# Information Reuse for Multi-goal Reachability Analyses*

Dirk Beyer[1], Andreas Holzer[2], Michael Tautschnig[3,4], and Helmut Veith[2]

[1] University of Passau, Germany
[2] Vienna University of Technology, Austria
[3] University of Oxford, UK
[4] Queen Mary, University of London, UK

**Abstract.** It is known that model checkers can generate test inputs as witnesses for reachability specifications (or, equivalently, as counterexamples for safety properties). While this use of model checkers for testing yields a theoretically sound test-generation procedure, it scales poorly for computing complex test suites for large sets of test goals, because each test goal requires an expensive run of the model checker. We represent test goals as automata and exploit relations between automata in order to reuse existing reachability information for the analysis of subsequent test goals. Exploiting the sharing of sub-automata in a series of reachability queries, we achieve considerable performance improvements over the standard approach. We show the practical use of our multi-goal reachability analysis in a predicate-abstraction-based test-input generator for the test-specification language FQL.

## 1 Introduction

Consider the problem of performing many reachability queries on a program that is given as source code. This problem is common in white-box test generation [3, 12, 13], where the goal is to obtain inputs for different paths in a given program. If, for instance, we want to achieve basic-block coverage, we will for each basic block $b$ generate a test-goal that asks if there is a program execution that reaches $b$ and, ultimately, the program exit. In previous work, we designed the coverage-specification language FQL [20, 22], which provides a concise specification of complex coverage criteria. Such a coverage criterion is then translated into a (possibly huge) set of test goals (cf. Table 2). Each such test goal is represented as a finite automaton, called *test-goal automaton*, and specifies a reachability query. Test-goal automata often have overlapping parts, i.e., identical parts of the automata, which let us reuse analysis results across several queries. In this paper, we present an approach that exploits the automaton structure of reachability queries to efficiently reuse information when solving multiple queries. In order to define the potentially shared behavior of two automata $A$ and $A'$, we introduce the notion of *similarity of $A$ and $A'$ modulo a set $X$ of transitions*, where $X$ is a subset of the transitions

* This work was supported by the Canadian NSERC grant RGPIN 341819-07, by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF), by the Vienna Science and Technology Fund (WWTF) grant PROSEED, and by the EPSRC project EP/H017585/1.

M. Felleisen and P. Gardner (Eds.): ESOP 2013, LNCS 7792, pp. 472–491, 2013.
© Springer-Verlag Berlin Heidelberg 2013

of $A'$. If $A$ *simulates* $A'$ *modulo* $X$, then we also have trace containment modulo $X$. That is, each sequence of transitions starting in an initial state of $A'$ and not including a transition from $X$ is also a sequence of transitions in $A$. This enables us to reason about reachability of program executions in $A'$ based on the reachability results for $A$ as long as we are investigating transition sequences shared by both automata. Using this notion of similarity, we face two challenges: *(i) how can we maximize the overlapping parts of automata* (for two automata we can always achieve similarity modulo $X$ by choosing a sufficiently large $X$, but the bigger $X$ is, the less information we can reuse), and *(ii) in which order shall we process the automata to achieve a minimal number of reachability analysis runs?*

The alphabet of a test-goal automaton is a finite set of observations of a program execution. An observation can be, for example, a specific code location that is visited during the execution or a predicate over program states. Each (partial) program execution can be mapped to a word over these observations and for each test-goal automaton we want to determine whether it describes a (partial) execution in the given program. Figure 2 shows a symbolic representation of the state space of program $P$ given in Figure 1. Each node consists of a program location and a description of the heap and the stack when entering that program location. The edges correspond to the execution of program statements and each target node represents the strongest postcondition of the respective statement when applied to the source node. For the moment, we restrict ourselves to observe only code locations (for simplicity, represented by line numbers). Then, the observable sequences of $P$ are $\langle 1, 2, 5, 7, 10\rangle$, $\langle 1, 2, 5, 6, 7, 10\rangle$, and $\langle 1, 4, 5, 7, 8\rangle$. $P$ satisfies test-goal automaton $A_1$ (cf. Figure 3) but not test-goal automata $A_2$ and $A_3$ (cf. Figure 4).

```
1  if (x > 10)
2    f1 = false;
3  else
4    f1 = true;
5  if (x == 100)
6    f2 = false;
7  if (f1)
8    s = f2;
9  else
10   s = f1;
```

**Fig. 1.** Example program $P$

Due to recursion and loops it is generally undecidable whether a test goal is satisfiable on an arbitrary program. We use reachability analyses, e.g., CEGAR-based predicate abstraction [7, 17], to approximate the set of executions of a program until we either (i) have found a partial program execution that is described by a word in the language of the test-goal *or* (ii) we have shown that there is no such execution. The test-goal automaton guides the reachability analysis, i.e., the analysis tracks program and automaton states simultaneously

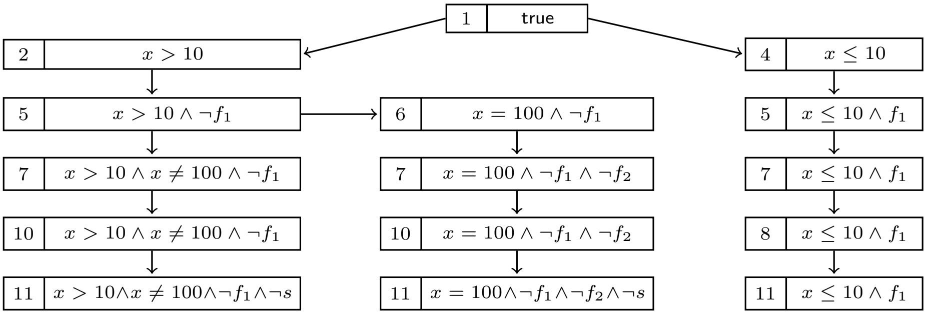
1 | true
2 | $x > 10$
5 | $x > 10 \wedge \neg f_1$
7 | $x > 10 \wedge x \neq 100 \wedge \neg f_1$
10 | $x > 10 \wedge x \neq 100 \wedge \neg f_1$
11 | $x > 10 \wedge x \neq 100 \wedge \neg f_1 \wedge \neg s$
6 | $x = 100 \wedge \neg f_1$
7 | $x = 100 \wedge \neg f_1 \wedge \neg f_2$
10 | $x = 100 \wedge \neg f_1 \wedge \neg f_2$
11 | $x = 100 \wedge \neg f_1 \wedge \neg f_2 \wedge \neg s$
4 | $x \leq 10$
5 | $x \leq 10 \wedge f_1$
7 | $x \leq 10 \wedge f_1$
8 | $x \leq 10 \wedge f_1$
11 | $x \leq 10 \wedge f_1$

**Fig. 2.** Reachable state space of $P$ (cf. Figure 1)

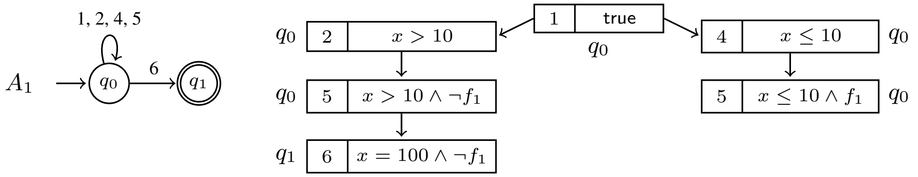
$A_1$
1, 2, 4, 5
$q_0$
6
$q_1$
1 | true
$q_0$
$q_0$ 2 | $x > 10$
$q_0$ 5 | $x > 10 \wedge \neg f_1$
$q_1$ 6 | $x = 100 \wedge \neg f_1$
4 | $x \leq 10$ $q_0$
5 | $x \leq 10 \wedge f_1$ $q_0$

**Fig. 3.** State space of $P$ restricted by $A_1$

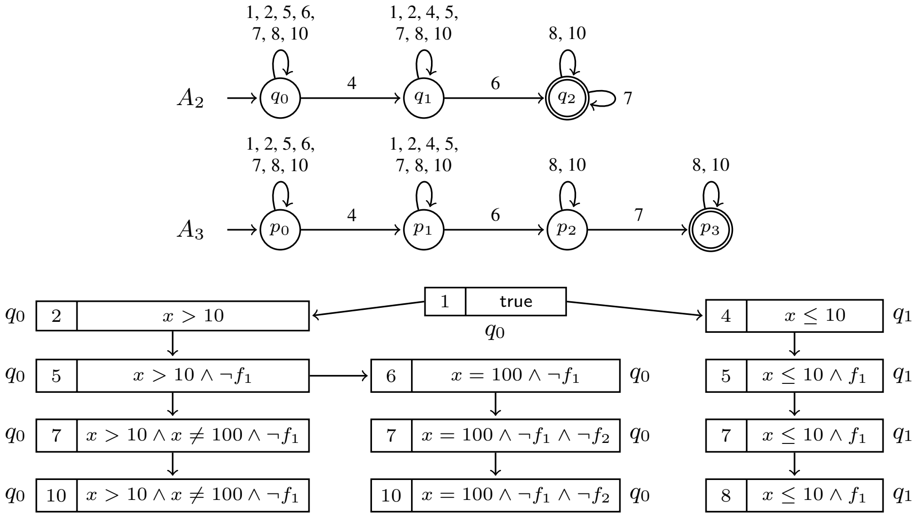
$A_2$
1, 2, 5, 6, 7, 8, 10
$q_0$
4
1, 2, 4, 5, 7, 8, 10
$q_1$
6
8, 10
$q_2$
7
$A_3$
1, 2, 5, 6, 7, 8, 10
$p_0$
4
1, 2, 4, 5, 7, 8, 10
$p_1$
6
8, 10
$p_2$
7
8, 10
$p_3$
1 | true
$q_0$
$q_0$ 2 | $x > 10$
$q_0$ 5 | $x > 10 \wedge \neg f_1$
$q_0$ 7 | $x > 10 \wedge x \neq 100 \wedge \neg f_1$
$q_0$ 10 | $x > 10 \wedge x \neq 100 \wedge \neg f_1$
6 | $x = 100 \wedge \neg f_1$ $q_0$
7 | $x = 100 \wedge \neg f_1 \wedge \neg f_2$ $q_0$
10 | $x = 100 \wedge \neg f_1 \wedge \neg f_2$ $q_0$
4 | $x \leq 10$ $q_1$
5 | $x \leq 10 \wedge f_1$ $q_1$
7 | $x \leq 10 \wedge f_1$ $q_1$
8 | $x \leq 10 \wedge f_1$ $q_1$

**Fig. 4.** State space of $P$ restricted by automaton $A_2$

and stops exploring the state space if there is no possible transition in the program state space or no possible next automaton transition (cf. the reduced state space in Fig. 3).

First, let us consider the case where the set $X$ of excluded automaton transitions is empty, i.e., $X = \emptyset$. Then simulation modulo $X$ amounts to the standard definition of simulation [25]. Let $H \subseteq Q' \times Q$ be a relation between the set of states $Q'$ of an automaton $A'$ and the set of states $Q$ of an automaton $A$ such that for each $(p, q) \in H$ there is for each outgoing transition $(p, a, p')$ an outgoing transition $(q, a, q')$ such that $(p', q') \in H$. We call $H$ a *simulation relation*. We say that $q$ *simulates* $p$ if $(p, q) \in H$ and we say that $A$ *simulates* $A'$ if for each initial state $p$ of $A'$, there is an initial state $q$ of $A$ such that $q$ simulates $p$. For example, in Fig. 4, automaton $A_2$ simulates automaton $A_3$. The simulation relation $H = \{(p_0, q_0), (p_1, q_1), (p_2, q_2), (p_2, q_3)\}$ witnesses this fact. The fact that $A_2$ simulates $A_3$ implies that each finite sequence of transitions starting in an initial state of $A_3$ corresponds to an equivalent sequence of transitions starting in an initial state of $A_2$. From the state space given in Fig. 4, we know that state $q_2$ is not reachable in $A_2$ and since $H$ relates $p_3$ only to $q_2$, we can conclude that $p_3$ is not reachable as well and that therefore no accepting trace in $A_3$ exists.

In general, test-goal automata do not simulate each other. For example, consider the situation where a reachability analysis involving $A_3$ is performed. Then, each node in

Fig. 4 labeled with $q_0$ would be labeled with $p_0$ and each node labeled with $q_1$ would be labeled with $p_1$. The automaton $A_3$ does not simulate $A_2$ (e.g., $A_2$ accepts a word $\langle 4, 6, 7, 7 \rangle$ which is not accepted by $A_3$). Nevertheless, we can still reuse the reachability information obtained for $A_3$ when solving $A_2$: Let $A'_2$ be the automaton $A_2$ where the transition $(q_2, 7, q_2)$ is removed. Then, $A_3$ simulates $A'_2$, witnessed by the relation $H' = \{(q_0, p_0), (q_1, p_1), (q_2, p_2)\}$, and we say that $A_3$ *simulates* $A_2$ *modulo the transition set* $\{(q_2, 7, q_2)\}$. From the fact that state $p_2$ is not reachable we can conclude that $q_2$ is not reachable and that therefore $A_2$ is unsatisfiable. Based on the set of excluded transitions one skips parts of the already analyzed state space (those parts which involve these transitions) or continues state-space exploration at points that have been skipped during the previous state space exploration.

In Sect. 4, we combine automaton-based reasoning techniques as introduced above into an approach for multi-goal reachability analysis. Before that, we will formally introduce the automata that we use for representing reachability queries (cf. Sect. 2) and discuss our automaton-based reasoning techniques individually (cf. Sect. 3). We implemented the test-input generator CPA/TIGER, which is based on predicate-abstraction. We show in our experiments (cf. Sect. 5) that we significantly improve over a naive approach to multi-goal reachability analysis by applying our information reuse techniques. Furthermore, we compare our implementation with existing test generation tools. In Sect. 6 we discuss related work and show how our approach can be integrated into other test generation methods. Finally, we conclude and discuss future work in Sect. 7.

## 2 Test-Goal Automata

We first introduce our program representation, then define test-goal automata, and finally discuss how we represent information gathered by a reachability analysis.

**Programs.** We represent a program as a *control-flow automaton* (CFA) [5]. A CFA $(L, E)$ is a directed, labeled graph, that consists of a finite set $L$ of nodes and a finite set $E \subseteq L \times \mathit{Ops} \times L$ of edges. A node $\ell \in L$ models a program location (program counter valuation) and an edge $(\ell, \mathit{op}, \ell') \in E$ models a program transfer (control flows from location $\ell$ to $\ell'$, while performing program operation $\mathit{op}$). A program operation $\mathit{op} \in \mathit{Ops}$ is either an assignment or an assume operation[1]. Program operations can read from, and write to, a finite set of integer and Boolean variables. Figure 5 shows the CFA representing the source code shown in Fig. 1.

A *program state* $c$ is a mapping from program counter and program variables to values. We denote the

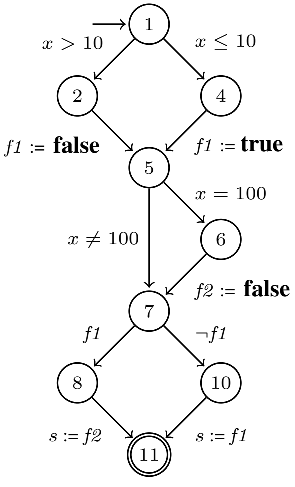

**Fig. 5.** CFA $P_0$ for code in Fig. 1

[1] Our implementation performs an interprocedural analysis (i.e., handles function call and function return), but for simplicity of presentation we limit the formalization to flat programs over integer and Boolean variables.

set of all program states by $C$. A set of program states is represented by a *state predicate* $\varphi$ over the program counter and program variables. We denote the set of state predicates by $\Phi$. We write $c \models \varphi$ (and say, $c$ satisfies state predicate $\varphi$) if program state $c \in C$ is in the state set represented by $\varphi \in \Phi$, and we write $[\![\varphi]\!] = \{c \in C \mid c \models \varphi\}$ for the set of concrete states represented by $\varphi$. The *concrete semantics* of a program operation $op \in Ops$ is given by the strongest postcondition $SP_{op}$, i.e., for a set of states represented by $\varphi$, the set of successor states is represented by $\varphi' = SP_{op}([\![\varphi]\!])$. A *program execution* is a sequence $c_0 \overset{e_0}{\to} c_1 \ldots c_i \overset{e_i}{\to} c_{i+1} \ldots$ of program states $c_i$, for $i \geq 0$, and consecutive CFA edges $e_i = (\ell_i, op_i, \ell_{i+1})$, for $i \geq 0$, such that $c_{i+1} \in SP_{op_i}(c_i)$ holds for each $c_i \overset{e_i}{\to} c_{i+1}$. We call a program execution *complete* if either the program location of the last state of the execution coincides with the program exit or the execution is infinite. Otherwise, we call the program execution *partial*.

**Test-Goal Automata.** A *test goal* describes a set of program traces. Test goals refer to the syntactic structure and semantics of a program. We characterize program traces syntactically by referring to CFA edges, and semantically by using state predicates. For example, a test goal can require to find a program execution (identified by an input assignment) to a particular program location (specified by a CFA edge), or to evaluate a certain expression to a specific value (specified by a CFA edge and a state predicate). We represent a test goal by a test-goal automaton:

A *test-goal automaton (TGA)* $A = (Q, \Sigma, \Delta, I, F)$ is a nondeterministic finite automaton, with a finite set $Q$ of states, an alphabet $\Sigma \subseteq E \times \Phi$ consisting of pairs of CFA edges and state predicates, a transition relation $\Delta \subseteq Q \times \Sigma \times Q$, a set $I \subseteq Q$ of initial states, and a set $F \subseteq Q$ of accepting states. We write $q \xrightarrow{a} q'$ in case $(q, a, q') \in \Delta$. We say that $A$ *accepts* a program execution $c_0 \xrightarrow{e_0} c_1 \cdots \xrightarrow{e_{n-1}} c_n$ if there is a sequence $q_0 \xrightarrow{(e_0,\psi_0)} q_1 \cdots \xrightarrow{(e_{n-1},\psi_{n-1})} q_n$ of TGA transitions starting in an initial state $q_0 \in I$ and ending in a final state $q_n \in F$ such that $c_{i+1} \models \psi_i$ holds for each $q_i \xrightarrow{(e_i,\psi_i)} q_{i+1}$. The last condition $c_{i+1} \models \psi_i$ means that if there is a program transition from state $c_i$ to state $c_{i+1}$, then the state predicate $\psi_i$ is evaluated on the successor state $c_{i+1}$. By adding an additional initial CFA edge, one can also restrict the initial program state.

*Example.* Figure 6 shows a TGA with initial state $q$ and final state $p$. The automaton requires that Line 11 (represented by CFA edge $(8, s{:=}f2, 11)$) is visited during a program execution and that flag $s$ is true after executing the statement $s := f2$. Due to the self-loops at $q$ and $p$ there are no restrictions to a program execution other than the one stated above. An execution satisfying this test goal has to make both variables *f1* and *f2* true and therefore $x$ can't have the value 100 in that execution. Figure 7 shows the set of TGAs representing basic-block coverage on program $P_0$ given in Fig. 5. For simplicity,

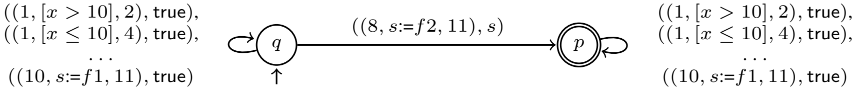
$((1, [x > 10], 2), \mathsf{true})$,
$((1, [x \leq 10], 4), \mathsf{true})$,
...
$((10, s{:=}f1, 11), \mathsf{true})$
$q$
$((8, s{:=}f2, 11), s)$
$p$
$((1, [x > 10], 2), \mathsf{true})$,
$((1, [x \leq 10], 4), \mathsf{true})$,
...
$((10, s{:=}f1, 11), \mathsf{true})$

**Fig. 6.** Example test-goal automaton $A_{10}$

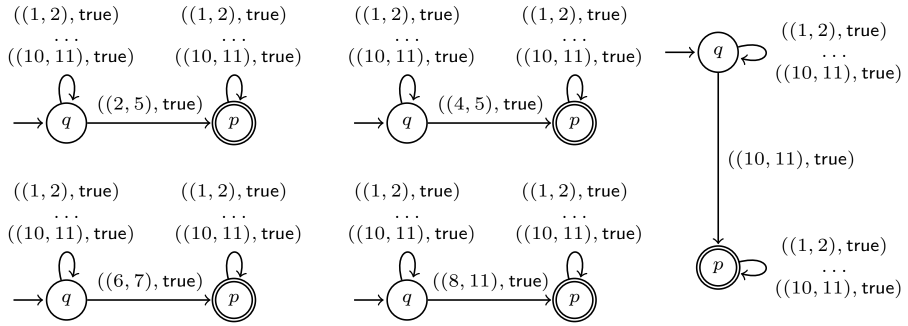

**Fig. 7.** Example test-goal automata for basic-block coverage

we omitted the operations labeling the CFA edges. For each entry of a basic block, i.e., CFA edges $(2, 5)$, $(4, 5)$, $(6, 7)$, $(8, 11)$, and $(10, 11)$, there is a respective automaton.

**Representing Reachability Information.** For our approach we consider reachability analyses that represent the reachable state space of a program by an *abstract reachability graph (ARG)* as it is done for example in predicate-abstraction-based model checkers [5, 6]. Let $P = (L, E)$ be a CFA and let $A = (Q, \Sigma, \delta, q_0, F)$ be a test-goal automaton. An abstract reachability graph $G_{P,A} = (S, T, s_0)$ consists of a finite set $S \subseteq ID \times L \times Q \times D$ of *abstract states*, where $ID$ is a set of identifiers and $D$ is an abstract data domain whose elements describe the heap and stack, finitely many transitions $T \subseteq S \times (E \times \Phi) \times S$ between these abstract states, and an *initial abstract state* $s_0$. The identifier is required to distinguish abstract states with otherwise equal values, which may be produced, e.g., by the ARG transformations of Sect. 3. To simplify the presentation, however, we omit the identifier in the remainder of this paper. An abstract state $s \in S$ induces a state predicate $\varphi_s$ over the same program location and a heap and stack described by the valuation in the abstract domain. Via $\varphi_s$ we obtain a set $[\![s]\!]$ of concrete program states. Given an abstract state $s$, all concrete states in $[\![s]\!]$ share the same program location, denoted by $\ell(s)$, and test-goal automaton state, denoted by $q(s)$. For the initial state it holds that $q(s_0) = q_0$. Let $t \in T$ be the transition $(s, (e, \varphi), s')$, then $t$ has a corresponding CFA edge, i.e., $(\ell(s), e, \ell(s')) \in E$, and $t$ has a corresponding TGA transition $(q(s), (e, \varphi), q(s')) \in \delta$. We denote $(\ell(s), e, \ell(s'))$ by $t_P$ and denote $(q(s), (e, \varphi), q(s'))$ by $t_A$. Then, each sequence $t_1 t_2 \ldots t_n$ of transitions in $G$ corresponds to a sequence $\langle t_{P1}, t_{P2}, \ldots, t_{Pn} \rangle$ of CFA edges and to a sequence $\langle t_{A1}, t_{A2}, \ldots, t_{An} \rangle$ of TGA transitions. Note, the reverse direction does not necessarily hold since a reachability analysis might terminate its state space exploration without explicitly enumerating all sequences of CFA edges or TGA transitions.

Figure 8 shows an example of an ARG $G$ obtained by a reachability analysis for the CFA depicted in Figure 5 and the TGA depicted in Figure 6. An ARG is a finite unwinding of the reachable state space. The unwinding stops in case no new behavior can be observed. In order to obtain a finite unwinding, abstraction might be applied. Paths through the ARG might be merged at points with the same program location and automaton state (cf. [6] for a detailed elaboration and formalization of merge and stop operators).

In the given example, the ARG $G$ contains the accepting state $(11, p, s)$ (with $s$ abstracting from *f1* and *f2* of a concrete execution of the program). We denote the directed acyclic graph reaching this state as the *witness of reachability* of $(11, p, s)$. A witness is *feasible* if there exists a real program execution encoded in the witness and *infeasible* otherwise. The witness given in Figure 8 (enclosed in a dotted line) is feasible, e.g., the input $x = 10, f2 = \mathsf{true}$ causes an execution following the program path $\langle 1, 4, 5, 7, 8, 11 \rangle$ which is accepted by the TGA given in Figure 6.

In Section 3, we discuss how to turn an ARG $G_{P,A_1}$ obtained for a TGA $A_1$ into an ARG $G_{P,A_2}$ for a TGA $A_2$.

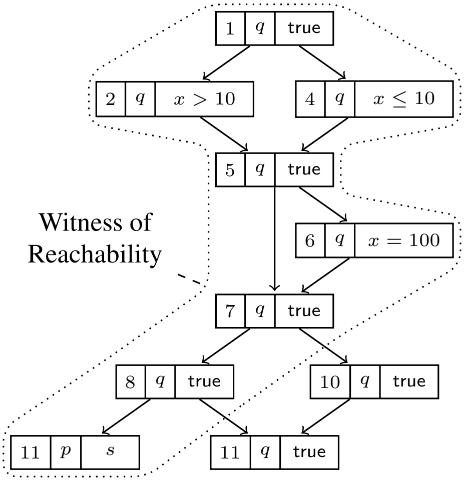

**Fig. 8.** Example ARG $G_{P_0,A_{10}}$ for CFA $P_0$ (see Figure 5) and TGA $A_{10}$ (see Figure 6)

## 3 Reasoning on Test Goals

As input to a multi-goal reachability analysis we are given a CFA $(L, E)$ and a set $\{A_1, A_2, \ldots, A_n\}$ of test-goal automata $A_i$. One way to tackle this problem is to invoke a reachability analysis for each test-goal automaton $A_i$ individually. This approach would have to rediscover lots of information again and again when analysing the program with respect to different test-goal automata. Therefore, we will now discuss a notion of simulation that enables us to identify information that is reusable across reachability analyses for different test-goal automata.

**Relating Test-Goal Automata.** We relate test-goal automata by adapting the notion of similarity [25] to identify the transitions that violate the similarity of two automata:

**Definition 1 (Similarity modulo $X$).** *Given two TGA $A_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, p_0, F_2)$ and a set $X \subseteq \delta_2$ of automaton transitions in $A_2$, we call a relation $H \subseteq Q_2 \times Q_1$ a* simulation relation modulo $X$ from $A_1$ to $A_2$ *if $H$ is a simulation relation from $A_1$ to $\bar{A}_2 = (Q_2, \Sigma, \delta_2 \setminus X, p_0, F_2)$. This means that for each $(p, q) \in H$ and for each transition $(p, a, p') \in (\delta_2 \setminus X)$ there is a transition $(q, a, q') \in \delta_1$ s.t. $(p', q') \in H$.*

*Example.* Figure 9 shows two test goal automata $A_1$ and $A_2$. There, the relation $H = \{(p_0, q_0), (p_1, q_1), (p_2, q_2)\}$ is a simulation relation modulo $X = \{(p_1, c, p_2)\}$ from $A_1$ to $A_2$. Each sequence of transitions contained in the automaton $\bar{A}_2$, as defined above, is also reflected by a corresponding sequence in $A_1$. Note, sets other then the chosen $X$ could be used to establish similarity from $A_1$ to $A_2$, e.g., by adding more transitions of $A_2$ to $X$. However, in order to increase the reuse of gathered reachability information small $X$ are preferable.

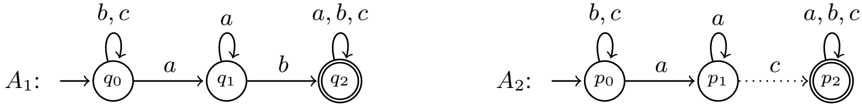

**Fig. 9.** $\{(p_0, q_0), (p_1, q_1), (p_2, q_2)\}$ is a simulation relation modulo $\{(p_1, c, p_2)\}$ from $A_1$ to $A_2$

**Algorithm 3.1.** transform — Transform an ARG into another ARG

```
Input: TGA $A_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, p_0, F_2)$, initial abstract state $\bar{s}_0$,
    automaton transition $(p_0, ((\ell, op, \ell'), \varphi), p') \in \delta_2$ (where $q(\bar{s}_0) = p_0$ and $\ell(\bar{s}_0) = \ell$),
    simulation relation $H$ modulo set of transitions $X$, ARG $G_{P,A_1}$, worklist $W_{A_1}$.
Output: Transformed ARG and worklist for further state-space exploration.
 1: $S' \leftarrow \bigcup_{s \in S} H(s)$
 2: $T' \leftarrow \bigcup_{t \in T} H(t)$
 3: if there is an abstract state $s'_0 \in S'$ such that $q(s') = p_0$ and $[\![s'_0]\!] \supseteq [\![\bar{s}_0]\!]$ then
 4:     choose such an $s'_0$
 5:     $S' \leftarrow \{s' \in S' \mid s'$ is reachable from $s'_0$ via $T'\}$
 6:     $T' \leftarrow \{t \in T' \mid t$ is reachable from $s'_0$ via $T'\}$
 7:     $W \leftarrow \{\langle s', d\rangle \mid \underbrace{(\ell, q, \psi)}_{=s'} \in S', \underbrace{(q, ((\ell, op, \ell'), \varphi), q')}_{=d} \in X\}$
 8:     $W \leftarrow W \cup \bigcup_{\langle \hat{s}, \hat{d}\rangle \in W_{A_1}} \Big((H(\hat{s}) \cap S') \times H(\hat{d})\Big)$
 9:     return $\langle (S', T', s'_0), W\rangle$
10: else
11:     return $\langle (\{\bar{s}_0\}, \emptyset, \bar{s}_0), \{(p_0, ((\ell, op, \ell'), \varphi), p')\}\rangle$
```

**Reusing Reachability Information.** Using simulation relations, we can transform a given ARG $G_{P,A_1}$ for a TGA $A_1$ into an ARG $G_{P,A_2}$ for a TGA $A_2$. We first describe the general principle of this transformation and then point out how to efficiently apply it when analysing multiple automata in a row.

Algorithm 3.1 takes an ARG $G_{P,A_1} = (S, T, s_0)$ and transforms it into an ARG $G_{P,A_2}$ based on a simulation relation $H$ from a TGA $A_1$ to TGA $A_2$ modulo a set of transitions $X$. To compute the transformation we furthermore need the two TGA $A_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, p_0, F_2)$, the abstract state $\bar{s}_0$ (with $q(\bar{s}_0) = p_0$ and $\ell(\bar{s}_0) = \ell$), where we will start with the information reuse, and a TGA transition $(p_0, ((\ell, op, \ell'), \varphi), p') \in \delta_2$ whose role we will discuss later in the context of Algorithm 4.3. For the moment, omitting it does not affect the overall understanding of the transformation process. Given these inputs, we will obtain an ARG $G_{P,A_2} = (S', T', s'_0)$. Before we discuss the algorithm, we will first define the sets $H(s)$ and $H(t)$, for $s \in S$ and $t \in T$, by

$$H(s) = \{s' \mid (p, q(s)) \in H \text{ and } s' \text{ coincides with } s \text{ except that } q(s') = p\}$$

and, for $t = (s_1, (e, \varphi), s'_1)$,

$$H(t) = \{(s_2, (e, \varphi), s'_2) \mid s_2 \in H(s_1), s'_2 \in H(s'_1), (q(s_2), (e, \varphi), q(s'_2)) \in \delta_2 \setminus X\}.$$

The set $X$ controls how many transitions from $T$ can be reused, i.e., the larger $X$ is, the less reachability information can be reused. In Lines 1 and 2 of the algorithm we define

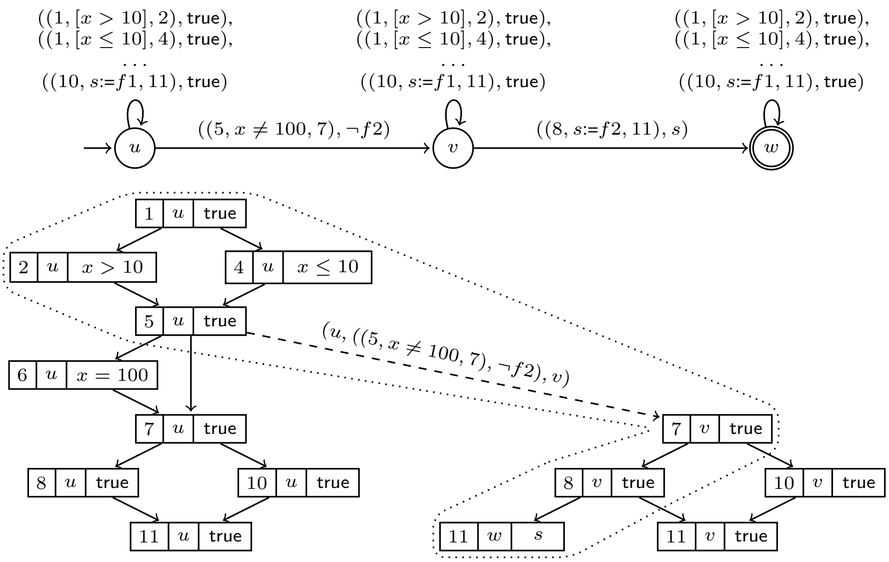

**Fig. 10.** ARG $G_{P_0,A_{11}}$ obtained from $G_{P_0,A_{10}}$ (cf. Figure 8) using the simulation relation $H = \{(u,q),(v,q),(w,p)\}$ modulo $X = \{(u,((5,x \neq 100,7),\neg f2),v)\}$

the set of transformed abstract states $S' = \bigcup_{s\in S} H(s)$ and the set of transformed ARG transitions $T' = \bigcup_{t\in T} H(t)$ by simply translating all abstract states and all transitions of $G_{P,A_1}$. This might lead to many subparts of the resulting ARG not being connected to the initial abstract state. In Lines 5 and 6 we will restrict these two sets to the reachable part only (in the implementation the transformation is only done for the reachable part of the new ARG). But, in order to determine the reachable part of the newly generated ARG, we first have to determine what the initial abstract state will be. We do this based on the given abstract state $\bar{s}_0$. We describe our algorithms from the point of view of an overapproximating reachability analysis, which leads to the condition that an initial abstract state $s'_0$ has to contain all concrete states of $\bar{s}_0$, i.e., $[\![s'_0]\!] \supseteq [\![\bar{s}_0]\!]$. The concepts behind the algorithms given in this paper also work for an underapproximating analysis, but one has to use a dual reasoning, e.g., we would need the condition $[\![s'_0]\!] \subseteq [\![\bar{s}_0]\!]$ instead. In case there is no such abstract state $s'_0$, then we can't reuse any information and we return the ARG $(\{\bar{s}_0\},\emptyset,\bar{s}_0)$. The role of the transition $(p_0,((\ell,op,\ell'),\varphi),p')$ is explained in the discussion of Algorithm 4.3 and we therefore skip it for the moment. Assume there is a suitable abstract state $s'_0$, then we restrict $S'$ and $T'$, as discussed above, to the part reachable from $s'_0$. Lines 7 and 8 create a worklist $W$ which contains abstract states and TGA transitions which have to be further explored. The transformation of these worklists will be explained in the discussion of Algorithm 4.3.

Figure 10 shows the ARG $G_{P_0,A_{11}}$ obtained from ARG $G_{P_0,A_{10}}$ by using the simulation relation $\{(u,q),(v,q),(w,p)\}$ modulo $\{(u,((5,x \neq 100,7),\neg f2),v)\}$. The resulting worklist is $\{\langle(5,u,\mathsf{true}),(u,((5,x \neq 100,7),\neg f2),v)\rangle\}$. First, the left part of ARG $G_{P_0,A_{11}}$ is computed, and then, a reachability analysis determines the

**Algorithm 3.2.** Compute $H, X$

```
Input: $A = (Q, \Sigma, \Delta, q_0, F)$, $A' = (Q', \Sigma, \Delta', p_0, F')$.
Output: $H$ and $X$ such that $A$ simulates $A'$ modulo $X$.
$H \leftarrow \{(p_0, q_0)\}$, $H' \leftarrow \emptyset$
$X \leftarrow \emptyset$, $X' \leftarrow \emptyset$
while $H \neq H'$ or $X \neq X'$ do
    $H' \leftarrow H$
    $X' \leftarrow X$
    if there is a $(p, q) \in H$ s.t. there is a $(p, a, p') \in \Delta' \setminus X$ but no $(q, a, q') \in \Delta$ then
        $X \leftarrow X \cup \{(p, a, p')\}$
    if there is a $(p, q) \in H$ s.t. there is a $(p, a, p') \in \Delta' \setminus X$ but $(p', q') \notin H$ for all
    $(q, a, q') \in \Delta$ then
        choose a subset $U \subseteq \{q' \mid (q, a, q') \in \Delta\}$
        if $U \neq \emptyset$ then
            $H \leftarrow H \cup \{(p', q') \mid q' \in U\}$
        else
            $X \leftarrow X \cup \{(p, a, p')\}$
return $(H, X)$
```

abstract state $(7, v, \mathsf{true})$ as successor of $(5, u, \mathsf{true})$ along TGA transition $(u, ((5, x \neq 100, 7), \neg f2), v)$. Then we again use reachability information from ARG $G_{P_0, A_{10}}$ this time starting in abstract state $(7, v, \mathsf{true})$. The DAG enclosed in the dotted line describes the witness after these three steps 'information reuse' — 'reachability analysis' — 'information reuse'. In Section 4 we describe how to combine these steps in an algorithm for multi-goal reachability analysis.

**Computing Simulation-Modulo-$X$ Relations.** The amount of possible information reuse is determined by (i) the set $X$ of transitions and (ii) the relation $H$. The bigger $X$ is, the bigger $H$ can be chosen, but at the same time the number of reusable transitions in an ARG decreases. Since the transitions encode the actual reachability information we have to find a balance between the size of $X$ and the size of $H$.

Algorithm 3.2 computes, given two TGA $A$ and $A'$, a set $X$ of transitions of $A'$ such that $A$ simulates $A'$ modulo $X$. Furthermore, it computes a corresponding relation $H$. The algorithm starts in the initial states $p_0$ and $q_0$ of $A'$ and $A$, respectively. Initially, the relation $H$ only contains the tuple $(p_0, q_0)$. Then, for each transition $(p_0, a, p')$ outgoing from $p_0$ we check whether we can find a transition $(q_0, a, q')$ outgoing from $q_0$ labeled with $a$ as well. If there is no such transition, then $A$ can't simulate $A'$ wrt. to this transition and we have to add $(p_0, a, p')$ to $X$. Algorithm 3.2 is parametric wrt. its behavior if there are such transitions: If $(p', q')$ is not contained in $H$ yet, the algorithm can decide whether it wants to add it to $H$ at all. A bigger relation $H$ might blow up the resulting translated state space. If the algorithm decides not to add any possible $(p', q')$ to $H$, then the transition $(p, a, p')$ has to be excluded, i.e., it has to be added to $X$. Otherwise, at least one of the $(p', q')$ are added to $H$. Depending on which $(p', q')$ are added to $H$ the final $H$ and $X$ can vary. In our implementation (cf. Section 5) we have implemented a breadth-first search which adds all possible $(p', q')$ to $H$. The automata we consider in our experiments have a tree-like structure and the different $(p', q')$ do

**Algorithm 4.1.** Multi-Goal Reachability Analysis

```
Input: Program P, a sequence of TGA $A_1, A_2, \ldots, A_i, \ldots, A_n$ (see Section 4), an initial program
    location $\ell_0$, and an initial data state $d_0$.
Output: Determines for each $1 \leq i \leq n$ whether $P$ satisfies $A_i$ and, if so, computes inputs.
for $i = 1 \to n$ do
    $s_0 \leftarrow (\ell_0, q_0, d_0)$ where $q_0$ is the initial state of $A_i$
    if $A_i$ is covered by an existing test input contained in the test suite then
        continue
    determine-feasibility($P$, $A_i$, $s_0$)                                  // Algorithm 4.2
```

not interfere with each other in the later exploration. The algorithm then continues with the elements added newly to $H$.

Note that the symbols in $\Sigma$ are actually interdependent, because they are tuples of CFA edges and state predicates. In case there is a transition $(p, a, p')$ and a transition $(q, b, q')$ for $(p, q) \in H$ and $a \neq b$ we might still have a simulation in case the CFA edges of $a$ and $b$ are the same and the state predicate of $b$ is weaker than the state predicate of $a$. For simplicity of presentation, we omitted this case. In case of using an underapproximating reachability analysis one would dually require that the state predicate of $b$ is stronger than the state predicate of $a$.

## 4 Multi-goal Reachability Analysis

In Section 3, we discussed how we can identify shared behavior of two TGA by simulation relations and how we can translate reachability information of one TGA into reachability information for another one. Now, we will use simulation relations as foundation for a reasoning engine that reuses already obtained reachability information. The input to our multi-goal reachability analysis is a sequence of TGA $A_1$, $A_2$, ..., $A_n$. At the end of this section, we will discuss how to exploit concise specifications of sets of TGA and how to obtain an order on these sets.

Algorithm 4.1 shows the main loop of our multi-goal reachability analysis. Its input is a program $P$, a sequence of TGA $A_1$, ..., $A_n$, the initial program location $\ell_0$, and an initial abstract data state $d_0$ describing the heap and stack at program entry. For each test-goal automaton $A_i$, we first check whether we already have inputs inducing a program execution that satisfies $A_i$ and, if so, we skip $A_i$ from further analysis and we continue with $A_{i+1}$. We do this check by simply executing the program simultaneously with the TGA $A_i$ with given inputs. If the execution reaches an accepting state of $A_i$, then, the inputs cover $A_i$. If $A_i$ is not covered, we will perform a feasibility check.

**Feasibility Check.** Algorithm 4.2 realizes the feasiblity check of a TGA. For storing already gathered reachability information we assume a database that stores quadruples $\langle A, G_{P,A}, W_A, \mathit{isFeasible} \rangle$ where $A$ is a TGA, $G_{P,A}$ is the ARG obtained for $A$, and $W_A$ is a worklist containing the abstract states of $G_{P,A}$ where state-space exploration has to continue in order to exhaustively investigate the state space, and *isFeasible* is either FEASIBLE or INFEASIBLE, depending on whether $P$ satisfies $A$ or not. If $W_A \neq \emptyset$ then the state space didn't have to be exhaustively explored to determine the value of

**Algorithm 4.2.** determine-feasibility — Determine Feasibility of TGA

**Input:** CFA $P$, TGA $A = (Q, \Sigma, \Delta, q(s_0), F)$, initial abstract state $s_0$
**Output:** Computes whether $A$ is feasible on $P$ and, if so, determines inputs.

```
 1: $W \leftarrow \{\langle s_0, (q(s_0), ((\ell(s_0), op, \ell'), \varphi), q')\rangle \mid (q(s_0), ((\ell(s_0), op, \ell'), \varphi), q') \in \Delta\}$
 2: $G_{P,A} \leftarrow (\{s_0\}, \emptyset, s_0)$
 3: while $W \neq \emptyset$ do
 4:     pick $\langle s, (q, ((\ell, op, \ell'), \varphi), q')\rangle \in W$ and remove it from $W$
 5:     $\langle G'_{P,A}, W'\rangle \leftarrow reuse(s, (q, ((\ell, op, \ell'), \varphi), q'), A)$        // Algorithm 4.3
 6:     insert $G'_{P,A}$ into $G_{P,A}$ at $s$
 7:     $\langle G_{P,A}, W'\rangle \leftarrow reach(W', A, G_{P,A})$
 8:     $W \leftarrow W \cup W'$
 9:     if there is an $s' \in G_{P,A}$ such that $q(s') \in F$ then
10:         $wit \leftarrow$ witness($s', G_{P,A}$)
11:         if $wit$ is feasible then
12:             derive inputs from $wit$ and store them in test suite
13:             store $\langle A, G_{P,A}, W,$ FEASIBLE$\rangle$ in reachability database
14:             return FEASIBLE
15:         else
16:             $\langle G_{P,A}, W\rangle \leftarrow refine(G_{P,A}, wit, W)$
17: store $\langle A, G_{P,A}, \emptyset,$ INFEASIBLE$\rangle$ in reachability database
18: return INFEASIBLE
```

*isFeasible*. A worklist is a set of tuples of abstract states and TGA transitions. A tuple $\langle s, (q, ((\ell, op, \ell'), \varphi), q')\rangle$ requires a state-space exploration starting in abstract state $s$, where $q(s) = q$ and $\ell(s) = \ell$ holds, and performed along CFA edge $(\ell, op, \ell')$ (the postcondition $\varphi$ has to be considered by the reachability analysis as well).

The algorithm maintains a worklist $W$ and an ARG $G_{P,A}$. Worklist $W$ is initialized with all transitions potentially leaving the initial abstract state. At the beginning, $G_{P,A}$ only consists of the initial abstract state $s_0$. As long as $W$ is not empty, the algorithm picks an element from the worklist and performs a state-space exploration. Before applying a reachability analysis the algorithm tries to reuse already computed reachability information. We call Algorithm 4.3 with the chosen abstract state and transition as well as the current TGA. The algorithm returns an ARG $G'_{P,A}$ and a worklist $W'$. $G'_{P,A}$ represents the reachability information reusable at abstract state $s$ and $W'$ contains the exploration points where reachability information is missing. In case no reuse is possible, the ARG containing only state $s$ is returned and the worklist contains the tuple which was initially picked at Line 4. After calling Algorithm 4.3, the returned sub-ARG is inserted into $G_{P,A}$ at abstract state $s$. As already mentioned, $W'$ contains the tuples where state-space exploration has to continue, therefore a reachability analysis is called with the task of exploring all points in $W'$. The reachability analysis might decide to return before having explored all tuples (or none at all) in $W'$. It returns an updated ARG and an updated worklist. The worklist $W'$ might be non-empty if the reachability analysis found a witness for the feasibility of $A$ or if we first want to check whether we can reuse some reachability information for the tuples in $W'$. We therefore add $W'$ to $W$ for later processing.

**Algorithm 4.3.** reuse — Reuse Stored Reachability Information

```
Input: Abs. state $s$, TGA $A = (Q, \Sigma, \Delta, q, F)$, transition $(q, ((\ell, op, \ell'), \varphi), q') \in \Delta$.
Output: ARG and worklist
 1: let $\hat{q}$ be a new state, i.e., $\hat{q} \notin Q$
 2: if $q \in F$ then
 3:     $F' \leftarrow F \uplus \{\hat{q}\}$
 4: else
 5:     $F' \leftarrow F$
 6: $A' \leftarrow (Q \uplus \{\hat{q}\}, \Sigma, \Delta \uplus \{(\hat{q}, ((\ell, op, \ell'), \varphi), q')\}, \hat{q}, F')$
 7: choose a tuple $\langle \bar{A}, G_{P,\bar{A}}, W, f \rangle$ from the database
 8: if $\langle \bar{A}, G_{P,\bar{A}}, W, f \rangle$ exists then
 9:     compute $H$, $X$ such that $\bar{A}$ simulates $A'$ modulo $X$                    // Algorithm 3.2
10:     if $(q, ((\ell, op, \ell'), \varphi), q') \notin X$ then
11:         $\langle G_{P,A'}, W \rangle \leftarrow transform(\bar{A}, A', s, (\hat{q}, ((\ell, op, \ell'), \varphi), q'), H, X, G_{P,\bar{A}}, W)$
                                                                                        // Algorithm 3.1
12:         replace each occurrence of $\hat{q}$ in $G_{P,A'}$ and $W$ by $q$
13:         return $\langle G_{P,A'}, W \rangle$
14: return $\langle (\{s\}, \emptyset, s), \{\langle s, (q, ((\ell, op, \ell'), \varphi), q') \rangle\} \rangle$
```

Lines 10 to 16 deal with the case where an abstract state with an accepting automaton state is found. Then, a witness is extracted and its feasibility is checked (the ARG encodes all information necessary to exactly represent the program paths in the witness as a formula $\psi$). If the witness is feasible we derive inputs from the model of $\psi$ and store these inputs in the test suite. Furthermore, we extend our reachability information database by the tuple $\langle A, G_{P,A}, W, \text{FEASIBLE} \rangle$. Then, we return to the main multi-goal reachability analysis algorithm. If the witness is infeasible it means the reachability analysis was not precise enough during state-space exploration and we therefore refine the precision of the analysis based on the infeasible witness. This may result in a change in the ARG as well as in the worklist. In our implementation we use a reachability analysis based on predicate abstraction and CEGAR [7, 17].

After all elements in $W$ are finally processed, we know that $P$ does not satisfy $A$ and store the according information in the database (cf. Line 17).

**Reusing Stored Reachability Information.** In order to determine reusable stored information, we first transform the given TGA $A$ into a TGA $A'$ which is the same as $A$ except that a new state $\hat{q}$ and a transition $(\hat{q}, ((\ell, op, \ell'), \varphi), q')$ is added. By doing this, we ensure that the resulting reused ARG starts with the automaton transition passed to Algorithm 4.3. In case $q$ is an accepting state we also make $\hat{q}$ an accepting state. In Line 7, a quadruple $\langle \bar{A}, G_{P,\bar{A}}, W, f \rangle$ is chosen from the reachability information database. The choice is parametric, i.e., one can use different strategies to select a quadruple, e.g., one can actually compute the maximal possible reuse for each stored quadruple and select the optimal one, or one can apply computationally cheaper heuristics based on the structure of the TGA stored in the quadruples. In case a strategy would decide not to reuse any information, it can just select some entry in the database and choose $X$ such that no information reuse will happen. If the database is empty, the algorithm returns the ARG containing only the passed abstract state $s$ and adds the tuple $\langle s, (q, ((\ell, op, \ell'), \varphi), q')\rangle$ to the worklist again.

**Enumerating Test-Goal Automata.** Algorithm 4.1 assumes a fixed order on the sequence of TGA. This is not a requirement for our approach. Actually, one of the central features of our multi-goal reachability analysis approach is not having all TGA available in advance in contrast to our previous test input generator FSHELL, where initially all TGA are encoded into the program. Depending on the nature of the TGA, this can drastically reduce the scalability of FSHELL, e.g., TGA encoding specific subpaths of the program decrease the performance considerably (cf. Table 1 in Section 5). FSHELL and CPA/TIGER derive their TGA from concise coverage specifications given in FQL. Concise means, the size of the specification might be logarithmic in the number of resulting TGA. But, the more concise the specification is, the more sharing the TGA have. This enables us to skip whole sets of TGA where we can infer infeasibility from other TGA (because the reason of infeasibility is in the shared part of these TGA).

In general, we could compare all TGA with each other and order the queries corresponding to the size of the resulting sets $X$. One can use computationally cheaper heuristics based on structural properties of the TGA and the program, e.g., an analysis of program dominators can exploit hidden connections between TGA. In Algorithm 4.2, the call to Algorithm 4.3 is interleaved with a reachability analysis. The degree of information reuse greatly varies based on the precision the analysis provides and the strategies that are used for computing simulation relations. Therefore, the order of TGA might be dynamically changed based on the results of the reachability analysis. In our implementation, we used a fixed order based on similarity of TGA and the structure of the CFA (see discussion of Table 5 in Sect. 5). We leave a systematic investigation of dynamic orders as future work.

## 5 Experiments

We implemented the tool CPA/TIGER to evaluate the performance of our approach. We use the Java-based verification framework CPACHECKER in order to reuse standard model-checking technology, and integrate our concepts as configurable program analyses. To demonstrate the capabilities of the new implementation, we compare it to the existing FQL backends FSHELL 1 [19] and FSHELL 2 [21]. Both versions are based on the C bounded model checker CBMC [9] and were implemented in C++. FSHELL takes as input a C program and an FQL query. FSHELL 1 instruments a C program with automata derived from an FQL query, whereas FSHELL 2 encodes these automata directly into the SAT-formula representing the C program under scrutiny. Since FSHELL 1 and 2 are based on bounded model checking (BMC), they require an explicit specification of loop bounds for programs with unbounded loops. For a fair comparison one has to consider that FSHELL 2 is written in C++ whereas CPA/TIGER is written in Java and additionally proves infeasibility of test goals.

**Path Coverage in Programs with Unbounded Loops.** To compare the scalability of the three tools in the context of different path lengths, we studied a small (26 lines of code) locking/unlocking example. The lock/unlock happens inside a loop that is

**Table 1.** $n$-bounded path coverage on `locks_1`

| $n$ | Nr. of test goals | Loop bound | FShell 1 t[s] | FShell 2 t[s] | CPA/Tiger t[s] |
|---|---|---|---|---|---|
| 1 | 7 | 2 | .6 | .8 | 2.8 |
| 2 | 31 | 3 | 11. | .3 | 2.1 |
| 3 | 127 | 4 | 390. | .6 | 3.3 |
| 4 | 511 | 5 | - | 1.4 | 5.1 |
| 5 | 2047 | 6 | - | 15. | 9.6 |
| 6 | 8191 | 7 | - | 230. | 24. |
| 7 | 32767 | 8 | - | 4600. | 94. |

only bounded by an input parameter, hence unwinding limits had to be specified for FShell 1 and 2, as stated in the loop-bound column in Table 1. We use FQL queries `cover PATHS(ID, n)`, where $n$ ranges from 1 to 8, to specify $n$-bounded path coverage, i.e., these queries require test suites that cover each path in the program that repeats a CFA edge at most $n$ times. The test-goal automata that are generated from these queries are mostly deterministic. Consequently, the guidance by test-goal automata yields very efficient analyses, which makes CPA/Tiger scale much better than FShell 2, which cannot exploit the fact of a highly deterministic guidance. FShell 1 cannot complete the experiments for $n > 3$ within a time limit of 15 minutes. CPA/Tiger scales sublinearly with the number of test goals, whereas the BMC-based approaches of FShell 1 and 2 are not well-suited for such programs and queries.

**(Basic Block)$^2$ Coverage in Programs with Unbounded Loops.** (Basic block)$^2$ coverage requires every pair of basic blocks to be covered by some test case and thereby is a better approximation of (unbounded) path coverage than simple basic-block coverage. Table 2 compares FShell 2 and CPA/Tiger with respect to (basic block)$^2$ coverage and (basic block)$^3$ coverage. In both cases, for loop bounds of 20, CPA/Tiger outperforms FShell 2. The FQL query `cover @BASICBLOCKENTRY->@BASICBLOCKENTRY` expresses this coverage criterion.

**NT-Drivers.** Table 3 summarizes the comparison of FShell 2 and CPA/Tiger with respect to simplified NT-drivers and basic block, (basic block)$^2$, and nodes-(basic block)$^2$ coverage. Nodes-(basic block)$^2$ coverage is similar to (basic block)$^2$ cover-

**Table 2.** (Basic block)$^2$ and (basic block)$^3$ coverage

| Source | LOC | Nr. of test goals | | FShell 2 (LB 20) t[s] | | CPA/Tiger t[s] | |
|---|---|---|---|---|---|---|---|
| | | BB$^2$ | BB$^3$ | BB$^2$ | BB$^3$ | BB$^2$ | BB$^3$ |
| `locks_5` | 70 | 961 | 29791 | 22. | 720. | 7.2 | 120. |
| `locks_6` | 81 | 1296 | 46656 | 31. | 2100. | 8.6 | 280. |
| `locks_7` | 92 | 1681 | 68921 | 38. | 3300. | 12. | 540. |
| `locks_8` | 103 | 2116 | 97336 | 57. | 3800. | 14. | 1200. |
| `locks_9` | 114 | 2601 | 132651 | 64. | 6700. | 20. | 1300. |

**Table 3.** Basic block, (basic block)$^2$, and nodes-(basic block)$^2$ coverage on NT-Drivers

| Source | LOC | Nr. of Test Goals | | | FSHELL 2 (LB 3) t[s] | | | CPA/TIGER t[s] | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | BB | BB$^2$ | NBB$^2$ | BB | BB$^2$ | NBB$^2$ | BB | BB$^2$ | NBB$^2$ |
| kbfilter1 | 771 | 118 | 13924 | 33124 | 2.9 | 7.3 | 1500. | 7.9 | 36. | 200. |
| kbfilter2 | 1352 | 203 | 41209 | 100489 | 5.2 | 24. | 2700. | 14. | 97. | 770. |
| kbfilter3 | 1349 | 202 | 40804 | 99856 | 5.1 | 19. | 2500. | 18. | 95. | 770. |
| floppy1 | 1510 | 209 | 43681 | 123904 | 3.5 | 21. | 8600. | 25. | 140. | 1100. |
| floppy2 | 1529 | 209 | 43681 | 124609 | 3.8 | 20. | 11000. | 23. | 130. | 1200. |
| floppy3 | 2198 | 291 | 84681 | 237169 | 12. | 58. | 10000. | 46. | 310. | 3300. |
| floppy4 | 2198 | 291 | 84681 | 238144 | 13. | 59. | 11000. | 41. | 270. | 3300. |
| cdaudio1 | 2997 | 420 | 176400 | 499849 | 48. | 100. | 11000. | 95. | 740. | 11000. |
| cdaudio2 | 2992 | 417 | 173889 | 495616 | 26. | 110. | 9600. | 100. | 770. | 12000. |
| diskperf | 1477 | 202 | 40804 | 114244 | 4.7 | 27. | 15000. | 45. | 280. | 2000. |

**Table 4.** Achieved line coverage

| Source | CPA/TIGER | | FSHELL 2 | |
|---|---|---|---|---|
| | Line Coverage [%] | Test Cases | Line Coverage [%] | Test Cases |
| kbfilter1 | 88.46 | 26 | 88.85 | 25 |
| kbfilter2 | 90.83 | 48 | 90.83 | 49 |
| kbfilter3 | 90.36 | 48 | 90.36 | 48 |
| floppy1 | 91.37 | 26 | 91.37 | 21 |
| floppy2 | 89.40 | 28 | 89.81 | 22 |
| floppy3 | 93.52 | 61 | 93.94 | 51 |
| floppy4 | 93.67 | 60 | 93.67 | 51 |
| cdaudio1 | 86.65 | 77 | 87.28 | 69 |
| cdaudio2 | 86.42 | 75 | 87.04 | 69 |
| diskperf | 86.49 | 27 | 83.66 | 21 |

age but only requires one CFA node of each basic block. For basic block and (basic block)$^2$ coverage FSHELL 2 performs better, but, for nodes-(basic block)$^2$ coverage CPA/TIGER performs better. CPA/TIGER does not implement some optimizations for coverage criteria involving nodes which are already implemented for coverage criteria involving CFA edges. Preliminary experiments showed that we can expect a speed-up factor between 2 and 3 for NBB$^2$ coverage.

**Achieved Coverage.** Table 4 compares the coverage achieved by the test generators CPA/TIGER and FSHELL 2. Due to overapproximation by predicate abstraction (no bit-precision) there are cases in which CPA/TIGER misses a test case and does not achieve the same coverage as FSHELL 2. On the other hand there is also the last case in Table 4 where CPA/TIGER achieves a higher coverage than FSHELL 2 because of insufficient loop unwindings.

**Effects of Information Reuse.** In Table 5 we show the effects of the information reuse approach described in this paper. Column A gives the runtime of CPA/TIGER

**Table 5.** Effects of CPA/TIGER Optimizations exemplified on BB$^2$ Coverage

| Source | A t[s] | B t[s] | C t[s] | D t[s] | E t[s] | F t[s] |
|---|---|---|---|---|---|---|
| kbfilter1 | 36. | 34. | 60. | 72. | 800. | 48. |
| kbfilter2 | 97. | 110. | 230. | 300. | 3700. | 170. |
| kbfilter3 | 95. | 110. | 230. | 350. | 3800. | 160. |
| floppy1 | 140. | 160. | 300. | 290. | 14000. | 390. |
| floppy2 | 130. | 140. | 310. | 300. | 13000. | 360. |
| floppy3 | 310. | 380. | 920. | 1200. | >15000. | 670. |
| floppy4 | 270. | 350. | 920. | 1100. | >15000. | 610. |
| cdaudio1 | 740. | 1100. | 2400. | 3000. | >15000. | 2300. |
| cdaudio2 | 770. | 1100. | 2400. | 2900. | >15000. | 2300. |
| diskperf | 280. | 260. | 470. | 550. | >15000. | 1700. |

**A**: all optimizations enabled, **B**: without automaton optimization, **C**: without infeasibility propagation, **D**: inverted order of test goals, **E**: without ARG reuse, **F**: without predicate reuse

with all optimizations enabled. Besides the described reachability information reuse, CPA/TIGER also performs several other optimizations. Column B shows the runtime without a TGA minimization step, in column C, the runtime for a fast infeasibility propagation technique is given. By infeasibility propagation we mean that we can infer from the infeasibility of a TGA the infeasibility of other TGA in case we can show that the feasibility of these automata would imply the feasibility of the infeasible automaton (e.g., by exploiting domination information). Column D shows the effect of different enumeration orders for test-goal automata. The default enumeration strategy for test-goal automata is based on a breadth-first search of the underlying CFA – column D shows the runtime when inverting the order. We also did experiments using a depth-first enumeration of test goals but this strategy was not beneficial. The reuse of parts of the ARG causes the biggest impact in the performance of CPA/TIGER (column E). The last column (F) shows the runtime when not reusing predicates from earlier runs.

**Availability.** CPA/TIGER is free software and available from the CPACHECKER[2] web page. FSHELL[3] is available as binary for several platforms. The experimental data that are discussed in this article are available on the supplementary webpage http://cpachecker.sosy-lab.org/cpa-tiger.

## 6 Related Work

In our multi-goal reachability analysis approach we unify query-driven program testing (e.g., [20]) with monitor-based safety checking (e.g., [4, 26]). An extension of BLAST embeds path automata in a relational querying language, for specifying safety

[2] http://cpachecker.sosy-lab.org

[3] http://code.forsyte.de/fshell

verification problems [4, 5], but not test coverage, as a set of single-goal reachability queries. In a model-based setting, automata-based specifications of coverage were presented by Blom et al. [8] for test-case generation using the model checker UPPAAL [23]. In contrast to directed testing [12] —where 'directed' means directed by branching conditions and randomization— the directedness in our approach stems from user-defined coverage specifications which separate the control from algorithmic issues.

The test-input generator FSHELL [19,20] encodes many TGA into a formula describing a finite unrolling of a program and tries to determine which of the TGA are feasible. Due to the underlying BMC engine of FSHELL, FSHELL can not determine whether a TGA is infeasible (it can only achieve that for a specific unrolling), while CPA/TIGER is able to infer infeasibility as well. Furthermore, the architecture of CPA/TIGER enables the combination of different kinds of reachability analysis (under- and overapproximation as well as different abstraction techniques). We see FSHELL as an complementary test input generation technique which we plan to integrate into the CPA/TIGER architecture.

Our approach generalizes the concepts of summaries where usually pre- and post-conditions for specific code parts are encoded. In principle, the summarized code part can be arbitrary, but so far summaries were usually generated at function level [1,2,11]. In this work, we propose the shared behavior of automata as criterion for summarizing and storing reachability information. Albarghouthi et al. [1] divide a reachability query into subqueries in order to parallelize the computation of *one* reachability analysis whereas our approach enables a parallelized analysis of different TGA as well. At the moment CPA/TIGER supports a very simple parallelization strategy: it splits the sets of test goals and performs separate multi-goal reachability analyses for these subsets of TGA. But, except for test inputs, we do not exchange any information between these analysis runs at the moment. Our approach has one potential benefit: the single reachability analyses do not have to finish in order to make reachability information available to other queries. We consider [1, 2, 11] as orthogonal work which is relevant in the context of the reachability analysis we perform. Their summarization and parallelization approach is only of limited use when we want to reason about information reuse across different reachability queries.

Extreme model checking [16] investigates the possible information reuse across different versions of a program. They reexplore a subtree of an abstract reachability tree at abstract states where their abstraction was affected by code changes. In constrast to our approach, they fix the specification across different analysis runs. In multi-goal reachability analysis the specification changes but the code remains the same. In their approach, only the prefix of an abstract reachability tree can be reused. We can reuse more than a prefix of an abstract reachability graph since we do not deal with code changes but changes in the specifications. As soon as the specifications behave the same for some part of the program, we can reuse the respective reachability information.

Different notions of simulation are used in work minimizing specifications given as Büchi automata [10, 27]. The simulation relations are formulated between the original automaton and a minimized version of it. In simulation modulo transition sets, we capture the situation that only parts of a specification are simulated by another automaton. Furthermore, we deal with automata over finite words instead of the infinite word automata used in the above mentioned work. We use simulation relations to identify shared parts of two automata and not to minimize an automaton. CPA/TIGER nevertheless performs some simple minimization steps in order to speed-up the reachability analysis process and increase the precision of the simulation relation computation.

Existing model-checking technology has been applied to test-case generation in a number of other projects, such as Java PathFinder [28] or SAL2 [15]. Recently, model checking and testing were given a more uniform view, combining over-approximating and under-approximating analyses [14] and using interpolation [24]. For hardware designs, [18] presented a coverage-driven test generation approach. As in our approach they reason about reachability as well as unreachability of coverage states (for a fixed coverage criterion) but use different techniques to achieve that: they use BDDs for encoding the state space and underapproximate the set of unreachable coverage states.

## 7 Conclusion and Future Work

This paper presents an approach for reusing reachability information based on the automaton structure of reachability queries. We introduced simulation modulo a transition set as central concept for identifying shared information of queries. This notion enables us to dynamically query for reachability information in a way similar to databases.

Future research on multi-goal reachability analysis based on our approach of information reuse has a theoretical and a practical side: on the theory side, a deeper investigation of how temporal logics and automata can be used to infer more facts from existing reachability information is of interest; on the practical side, our approach enables the parallelization of the reachability-analysis step and the information-reuse reasoning step which we have not investigated in depth yet. Furthermore, our approach enables the use of offline storage of reachability information which we want to investigate to improve scalability to large programs. At the moment, CPA/TIGER integrates over- and underapproximation in a very simplistic way. Since FSHELL 2 and CPA/TIGER showed complementary strengths in the experiments, a deeper investigation on how to combine over- and underapproximations in reachability analyses is needed.

## References

1. Albarghouthi, A., Kumar, R., Nori, A.V., Rajamani, S.K.: Parallelizing Top-down Interprocedural Analyses. In: Proc. PLDI, pp. 217–228. ACM (2012)
2. Anand, S., Godefroid, P., Tillmann, N.: Demand-Driven Compositional Symbolic Execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
3. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: Proc. ICSE, pp. 326–335. IEEE (2004)
4. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST Query Language for Software Verification. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker BLAST. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)
6. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program Analysis with Dynamic Precision Adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008)

7. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate Abstraction with Adjustable-block Encoding. In: Proc. FMCAD 2010, pp. 189–198. FMCAD Inc. (2010)
8. Blom, J., Hessel, A., Jonsson, B., Pettersson, P.: Specifying and Generating Test Cases Using Observer Automata. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 125–139. Springer, Heidelberg (2005)
9. Clarke, E., Kröning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
10. Etessami, K., Wilke, T., Schuller, R.A.: Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. SIAM J. Comput. 34(5), 1159–1175 (2005)
11. Godefroid, P.: Compositional Dynamic Test Generation. In: Proc. POPL, pp. 47–54. ACM (2007)
12. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proc. PLDI, pp. 213–223. ACM (2005)
13. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated Whitebox Fuzz Testing. In: Proc. NDSS, pp. 151–166. The Internet Society (2008)
14. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional May-must Program Analysis: Unleashing the Power of Alternation. In: Proc. POPL, pp. 43–56. ACM (2010)
15. Hamon, G., de Moura, L.M., Rushby, J.M.: Generating Efficient Test Sets with a Model Checker. In: Proc. SEFM, pp. 261–270. IEEE (2004)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme Model Checking. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358. Springer, Heidelberg (2004)
17. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proc. POPL, pp. 58–70. ACM (2002)
18. Ho, P.H., Shiple, T., Harer, K., Kukula, J., Damiano, R., Bertacco, V., Taylor, J., Long, J.: Smart Simulation using Collaborative Formal and Simulation Engines. In: Proc. ICCAD, pp. 120–126. IEEE Press (2000)
19. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 209–213. Springer, Heidelberg (2008)
20. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-Driven Program Testing. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 151–166. Springer, Heidelberg (2009)
21. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How Did You Specify Your Test Suite. In: Proc. ASE, pp. 407–416. ACM (2010)
22. Holzer, A., Tautschnig, M., Schallhart, C., Veith, H.: An Introduction to Test Specification in FQL. In: Barner, S., Kröning, D., Raz, O. (eds.) HVC 2010. LNCS, vol. 6504, pp. 9–22. Springer, Heidelberg (2010)
23. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Int. J. Softw. Tools Technol. Transfer 1(1-2), 134–152 (1997)
24. McMillan, K.L.: Lazy Annotation for Program Testing and Verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)
25. Milner, R.: An Algebraic Definition of Simulation Between Programs. In: Proc. IJCAI 1971, pp. 481–489. Morgan Kaufmann Publishers Inc. (1971)
26. Šerý, O.: Enhanced Property Specification and Verification in BLAST. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 456–469. Springer, Heidelberg (2009)
27. Somenzi, F., Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)
28. Visser, W., Pasareanu, C.S., Khurshid, S.: Test Input Generation with Java PathFinder. In: Proc. ISSTA, pp. 97–107. ACM (2004)