Transitive Closures of Affine Integer Tuple Relations and their Overapproximations

Sven Verdoolaege, Albert Cohen, and Anna Beletska

INRIA and École Normale Supérieure

Abstract. The set of paths in a graph is an important concept with many applications in system analysis. In the context of integer tuple relations, which can be used to represent possibly infinite graphs, this set corresponds to the transitive closure of the relation representing the graph. Relations described using only affine constraints and projection are fairly efficient to use in practice and capture Presburger arithmetic. Unfortunately, the transitive closure of such a quasi-affine relation may not be quasi-affine and so there is a need for approximations. In particular, most applications in system analysis require overapproximations. Previous work has mostly focused either on underapproximations or special cases of affine relations. We present a novel algorithm for computing overapproximations of transitive closures for the general case of quasi-affine relations (convex or not). Experiments on non-trivial relations from real-world applications show our algorithm to be on average more accurate and faster than the best known alternatives.

1 Introduction

Computing the transitive closure of a relation is an operation underlying many important algorithms, with applications to computer-aided design, software engineering, scheduling, databases and optimizing compilers. In this paper, we consider the class of parametrized relations over integer tuples whose constraints consist of affine equalities and inequalities over variables, parameters and existentially quantified variables. This class has the same expressivity as Presburger arithmetic. Such quasi-affine relations typically describe infinite graphs, with the transitive closure corresponding to the set of all paths in the graph, and are widespread in decision and optimization problems with infinite domains, with applications to static analysis, formal verification and automatic parallelization [3, 4, 7, 15, 17, 18, 23, 25]. In this context, the use of quasi-affine relations is preferred because most operations on such relations can be performed exactly and fairly efficiently. However, as shown by Kelly et al. [25], the transitive closure of a quasi-affine relation may not be representable as a quasi-affine relation, or may not be computable at all. This leads to the design of approximation techniques [1, 6, 9, 24, 25]. and/or the study of sub-classes, including sub-polyhedral domains, where an exact computation is possible [2, 10–14, 18, 20]. Our approach belongs to the first group. That is, our goal is not to investigate classes of relations for which the transitive closure is guaranteed to be exact, but rather to obtain a general technique for quasi-affine relations that always produces an overapproximation, striking a balance between accuracy and speed.

Until recently, approximation for the general case of quasi-affine relations has only been investigated by Kelly et al. [25], and they focus on computing underapproxima-

tions.Yet the vast majority of the applications require overapproximations, and the algorithm proposed by Kelly et al. for computing overapproximations is very inaccurate, both in our own implementation and in an independent one [9]. Overapproximations have been considered by Beletska et al. [6], but in a more limited setting.

```
for (i = 3; i <= n; i++)
  a[i] = f(a[i - 3]);
```

Fig. 1. A sequential loop

```
#pragma omp parallel for
for (i = 3; i <= min(n, 5); i++)
  for (j = i; j <= n; j += 3)
    a[j] = f(a[j - 3]);
```

Fig. 2. A parallelized version of the loop in Figure 1

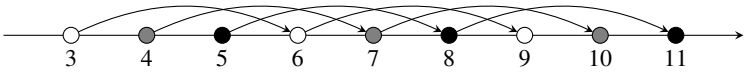

Fig. 3. The dependences and slices of the loop in Figure 1

We use Iteration Space Slicing (ISS) [7] to illustrate the application of the transitive closure to quasi-affine relations. The objective of this technique is to split up the iterations of a loop nest into slices that can be executed in parallel. Applying the technique to the code in Figure 1, we see that some iterations of the loop use a result computed in earlier iterations and can therefore not be executed independently. These dependences are shown as arrows in Figure 3 for the case where $n = 11$. The intuition behind ISS is to group all iterations that are connected through dependences and to execute the resulting groups in parallel. These groups form a partition of the iterations, which is defined by an equivalence relation based on the dependences. In particular, the equivalence relation needs to be transitively closed, which requires the computation of the transitive closure of a relation representing the (extended) dependence graph. The resulting relation connects iterations to directly or indirectly depending iterations. In the example, three such groups can be discerned, indicated by different colors of the nodes in Figure 3. The resulting parallel program, with the outer parallel loop running over the different groups and the inner loop running over all iterations that belong to a group, is shown in Figure 2. It is important to note here that if the transitive closure cannot be computed exactly, then an overapproximation should be computed. This may result in more iterations being grouped together and therefore fewer slices and less parallelism, but the resulting program would still be correct. Underapproximation, on the other hand, would lead to invalid code. Furthermore, the overapproximation needs to be transitively closed since the slices should not overlap. Most of our target applications are based on dependence relations. For more information, we refer to our report [33].

In this paper, we present an algorithm for computing overapproximations of transitive closures. The algorithm subsumes those of [6] and [1]. Furthermore, it is experimentally shown to be exact in more instances from our applications than that of [25] and generally also faster on those instances where both produce an exact result. Our algorithm includes three decomposition methods, two of which are refinements of those

of [25], while the remaining one is new. Finally, we provide a more extensive experimental evaluation on more difficult instances. As an indication, Kelly et al. [25] report that they were able to compute exact results for 99% of their input relations, whereas they can only compute exact results for about 60% of our input relations and our algorithm can compute exact results for about 80% of them. This difference in accuracy is shown to have an impact on the final outcome of some of our applications.

Section 2 gives background information on affine relations and transitive closures. We discuss related work in Section 3. Section 4 details the core of our algorithm. Section 5 studies a decomposition method to increase accuracy and speed. Section 6 describes the implementation. The results of our experiments are shown in Section 7.

2 Background

We use bold letters to denote vectors, and we write $\langle \mathbf{a}, \mathbf{x} \rangle$ for the scalar product of $\mathbf{a}$ and $\mathbf{x}$. We consider binary relations on $\mathbb{Z}^d$, i.e., relations mapping d-tuples of integers to d-tuples of integers. $S \circ R$ denotes the composition of two relations R and S. A relation R is *transitively closed* if $R \circ R = R$. The *transitive closure* of R, denoted R^+, is the (inclusion-wise) smallest transitively closed relation T such that $R \subseteq T$. The transitive closure R^+ can be constructed as the union of all positive integer powers of R:

$$R^+ := \bigcup_{k \geq 1} R^k, \quad \text{with} \quad R^k := \begin{cases} R & \text{if } k = 1 \\ R \circ R^{k-1} & \text{if } k \geq 2. \end{cases} \tag{1}$$

A relation R is *reflexively closed* on a set D if the identity relation Id_D is a subset of R. The *reflexive closure* of R on D is $R \cup \mathrm{Id}_D$. The *reflexive and transitive closure* of R on D is $R^*_D := R^+ \cup \mathrm{Id}_D$. The *cross product* of two relations R and S is the relation $R \times S = \{ (\mathbf{x}_1, \mathbf{y}_1) \to (\mathbf{x}_2, \mathbf{y}_2) \mid \mathbf{x}_1 \to \mathbf{x}_2 \in R \wedge \mathbf{y}_1 \to \mathbf{y}_2 \in S \}$. Occasionally, we will also consider binary relations over labeled integer tuples, i.e., subsets of $\bigcup_{d_1, d_2 \geq 0} (\Sigma \times \mathbb{Z}^{d_1}) \to (\Sigma \times \mathbb{Z}^{d_2})$, with Σ a finite set of labels. By assigning an integer value to each label, any such relation can be encoded as a relation over the $(1 + d)$-tuples with d the largest of the d_1s and d_2s over all elements in the relation.

We work with relations that have a finite representation. A commonly used class of such relations are those that can be represented using affine constraints. We consider finite unions of *basic relations* $R = \bigcup_i R_i$, each of which is represented as

$$R_i = \mathbf{s} \mapsto \{ \mathbf{x}_1 \to \mathbf{x}_2 \in \mathbb{Z}^d \times \mathbb{Z}^d \mid \exists \mathbf{z} \in \mathbb{Z}^e : A_1 \mathbf{x}_1 + A_2 \mathbf{x}_2 + B\mathbf{s} + D\mathbf{z} + \mathbf{c} \geq \mathbf{0} \}, \tag{2}$$

with $\mathbf{s}$ a vector of n free parameters, $A_i \in \mathbb{Z}^{m \times d}$, $B \in \mathbb{Z}^{m \times n}$, $D \in \mathbb{Z}^{m \times e}$ and $\mathbf{c} \in \mathbb{Z}^m$. The explicit declaration of the parameters ($\mathbf{s} \mapsto$) only serves to stress that the relation is parameteric and will sometimes be omitted. To emphasize that the description may involve existentially quantified variables $\mathbf{z}$, we call such relations *quasi-affine*. Any Presburger relation can be put in this form.

Unfortunately, the transitive closure of a quasi-affine relation may not be representable using affine constraints [25]. Similarly, a description of *all* positive integer powers k of R, parametrically in k, may not be representable either. We will refer to this description as simply the *power* of R and denote it as R^k. Since the power R^k as well as

the transitive closure R^+ may not be representable, we will compute approximations, in particular overapproximations, denoted as $\mathcal{P}_k(R)$ and $\mathcal{T}(R)$, respectively.

Next to quasi-affine relations, our computations also make use of quasi-affine sets. These sets are defined essentially the same way: the only difference is that sets are unary relations on integer tuples instead of binary relations. The variables representing these tuples are called *set variables*. Sets can be obtained from relations in the following ways. The *domain* of a relation R is the set that results from *projecting out* the variables $\mathbf{x}_2$, i.e., $\operatorname{dom} R \coloneqq \{\, \mathbf{x}_1 \in \mathbb{Z}^d \mid \exists \mathbf{x}_2 \in \mathbb{Z}^d : \mathbf{x}_1 \to \mathbf{x}_2 \in R \,\}$. The *range* of a relation R is the set $\operatorname{ran} R \coloneqq \{\, \mathbf{x}_2 \in \mathbb{Z}^d \mid \exists \mathbf{x}_1 \in \mathbb{Z}^d : \mathbf{x}_1 \to \mathbf{x}_2 \in R \,\}$. The *difference set* of a relation R is the set $\Delta R \coloneqq \{\, \boldsymbol{\delta} \in \mathbb{Z}^d \mid \exists \mathbf{x} \to \mathbf{y} \in R : \boldsymbol{\delta} = \mathbf{y} - \mathbf{x} \,\}$. We also need the following operations on sets. The *Minkowski sum* of S_1 and S_2 is the set of sums of pairs of elements from S_1 and S_2, i.e., $S_1 + S_2 = \{\, \mathbf{a} + \mathbf{b} \mid \mathbf{a} \in S_1 \wedge \mathbf{b} \in S_2 \,\}$. The kth multiple of a set, with k a positive integer is defined as $1\,S = S$ and $k\,S = (k-1)\,S + S$ for $k \geq 2$. Note that as with the kth power of a relation, the kth multiple of a quasi-affine set, with k a parameter, may not be representable as a quasi-affine set.

Most of the applications we consider operate within the context of the polyhedral model [22], where single-entry single-exit program regions are represented, analyzed and transformed using quasi-affine sets and relations. In particular, the set of all iterations of a loop for which a given statement is executed is called the *iteration domain*. When the loop iterators are integers and lower and upper bounds of all enclosing loops as well all enclosing conditions are quasi-affine, then the iteration domain can be represented as a quasi-affine set. For example, the iteration domain of the single statement in Figure 1 is $n \mapsto \{\, i \mid 3 \leq i \leq n \,\}$. *Dependence relations* map elements of an iteration domain to elements of another (or the same) iteration domain which depend on them for their execution. In the example, we have as dependence relation $n \mapsto \{\, i \to i + 3 \mid 3 \leq i, i + 3 \leq n \,\}$. The graph with the statements and their iterations domains as nodes and the dependence relations as edges is called the *dependence graph*.

3 Related Work

The seminal work of Kelly et al. [25] introduced many of the concepts and algorithms in the computation of transitive closures that are also used in this paper. In particular, we use a revised version of their incremental computation and we apply their modified Floyd-Warshall algorithm internally. However, the authors consider a different set of applications which require underapproximations of the transitive closures instead of overapproximations. Their work therefore focuses almost exclusively on underapproximations. For overapproximations, they apparently consider some kind of "box-closure", which we recall in Section 6 and which is considerably less accurate than our algorithm.

Bielecki et al. [8] aim for exact results, which may therefore be non-affine. In our applications, affine results are preferred as they are easier to manipulate in further calculations. Furthermore, the authors only consider bijective relations over a convex domain. We consider general quasi-affine relations, which may be both non-bijective and defined over finite unions of domains.

Beletska et al. [6] consider finite unions of translations, for which they compute quasi-affine transitive closure approximations, as well as some other cases of finite unions of bijective relations, which lead to non-affine results. Their algorithm applied to unions of translations forms a special case of our algorithm for general affine relations.

Bielecki et al. [9] propose to compute the transitive closure using the classical iterative least fixed point computation and if this process does not produce the exact result after a fixed number of iterations, they resort to a variation of the "box-closure" of [25]. To increase the chances of the least fixed point computation, they first replace each disjunct in the input relation by its transitive closure, provided it can be computed exactly using available techniques [8, 25].

Transitive closures are also used in the analysis of counter systems to accelerate the computation of reachable sets. In this context, the power of a relation is known as a "counting acceleration" [20], while our relations over labeled tuples correspond to Presburger counter systems [20], extended to the integers. Much of the work on counter systems is devoted to the description of classes of systems for which the computations can be performed exactly. See, e.g., the work of Bardin et al. [2] and their references or the work of Bozga et al. [13]. By definition, these classes do not cover the class of input relations that we target in our approach. Other work on counter systems, e.g., that of Sankaranarayanan et al. [28], Feautrier and Gonnord [24] or Ancourt et al. [1], focuses on the computation of invariants and therefore allows for overapproximations. However, the analysis is usually performed on (non-parametric) polyhedra. That is, the relations for which transitive closures are computed do not involve parameters, existentially quantified variables or unions. The transitive closure algorithm proposed by Ancourt et al. [1] is essentially the same as that used by Boigelot and Herbreteau [12], except that the latter apply it on hybrid systems and only in cases where the algorithm produces an exact result. The same algorithm also forms the core of our transitive closure algorithm for single disjunct relations.

4 Powers and Transitive Closures

We present our core algorithm for computing overapproximations of the parametric power and the transitive closure of a relation. We first discuss the relationship between these two concepts and provide further evidence for the need for overapproximations. Then, we address the case where R is a single basic relation, followed by the case of multiple disjuncts. Finally, we explain how to check the exactness of the result and why the overapproximation is guaranteed to be transitively closed.

4.1 Introduction

There is a close relationship between parametric powers and transitive closures. Based on (1), the transitive closure R^+ can be computed from the parametric power R^k by projecting out the parameter k. Conversely, an algorithm for computing transitive closures can also be used to compute parametric powers. In particular, given a relation R, compute C^+ with $C = R \times \{ i \to i + 1 \}$. For each pair of integer tuples in C, the difference between the final coordinates is 1. The difference between the final coordinates of pairs

in C^+ is therefore equal to the number of steps taken. To compute R^k, one may equate k to this difference and subsequently project out the final coordinates.

As mentioned in Section 2, it is not always possible to compute powers and closures exactly, and we may aim instead for overapproximations $\mathcal{P}_k(R)$ and $\mathcal{T}(R)$. It should be clear that both conversions above map overapproximations to overapproximations. Important: note that the transitive closure may not be affinely representable even if the input relation is a union of constant-distance translations. A well know case can be built by considering the lengths of dependence paths associated to SUREs [19, Theorem 23].

4.2 Single Disjunct

Given a single basic relation R of the form (2), we look for an overapproximation of R^+ and we will derive it from an overapproximation of R^k. Furthermore, we want to compute the approximation efficiently and we want it to be as close to exact as possible.

We will treat the input relation as a (possibly infinite) union of translations. The distances covered by these translations are the elements of the difference set $\Delta = \Delta R$. We will assume here that Δ also consists of a single basic set; our implementation of the ΔR operation may result in a proper union due to our treatment of existentially quantified variables discussed below. The union case is treated in Section 4.3. Our approximation of the kth power contains translations over distances that are the sums of k distances in Δ. In particular, it contains those translations starting from and ending at the same points as those of the input relation. That is, we compute all paths along distances in Δ

$$P^k = \{\, \mathbf{x} \to \mathbf{y} \mid \exists \boldsymbol{\delta} \in \mathcal{D}^k : \mathbf{y} = \mathbf{x} + \boldsymbol{\delta} \,\}, \quad \text{with} \quad \mathcal{D}^k = k\Delta \quad \text{and} \quad k \in \mathbb{Z}_{\geq 1}, \tag{3}$$

and intersect domain and range with those of R,

$$\mathcal{P}_k(R) = P^k \cap (\operatorname{dom} R \to \operatorname{ran} R)\,. \tag{4}$$

Example 1. To see the importance of this intersection with domain and range, consider the relation $R = \{\, (x, y) \to (x, x) \,\}$. First note that this relation is transitively closed already, so in our implementation we would not apply the algorithm here. If we did, however, then we would have $\Delta R = \{\, 0 \,\} \times \mathbb{Z}$, whence $P^k = \{\, (x, y) \to (x, y') \,\}$. On the other hand, $\operatorname{ran} R = \{\, (x, x) \,\}$ and so $\mathcal{T}(R) = \mathcal{P}_k(R) = \{\, (x, y) \to (x, x) \,\}$.

Unfortunately, the set $k\Delta$ in (3) may not be affine in general and then the same holds for P^k. As a trivial example of $k\Delta$ not being affine, take Δ to be the parametric singleton $n \to \{\, n \,\}$. If, however, Δ is a non-parametric singleton $\Delta = \{\, \boldsymbol{\delta} \,\}$, i.e., $\boldsymbol{\delta}$ does not depend on the parameters, then $k\Delta$ is simply $\{\, k\,\boldsymbol{\delta} \,\}$ and we can compute our approximation of the power according to (4). Otherwise, we drop the definition of $\mathcal{D}^k$ in (3) and compute $\mathcal{D}^k$ as an approximation of $k\Delta$, essentially copying some constraints of (a projection of) Δ. This process ensures that $\mathcal{D}^k$ is easy to compute, although it may in some cases not be the most accurate affine approximation of $k\Delta$.

Let us first assume that the description of Δ does not involve any existentially quantified variables or parameters. The constraints then have the form $\langle \mathbf{a}, \mathbf{x} \rangle + c \geq 0$. Any element in $k\Delta$ can be written as the sum of k elements $\boldsymbol{\delta}_i$ from Δ. Each of these satisfies the constraint. The sum therefore satisfies the constraint

$$\langle \mathbf{a}, \mathbf{x} \rangle + c\,k \geq 0, \tag{5}$$

meaning that the constraint in (5) is valid for $k\,\Delta$. Our approximation $\mathcal{D}^k$ of $k\,\Delta$ is then the set bounded by the constraints in (5). In this special case, we compute essentially the same approximation as [1]. Note that if Δ has integer vertices, then the vertices of $\Delta \times \{ 1 \}$ generate the rational cone $\{ (\mathbf{x}, k) \in \mathbb{Q}^{d+1} \mid \langle \mathbf{a}, \mathbf{x} \rangle + c\,k \geq 0 \}$. This means that $\Delta \times \{ 1 \}$ is a Hilbert basis of this cone [29, Theorem 16.4] and that therefore $\mathcal{D}^k = k\,\Delta$.

Example 2. As a trivial example, consider the relation $R = \{ x \to y \mid 2 \leq y - x \leq 3 \}$. We have $\Delta = \Delta\, R = \{ \delta \mid 2 \leq \delta \leq 3 \}$ and $\mathcal{D}^k = k \mapsto \{ \delta \mid 2\,k \leq \delta \leq 3\,k \}$. Therefore, $\mathcal{P}_k(R) = P^k = k \mapsto \{ x \to y \mid 2\,k \leq y - x \leq 3\,k \}$ and $\mathcal{T}(R) = \{ x \to y \mid y - x \geq 2 \}$.

If the description of Δ does involve parameters, we cannot simply multiply the parametric constant by k: that would result in non-affine constraints. One option is to treat parameters as variables that just happen to remain constant. That is, instead of considering the set $\Delta = \Delta\, R = \mathbf{s} \mapsto \{ \boldsymbol{\delta} \in \mathbb{Z}^d \mid \exists \mathbf{x} \to \mathbf{y} \in R : \boldsymbol{\delta} = \mathbf{y} - \mathbf{x} \}$, we consider the set

$$\Delta' = \Delta\, R' = \{ \boldsymbol{\delta} \in \mathbb{Z}^{n+d} \mid \exists (\mathbf{s}, \mathbf{x}) \to (\mathbf{s}, \mathbf{y}) \in R' : \boldsymbol{\delta} = (\mathbf{s} - \mathbf{s}, \mathbf{y} - \mathbf{x}) \}. \tag{6}$$

The first n coordinates of every element in Δ' are zero. Projecting out these zero coordinates from Δ' is equivalent to projecting out the parameters in Δ. The result is obviously a superset of Δ, but all its constraints only involve the variables $\mathbf{x}$ and can therefore be treated as above.

Another option is to categorize the constraints of Δ according to whether they involve set variables, parameters or both. Constraints involving only set variables are treated as before. Constraints involving only parameters, i.e., constraints of the form

$$\langle \mathbf{b}, \mathbf{s} \rangle + c \geq 0. \tag{7}$$

are also valid for $k\,\Delta$. (Δ is empty for values of the parameters not satisfying these constraints and therefore so is $k\,\Delta$.) For constraints of the form

$$\langle \mathbf{a}, \mathbf{x} \rangle + \langle \mathbf{b}, \mathbf{s} \rangle + c \geq 0, \tag{8}$$

involving both set variables and parameters, we need to consider the sign of $\langle \mathbf{b}, \mathbf{s} \rangle + c$. If this expression is non-positive for all values of $\mathbf{s}$ for which Δ is non-empty, i.e.,

$$\Delta \cap \mathbf{s} \mapsto \{ \boldsymbol{\delta} \mid \langle \mathbf{b}, \mathbf{s} \rangle + c > 0 \} = \emptyset, \tag{9}$$

then $\langle \mathbf{a}, \mathbf{x} \rangle$ will always have a non-negative value v and we have $k \langle \mathbf{a}, \mathbf{x} \rangle \geq v$ for $k \geq 1$. The constraint in (8) is therefore also valid for $k\,\Delta$ if this condition holds. Our approximation $\mathcal{D}^k$ of $k\,\Delta$ is the set bounded by the constraints in (5), (7) and (8). Constraints of the form (8) for which (9) does not hold are simply dropped. Since this may result in a loss of accuracy, we add the constraints derived from Δ' above if any constraints of the form (8) get dropped.

Example 3. Consider the relation $R = n \to \{ (x, y) \to (1 + x, 1 - n + y) \mid n \geq 2 \}$. We have $\Delta\, R = n \to \{ (1, 1 - n) \mid n \geq 2 \}$ and so, by specifically treating parameters as described above, we obtain the following approximation for R^+: $n \to \{ (x, y) \to (x', y') \mid n \geq 2 \wedge y' \leq 1 - n + y \wedge x' \geq 1 + x \}$. If we consider instead $R' = \{ (n, x, y) \to$

$(n, 1 + x, 1 - n + y) \mid n \geq 2\}$ then $\Delta R' = \{(0, 1, y) \mid y \leq -1\}$ and we obtain the approximation $n \to \{(x, y) \to (x', y') \mid n \geq 2 \wedge x' \geq 1 + x \wedge y' \leq x + y - x'\}$. If we consider both ΔR and $\Delta R'$, then we obtain $n \to \{(x, y) \to (x', y') \mid n \geq 2 \wedge y' \leq 1 - n + y \wedge x' \geq 1 + x \wedge y' \leq x + y - x'\}$. Note however that this is not the most accurate affine approximation: $n \to \{(x, y) \to (x', y') \mid y' \leq 2 - n + x + y - x' \wedge n \geq 2 \wedge x' \geq 1 + x\}$ is a more accurate one.

If the description of Δ does involve existentially quantified variables, we compute unique representatives for these variables, picking the lexicographically minimal value for each of them using parametric integer programming [21]. The result is an explicit representation of each existentially quantified variable as the greatest integer part of an affine expression in the parameters and set variables. This representation may involve case distinctions, leading to a partitioning of Δ. If the representation involves only parameters, then the existentially quantified variable can be treated as a parameter. Similarly, if it only involves set variables, the existentially quantified variable can be treated as a set variable too. Otherwise, any constraints involving the variable are discarded. If this happens then, as before, we add the constraints derived from Δ' (6).

Example 4. Consider $R = n \to \{x \to y \mid \exists \alpha_0, \alpha_1 : 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -1 - x + y \wedge y \geq 6 + x\}$. The difference set of this relation is $\Delta = \Delta R = n \to \{x \mid \exists \alpha_0, \alpha_1 : 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -1 + x \wedge x \geq 6\}$. The existentially quantified variables can be defined in terms of the parameters and variables as $\alpha_0 = \lfloor(-2 + n)/7\rfloor$ and $\alpha_1 = \lfloor(-1 + x)/5\rfloor$. α_0 can therefore be treated as a parameter, while α_1 can be treated as a variable. This in turn means that $7\alpha_0 = -2 + n$ can be treated as a purely parametric constraint, while the other two constraints are non-parametric. The corresponding P^k is therefore $(n, k) \to \{x \to y \mid \exists \alpha_0, \alpha_1, f : k \geq 1 \wedge y = x + f \wedge 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -k + f \wedge f \geq 6k\}$. Projecting out the parameter k and simplifying the result, we obtain the exact transitive closure $R^+ = n \to \{x \to y \mid \exists \beta_0, \beta_1 : 7\beta_0 = -2 + n \wedge 6\beta_1 \geq -x + y \wedge 5\beta_1 \leq -1 - x + y\}$.

4.3 Multiple Disjuncts

When the set of distances Δ is a proper union of basic sets $\Delta = \bigcup_i \Delta_i$, we apply the technique of Section 4.2 to each Δ_i separately, yielding approximations $\mathcal{D}_i^k$ of $k_i \Delta_i$ and corresponding paths P_i^k from (3). The set of global paths should take a total of k steps along the Δ_is, which can be obtained by essentially composing the P_i^ks and taking k to be the sum of all k_is. However, we need to allow for some k_is to be zero, so we introduce stationary paths $S_i = \mathrm{Id}_{\mathbb{Z}^d} \cap \{\mathbf{x} \to \mathbf{y} \mid k_i = 0\}$ and compute the set of global paths as

$$P^k = \left((P_m^{k_m} \cup S_m) \circ \cdots \circ (P_2^{k_2} \cup S_2) \circ (P_1^{k_1} \cup S_1)\right) \cap \{\mathbf{x} \to \mathbf{y} \mid k = \sum_i k_i > 0\}. \quad (10)$$

The final constraint ensures that at least one step is taken. The approximation of the power is then again computed according to (4). As explained in Section 4.1, $\mathcal{P}_k(R)$ can be represented as $\mathcal{T}(C)$, with $C = R \times \{i \to i + 1\}$. Using this representation, all Δ_i have 1 as their final coordinate and S_i above is simply $\mathrm{Id}_{\mathbb{Z}^{d+1}}$.

We need to be careful about scalability at this point. Given a set of distances Δ with m disjuncts, a naive application of (10) results in a P^k relation with $2^m - 1$ disjuncts. We

try to limit this explosion in three ways. First, we handle all singleton Δ_i together; second, we try to avoid introducing a union with S_i; and third, we try to combine disjuncts. In particular, the paths along $\Delta_i = \{ \boldsymbol{\delta}_i \}$ can be computed as

$$P^k = \{ \mathbf{x} \to \mathbf{y} \mid \exists k_i \in \mathbb{Z}_{\geq 0} : \mathbf{y} = \mathbf{x} + \sum_i k_i \boldsymbol{\delta}_i \wedge \sum_i k_i = k > 0 \}.$$

In this special case, we compute essentially the same approximation as [6]. For the remaining Δ_i, if the result of replacing constraint $k \geq 1$ by $k = 0$ in the computation of P^k yields the identity mapping, then $P_i^k \cup S_i$ is simply Q_i^k with Q_i^k the result of replacing $k \geq 1$ by $k \geq 0$. It is tempting to always replace $P_i^k \cup S_i$ by this Q_i^k, even if it is an overapproximation, but experience has shown that this leads to a significant loss in accuracy. Finally, if neither of these optimizations apply, then after each composition in (10) we "coalesce" the resulting relation. Coalescing detects pairs of disjuncts that can be replaced by a single disjunct without introducing any spurious elements [32].

4.4 Properties

By construction (Section 4.2 and Section 4.3), we have the following lemma.

Lemma 1. *$\mathcal{P}_k(R)$ is an overapproximation of R^k, i.e., $R^k \subseteq \mathcal{P}_k(R)$.*

The transitive closure approximation is obtained by projecting out the parameter k. If $\mathcal{P}_k(R)$ is represented as $\mathcal{T}(C)$, with $C = R \times \{ i \to i + 1 \}$, then $\mathcal{T}(R)$ can be obtained from $\mathcal{T}(C)$ by projecting out the final coordinates. The following lemma immediately holds.

Lemma 2. *$\mathcal{T}(R)$ is an overapproximation of R^+, i.e., $R^+ \subseteq \mathcal{T}(R)$.*

In many cases, $\mathcal{P}_k(R)$ will be exactly R^k. Given a particular R it is instructive to know whether the computed $\mathcal{P}_k(R)$ is exact or not, either for applications working directly with powers or as a basis for an exactness test on closures detailed below. The exactness test on powers amounts to checking whether $\mathcal{P}_k(R)$ satisfies the definition of R^k in (1):

$$\mathcal{P}_1(R) \subseteq R \quad \text{and} \quad \mathcal{P}_k(R) \subseteq R \circ \mathcal{P}_{k-1}(R) \text{ for } k \geq 2.$$

The reverse inclusion is guaranteed by Lemma 1. If $\mathcal{P}_k(R)$ is exact, then $\mathcal{T}(R)$ is also exact since the projection is performed exactly. However, if $\mathcal{P}_k(R)$ is *not* exact then $\mathcal{T}(R)$ might still be exact. We therefore prefer the more accurate test of [25, Theorem 5]:

$$\mathcal{T}(R) \subseteq R \cup (R \circ \mathcal{T}(R)) .$$

However, this test can only be used if R is acyclic, i.e., if R^+ has no fixed points. Since $\mathcal{T}(R)$ is an overapproximation of R^+, it is sufficient to check that $\mathcal{T}(R)$ has no fixed points, i.e., that $\mathbf{0} \notin \Delta \mathcal{T}(R)$. If $\mathcal{T}(R)$ does have fixed points, then we apply the exactness test on $\mathcal{P}_k(R)$ instead.

Some applications also require the computed approximation of the transitive closure to be a transitively closed one [3, 7, 17]. The power approximation $\mathcal{P}_k(R)$ computed above is transitively closed as soon as P^k is transitively closed: if $\mathbf{x} \to \mathbf{y} \in \mathcal{P}_{k_1}(R)$

and $\mathbf{y} \to \mathbf{z} \in \mathcal{P}_{k_2}(R)$, then $\mathbf{x} \to \mathbf{z} \in \mathcal{P}_{k_1+k_2}(R)$, because P^k is transitively closed (and so $\mathbf{x} \to \mathbf{z} \in P^{k_1+k_2}$), $\mathbf{x} \in \operatorname{dom} R$ and $\mathbf{z} \in \operatorname{ran} R$. If $\mathbf{x}_1 \in \mathcal{D}^{k_1}$ and $\mathbf{x}_2 \in \mathcal{D}^{k_2}$, then both combinations satisfy (5) and so does their sum. Constraint (8) is also satisfied for $\mathbf{x}_1 + \mathbf{x}_2$, hence $\mathbf{x}_1 + \mathbf{x}_2 \in \mathcal{D}^{k_1+k_2}$. We conclude that in the single disjunct case, P^k in (3) is transitively closed, which in turn implies that also P^k in (10) is transitively closed in the multiple disjunct case. $\mathcal{T}(R)$ is transitively closed because for any $\mathbf{x} \to \mathbf{y}$ and $\mathbf{y} \to \mathbf{z}$ in $\mathcal{T}(R)$, there is some pair k_1, k_2 such that $\mathbf{x} \to \mathbf{y} \in \mathcal{P}_{k_1}(R)$ and $\mathbf{y} \to \mathbf{z} \in \mathcal{P}_{k_2}(R)$ and so $\mathbf{x} \to \mathbf{z} \in \mathcal{P}_{k_1+k_2}(R)$. We therefore have the following theorem.

Theorem 1. *$\mathcal{T}(R)$ is a transitively closed overapproximation of R^+.*

5 Strongly Connected Components

In order to improve accuracy, we apply several methods for breaking up the transitive closure computation. The first one is a decomposition into strongly connected components. The other two are variations of methods in [25]: we apply the modified Floyd-Warshall algorithm internally after partitioning the domain and we apply an incremental computation method. Our variations on these methods are explained in [33, Section 6].

Computations in Section 4.2 and Section 4.3 focus on the distance between elements in relation. The domain and range of the input relation are only taken into account at the very last step in (4). This means that translations described by one disjunct are applied to domain elements of other disjuncts, even if the domains are completely disjoint. In this section, we describe how the accuracy of $\mathcal{P}_k(R)$ and $\mathcal{T}(R)$ can be improved by decomposing the disjuncts of R into strongly connected components (SCCs).

The translations of R^+ are compositions of translations in the disjuncts of R. Two disjuncts R_i and R_j should be lumped into a connected component if there exist translations in R^k that first go through R_i and then through R_j, and translations that first go through R_j and then through R_i. Formally, we consider the directed graph whose vertices are the disjuncts in R and whose arcs connect pairs of vertices (R_i, R_j) if R_i can immediately follow R_j. The SCCs can be computed from this graph using Tarjan's algorithm [30]. In principle R_i can immediately follow R_j if the range of R_j intersects the domain of R_i, i.e., if $R_i \circ R_j \neq \emptyset$. However, if $R_i \circ R_j \subseteq R_j \circ R_i$ then one may always interchange R_i and R_j in any sequence leading to an element of R^+ where R_i immediately follows R_j. It is therefore sufficient to introduce an edge between R_i and R_j only if

$$R_i \circ R_j \not\subseteq R_j \circ R_i. \tag{11}$$

Once the components have been obtained, we compute $\mathcal{T}(R_c)$ on each component R_c separately. These $\mathcal{T}(R_c)$ can be combined into a global $\mathcal{T}(R)$ in the same way the paths are combined in (10). The combination must be performed according to a topological ordering of the components, obtained as a byproduct of Tarjan's algorithm. The decomposition preserves the validity of Lemma 1. The exactness check of Section 4.4 is performed on each component separately. If the approximation turns out to be inexact for any of the components, then the entire result is marked inexact and the exactness check is skipped on the remaining components.

To ensure closedness of $\mathcal{T}(R)$, we need to make a minor modification. If we are to perform the decomposition based solely on criterion $R_i \circ R_j \neq \emptyset$, then the same property

will also hold for the components and, because of (4), for the powers of the components, implying that the final result is also transitively closed. If (11) is ever used, however, then transitive closedness of the result is not guaranteed unless all computations are performed exactly. We therefore explicitly check whether the result is transitively closed when the computation is not exact and when (11) has been used. If the check fails, we recompute the result without a decomposition into SCCs.

6 Implementation Details

The algorithms described in the previous sections have been implemented in the `isl` library [32]. For details about the algorithms, the design and the implementation of this integer set library, the reader is referred to the documentation and dedicated paper: `http://freshmeat.net/projects/isl`. The `isl` library supports both a parametric power ($\mathcal{P}_k(R)$) and a transitive closure ($\mathcal{T}(R)$) operation. Most of the implementation is shared between the two operations. The transitive closure operation first checks if the input happens to be transitively closed already and, if so, returns immediately. Both operations then check for strongly connected components.Within each component, either the modified Floyd-Warshall algorithm is applied or an incremental computation is attempted, depending on whether the domain and range can be partitioned. For practical reasons, incremental computation of powers has not been implemented. In the case of the power or in case no incremental computation can be performed, the basic single or multiple disjunct algorithm is applied. The exactness test is performed on the result of this basic algorithm. In the case of the transitive closure, the final coordinates encoding the path lengths are projected out on the same result. In the case of the power, the final coordinates are only projected out at the very end, after equating their difference to the exponent parameter. The `isl` library has direct support for unions of relations over pairs of labeled tuples. When the transitive closure of such a union is computed, we first apply the modified Floyd-Warshall algorithm on a partition based on the label and tuple size. Each recursive call is then handled as described above.

We also implemented a variation of the "box-closure" of Kelly et al. [25], which is a simplified version of the algorithm in Section 4.2. They overapproximate Δ by a rectangular box, possibly intersected with a rectangular lattice, with the box having fixed (i.e., non-parametric), but possibly infinite, lower and upper bounds. This overapproximation therefore has only non-parametric constraints and the corresponding $\mathcal{D}^k$ can be constructed using some very specific instances of (5). This algorithm clearly results in an overapproximation of R^k and therefore, after projection, of R^+. To improve accuracy, we also apply their incremental algorithm, but only in case the result is exact. The `ApproxClosure` operation which appeared in very recent versions of `Omega+` applies a similar algorithm. The main differences are that it does not perform an incremental computation and that it computes a box-closure on each disjunct individually.

7 Experiments

In all our experiments, we have used `isl` version isl-0.05.1-125-ga88daa9, `Omega+` version 2.1.6 [16], `Fast` version 2.1, `Aspic` version 3.2 and the latest version of `StInG`

[28].[1] Version 2.1.6 of Omega+ provides three transitive closure operations: the original TransitiveClosure (TC), which computes an underapproximation of the transitive closure; ApproxClosure (AC), which computes an overapproximation of the reflexive and transitive closure; and calculateTransitiveClosure (CTC), which appears to first try the least fixed point algorithm of [9] and then falls back on ApproxClosure. The execution times of the Omega+ transitive closure operations include the time taken for an extra exactness test. For TransitiveClosure, this test is based on [25, Theorem 1]. Presumably, a similar exactness test is performed internally, but the result of this test is not available to the user. In some cases, Omega+ returns a result containing UNKNOWN constraints and then it is clear that the result is not exact. In other cases, the user has no way of knowing whether the result is exact except by explicitly applying an exactness test. The isl library, by contrast, returns the exactness as an extra result. For ApproxClosure, we apply the test of [25, Theorem 5]. Note that this test may result in false positives when applied to cyclic relations. The exactness of the Aspic results is evaluated in the same way. Recall from Section 4.4 that we do not apply this test inside isl on relations that may be cyclic. Since it is not clear whether calculateTransitiveClosure will always produce an overapproximation, we apply both tests when checking its exactness. For the Fast results, no exactness test is needed since Fast will only terminate if it has computed an exact result. On the other hand, the execution time of Fast includes a conversion of the resulting Armoise formula to a quasi-affine relation, i.e., a disjunctive normal form. Since Fast only supports non-negative variables, we split all variables into a pair of non-negative variables whenever the input relation contains any negative value. Below, we discuss our experiments on inputs from our target domains. We have also performed some experiments [33] on the Aspic and Lever [31] test sets. On the first, isl performs comparably to Aspic, while on the second, isl only outperforms Lever on a small minority of the cases.

7.1 Type Size Inference

Chin and Khoo [17] apply the transitive closure operation to the following relation, derived from their Ackermann example: $\{(i,j) \to (i-1,j_1) \mid i \geq 1 \wedge j \geq 1\} \cup \{(i,j) \to (i,j-1) \mid i \geq 1 \wedge j \geq 1\} \cup \{(i,0) \to (i-1,1) \mid i \geq 1\}$. Omega produces an underapproximation and the authors heuristically manipulate this underapproximation to arrive at the following overapproximation: $\{(i,j) \to (i_1,j_1) \mid i_1 \geq 0 \wedge i_1 \leq i-1 \wedge j \geq 0\} \cup \{(i,j) \to (i,j_1) \mid j_1 \geq 0 \wedge j_1 \leq j-1 \wedge i \geq 1\}$. We compute the exact transitive closure: $\{(i,j) \to (o_0,o_1) \mid o_0 \geq 0 \wedge o_0 \leq -1+i \wedge j \geq 0 \wedge o_0 \leq -2+i+j\} \cup \{(i,j) \to (o_0,1) \mid o_0 \leq -1+i \wedge j \geq 0 \wedge o_0 \geq 0\} \cup \{(i,j) \to (i,o_1) \mid i \geq 1 \wedge o_1 \geq 0 \wedge o_1 \leq -1+j\} \cup \{(i,j) \to (o_0,0) \mid o_0 \leq -1+i \wedge j \geq 0 \wedge o_0 \geq 1\}$.

7.2 Equivalence Checking

Our most extensive set of experiments is based on the algorithm of [4] for checking the equivalence of a pair of static affine programs. Since the original implementation was not available to us, we have reimplemented the algorithm using VAUCANSON [26]

[1] The Fast and Aspic tests are based on the encoding described in [33, Section 8].

to compute regular expressions and `isl` to perform all set and relation manipulations. For the transitive closure operation we use the algorithm presented in this paper, the "box" implementation described in Section 6 or one of the implementations in `Omega+`. Since it is not clear whether `calculateTransitiveClosure` will always produce an overapproximation, we did not test this implementation in this experiment. The equivalence checking procedure requires overapproximations of transitive closures and using `calculateTransitiveClosure` might therefore render the procedure unsound. Since `TransitiveClosure` computes an underapproximation, we only use the results if they are exact. If not, we fall back on `ApproxClosure`. We will refer to this implementation as "TC+AC". For the other methods, we omit the exactness test in this experiment.

			Omega+	
	isl	box	TC+AC	AC
proved equivalent	72	46	49	50
not proved equivalent	15	51	28	45
out-of-memory	17	12	14+18	4+12
time-out	9	4	4	2

Table 1. Results for equivalence checking

			Omega+					
	isl	box	TC	AC	CTC	Fast	Aspic	StInG
exact	472	334	366	267	274	139	201	215
inexact	67	227	157	266	245	0	268	240
failure	34	12	50	40	54	434	104	118

Table 2. Outcome of transitive closure operations from equivalence checking

The equivalence checking procedure was applied to the output of `CLooG` [5] on 113 of its tests. In particular, the output generated when using the `isl` backend was compared against the output when using the `PPL` backend. These outputs should be equivalent for all cases, as was confirmed by the equivalence checking procedure of [34]. Table 1 shows the results. Using `isl`, 72 cases could be proven equivalent, while using `Omega+` this number was reduced to only 49 or 50. This does not necessarily mean that all transitive closures were computed exactly; it just means that the results were accurate enough to prove equivalence. In fact, using `ApproxClosure` on its own, we can prove one more case equivalent than first using `TransitiveClosure` and then, if needed, `ApproxClosure`. On the other hand, as we will see below, `TransitiveClosure` is generally more accurate than `ApproxClosure`. A time limit of 1 hour was imposed, resulting in some cases timing out, and memory usage was capped at 2GB, similarly resulting in some out-of-memory conditions. For the `Omega+` cases, we distinguish the real out-of-memory and maxing out the number of constraints (2048). The `isl` library does not impose a limit on the number of constraints. For those cases that `Omega+`'s `ApproxClosure` was able to handle (a strict subset of those that could be handled by `isl`), Figure 4 compares the running times. Surprisingly, `isl` is faster than `Omega+`'s `ApproxClosure` in all but one case. What is no surprise is that the running times (not shown in the figure) of the combined `TransitiveClosure` and `ApproxClosure` method are much higher still because it involves an explicit exactness test.

In order to compare the relative performance of the transitive closure operations themselves, we collected all transitive closure instances required in the above experiment. This resulted in a total of 573 distinct cases. The results are shown in Table 2, where failure may be out-of-memory (1GB), time-out (60s), or in case of `Omega+`, max-

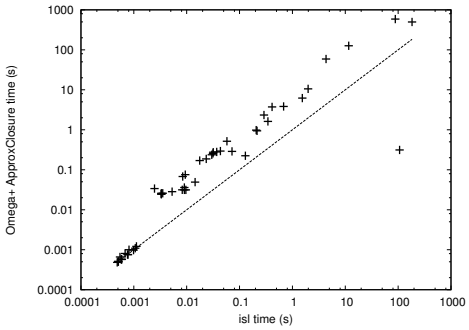

Fig. 4. Equivalence checking time

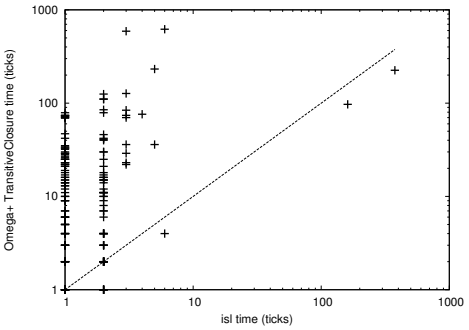

Fig. 5. Transitive closure computation time

ing out the number of constraints. Since only isl, box and Fast give an indication of whether the computed result is exact or not the results of the other methods are explicitly checked for exactness. This exactness test may also contribute to some failures. Interestingly, our "box" implementation is more accurate than both `ApproxClosure` and `calculateTransitiveClosure` on this test set. On average, the `isl` implementation is more accurate than any of the `Omega+` implementations on the test set. There are also some exceptions, however. There are two cases where one or two of the `Omega+` implementations computes an exact result while both `isl` and the box implementation do not. In all those cases where `isl` fails, the other implementations either also fail or compute an inexact result. This observation, together with the higher failure rate (compared to the box implementation), suggests that our algorithm may be trying a little bit too hard to compute an exact result.

Figure 5 shows that for those transitive closures that both `TransitiveClosure` and `isl` compute exactly, `isl` is as fast as or faster than `Omega+` in all but a few exceptional cases. This result is somewhat unexpected since `Omega+`'s `TransitiveClosure` performs its operations in machine precision, while `isl` performs all its operations in exact integer arithmetic using GMP.

		top-level								nested							
				Omega+								Omega+						
		isl	box	TC	AC	CTC	Fast	Aspic	StInG	isl	box	TC	AC	CTC	Fast	Aspic	StInG
mem	exact	70	44	58	43	53	25	15	39	37	25	35	7	31	1	1	15
	inexact	7	60	11	50	6	0	87	22	10	42	17	50	19	0	67	43
	failure	57	30	65	41	75	109	32	73	21	1	16	11	18	67	0	10
val	exact	72	44	57	43	57	28	37	39	53	35	47	23	37	7	8	28
	inexact	2	73	26	56	12	0	41	22	12	41	20	48	33	0	59	36
	failure	60	17	51	35	65	106	56	73	12	1	10	6	7	70	10	13

Table 3. Success rate of transitive closure operations from ISS experiment

7.3 Iteration Space Slicing

The ISS experiments were performed on loop nests previously used in [7] and extracted from version 3.2 of NAS Parallel Benchmarks [35] consisting of five kernels and three pseudo-applications derived from computational fluid dynamics applications. In total, 257 loops could be analyzed, but 123 have no dependences. For each of the remaining 134 loops, a dependence graph was computed using either value based dependence analysis or memory based dependence analysis [27]. Each of these dependence graphs was encoded as a single relation and passed to the transitive closure operation. The results are shown in Table 3. Since the input encodes an entire dependence graph, `isl` is expected to produce more accurate results than `Omega+` as `isl` implements Floyd-Warshall internally. We therefore also show the results on all the nested transitive closure operations computed during the execution of Floyd-Warshall. It should be noted, though, that `isl` also performs coalescing on intermediate results, so an implementation of Floyd-Warshall on top of `Omega+` may not produce results that are as accurate.

8 Conclusions and Future Work

We presented a novel algorithm for computing overapproximations of transitive closures for the general case of affine relations. The overapproximations computed by the algorithm are guaranteed to be transitively closed. The algorithm was experimentally shown to be significantly more accurate than the best known alternatives on representative benchmarks from our target applications, and our implementation is generally also faster despite performing all computations in exact integer arithmetic.

Although our algorithm can be applied to any affine relation, we have observed that the results are not very accurate if the input relation is cyclic. As part of future work, we therefore want to devise improved strategies for handling such cyclic relations. The comparison with tools for reachability or invariant analysis has revealed that our problems have quite different characteristics, in that our algorithm does not work very well on their problems while their algorithms do not work very well on ours. The design of a combined approach that could work for both classes of problems is therefore also an interesting line of research.

Acknowledgments

This work was partly funded by the European FP7 project TERAFLUX id. 249013. We would like to thank Louis-Noël Pouchet for showing us how to use `VAUCANSON`, Laure Gonnord for extending `Aspic` to produce `isl` compatible output and Jérôme Leroux for explaining how to encode relations in `Fast`.

References

1. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. Electron. Notes Theor. Comput. Sci. 267, 3–16 (October 2010)

2. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. STTT 10(5), 401–424 (2008)
3. Barthou, D., Cohen, A., Collard, J.F.: Maximal static expansion. Int. J. Parallel Programming 28(3), 213–243 (2000)
4. Barthou, D., Feautrier, P., Redon, X.: On the equivalence of two systems of affine recurrence equations. In: Euro-Par Conference. Lect. Notes in Computer Science, vol. 2400, pp. 309–313. Springer-Verlag, Paderborn (Aug 2002)
5. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. pp. 7–16. IEEE Computer Society, Washington, DC, USA (2004)
6. Beletska, A., Barthou, D., Bielecki, W., Cohen, A.: Computing the transitive closure of a union of affine integer tuple relations. In: COCOA '09: Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications. pp. 98–109. Springer-Verlag, Berlin, Heidelberg (2009)
7. Beletska, A., Bielecki, W., Cohen, A., Palkowski, M., Siedlecki, K.: Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. Parallel and Distributed Computing, International Symposium on 0, 73–80 (2009)
8. Bielecki, W., Klimek, T., Trifunovic, K.: Calculating exact transitive closure for a normalized affine integer tuple relation. Electronic Notes in Discrete Mathematics 33, 7–14 (2009)
9. Bielecki, W., Klimek, T., Palkowski, M., Beletska, A.: An iterative algorithm of computing the transitive closure of a union of parameterized affine integer tuple relations. In: Proceedings of the 4th international conference on Combinatorial optimization and applications - Volume Part I. pp. 104–113. COCOA'10, Springer-Verlag, Berlin, Heidelberg (2010)
10. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces. Ph.D. thesis, Université de Liège (1998)
11. Boigelot, B., Wolper, P.: Symbolic verification with periodic sets. In: Proceedings of the 6th International Conference on Computer-Aided Verification. Lecture Notes in Computer Science, vol. 818, pp. 55–67. Springer-Verlag (1994)
12. Boigelot, B., Herbreteau, F.: The Power of Hybrid Acceleration. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, 18th International Conference. Lecture Notes in Computer Science, vol. 4144, pp. 438–451. Springer, Seattle, WA United States (2006)
13. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,. pp. 337–351. TACAS '09, Springer-Verlag, Berlin, Heidelberg (2009)
14. Bozga, M., Iosif, R., Konecný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 227–242. Springer (2010)
15. Bultan, T., Gerber, R., Pugh, W.: Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. ACM Trans. Program. Lang. Syst. 21(4), 747–789 (1999)
16. Chen, C.: Omega+ library (2009), http://www.chunchen.info/omega/
17. Chin, W.N., Khoo, S.C.: Calculating sized types. Higher Order Symbol. Comput. 14(2-3), 261–300 (2001)
18. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: CAV'98, LNCS 1427. pp. 268–279. Springer (1998)
19. Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser Boston (2000)

20. Demri, S., Finkel, A., Goranko, V., van Drimmelen, G.: Towards a model-checker for counter systems. In: Graf, S., Zhang, W. (eds.) Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006, Beijing, China, October 23-26, 2006. Lecture Notes in Computer Science, vol. 4218, pp. 493–507. Springer (2006)
21. Feautrier, P.: Parametric integer programming. Operationnelle/Operations Research 22(3), 243–268 (1988)
22. Feautrier, P.: The Data Parallel Programming Model, LNCS, vol. 1132, chap. Automatic Parallelization in the Polytope Model, pp. 79–100. Springer-Verlag (1996)
23. Feautrier, P., Griebl, M., Lengauer, C.: On index set splitting. In: Parallel Architectures and Compilation Techniques (PACT'99). Newport Beach, CA (Oct 1999)
24. Feautrier, P., Gonnord, L.: Accelerated invariant generation for c programs with aspic and c2fsm. Electron. Notes Theor. Comput. Sci. 267, 3–13 (October 2010)
25. Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive closure of infinite graphs and its applications. In: Huang, C.H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) Languages and Compilers for Parallel Computing, 8th International Workshop, LCPC'95, Columbus, Ohio, USA, August 10-12, 1995, Proceedings. Lecture Notes in Computer Science, vol. 1033, pp. 126–140. Springer (1996)
26. Lombardy, S., Régis-Gianas, Y., Sakarovitch, J.: Introducing VAUCANSON. Theor. Comput. Sci. 328(1-2), 77–96 (2004)
27. Pugh, W., Wonnacott, D.: An exact method for analysis of value-based array data dependences. In: Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing. pp. 546–566. Springer-Verlag (1994)
28. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: SAS'04. pp. 53–68 (2004)
29. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons (1986)
30. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
31. Vardhan, A., Viswanathan, M.: Lever: A tool for learning based verification. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4144, pp. 471–474. Springer (2006)
32. Verdoolaege, S.: isl: An integer set library for the polyhedral model. In: Fukuda, K., Hoeven, J., Joswig, M., Takayama, N. (eds.) Mathematical Software - ICMS 2010, Lecture Notes in Computer Science, vol. 6327, pp. 299–302. Springer Berlin / Heidelberg (2010)
33. Verdoolaege, S., Cohen, A., Beletska, A.: Transitive closures of affine integer tuple relations and their overapproximations. Tech. Rep. RR-7560, INRIA (Mar 2011)
34. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. In: Computer Aided Verification 21. pp. 599–613. Springer (Jun 2009)
35. NAS benchmarks suite. `http://www.nas.nasa.gov`