\vec{\tau} = (\vec{l}, P) \rightarrow (\vec{l'}, P')$ *in* M' *the transition relation* $\vec{v}$ *is safe for the trace automaton* $\vec{A}$, *where* $(\vec{A}, \vec{v}) = \vec{\mathcal{S}}[\![\vec{\tau}]\!]\psi$.

Composition of two program transition predicates can trivially be defined as the pairwise composition of the process transition predicates they contain.

Definition 14.3. (Composition)*: Suppose that the program transition predicates* $\vec{v_1}$, $\vec{v_2}$ *are safe for the trace automata* $\vec{A_1}$, $\vec{A_2}$ *respectively then define their composition by*

$$\begin{aligned}&(\vec{A_1}, \vec{v_1}) \circ (\vec{A_2}, \vec{v_2})\\ &:= (\vec{A_1} \circ \vec{A_2}, [(\vec{v_1})_i \circ (\vec{v_1})_i]_{i=1}^k])\end{aligned}$$

Lemma 14.2. *Suppose that the program transition predicates* $\vec{v_1}$, $\vec{v_2}$ *are safe for the trace automata* $\vec{A_1}$, $\vec{A_2}$ *respectively and let* $(\vec{A'}, \vec{v'}) = (\vec{A_1}, \vec{v_1}) \circ (\vec{A_2}, \vec{v_2})$ *be their composition, then* $\vec{v'}$ *is safe for* $\vec{A'}$.

Proof. The safety of the composition of program transition relations follows directly from the safety of the composition of each process transition relation, so consider the composition for the i^{th} process: $((\vec{A'})_i, (\vec{v'})_i) = ((\vec{A_1})_i, (\vec{v_1})_i) \circ ((\vec{A_2})_i, (\vec{v_2})_i)$. If either $(\vec{v_1})_i$ or $(\vec{v_2})_i$ is an identity transition relation then safety follows trivially, so suppose that is not the case. To prove safety of the composition, let $\tau = l_1 \rightarrow \ldots \rightarrow l_{n'}$ be path such that for some index $j_2 \in \{1, \ldots, n'\}$ the following holds (**Fig. 22** illustrate how the indices into τ relate):

- $(l_1 \rightarrow \ldots \rightarrow l_{j_2}) \in (\vec{A'})_i$ and
- $l_1, l_{n'} \in L(\vec{c})$, $l_m \notin L(\vec{c})$ for $m \in \{j_2, \ldots, n' - 1\}$.

By the safety of $(\vec{v_2})_i$ there exists an index $j_1 < j_2$ such that

- $(l_{j_1} \rightarrow \ldots \rightarrow l_{j_2}) \in (\vec{A_2})_i$ and
- $l_{j_1} \in L(\vec{c})$.

Since $\vec{A'} = \vec{A_1} \circ \vec{A_2}$ it follows from the safety of $(\vec{v_1})_i$ that for some $n_1 \in \{j_1, \ldots, n'\}$:

- $(l_1 \rightarrow \ldots \rightarrow l_{j_1}) \in (\vec{A_1})_i$ and
- $l_{n_1} \in L(\vec{c})$, $l_m \notin L(\vec{c})$ for $m \in \{j_1, \ldots, n_1 - 1\}$,

so we can conclude that $n_1 = j_1$ (because $l_{j_1} \in L(\vec{c})$). In other words, $(\vec{v_1})_i$ is safe for $l_1 \rightarrow \ldots \rightarrow l_{j_1}$ and $(\vec{v_2})_i$ is safe for $l_{j_1} \rightarrow \ldots \rightarrow l_{n'}$, so by Lemma 3.2 their composition must be safe for $\tau = l_1 \rightarrow \ldots \rightarrow l_{n'}$. □

Just as in the non-parallel analysis, the closure set must be finite in order to insure termination of the verification algorithm, so we extend the definition of convergent process transition relations to convergent program transition relations.

Definition 14.4. (Convergent Program Transition Relations)*: A convergent program transition relation* $\vec{v} = (v_1, \ldots, v_k)$ *is a program transition relation such that each* v_i *is a convergent process transition relation. The domain of convergent program transition relations is denoted* $\vec{V}_{con}$. *The abstraction operator* $\vec{\alpha}_{con}$ *is given by* $\vec{\alpha}_{con}(\vec{v}) = [\alpha_{con}((\vec{v})_i)]_{i=1}^k$ *and the convergent composition operator* $\circ_{con}$ *is given by* $\vec{v_1} \circ_{con} \vec{v_2} = [(\vec{v_1})_i \circ_{con} (\vec{v_2})_i]_{i=1}^k$. *Finally let* $\vec{\overline{\alpha}}_{con}$ *denote the extension of* $\vec{\alpha}_{con}$ *to pairs of trace automata and program transition relations:* $\vec{\overline{\alpha}}_{con}(\vec{A}, \vec{v}) := (\vec{A}, \vec{\alpha}_{con}(\vec{v}))$.

Lemma 14.3. *If the transition relation* $\vec{v}$ *is safe for the trace automaton* $\vec{A}$ *then* $\vec{\alpha}_{con}(\vec{v})$ *is also safe for* $\vec{A}$.

Proof. Follows trivially since α_{con} preserves safety by Lemma 7.2. □

15. Inter-Recursion Paths

The concept of cut-point labels can also be extended to the parallel setting in order to speed up the closure set computation. A *cut-point location* is simply a location where at least one process is at a cut-point label.

Definition 15.1. (Cut-Point Location)*: Given a program sub-computation* $\vec{c}$, *define the set of* cut-point locations $\vec{L}_{cpt}(\vec{c})$ *such that* $\vec{l} \in \vec{L}_{cpt}(\vec{c})$

```
Input:  Program $\mathcal{P}$ and fairness test $fair$.
decompose $FLOW_{inf}(\mathcal{P})$ into a set of strongly connected components $C$
$\psi := \emptyset$
foreach $c \in C$ in bottom-up order:
    compute safe bounds $\psi_0$ for $c$.
    $\psi := \psi \cup \psi_0$.
let $\vec{C}$ be the set of program sub-computations.
foreach $\vec{c} \in \vec{C}$ in bottom-up order:
    compute the State Predicate Model $M'$ for $\vec{c}$
    $W_0 := \{\overline{\alpha}_{con}(\vec{\mathcal{S}}[\![\vec{s_1} \rightsquigarrow \ldots \rightsquigarrow \vec{s_n}]\!]\vec{c}, \psi) \mid$ for all inter-recursion paths $(\vec{s_1} \rightsquigarrow \ldots \rightsquigarrow \vec{s_n}) \in M'$ $\}$
    compute the closure set $(V, E)$ of $W_0$, as per Fig. 5 using $\circ_{con}$
    foreach idempotent $\vec{v} \in V$:
        if( $fair_{A_F}((A_{\vec{v}})^{\omega}) \not\Rightarrow$ well-founded($\vec{v}$) )
            return "Requirement Potentially Violated for $A_{\vec{v}}$"
return "Requirement is Satisfied"
```

Fig. 23 Sub-computation based parallel verification algorithm.

iff $(\vec{l})_i \in L_{cpt}((\vec{c})_i)$ *for some* i.

The concept of inter-recursion traces (Definition 6.2) can then be applied to progressive paths in the state predicate model.

Definition 15.2. (Inter-Recursion Path)*: For a given state predicate model* M' *and a set of cut-point locations* $\vec{L}_{cpt}(\vec{c})$*, an* inter-recursion path *is a path* $(\vec{l_1}, P_1) \rightsquigarrow \ldots \rightsquigarrow (\vec{l_n}, P_n)$ *in* M' *where* $\vec{l_1}$ *and* $\vec{l_n}$ *are cut-point locations and* $\vec{l_i}$ *is non-cut-point for all* $2 \leq i \leq n-1$.

Example: *Returning to the state predicate model in Fig. 18, we note that since labels* 1 *and* 13 *are cut-point labels,*

$$(1,13), (2,13), (1,14), (1,16), (1,17), (3,13)$$

are cut-point locations. An inter-recursion path is thus any path between them, e.g.,

$$((1,13), P) \rightarrow ((2,13), P)$$

and

$$\begin{aligned} &((2,14), P) \rightarrow ((2,16), P) \\ \rightarrow\ &((3,17), P) \rightarrow ((3,19), P) \\ \rightarrow\ &((3,20), P) \rightarrow ((3,13), P), \end{aligned}$$

where $P = \{0 \leq \mathtt{x}, 0 \neq \mathtt{y}\}$. *Notice that the latter inter-recursion path is a concatenation of the two progressive paths which were discussed in the previous example:*

$$((2,14), P) \rightarrow ((2,16), P)$$

and

$$\begin{aligned} &((2,16), P) \rightarrow ((3,17), P) \\ \rightarrow\ &((3,19), P) \rightarrow ((3,20), P) \\ \rightarrow\ &((3,13), P). \end{aligned}$$

Lemma 15.1. *For any sub-computation* $\vec{c}$ *and any infinite computation* $\vec{\pi}$ *for which* $trace(\vec{\pi}) = \vec{l_1} \rightarrow \vec{l_2} \rightarrow \ldots$ *and* $\vec{l_i} \in \vec{c}$ *from some point onwards* ($\exists i_0 \in I\!N : \forall i > i_0 : \vec{l_i} \in \vec{c}$)*, there exists an infinite set of indices* $I \subseteq I\!N$ *such that* $\vec{l_i} \in \vec{L}_{cpt}(\vec{c})$ *for all* $i \in I$.

16. Parallel Verification Algorithm

The algorithm for verifying liveness properties, given in **Fig. 23**, processes all the program sub-computations in turn, so the process sub-computation bound environment ψ must be computed in a separate loop over process sub-computations. Transition predicates are extracted for each inter-recursion path in the model. Since the transition predicates are of (virtually) the same form as for the non-parallel algorithm, the closure set computation and the fairness test is identical to the non-parallel algorithm. The essential difference between the parallel verification algorithm and the sequential algorithm is that the parallel algorithm must compute the parallel composition of the processes in the program prior to the closure set computation. This leads to a potentially exponential blow-up in the size of the model, which naturally has a large impact on the efficiency of the analysis. While the sub-computation based parallel verification algorithm does not address the *complexity* of the closure set computation, the efficiency can be improved if the exponential blow-up can be limited.

Theorem 16.1. *Given a program* $\mathcal{P}$ *and a fairness test* $fair_{A_F}$ *on trace automata, if the algorithm in Fig. 23 returns "Requirement is Satisfied" then* $\mathcal{P}$ *is fairly terminating.*

17. Conclusion

We have presented an extension of the transition predicate abstraction framework of Podelski, et al.[11)] that improves on both precision and performance of the original algorithm. The main contribution is the method for identification of sub-computations via the infinite flow graph and its integration into the verification algorithm. We introduced two concrete levels of abstractions: One expressive abstraction sufficiently precise to handle certain types of non-monotone progress and another which ensures convergence of the closure set computation, allowing inter-recursion traces to be collapsed prior to the closure set computation. Further we reported on experimental results which indicate the potential performance gain. Finally we extended the sub-computation algorithm to an parallel language with synchronous communication via statically named channels. By using state predicate abstraction to guide the extraction of transition relations for the parallel composition of a given program, the number of feasible transition relations can be reduced prior to the closure set computation, thus improving efficiency of the algorithm. Many of the concepts of the non-parallel algorithm were shown to carry over into the parallel setting.

17.1 Related and Future Work

In our algorithm the set of preconditions are used directly as a basis for the termination test, so termination can only be shown for programs that already contain suitable tests. The preconditions can however be strengthened by automatic discovery of invariant linear relations using the polyhedral cones, as developed by Cousot and Halbwachs[5], which is a powerful framework for approximating state space. In Ref. 4) the polyhedral analysis is used to augment a termination analysis for an imperative language with well-structured control flow. The termination analysis developed is strong enough to handle lexicographical termination arguments, but fail other termination arguments that size-change termination can handle. Bridging the gap between size-change termination for functional languages and polyhedral analysis for imperative language, Avery[1] applies the polyhedral analysis to strengthen size-change termination analysis for a small imperative language. The method is directly applicable to the analysis of this paper since we use the same definition of well-foundedness. In Ref. 2) Bradley, et al. develops a technique for proving termination of programs with polynomial ranking functions, which might lead to a technique for automatic discovery of polynomial invariants.

Rather than using the notion of decreasing expressions[1] applied to preconditions, it is possible to use more powerful methods for finding "ranking functions" for transition relations. Podelski and Rybalchenko[10] introduce a *complete* method for finding linear ranking functions.

The present algorithm only deals with variables over integers, but the framework can be extended to handle recursive data types by introducing "size measures" that map a data value to an appropriate integer "size", cf. Ref. 6). With such an extension, the framework would be able to handle programs over data structures such as lists and trees in a natural way.

The time complexity for proving size-change termination for first order functional programs with well-founded data has been proven to be PSPACE complete. This result unfortunately also carries over to fair termination. Codish, et al.[3] have developed an efficient method for computing the closure set by using binary decision diagrams leads to a concise representation, though the complexity remains PSPACE complete. Lee has developed[7] two PTIME approximations for size-change termination: One quadratic in the size of the input (which requires certain restrictions on the abstractions) and a general cubic approximation. The two have proven to be surprisingly precise in practice and it is possible that the methods can be applied to speed up the closure set computation for transition relations. However one open question is how to uncover the trace automata that a given transition relation is safe for, in order to prove fair termination.

References

1) Avery, J.: Size-Change Termination and Bound Analysis, *Proc. FLOPS 2006*, LNCS 3945, Hagiya, M. and Wadler, P. (Eds.), Fuji Susono, pp.192–207, Springer (2006).

2) Bradley, A.R, Manna, Z. and Sipma, H.B.: Termination of Polynomial Programs, *Proc. VMCAI '05*, LNCS 3385, Cousot, R. (Ed.), Paris, France, pp.113–129, Springer (2005).

3) Codish, M, Lagoon, V, Schachte, P. and Stuckey, P.: Size-Change Termination Analysis in k-Bits, *Proc. ESOP 2006*, LNCS 3924, Sestoft, P. (Ed.), Vienna, Austria, pp.230–245, Springer (2006).

4) Cólon, M.A. and Sipma, H.B.: Practical Methods for Proving Program Termination, *Proc. CAV 2002*, LNCS 2404, Brinksma, E. and Larsen, K.G. (Eds.), Copenhagen, Denmark,

pp.442–454, Springer (2002).

5) Cousot, P. and Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program, *Proc. POPL 1978*, Tucson, Arizona, pp.84–97, ACM (1978).

6) Frederiksen, C.C.: Automatic Runtime Analysis for First Order Functional Programs, Technical Report D-470, University of Copenhagen (2002).

7) Lee, C.S.: Program Termination Analysis in Polynomial Time, *Proc. GPCE 2002*, LNCS 2487, Batory, D.S, Consel, C. and Taha, W. (Eds.), Pittsburgh, PA, USA, pp.218–235, Springer (2002).

8) Lee, C.S, Jones, N.D. and Ben-Amram, A.M.: The Size-Change Principle for Program Termination, *Proc. POPL 2001*, London, England, pp.81–92, ACM (2001).

9) Podelski, A. and Rybalchenko, A.: Software Model Checking of Liveness Properties via Transition Invariants, Technical Report MPI-TR-2003-2-004, Max Planck Insitiute für Informatik (2003).

10) Podelski, A. and Rybalchenko, A.: A Complete Method for Synthesis of Linear Ranking Functions, *Proc. VMCAI 2004*, LNCS 2937, Steffen, B. and Levi, G. (Eds.), Venice, Italy, pp.239–251, Springer (2004).

11) Podelski, A. and Rybalchenko, A.: Transition Predicate Abstraction and Fair Termination, *Proc. POPL 2005*, Long Beach, California, USA, pp.132–144, ACM (2005).

12) Ramsey, F.P.: On a Problem of Formal Logic, *Procedures of London Mathematical Society*, Vol.30, London, England, pp.264–285 (1930).

Appendix

A.1 Proof of Theorem 4.2

Proof. Assume for all idempotent v are well-founded if $\mathit{fair}_{A_F}((A_v)^\omega)$ and that $\mathcal{P}$ is not fairly terminating. Let $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ be an infinite computation satisfying the fairness requirement: fair_{A_F} holds from some point onwards. If $\mathit{trace}(\pi) = l_1 \rightarrow l_2 \rightarrow \ldots$ then there exists a $k' \geq 1$ such that $\mathit{fair}_{A_F}(l_k \rightarrow l_{k+1} \rightarrow \ldots)$ holds for all $k \geq k'$. Define $w_{(i,i+1)} := v$ to be the transition relation $v \in V$ corresponding to the transition $l_i \rightarrow l_{i+1}$. Define a *2-set* to be a two element set $\{i, j\}$ of positive integers and assume without loss of generality that $i < j$. For each $v \in V$ define the class P_v of 2-sets yielding v by:

$$P_v := \{(i,j) \mid v = w_{(i,i+1)} \circ \ldots \circ w_{(j-1,j)}\}.$$

The set of classes $\{P_v \mid v \in V\}$ partitions subsequences of $w_{(1,2)}, w_{(2,3)}, \ldots$ into equivalence classes depending on which transition relation they compose to. The set of classes $\{P_v \mid v \in V\}$ is mutually disjoint and every 2-set belongs of exactly one of them. The closure set (V, E) was assumed finite, so by the Infinite Ramsey Theorem [12] there thus exists an infinite set of indices I such that any 2-set of elements in I belong to the same equivalence class. That is, there exists a transition relation $v' \in V$ such that for all indices $i, j \in I$ such that $i < j$ we have that $(i, j) \in P_{v'}$. This implies that v' is idempotent: let i, i' and i'' be indices in I such that $i < i' < i''$:

$$\begin{aligned} v' &= w_{(i,i+1)} \circ \ldots \circ w_{(i''-1,i'')} \\ &= (w_{(i,i+1)} \circ \ldots \circ w_{(i'-1,i')}) \\ &\quad \circ (w_{(i',i'+1)} \circ \ldots \circ w_{(i''-1,i'')}) \\ &= v' \circ v' \end{aligned}$$

Fairness for a given trace automaton A is defined by $\mathit{fair}_{A_F}(A) \iff A_F \cap A \neq \emptyset$, where A_F is the Büchi-automaton encoding all fair traces. It follows that A is fair iff some trace $\mathit{tr} \in A$ is fair: $\mathit{fair}_{A_F}(A) \iff \exists \mathit{tr} \in A : \mathit{fair}_{A_F}(\mathit{tr})$. By Theorem 3.1 $A_{v'}$ is safe for v', so we have that $(l_k \rightarrow l_{k+1} \rightarrow \ldots) \in (A_{v'})^\omega$ for some $k \in \{i \in I \mid i \geq k'\}$. Since $\mathit{fair}_{A_F}(l_k \rightarrow l_{k+1} \rightarrow \ldots)$ holds by assumption, it thus follows that $\mathit{fair}_{A_F}((A_{v'})^\omega)$ must hold. Again, by assumption $\mathit{fair}_{A_F}((A_{v'})^\omega)$ implies $well-founded(v')$, so v' must be well-founded. That is, there exists a precondition $0 < e$ which decreases for v. By Theorem 4.1 for any two indices i, j in I such that $i < j$ we have $\rho_i \models_{exp} e \rightarrow a_i$ and $\rho_j \models_{exp} e \rightarrow a_j$ implies $a_i > a_j$, where ρ_i is the store on the top of stack s_i. The index set I is infinite so there exists an infinite sequence $a_1 > a_2 > \ldots$ of decreasing numbers. However the safety of v' implies that if $\rho_i \models_{exp} e \rightarrow a_i$ then $0 < a_i$ for all i so the sequence is bounded from below. Thus the sequence is well-founded and must be finite, contradicting the original assumption. $\square$

A.2 Proof of Theorem 8.1

Proof. Again the correctness of the algorithm depends on Theorem 4.2. By Lemma 6.1 any trace that is restricted to a sub-computation c from some point onwards can be partitioned into subsequences that are accepted by some trace automaton A where $(A, v) \in W_0$ by choosing $I = L_{cpt}(c)$. Since the algorithm was assumed to output “Requirement is Satisfied” the closure set computation of W_0 must terminate, so the closure set must be finite. Any $(A, v) \in W_0$ is safe since extraction of transition relations is safe by Lemma 3.1, composition over inter-recursion traces is safe by Lemma 3.2 and α_{con} preserves safety by Lemma 7.2. The sub-computation bound environment is modified as the algorithm iterates over sub-computations, so we show safety of ψ by induction. Clearly the empty sub-computation bound environment is safe and by assumption the new bounds ψ_0 are safe for each iteration so $\psi \cup \psi_0$ must also be safe. By definition the convergent composition operator $\circ_{con}$ preserves safety, so by Theorem 4.2 the program is fairly terminating if for all idempotent $v \in V$: $fair_{A_F}((A_v)^\omega) \Rightarrow well-founded(v)$. Since this is precisely the condition tested, the algorithm outputs "Requirement is Satisfied" only if the program is fairly terminating. □

A.3 Proof of Theorem 16.1

Proof. The correctness of the algorithm depends on Theorem 4.2. The sub-computation bound environment is modified as the algorithm iterates over sub-computations, so we show safety of ψ by induction. Clearly the empty sub-computation bound environment is safe and by assumption the new bounds ψ_0 are safe for each iteration, so $\psi \cup \psi_0$ must also be safe.

By Lemma 15.1 any trace that is restricted to a program sub-computation $\vec{c}$ from some point onwards can be partitioned into subsequences that are accepted by some trace automaton $\vec{A}$ where $(\vec{A}, \vec{v}) \in W_0$ by choosing $I = \vec{L}_{cpt}(\vec{c})$. Since the algorithm was assumed to output "Requirement is Satisfied" the closure set computation of W_0 must terminate, so the closure set must be finite. Any $(\vec{A}, \vec{v}) \in W_0$ is safe since extraction of transition relations is safe by Lemma 14.1, composition over inter-recursion paths is safe by Lemma 14.2 and $\vec{\alpha}_{con}$ preserves safety by Lemma 14.3. By definition the convergent composition operator $\circ_{con}$ preserves safety, so by Theorem 4.2 the program is fairly terminating if for all idempotent $\vec{v} \in \vec{V}$: $fair_{A_F}((A_{\vec{v}})^\omega) \Rightarrow well-founded(\vec{v})$. Since this is precisely the condition tested, the algorithm outputs "Requirement is Satisfied" only if the program is fairly terminating. □

(Received December 15, 2006)
(Accepted March 26, 2007)

Carl Christian Frederiksen was born in 1976. During his studies in the TOPPS group (Theory and Practise of Programs) at the University of Copenhagen, he worked as a research assistant on model checking, size-change termination and running time analysis projects. After receiving his M.Sc. degree in 2002, he joined Hagiya Lab at the University of Tokyo where he is currently working as a Ph.D. student. His research interests include size-change termination based program analysis and verification of liveness properties.

Masami Hagiya is a professor at Department of Computer Science, the University of Tokyo. After receiving M.Sc. from the University of Tokyo, he worked for Research Institute for Mathematical Sciences, Kyoto University, and received Dr.Sc. He has been working on modeling, formalization, simulation, and verification of computer systems, including various kinds of software systems and programming languages, mainly using deductive approaches. Recently, he is not only dealing with systems composed of electronic computers, but also biological and molecular systems with respect to their computational aspects, and is now working on DNA and molecular computing.