Reachability Analysis of Procedural Programs with Affine Integer Arithmetic

Michael Luttenberger

Institute for Formal Methods in Computer Science, University of Stuttgart
`luttenml@informatik.uni-stuttgart.de`

Abstract. We present a tool for reachability analysis of procedural programs whose statements consist of affine equations and inequations. The algorithms uses finite automata to finitely represent possibly infinite sets of both stack contents and memory valuations. We illustrate our program on some examples and compare it to Moped, a model checker for pushdown systems.

1 Introduction

In [1] Esparza and Schwoon presented Moped, a model-checker for boolean programs. A boolean program consists of a set of (possibly recursive) procedures acting on boolean variables. Moped translates boolean programs into symbolic pushdown systems, a combination of pushdown systems and binary decision diagrams (BDDs, cf. [2]). Pushdown systems (a formalism closely related to pushdown automata) are used to model the program's control flow. In particular, the pushdown stack is used to store the return addresses of the procedures whose execution has not finished yet; procedure calls push a new return address into the stack, while return instructions pop the address at the top of the stack. BDDs are used to succinctly represent memory transformations. For instance, an assignment $x := f(x, y)$, where f is a boolean function, induces a memory transformation given by the relation $\{(x, y, x', y') \mid x' = f(x, y), y' = y\}$, and this relation is encoded as a BDD. Moped's core algorithm computes the (possibly infinite) set of reachable configurations of a given boolean program. This possibly infinite set turns out to be regular, and Moped represents it as a *symbolic finite automaton*, a finite automaton whose transitions are labeled by BDDs.

In this paper we extend Moped so that it can analyze *affine integer programs*. An affine integer program is a set of (possibly recursive) procedures acting on *integer* variables, and such that all memory transformations are affine relations on integers. For the succinct representation of affine relations we use a subclass of finite automata called *number decision diagrams* (NDDs) [3–6]. NDDs are capable of representing the set of solutions of an arbitrary Presburger formula [3, 7]. We extend Moped's core algorithm for the computation of the reachable configurations. Since affine integer programs are Turing-powerful, the algorithm is not guaranteed to terminate, but, when it does, it returns a finite representation of all reachable configurations in the form of a finite automaton whose transitions are labeled by NDDs.

The paper is structured as follows. Section 2 introduces basic notations, pushdown systems and NDDs. Section 3 shows how to model procedural programs as pushdown systems and memory transformations as NDDs. The algorithm for calculating the reachable configurations is presented in Section 4. Section 5 gives a short overview of our implementation and some experimental results. We close with a discussion of future work.

2 Preliminaries

This section serves for introducing the notation used in the following and provides the basic definitions and results our work is based on. We start with our basic notation.

2.1 Notation

Let $\mathbb{N}$ denote the set of natural numbers including 0, $\mathbb{Z}$ the set of integers, and $\mathbb{R}$ the set of real numbers. We use $\langle\cdot,\cdot\rangle$ to denote the euclidean scalar-product on $\mathbb{R}^n$. As usual, for Σ a set Σ^* denotes the set $\bigcup_{n\in\mathbb{N}} \Sigma^n$ of finite *words* over Σ, especially ε denotes the empty word. Further for $n \in \mathbb{N}$ we use $\Sigma^{<n}$ for the set $\bigcup_{k\in n} \Sigma^k$. For $u \in \Sigma^*$ let $|u|$ denote the length of u. We refer to the single letters of a word $w \in \Sigma^*$ by w_i, for $1 \leq i \leq |u|$. Let A, B, C be sets. For a relation $R \subseteq A \times B$ and $a \in A$, $b \in B$ we write aRb for $(a, b) \in R$, aR for $\{b \in B \mid aRb\}$, and Rb for $\{a \in A \mid aRb\}$. We extend this notation for subsets of A, respectively B, e.g. for $M \subseteq A$ we have $MR = \{b \in B \mid \exists a \in M : aRb\}$. For a second relation $R' \subseteq B \times C$ we write $R \circ R'$ or RR' for the relational product $\{(x, z) \in A \times C \mid \exists y \in B : xRy \wedge yR'z\}$. Especially, if $R \subseteq A \times A$ we write $R^0 := \mathrm{Id}_A := \{(a, a) \mid a \in A\}$, $R^n := RR^{n-1}$, and $R^* = \bigcup_{n\in\mathbb{N}} R^n$, i.e. the reflexive, transitive closure of R.

2.2 Pushdown Systems

For modeling the control flow of a recursive program we use the concept of pushdown systems. This approach has been used very successfully for example in [8] and [9].

Definition 1. *Pushdown system (PDS). A* pushdown system (PDS) $\mathcal{P}$ *is given by a tuple* $(Q, \Gamma, \hookrightarrow)$ *where* Q *and* Γ *are finite, and wlog. disjoint sets, and* $\hookrightarrow \subseteq (Q \times \Gamma) \times (Q \times \Gamma^{<3})$*. The elements of* Q*, respectively* Γ*, are called* control locations*, respectively* stack symbols*. We call* $q\gamma \hookrightarrow q'u$ *a* transition rule*, more exactly we call it a* push rule *if* $|u| = 2$*, a* pop rule *if* $u = \varepsilon$*, and a* simple rule *if* $|u| = 1$*. We call* $\mathcal{C} := Q \times \Gamma^*$ *the set of all* control configurations*.*

We extend the relation $\hookrightarrow$ to $\mathcal{C} \times \mathcal{C}$ in the natural way, i.e. $q\gamma\omega \hookrightarrow q'u\omega$ iff $q\gamma \hookrightarrow q'u$, especially $q\varepsilon \hookrightarrow = \emptyset$. For $C \subseteq \mathcal{C}$, we write also $\mathrm{pre}^*(C)$ for $\hookrightarrow^* C$ and $\mathrm{post}^*(C)$ for $C \hookrightarrow^*$.

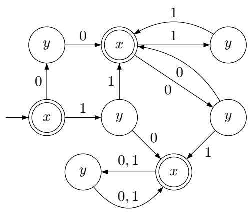


Fig. 1. An NDD representing the set of integers $\{(x, y) \in \mathbb{Z}^2 \mid x \leq y\}$

2.3 Number Decision Diagrams

Inspired by the work of Wolper and Boigelot ([10, 3, 4]), we use a subclass of finite automata, called number decision diagrams (NDD), for modeling the memory relations between two control locations of a program. Before giving a formal definition for NDDs, we need to recall some basics about the two-complement representation of the integers as strings over $\{0, 1\}$. For $\sigma \in \{0, 1\}^*$, let $\rho(\sigma) = 0$, if $\sigma = \varepsilon$, otherwise $\rho(\sigma) = 2^{|\sigma|-1} \cdot (-\sigma_1 + \sum_{k=2}^{|\sigma|} \sigma_k 2^{-k})$. Obviously, for $\sigma \neq \varepsilon$ $\rho(\sigma) = \rho(\sigma_1^* \sigma)$ holds. This simple fact is used for extending the two-complement representation from $\mathbb{Z}$ to $\mathbb{Z}^n$. For $\sigma \in \{0, 1\}$ a string of length $|\sigma| \mod n = 0$, we partition σ into the bit-vectors $\sigma_{[l]} := (\sigma_{ln}, \ldots, \sigma_{(l+1)n-1})$, for $1 \leq l \leq \frac{|\sigma|}{n}$, and set $\rho_n(\sigma) = 2^{\frac{|\sigma|}{n}-1}(-\sigma_{[1]} + \sum_{k=2}^{\frac{|\sigma|}{n}} \sigma_{[k]} 2^{-k})$. For example, we have $\rho_2^{-1}((3, 1)) = (00)^* 1011$.

Definition 2. *Number decision diagram (NDD). An NDD of dimension $n \in \mathbb{N}$ is a finite automaton $\mathcal{A} = (Z, \{0, 1\}, \delta, I, F)$ with the following restrictions. For $\sigma \in \{0, 1\}^*$ are string accepted by $\mathcal{A}$, it has to hold that (1) $|u| \in n\mathbb{N}$, and (2) all strings $\sigma' \in \{0, 1\}^*$ with $\rho_n(\sigma) = \rho_n(\sigma')$ are accepted by $\mathcal{A}$, too. (see fig. 1). We denote with $\mathcal{Z}(\mathcal{A})$ the set of integers represented by an NDD $\mathcal{A}$.*

For $n \in \mathbb{N}$, $a \in \mathbb{Z}^n$, $c \in \mathbb{Z}$, constructing an NDD accepting $\rho_n^{-1}(\{z \in \mathbb{Z}^n \mid \langle z, a\rangle \leq c\})$ is done similar to calculating weakest preconditions. We refer the reader to [4]. Note that in order to create an NDD for an (in)equation, for example $x \leq y$, one has to fix a *variable order*, i.e. a total order on the variables used. This order simply determines when an NDD reads a bit for a given variable. As in the case of BDDs ([2]), the size of an NDD for a given linear constraint might vary exponentially with the chosen variable order[1].

3 Modeling Recursive Programs with Integer Variables

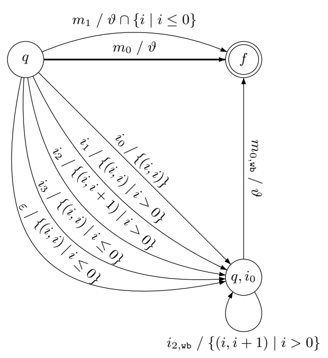
We restrict ourselves to the following class of recursive programs: (1) every statement is either an assignment, an if-statement, a while-loop, a procedure call, or a return-statement; (2) every variable has $\mathbb{Z}$ as domain; (3) every assignment is an affine combination of the variables in scope, e.g x := 5∗x + 3∗y + 2; (4)

[1] The standard example is the formula $\wedge_{i=1}^n x_i = x_i'$.

```
main() {
int x;
m0 : infinity(x);
m1 : return;
}
```

```
infinity( int i ) {
i0 : if( i > 0 ) {
i1 :     i+=1;
i2 :     infinity(i);
     }
i3 : return;
}
```

$$
\begin{array}{lll}
qm_0 & \hookrightarrow qi_0 m_{0,\mathtt{wb}} & : i' = x \\
qm_{0,\mathtt{wb}} & \hookrightarrow qm_1 & : x' = i \\
qi_0 & \hookrightarrow qi_1 & : i > 0 \wedge i' = i \\
qi_0 & \hookrightarrow qi_3 & : i \leq 0 \wedge i' = i \\
qi_1 & \hookrightarrow qi_2 & : i' = i + 1 \\
qi_2 & \hookrightarrow qi_0 i_{2,\mathtt{wb}} & : i' = i \\
qi_{2,\mathtt{wb}} & \hookrightarrow qi_3 & : i' = i \\
qi_3 & \hookrightarrow q\varepsilon & : i' = i
\end{array}
$$



$m_1 \;/\; \vartheta \cap \{i \mid i \leq 0\}$
$m_0 \;/\; \vartheta$
q
f
$i_0 \;/\; \{(i, i)\}$
$i_1 \;/\; \{(i, i) \mid i > 0\}$
$i_2 \;/\; \{(i, i+1) \mid i > 0\}$
$i_3 \;/\; \{(i, i) \mid i \leq 0\}$
$\varepsilon \;/\; \{(i, i) \mid i \leq 0\}$
$m_{0,\mathtt{wb}} \;/\; \vartheta$
q, i_0
$i_{2,\mathtt{wb}} \;/\; \{(i, i+1) \mid i > 0\}$

Fig. 2. We start with $\mathcal{A}_0$ consisting of the single transition $q \xrightarrow[\vartheta]{m_0} f$. Here, ϑ is an NDD with $\mathrm{Sig}(\vartheta) = \{x'\}$, describing the initial values of x. In general, we won't be able to reach the fixed point $\mathcal{A}_{\mathrm{post}^*}$ when using $[\![\cdot]\!]$ in rule (3), as we will discover with each iteration of the saturation process a new value the procedure infinity is called with. When using $[\cdot]$ instead, we can calculate $\mathcal{A}_{\mathrm{post}^*}$. Because of the weight of the transition $q \xrightarrow{m_1} f$, we know that the program only terminates for non-positive values of x.

every condition in an if-statement or while-loop is a boolean combination of affine (in)equations, e.g. x $\leq$5 $\wedge$y $\neq$3. For the sake of clarity, we further make the following assumptions: every end of a procedure is explicitly marked by a return-statement; every statement of the program is uniquely labeled; the program only uses procedures; parameters are passed by reference; only variables are passed as parameters. Finally, we make the assumption that no global variable is passed directly as parameter in a procedure call. We need this assumption as we model memory transformations locally. Thus, we do not know if a parameter variable is currently used as a reference to some global variable[2]. It is easy to see that this model is already Turing-powerful as a straight forward reduction shows that two-counter machines can be simulated. Hence, reachability analysis is in general only semi-decidable, see e.g. also [3].

We will use an extension of pushdown systems, referred to as *weighted pushdown systems (wPDS)* (cf. [11]), to model a given program. Here, a wPDS is obtained from a PDS by assigning each rule a weight of a previously fixed set. In our case, we will model the control flow of a given program as a PDS and then assign each rule of this PDS an NDD which represents the memory transformation of that statement which is modeled by the respective rule. This way, we will obtain a PDS weighted by NDDs. This wPDS has exactly one control location. Its stack alphabet consists exactly of the labels of the given program plus so-called write-back labels. Fig. 2 gives a preview of this. Due to space limitations, we only mention that this model allows us to incorporate safety specifications as usual

[2] As we are considering only single threaded programs, this boils down to using call-by-copy-restore, aka call-by-value-result as evaluation strategy.

by intersecting the wPDS, constructed above, with a weighted finite automaton representing the safety properties. Here, a transition $z \xrightarrow{l} z'$ of such an specification automaton, reading a program label l, additionally consists of an NDD representing an boolean combination of affine inequations[3] over the variables in scope at l. We first turn to how memory transformations are modeled.

3.1 Modeling Memory Transformations

We use NDDs for representing the memory transformation corresponding to a statement. Let $\mathcal{V}$ be the set of variables used in the program. Then we introduce the sets of variables $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$, $\mathcal{V}_s = \{v_s \mid v \in \mathcal{V}\}$, and $\mathcal{V}_s' = \{v_s' \mid v \in \mathcal{V}\}$. $v \in \mathcal{V}$ is used for representing the value of the program variable v before executing a memory transformation, while $v' \in \mathcal{V}'$ represents its value after the applying the transformation. Similarly, v_s and v_s' are used for the value of v saved on the top of the stack. Finally, we set $\hat{\mathcal{V}} = \mathcal{V} \cup \mathcal{V}' \cup \mathcal{V}_s \cup \mathcal{V}_s'$. Fix any total ordering $\prec$ on $\hat{\mathcal{V}}$.

We extend an NDD with a signature consisting of those variables read and written by the memory transformation. We assume that these signatures are sorted increasingly with respect to $\prec$. Variables not "touched" by a transformation do not need to be included in the signature. For example, the statement x:= 2 $*$ y is translated into an NDD representing the relation $x' = 2y \wedge y' = y$ having the signature (x', y, y') assuming $x \prec y \prec y'$. Given an NDD Θ, we use $\mathrm{Sig}(\Theta) \subseteq \hat{\mathcal{V}}$ to denote the set of variables induced by the signature of Θ.

3.2 Modeling Programs

Assignments Consider any assignment, e.g. l_1 :x := 5$*$x + 3$*$y + 2; l_2 :.... In order to describe the memory transformation associated with this assignment, we create a simple rule $ql_1 \hookrightarrow ql_2$, and extend this rule by an NDD Θ representing the relation $x' = 5x + 3y + 2 \wedge y' = y$. We extend Θ by the signature (x, y, x', y') assuming $x \prec y \prec x' \prec y'$.

If-Statements An if-statement l_1 :**if**(β) { l_2 :...} **else** { l_3 :...} is translated into to the simple rules $ql_1 \hookrightarrow ql_2$ and $ql_1 \hookrightarrow ql_3$. Let $\mathcal{X}$ be the set of variables appearing in β. We then create two NDDs Θ, Θ' with $\mathrm{Sig}(\Theta) = \mathrm{Sig}(\Theta') = \{x, x' \mid x \in \mathcal{X}\}$ representing $\beta \wedge \bigwedge_{x \in \mathcal{X}} x' = x$, and $\neg\beta \wedge \bigwedge_{x\mathcal{X}} x' = x$, respectively. Finally, we assign the rule $ql_1 \hookrightarrow ql_2$ ($ql_1 \hookrightarrow ql_3$) Θ (Θ') as weight.

While-Loops We proceed similarly as in the case of if-statements.

Procedure Calls We now consider the call l_1 :proc(x_1, ..., x_n); l_2 :... of the procedure proc(p_1, ..., p_n). Here, x_1 to x_n are variables local[4] to the current procedure whereas p_1 to p_n are variables local to proc. Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ and $\mathcal{V}^{\mathrm{gob}}$ the

[3] Or more generally, a Presburger formula.

[4] See the assumptions made at the beginning of section 3.

set of global variables. Note, by assumption $\mathcal{X} \cap \mathcal{V}^{\text{gob}} = \emptyset$. Then, the passing of parameters is described by the relation $(\bigwedge_{i=1}^{n}(p'_i = x_i)) \wedge \bigwedge_{x \in \mathcal{V}^{\text{loc}} \setminus \mathcal{X}} x'_s = x$. Let $\Theta_{\texttt{call}}$ be an NDD representing this relation with the respective signature. The write-back of variables is described similarly by the relation $\bigwedge_{i=1}^{n}(x'_i = p_i) \wedge \bigwedge_{x \in \mathcal{V}^{\text{loc}} \setminus \mathcal{X}} x' = x'_s$. Let $\Theta_{\texttt{wb}}$ be an NDD representing the later relation[5]. Finally, we add the the pushdown rules $l_1 \xhookrightarrow[\Theta_{\texttt{call}}]{} l_{\texttt{proc}} l_{1,\texttt{wb}}$ and $l_{1,\texttt{wb}} \xhookrightarrow[\Theta_{\texttt{wb}}]{} l_2$ where $l_{\texttt{proc}}$ is the label of the first statement occurring in the procedure proc and $l_{1,\texttt{wb}}$ is a previously unused label.

Return Statements For a return statement l :**return**, we simply add the pop rule $ql \to q\varepsilon$. As we are assuming that the given program uses only procedures where parameters are passed by reference, we have to extend this rule by the following NDD. Let $\mathcal{X}$ be the set of all variables local to the procedure l belongs except those variables used as parameters. We then extend the rule $ql \to q\varepsilon$ by an NDD Θ with $\text{Sig}(\Theta) = \mathcal{X}$, representing the set $\mathbb{Z}^{\#\mathcal{X}}$. This tells our extend-operation, introduced in the next section, that we want to project all variable values which do not need to be returned by the call-by-copy-result mechanism.

4 Reachability Analysis

For reachability analysis, we use the algorithms presented in [11]. There, wPDSs are considered whose transitions are weighted by elements of a bounded semiring. As the weights do not have to form an semiring necessarily for the algorithm to work, we skip the definition of it, referring the reader to [11] instead. Still, the boundedness of the semiring is used to show termination in general. In our case, these algorithms will only allow us to approximate the set of reachable configurations in general. For instantiating these algorithms, we need two operations over NDDs, referred to as combine- and extend-operations there. The combine-operator is used for updating, in our case, the memory configurations encountered in the exploration process for a given control configuration. This means, the combine-operations simply has to return an NDD representing the union of two given NDDs. The extend-operations, here, is used for composing two NDDs representing memory relations[6]. With respect to our model, this extend-operation has to preserve the values of variables not touched by a memory transformation and has to take care of projecting variables, e.g. in the case of a return statement. We, further, need an restriction operator over NDDs and the concept of $\mathcal{P}$-automaton. We will refer to this restriction operator also as

[5] The need for the variables x_s and x'_s simply arises by the fact that the called procedure proc might be the current procedure. This models the process of saving the current activation record on the stack. Note, that we do not need to save every local variable, usually a small subset suffices, e.g. in the case of quicksort, where we only have to remember the sorting range for the second recursive call.

[6] Where memory relation refers to both transformations and configurations.

call-operator[7], as it is used in the forward reachability analysis for specifying the parameters for which a procedure should be evaluated. We give an example of this below. The $\mathcal{P}$-automata, on the other hand, are used for representing both the set of initial configurations, from which our reachability analysis starts, and the set of configurations reached after a finite number of iterations. With respect to our model, we have to apply some restrictions to these $\mathcal{P}$-automata, as discussed later. The next paragraph describes our extend-operator in greater detail. We then introduce the call-operator and $\mathcal{P}$-automata. Finally, we instantiate the algorithm for forward reachability from [11]. On the basis of this algorithm, we discuss some aspects of the choice of the restriction operator used.

Extend-Operation As we have extended the NDDs by signature describing the variables accessed by the represented relation, we can associate each state q of an NDD Θ with a variable $\mathrm{var}(q) \in \mathrm{Sig}(\Theta)$, and, hence, can assign to q the rank $\mathrm{rk}(q)$ of its variable $\mathrm{var}(q)$ w.r.t. $\prec$[8]. With this at hand, we can define the extend-operation $\otimes$ for two NDDS, say L and R, by the following adapted product-automaton construction. We start with the state pair (q_L^0, q_R^0) where q_L^0 (q_R^0) the initial state of L (R). In a state (q_L, q_R), the variable of the state having the lower rank determines what to be done next. With respect to $x \in \mathcal{V} \cup \mathcal{V}_s$, we discern four cases:

(1) $x' \in \mathrm{Sig}(L) \wedge x \in \mathrm{Sig}(R)$ (e.g. $L \equiv x' = 5x$, $R \equiv y' = x + 1 \wedge x' = x$).
(2) $x' \in \mathrm{Sig}(L) \cap \mathrm{Sig}(R) \wedge x \notin \mathrm{Sig}(R)$ (e.g. $L \equiv x' = 3$, $R \equiv x' > 5$).
(3) $x' \notin \mathrm{Sig}(L) \wedge x \in \mathrm{Sig}(L) \cap \mathrm{Sig}(R)$ (e.g. $L \equiv y' = x$, $R \equiv z' = x$).
(4) $x, x' \notin \mathrm{Sig}(L) \cap \mathrm{Sig}(R)$.

We assume $x' \prec x$ in the following, the other case being symmetrical. Further, let $b, b' \in \{0, 1\}$. Now, in case (1), while exploring L and R in parallel, assume we have reached a state pair (q_L, q_R) with $\mathrm{var}(q_L) = x'$ and $\mathrm{rk}(q_L) < \mathrm{rk}(q_R)$. For a transition $q_L \xrightarrow{b} q'_L$, we add a transition $(q_L, q_R) \xrightarrow{\varepsilon} (q'_L, q_R, [x' = b])$ remembering that L read an b for x'. As $x' \prec x$ and $x \in \mathrm{Sig}(R)$, we eventually reach a state pair $(q''_L, q''_R, [x' = b])$ with $\mathrm{var}(q''_R) = x$ and $\mathrm{rk}(q''_R) \leq \mathrm{rk}(q''_L)$. For a transition $q''_R \xrightarrow{b} q'''_R$, we add the transition $(q''_L, q''_R, [x' = b]) \xrightarrow{\varepsilon} (q''_L, q'''_R)$[9]. In case (2), assume we have reached a state pair (q_L, q_R) with $\mathrm{var}(q_L) = \mathrm{var}(q_R) = x'$. For any transitions $q_L \xrightarrow{b'} q'_L$ and $q_R \xrightarrow{b} q'_R$, we add a transition $(q_L, q_R) \xrightarrow{b} (q'_L, q'_R)$, i.e. we simply forget the values specified by L for x', taking those specified by R instead. Should we eventually reach a state pair (q''_L, q''_R) with $\mathrm{var}(q''_L) = x$ and $\mathrm{rk}(q''_L) < \mathrm{rk}(q''_R)$, we will simply copy the behavior of L, i.e. for a transition $q''_L \xrightarrow{b} q'''_L$, we add $(q''_L, q''_R) \xrightarrow{b} (q'''_L, q''_R)$ in the product automaton. For case (3), when reaching a state pair (q_L, q_R) with $\mathrm{var}(q_R) = x'$ and $\mathrm{rk}(q_R) <$

[7] This call-operator is not explicitly mentioned in [11]. Still, in the library description of [12], it is referred to as quasi-one.

[8] The rank of $v \in \hat{\mathcal{V}}$ is as usual $\#\{v' \in \hat{\mathcal{V}} \mid v' \prec v\}$.

[9] In the case that x' precedes x directly w.r.t. $\prec$, we can merge these two states, reducing the number of intermediate states.

$\mathrm{rk}(q_L)$, we, again, simply copy the behavior of R. When reaching a state pair (q''_L, q''_R) with $\mathrm{var}(q''_L) = \mathrm{var}(q''_R) = x$, we synchronize both automata by adding a transition $(q''_L, q''_R) \xrightarrow{b} (q'''_L, q'''_R)$ for every pair of transitions $q''_L \xrightarrow{b} q'''_L$ and $q''_R \xrightarrow{b} q'''_R$. Finally, in case (4), we have to copy the behavior of the respective automaton, again.

This extend-operation slightly differs from the usual approach as known from BDDs. This allows us to dispense with explicitly encoding in the automaton that variables not touched by the represented transformation have to be copied. Hence, reducing the size of the NDDs representing the memory transformations[10].

Call-Operator As we will see in the following, we need an operator for specifying for which values a procedure should be evaluated. Let Θ be any NDD. We denote with $\Theta|_{\mathcal{V}'}$ the NDD resulting from projecting all states of Θ associated with variables in $\mathrm{Sig}(\Theta) \setminus \mathcal{V}'$, i.e. $\Theta|_{\mathcal{V}'}$ represents only those values of non-stack variables output by Θ. Then, χ is called a *call-operator*, if $\chi(\Theta)$ represents the identity relation $\bigwedge_{x' \in \mathrm{Sig}(\Theta) \cap \mathcal{V}'} x = x'$ restricted to some superset of $\mathcal{Z}(\Theta\ |_{\mathcal{V}'})$. Especially, let $[\![\cdot]\!]$ denote the call-operator restricted to $\mathcal{Z}(\Theta\ |_{\mathcal{V}'})$ and $[\cdot]$ the unrestricted identity. Both operators, $[\![\cdot]\!]$ and $[\cdot]$, can be implemented similarly to the extend-operations.

$\mathcal{P}$-automaton For representing the initial states and reachable states, we further need the concept of $\mathcal{P}$-automata.

Definition 3. *$\mathcal{P}$-automaton. For a given PDS $\mathcal{P}$, a* $\mathcal{P}$-automaton *$\mathcal{A}$ is a tuple $(Z, \Gamma, \delta, Q, F)$ where Z is the finite set of states with $P \subseteq Z$ where P is the set of control locations of $\mathcal{P}$ and also the set of* initial states *of $\mathcal{A}$. $F \subseteq Z$ is called the set of* final states. *$\delta \subseteq Z \times \Gamma \times Z$ is the transition relation of $\mathcal{A}$.*

As usual, $\mathcal{A}$ can be identified with a directed labeled graph with nodes Z having an edge labeled by $\gamma \in \Gamma$ from z to z' if and only if $(z, \gamma, z') \in \delta$. A configuration $q\omega \in \mathcal{C}$ of $\mathcal{P}$ is accepted by $\mathcal{A}$ iff there exists a path from q to a final state $f \in F$ labeled by ω in the graph introduced by $\mathcal{A}$. A $\mathcal{P}$-Automaton *represents* exactly those configurations $\mathcal{C}(\mathcal{A})$ of $\mathcal{C}$ which it accepts. A set $C \subseteq \mathcal{C}$ of configurations is called *regular* iff there exists an $\mathcal{P}$-Automaton representing C. With respect to our model, we have to restrict the set of control configurations an $\mathcal{P}$-Automaton $\mathcal{A}$ may represent. Let q be the single, implicit control location of the extended PDS representing our program[11]. We then require that (1) $\mathcal{C}(\mathcal{A}) \subseteq \{q\}(\Gamma \setminus \Gamma_{\mathtt{wb}})\Gamma^*_{\mathtt{wb}}$ and (2) that the procedure, the label ω_{i-1} belongs to, is called by the statement at l, for $q\omega \in \mathcal{C}(\mathcal{A})$ and $l_{\mathtt{wb}} = \omega_i$ (with $i > 0$). Wlog., we may assume that $\mathcal{A}$ does not possess any ε-transitions. Then, every transition leaving the initial state q reads labels corresponding to original statements of the program,

[10] We are currently working on a second version of our NDD library, inter alia, aiming at dispensing with all explicit copy statements.

[11] More exactly, q will be the initial state of the PDS, if safety properties are incorporated in the PDS.

while all other states of $\mathcal{A}$ read write-back labels. We extend a transition $q \xrightarrow{l} z$ by an NDD ϑ with $\mathrm{Sig}(\vartheta) = \{v' \mid v \in \mathcal{X}\}$ representing memory valuations for the variables in $\mathcal{X}$. Here, $\mathcal{X}$ denotes the set of variables in scope at l. Similarly, a transition $z \xrightarrow{l_{\mathtt{wb}}} z'$ is extended by an NDD representing valuations of the variables saved on the stack before the call at l with the respective signature. We write $\xrightarrow[\vartheta]{l}$ for these extended transitions.

Forward reachability With these concepts at hand, we can instantiate the post*-algorithm presented in [11] resulting in the following algorithm (here, we chose to use $[\![\cdot]\!]$ as call-operator):

Algorithm 1. *Let $\mathcal{A}_0$ be any such extended $\mathcal{P}$-automaton satisfying the above requisites. For $i \geq 0$, $\mathcal{A}_{i+1}$ results from $\mathcal{A}_i$ by either introducing new or updating existing transitions of $\mathcal{A}_i$ as specified by these rules: Let $q \xrightarrow[\vartheta]{l} z$ be a transition of $\mathcal{A}_i$.*

(1) if $ql \xhookrightarrow[\Theta]{} q'l'$, update the transition $q \xrightarrow{l'} z$ with the weight $\vartheta \otimes \Theta$.
(2) if $ql \xhookrightarrow[\Theta_{ret}]{} q'\varepsilon$, update the transition $q \xrightarrow{\varepsilon} z$ with the weight $\vartheta \otimes \Theta$.
(3) if $ql \xhookrightarrow[\Theta_{call}]{} q'l'l_{wb}$, introduce, if necessary, the state (q', l'), then update the transitions $q \xrightarrow{l'} (q', l') \xrightarrow{l_{wb}} z$ with the weights $[\![\vartheta \otimes \Theta_{\mathtt{call}}]\!]$ and $\vartheta \otimes \Theta_{\mathtt{call}}$, respectively.
(4) if $q \xrightarrow[\Theta]{\varepsilon} z$ and $z \xrightarrow[\vartheta]{l_{wb}} z'$, update the transition $q \xrightarrow{\varepsilon} z'$ with $\vartheta \otimes \Theta$.

If $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ differ in any edge weight, repeat the above process.

We give a short explanation what these rules do in order to motivate the necessity of the call-operator. Rules (1) and (2) correspond to intraprocedural steps and are self-explainatory. The third rule matches a call to the procedure with entry label l'. The transition $q \xrightarrow{l'} (q', l')$ is weighted by the NDD $[\![\vartheta\Theta_{\mathtt{call}}]\!]$, i.e. in general the identity over the global variables and parameters of the called procedure restricted to the values of these variables encountered up to now in the exploration process. We need to use a call-operator for rule (3), as otherwise we would have no possibility to discern between different calls to the same procedure. The transition $(q', l') \xrightarrow[\vartheta\Theta_{\mathtt{call}}]{l_{\mathtt{wb}}} z'$, therefore, is used to remember the values of the variables saved on the stack right before the call, of the global variables and the parameters of the called procedure. Rule (4), finally, besides reintroducing the information about the variables saved on the stack before the call, evaluates, in general, the procedure corresponding to l' on the values specified by ϑ. Here, our composition operator ensures that the values of the stack variables are copied from ϑ to $\vartheta\Theta$. For approximating the set of reachable configurations, any call-operator might be used. We have chosen to use $[\![\cdot]\!]$ in rule (3), as this evaluates the corresponding procedure only on those values the procedure is eventually called with. Hence, if $\mathcal{A}_i$ contains a transition $q \xrightarrow{l} z$, we know that starting from the configurations represented by $\mathcal{A}_0$, we can reach the statement associated with l. This does not need to hold when using, e.g., $[\cdot]$. In the case of safety properties, we are mainly interested in whether any control configuration starting with $q'l'$ is reachable. Therefore, $[\![\cdot]\!]$ seems to be the natural choice. Still, in certain cases choosing $[\cdot]$ instead of $[\![\cdot]\!]$ is feasible, and may even allow to proof non-termination (cf. fig. 2).

5 Implementation and Experimental Results

The creation, composition, and merging of NDDs is done by our own NDD library. Besides of the signatures needed by our approach, the library further supports reference counting and removal of redundant states. The latter technique is usually used for BDDs. There, a state is redundant iff it has exactly one successor. This means, such a state simply carries no information regarding the decision process whether a word is to be accepted or not. We adopted this in our NDD library for all but those states associated with the first of the signature. A drawback of removing redundant states is that one has to remember explicitly which state is associated to which variable of the signature. As we are mainly concerned with memory usage, we sort the states with respect to the rank of the variable they are associated with. This allows us only to store the information about which range of states is associated with a given variable of the signature, limiting the memory overhead to the size of the signature which is usual negligible compared to the size of the automaton[12]. As stated previously, our scheme for composing two NDDs L and R differs slightly from the usual approach of renaming, intersecting, and projecting, as known, e.g., from BDDs. In our approach, renaming is done implicitly and on the fly, while exploring the two automata being composed. Therefore, we can dispense with the intermediate automata resulting from renaming. We further try to minimize the number of states associated to project variables. For example, considering the first case discussed in the paragraph regarding the extend-operations, if the ranks of the variables $x, x' \in \hat{\mathcal{V}}$ only differ by one, we do not need to introduce the intermediate state $(q'_L, q_R, [x' = b])$.

With this NDD library at hand, we can directly instantiate the wPDS library by Stefan Schwoon [12] for implementing the algorithm described in sect. 4. In the next subsection, we give some experimental results.

5.1 Experiments

We use quicksort to illustrate the usefulness of our approach. Our model of quicksort is based on the algorithm depicted in 3 taken from [13]. This algorithm is faulty[13] as the loop for decreasing `j` does not terminate, if the pivot `v` is the

[12] Of course, sorting and storing the states in such a way might lead to an increased cache-miss ratio. We have a second version of our tool under development currently, taking a slightly different approach.

[13] Mentioned in [13].

```
int a[];
qs( int l, int r ) {
  if(r <= l) return;
  int i,j,v;
  i = l-1; j = r; v = a[r];
  while(true) {
    do {++i;} while(a[i] < v);
    do {--j;} while(a[j] > v);
    if(i >= j) break;
    swap(a[i], a[j]);
  }
  swap(a[i], a[j]);
  qs(l, i-1);
  qs(i+1, r);
}
```

	setup		post*	
N	time	memory	time	memory
	faulty version			
3	< 1s	1.3M	2s	2.4M
4	< 1s	1.7M	68s	14.4M
5	5s	2.5M	2690s	496M
	corrected version			
3	< 1s	1.4M	2s	2.4M
4	< 1s	1.7M	107s	24.3M
5	7s	4.4M	4202s	843M

Fig. 3. Left: Faulty version of quicksort. Right: Results for termination and correctness of sorting.

minimum of the elements in a[]. For a fixed array size N, we can model the array as N variables $a_1, \ldots, a_N$. The condition of the loop for decreasing i then becomes $\bigvee_{\alpha=1}^{N}(a_\alpha = v \wedge \alpha = j)$[14]. With this at hand, we can apply our approach and easily calculate the set of all array valuations for which this algorithm terminates. For this, we have only to add a second array b[] remembering the initial values of a[]. By definition of our extend-operation, we do not need to insert any additional relations into the model for copying the values of b[], as no statement of the original algorithms accesses b[]. After the saturation process has terminated, we obtain the input-output-relation of the presented algorithm restricted to those initial values for which the algorithm terminates. Fig. 3 summarizes the memory consumption and time spent both for the preprocessing step of creating the extended PDS and for the reachability analysis.

6 Summary and Future Work

We have presented, to the best of our knowledge, the first tool for reachability analysis of systems having both an infinite control-state space and an infinite data-domain. We see the contribution of our work as an refinement of the model-checking process, where our tool can be plugged in as an additional step after a finite model has been verified.

Our future work will encompass several objectives. With respect to our implementation, there are lots of options left to tweak the performance, and a second version of our tool is already work-in-progress. Further, Moped has been recently extended to support abstraction and the counter example guided abstraction refinement process (CEGAR) (cf. [14]). Right now, BDDs are used within this CEGAR process. It is just natural to extend this by using NDDs

[14] As this formula restricts j to values in $\{1, \ldots, N\}$, the saturation process of sect. 4 always terminates.

instead. There, NDDs would allow us on the one hand to model infinite data domains, and on the other hand we would be able to derive abstract predicates over infinite sets. Another important and logical evolution is to implement, in the case of intraprocedural loops, and extend, in the case of interprocedural ,,loops", the acceleration techniques proposed in [3] and later refined in [5].

7 Acknowledgments

We would like to thank Stefan Schwoon and Javier Esparza for many helpful suggestions and discussions.

References

1. Esparza, J., Schwoon, S.: A bdd-based model checker for recursive programs. In: Proceedings of CAV '01. Number 2012 in LNCS (2001)
2. Somenzi, F.: Colorado university decision diagram package. Technical report, University of Colorado (1998)
3. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces. PhD thesis, University of Liege (1999)
4. Wolper, P., Boigelot, B.: On the construction of automata from linear arithmetic constraints. In: Proceedings of TACAS '00. Number 1785 in LNCS (2000)
5. Finkel, A., Leroux, J.: How to compose presburger-accelerations: Applications to broadcast protocols. In: Proceedings of FST&TCS'02. Number 2556 in LNCS (2002)
6. Bardin, S., Finkel, A., Leroux, J.: Faster acceleration of counter-automata in practice. In: Proceedings of TACAS'04. Number 2988 in LNCS (2004)
7. Presbuger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: C. R. ler congres des Mathematiciens des pays slaves. (1927) 92–101
8. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Proceedings of CONCUR '97. Number 1243 in LNCS (1997) 135–150
9. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Proceedings of CAV '00. Number 1855 in LNCS (2000)
10. Wolper, P., Boigelot, B.: Verifying system with infinite but regular state space. In: Proceedings of CAV '98. Number 1427 in LNCS (1998) 88–97
11. Reps, T., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: Proceedings of SAS '03. Number 2694 in LNCS (2003) 189–213
12. Schwoon, S.: (weighted pds library: http://www.fmi.uni-stuttgart.de/szs/tools/wpds/)
13. Ottmann, T., Widmayer, P. In: Algorithmen und Datenstrukturen. Spektrum (2002) 87
14. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. In Hermanns, H., Palsberg, J., eds.: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Vienna, Austria (2006) To appear.