Transitive Closures of Affine Integer Tuple Relations and their Overapproximations

Sven Verdoolaege, Albert Cohen, Anna Beletska

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Transitive Closures of Affine Integer Tuple Relations and their Overapproximations

Sven Verdoolaege — Albert Cohen — Anna Beletska

N° 7560

March 2011

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7560--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

Transitive Closures of Affine Integer Tuple Relations and their Overapproximations

Sven Verdoolaege , Albert Cohen , Anna Beletska

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet ALCHEMY

Rapport de recherche n° 7560 — March 2011 — 35 pages

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. 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.

Key-words: affine integer tuple relation, dependence graph, Floyd-Warshall, maximal reaching path length, polyhedral model, strongly connected components, transitive closure

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Transitive Closures of Affine Integer Tuple Relations and their Overapproximations

Résumé : L'ensemble des chemins dans un graphe joue un rôle important pour de nombreuses applications dans le domaine de l'analyse des systèmes. Dans le cas des relations entre tuples d'entiers, lesquelles permettent de représenter des graphes potentiellement infinis, cet ensemble correspond à la clôture transitive de la relation. Lorsque ces relations sont décrites uniquement à l'aide de contraintes affines et de projections, elles ont la puissance d'expression de l'arithmétique de Presburger, et elles donnent lieu à des algorithmes relativement efficaces en pratique. Malheureusement, la clôture transitive d'une telle relation quasi-affine n'est pas forcément quasi-affine, impliquant le recours à des approximations. En particulier, la plupart des applications à l'analyse des systèmes requiert des sur-approximations. Les résultats antérieurs se concentrent soit sur des sous-approximations soit sur des cas particuliers de relations affines. Nous proposons un nouvel algorithme pour le calcul de sur-approximations de clôtures transitives, dans le cas général des relations quasi-affines (convexes ou non). Nos résultats expérimentaux portent sur des relations non-triviales issues d'applications réelles, et démontrent que notre algorithme est plus précis et plus rapide en moyenne que les meilleures alternatives connues.

Mots-clés : relation affine entre des tuples d'entiers, graphe de dépendence, Floyd-Warshall, longueur maximale d'un chemin d'accessibilité, modèle polyhédrique, composantes fortement connexes, clôture transitive

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

Figure 1: A sequential loop

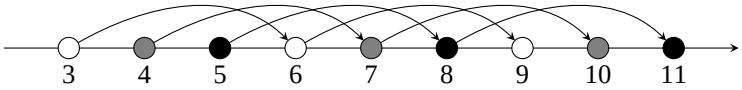

Figure 2: The dependences and slices of the loop in Figure 1

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

Figure 3: A parallelized version of the loop in Figure 1

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 [4,5,8,16,18,19,24,28]. 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. [28], 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 [2,7,10,25,28]. and/or the study of sub-classes, including sub-polyhedral domains, where an exact computation is possible [3, 11–15, 19, 21]. 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.

Approximation for the general case of quasi-affine relations has only been investigated by Kelly et al. [28], and their technique only provides an underapproximation (which is sufficient for their applications). Yet the vast majority of the applications require overapproximations, and the (unimplemented) algorithm proposed by Kelly et al. computing overapproximations is very inaccurate. Overapproximations have been considered by Beletska et al. [7], but in a more limited setting.

We use Iteration Space Slicing (ISS) to illustrate the application of the transitive closure to quasi-affine relations [8]. 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 2 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. The construction of these groups can be formulated as a transitive closure on a relation representing the (extended) dependence graph. The resulting relation connects iterations to directly or indirectly depending iterations, from which the groups can be derived. In the example, three such groups can be discerned, indicated by different colors of the nodes in Figure 2. 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 3. 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.

In this paper, we present an algorithm for computing overapproximations of transitive closures. The algorithm subsumes those of [7] and [2]. Furthermore, it is experimentally shown to be exact in more instances from our applications than that of [28] 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 [28], while the remaining one is new. Finally, we provide a more extensive experimental evaluation on more difficult instances. As an indication, Kelly et al. [28] 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 briefly explain some of our target applications in Section 3 and we discuss related work in Section 4. Section 5 details the core of our algorithm. Section 6 studies decomposition methods to increase accuracy and speed. Section 7 describes the implementation, while Section 8 explains the relationship with reachability analysis. The results of our experiments are shown in Section 9.

2 Background

We consider binary relations on $\mathbb{Z}^d$, i.e., relations mapping d-tuples of integers to d-tuples of integers. The composition of two relations R and S is denoted $S \circ R$. A relation R is *transitively closed* if $R \circ R = R$. The *transitive closure* of R, denoted R^+, is the (inclusion-wise) smallest relation T such that $R \subseteq T$ and T is transitively closed. 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

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

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 $\mathbf{c} \in \mathbb{Z}^m$. 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 [28]. 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, we will also make use of quasi-affine sets during our computations. These sets are defined in essentially the same way as quasi-affine relations. The only difference is that sets are unary relations on integer tuples instead of binary relations. Sets can be obtained from relations in the following ways. The *domain* of a relation R is the set $\mathrm{dom}\, R := \{ \mathbf{x}_1 \in \mathbb{Z}^d \mid \exists \mathbf{x}_2 \in \mathbb{Z}^d : \mathbf{x}_1 \to \mathbf{x}_2 \in R \}$. The *range* of a relation R is the set $\mathrm{ran}\, R := \{ \mathbf{x}_2 \in \mathbb{Z}^d \mid \exists \mathbf{x}_1 \in \mathbb{Z}^d : \mathbf{x}_1 \to \mathbf{x}_2 \in R \}$. The *difference set* of a relation R is the set $\Delta R := \{ \boldsymbol{\delta} \in \mathbb{Z}^d \mid \exists \mathbf{x} \to \mathbf{y} \in R : \boldsymbol{\delta} = \mathbf{y} - \mathbf{x} \}$. 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., $S_1 + S_2 = \{ \mathbf{a} + \mathbf{b} \mid \mathbf{a} \in S_1 \wedge \mathbf{b} \in S_1 \}$. 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 [23], 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 somehow depend on them for their execution. Two common types of dependences are *memory based* dependences and *value based* dependences. There is a memory based dependence between iteration **i** and iteration **j** if **i** is executed before **j** and both access the same memory location, with at least one of the accesses being a write. The meaning of such a dependence is that after transformation, **i** should still be executed before **j**. There is a value based dependence between iteration **i** and iteration **j** if **i** performs the

last write to a memory location read by **j**, meaning that **j** reads the value written by **i**. In Figure 1, each array element is only written once, so the two types of dependences result in the same relation: $n \mapsto \{ i \rightarrow i + 3 \mid 3 \leq i, i + 3 \leq n \}$. The graph with the statements and their iterations domains as nodes and the dependence relations as edges is called the *dependence graph*.

3 Applications

3.1 Sized Types

Chin et al. [18] present a technique for computing relationships between the sizes of the arguments of a function in the context of a functional programming language. In particular, they are interested in the relationship between the sizes of input and output arguments and in the relationship between the sizes of the original input arguments of a function and the sizes of the inputs to any (possibly indirect) recursive call of the same function. In case a function does not call itself recursively, the relationship between input and output sizes can be derived using some inference rules. If these relationships can be represented using affine constraints, then the possible values of these sizes can be represented using an affine set. In case a function does call itself recursively, similar rules can be used to derive an affine relation that maps the sizes of the original call to the sizes of the direct recursive call. The relationship between the original sizes and those of *any* recursive call are then obtained by computing the transitive closure of this relation. Combining the exact transitive closure with the affine set representing the leaf call results in the least fixed point of the sized type. If the transitive closure cannot be computed exactly, approximations are allowed, but the sized type should still be a fixed point. This means that the transitive closure should be overapproximated and that furthermore the result should be closed. Realizing that `Omega` computes underapproximations instead of overapproximations, the authors propose a heuristic method that manipulates the output of `Omega` until an overapproximation is obtained. As we will show in Section 9.1, our algorithm produces better results.

3.2 Iteration Space Slicing

As already illustrated in the introduction, the purpose of iteration space slicing is to partition the iteration domains of a program fragment into slices that are not connected through any dependences. The simplest way to reduce this problem to a transitive closure computation is to consider the union of the dependence relation and its inverse. Any pair of iterations that is connected through one or more applications of this union belongs to the same slice in the partition. The transitive closure of this relation therefore connects each iteration to each other iteration in the same slice. Taking the union with the identity relation on the iteration domains, we obtain a relation between each pair of iterations in the same slice. In the example, the transitive closure of the union of $n \mapsto \{ i \rightarrow i + 3 \mid 3 \leq i, i + 3 \leq n \}$ and its inverse is

$$n \mapsto \{ i \rightarrow j \mid \exists \alpha : i - j = 3\alpha \wedge 3 \leq i, j \leq n \wedge (i > j \vee i < j) \}.$$

This transitive closure can be computed exactly using the techniques described below, although the representation of the result will be slightly more complicated than the representation shown above. Taking the union with the identity relation on $n \mapsto \{ i \mid$

$3 \leq i \leq n\,\}$ results in

$$n \mapsto \{\, i \rightarrow j \mid \exists \alpha : i - j = 3\alpha \wedge 3 \leq i, j \leq n \,\}.$$

This relation can then be further manipulated to obtain a representative of each slice. The most natural choice is to compute the (lexicographically) minimal element of the image of a given element in the domain. All elements in the same slice will then be mapped to the same minimal element, which can then be used to identify the slice. In the example, we obtain

$$n \mapsto \{\, i \rightarrow j \mid \exists \alpha : i - j = 3\alpha \wedge 3 \leq i \leq n \wedge 3 \leq j \leq 5 \,\}.$$

The actual algorithm for computing slices proposed by Beletska et al. [8] is slightly more complicated as it avoids computing the transitive closure of a cyclic relation. In general, this more complicated algorithm produces more accurate results.

3.3 Maximal Static Expansion

Scalar variables or array elements are often reused throughout the course of a program to store different results. This reuse leads to extra memory based dependences that can hamper parallelization. Maximal static expansion [4] is a technique for removing some of these memory based dependences by allocating independent results to different memory locations. In particular, the original storage mapping of the program, mapping accesses to memory locations, is expanded in the sense that some accesses that used to be mapped to the same memory location may now be mapped to distinct memory locations. Essentially, the expansion is obtained by adding extra dimensions to the arrays or by turning scalars into arrays. The expansion is static if no expansion is performed on write accesses that, based on a conservative analysis, may have been mapped to the same memory location in the original program and that may end up being read by the same read access. That is, the extra dimensions, if any, are not allowed to differentiate between such accesses. Since an expansion is a partition, it needs to be defined in terms of an equivalence relation. In particular, the defining relation needs to be transitively closed. We therefore need to compute the transitive closure of both relations involved, i.e., that of possibly mapping to the same memory location in the original program and that of possibly reaching the same read access. Note that, as in the case of iteration space slicing, it is essential to obtain a representation of the actual transitive closure and not just of some reachable set.

3.4 Free Schedules

A free schedule [20] executes each operation as soon as all the operations on which it depends have been executed. Given a dependence graph G, the free schedule can be obtained by first computing the lengths of all reaching paths in G to a given iteration and then computing the maximum of those lengths. The length of a reaching path is the exponent k associated to an element in the power G^k of the dependence graph. These lengths can therefore be obtained by projecting out the domain of G^k and subsequently treating k as the image of a mapping defined on the range of G^k. As explained in Section 5.1, the power of a relation can be computed from the transitive closure of a related relation. If the transitive closure and the power are overapproximations, then there may be extra reaching path lengths associated to a given operation, possibly affecting (increasing) the maximum length. The resulting schedule may no longer be the

free schedule, but it will still be a valid schedule. Underapproximations, on the other hand, could lead to invalid schedules.

3.5 Equivalence Checking

Assume that we are given two programs that can be represented in the polyhedral model and that we want to check whether those two programs are equivalent, i.e., that their output arrays will contain the same values when given identical values for their input arrays. Both programs can be represented as an inverted dependence graph that has been annotated with the operations that are performed in each statement [37]. The two programs are equivalent if every pair of paths that start out from the same elements of the same output arrays are such that they pass through nodes that perform the same operation and end up either in nodes that compute the same constant or in nodes that read the same elements from the same input arrays. We can perform this check on an automaton that is the cross product of the two annotated dependence graphs. This check is essentially a reachability analysis, but Barthou et al. [5] propose to perform this reachability analysis by first computing a regular expression for all paths in the automaton from initial states to leaf states, subsequently translating this regular expression into operations on the dependence relations and finally applying the result of these operations on an initial state that represents pairs of identical output array elements. If the regular expression contains cycles, then these are translated into transitive closures.

Note that unlike most of our other target applications, transitive closures are not essential for solving the equivalence checking problem. In fact, much better results can be achieved using different approaches [37]. However, given our experience with this problem, we have easy access to a set of problem instances. Furthermore, as in most of our other target applications, the transitive closures involved are based on dependence relations and are therefore representative for the kinds of relations for which we want to compute transitive closures. In fact, they represent the more difficult kinds of relations as they are based on pairs of dependence relations and may be the result of nested transitive closure operations.

4 Related Work

The seminal work of Kelly et al. [28] 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 these underapproximations. For overapproximations, they apparently only consider some kind of "box-closure", which we recall in Section 7 and which is considerably less accurate than our algorithm.

Bielecki et al. [9] 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. [7] 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. [10] 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 [28]. 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 [9, 28].

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" [21], while our relations over labeled tuples correspond to Presburger counter systems [21], 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. [3] and their references or the work of Bozga et al. [14]. 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. [30], Feautrier and Gonnord [25] or Ancourt et al. [2], 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. [2] is essentially the same as that used by Boigelot and Herbreteau [13], 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.

5 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.

5.1 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 closure 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.

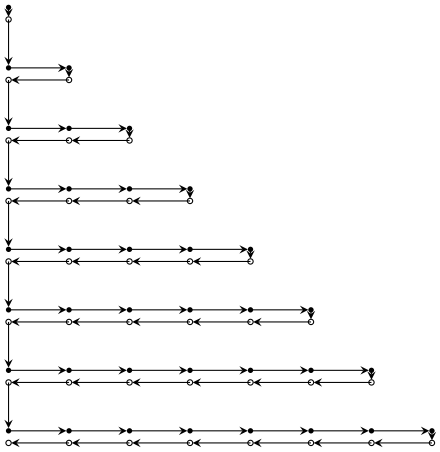

Figure 4: A graphical representation of the relation in (3). The R domain is marked with filled circles, while the L domain is marked with open circles.

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 [20, Theorem 23].

As a related example, let us first consider the relation

$$\begin{aligned}
&\{\,\mathrm{R}(i,j) \to \mathrm{R}(i,1+j) \mid j \geq 0 \wedge j \leq -1+i\,\} \cup \\
&\{\,\mathrm{R}(i,i) \to \mathrm{L}(i,i) \mid i \geq 0\,\} \cup \\
&\{\,\mathrm{L}(i,j) \to \mathrm{L}(i,-1+j) \mid j \geq 1 \wedge j \leq i\,\} \cup \\
&\{\,\mathrm{L}(i,0) \to \mathrm{R}(1+i,0) \mid i \geq 0\,\},
\end{aligned} \tag{3}$$

shown graphically in Figure 4. It should be fairly clear from the figure that the transitive closure of this relation *is* affine. Indeed, this transitive closure can be represented as

$$\begin{aligned}
&\{\,\mathrm{R}(i,j) \to \mathrm{R}(i,j') \mid j \geq 0 \wedge j' \geq 1+j \wedge j' \leq i\,\} \cup \\
&\{\,\mathrm{R}(i,j) \to \mathrm{R}(i',j') \mid j \geq 0 \wedge j \leq i \wedge j' \geq 0 \wedge j' \leq i' \wedge i' \geq 1+i\,\} \cup \\
&\{\,\mathrm{R}(i,j) \to \mathrm{L}(i',j') \mid j \geq 0 \wedge j \leq i \wedge j' \geq 0 \wedge j' \leq i' \wedge i' \geq i\,\} \cup \\
&\{\,\mathrm{L}(i,j) \to \mathrm{L}(i,j') \mid j' \geq 0 \wedge j' \leq -1+j \wedge j \leq i\,\} \cup \\
&\{\,\mathrm{L}(i,j) \to \mathrm{L}(i',j') \mid j \geq 0 \wedge j \leq i \wedge j' \geq 0 \wedge j' \leq i' \wedge i' \geq 1+i\,\} \cup \\
&\{\,\mathrm{L}(i,j) \to \mathrm{R}(i',j') \mid j \geq 0 \wedge j \leq i \wedge j' \geq 0 \wedge j' \leq i' \wedge i' \geq 1+i\,\}.
\end{aligned}$$

However, it should be equally clear from the figure that the path lengths are *not* affine. For example the length of a path from $\mathrm{L}(i_1,0)$ to $\mathrm{L}(i_2,0)$, with $i_2 > i_1$ is equal to $i_2^2+3i_2-i_1^2-3i_1-1$, a fact that can easily be verified using `barvinok` [35]. This means that the transitive closure of the following relation is not affine:

$$\begin{aligned}
&\{\,\mathrm{R}(i,j,k) \to \mathrm{R}(i,1+j,k+1) \mid j \geq 0 \wedge j \leq -1+i\,\} \cup \\
&\{\,\mathrm{R}(i,i,k) \to \mathrm{L}(i,i,k+1) \mid i \geq 0\,\} \cup \\
&\{\,\mathrm{L}(i,j,k) \to \mathrm{L}(i,-1+j,k+1) \mid j \geq 1 \wedge j \leq i\,\} \cup \\
&\{\,\mathrm{L}(i,0,k) \to \mathrm{R}(1+i,0,k+1) \mid i \geq 0\,\}.
\end{aligned}$$

This example was inspired by Example 2 of [31, Section 3.1], which involves another union of translations over fixed distances such that the transitive closure is affine, but the parametric power is not.

5.2 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 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 5.3. 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

$$P^k = \{\, \mathbf{x} \to \mathbf{y} \mid \exists \boldsymbol{\delta} \in \mathcal{D}^k : \mathbf{y} = \mathbf{x} + \boldsymbol{\delta} \,\}, \quad \text{with} \quad \mathcal{D}^k = k\,\Delta \quad \text{and} \quad k \in \mathbb{Z}_{\geq 1}, \tag{4}$$

and intersect domain and range with those of R,

$$\mathcal{P}_k(R) = P^k \cap (\operatorname{dom} R \to \operatorname{ran} R)\,. \tag{5}$$

Example 5.1 *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 (4) may not be affine in general and then the same holds for P. 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 (5). Otherwise, we drop the definition of $\mathcal{D}^k$ in (4) 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 $k\,\Delta$.

5.2.1 No parameters or existentially quantified variables

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

$$\langle \mathbf{a}, \mathbf{x} \rangle + c\,k \geq 0, \tag{6}$$

meaning that the constraint in (6) is valid for $k\,\Delta$. Our approximation $\mathcal{D}^k$ of $k\,\Delta$ is then the set bounded by the constraints in (6). In this special case, we compute essentially

the same approximation as [2]. 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 [32, Theorem 16.4] and that therefore $\mathcal{D}^k = k\,\Delta$.

Example 5.2 *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* $\mathcal{D}^k = \{\delta \mid 2k \leq \delta \leq 3k\}$. *Therefore,* $\mathcal{P}_k(R) = P^k = \{x \to y \mid 2k \leq y - x \leq 3k\}$ *and* $\mathcal{T}(R) = \{x \to y \mid y - x \geq 2\}$.

5.2.2 Parameters

If the description of Δ does involve parameters then we cannot simply multiply the parametric constant by k as 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

$$\Delta' = \Delta R' = \{\boldsymbol{\delta} \in \mathbb{Z}^{n+d} \mid \exists (\mathbf{s}, \mathbf{x}) \to (\mathbf{s}, \mathbf{y}) \in R' : \boldsymbol{\delta} = (\mathbf{s} - \mathbf{s}, \mathbf{y} - \mathbf{x})\}. \tag{7}$$

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

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

are also valid for $k\,\Delta$. For constraints of the form

$$\langle \mathbf{a}, \mathbf{x} \rangle + \langle \mathbf{b}, \mathbf{s} \rangle + c \geq 0, \tag{9}$$

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.,

$$\Delta \cap \{\boldsymbol{\delta} \mid \langle \mathbf{b}, \mathbf{s} \rangle + c > 0\} = \emptyset, \tag{10}$$

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 (9) 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 (6), (8) and (9). Constraints of the form (9) for which (10) 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 (9) get dropped.

Example 5.3 *Consider the relation from [28, Figure 12]:*

$$\begin{aligned} &n \to \{(i, j) \to (i, 1 + j) \mid i \geq 1 \wedge j \geq 1 \wedge j \leq -1 + n \wedge i \leq n\} \cup \\ &n \to \{(i, n) \to (1 + i, 1) \mid i \geq 1 \wedge i \leq -1 + n\}. \end{aligned} \tag{11}$$

Kelly et al. [28] compute the exact transitive closure through an application of their variation of the incremental computation. However, the transitive closure can also be

computed exactly using the basic technique of this section. The difference set of the second disjunct is

$$\Delta = n \to \{ (1, 1-n) \mid n \geq 2 \}.$$

The second coordinate corresponds to an equality $x_2 = 1 - n$*, which in turn can be split up into two inequalities* $x_2 \geq 1 - n$ *and* $x_2 \leq 1 - n$*. Only the latter satisfies the condition in* (10)*. Our approximation for the non-affine* $k\Delta = n \to \{ (k, k-kn) \mid n \geq 2 \}$ *is therefore* $\mathcal{D}^k = n \to \{ (k, x_2) \mid k \geq 1 \wedge x_2 \leq 1 - n \wedge n \geq 2 \}$*. This means that any negative movement of at least* $n - 1$ *is allowed in the second coordinate as long as the first coordinate is increased. The actual transitive closure of the relation in* (11) *allows movements to* any *element with a higher first coordinate. The inaccurateness in* $\mathcal{D}^k$ *therefore does not result in any inaccurateness of the transitive closure on this example. We find that this kind of behavior is fairly typical for transitive closures of dependence relations.*

Example 5.4 *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 treating the parameters in a special way, 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 that can be obtained. In particular, the following approximation is more accurate

$$n \to \{ (x, y) \to (x', y') \mid y' \leq 2 - n + x + y - x' \wedge n \geq 2 \wedge x' \geq 1 + x \}.$$

5.2.3 Existentially quantified variables

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 [22]. The result is an explicit representation of each existentially quantified variables as greatest integer parts

of affine expressions 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' (7).

Example 5.5 *Consider the relation*

$$R = n \rightarrow \{ x \rightarrow y \mid \exists \alpha_0, \alpha_1 : 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -1 - x + y \wedge y \geq 6 + x \}.$$

The difference set of this relation is

$$\Delta = \Delta R = n \rightarrow \{ x \mid \exists \alpha_0, \alpha_1 : 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -1 + x \wedge x \geq 6 \}.$$

The existentially quantified variables can be defined in terms of the parameters and variables as

$$\alpha_0 = \left\lfloor \frac{-2+n}{7} \right\rfloor \qquad \text{and} \qquad \alpha_1 = \left\lfloor \frac{-1+x}{5} \right\rfloor.$$

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

$$n \rightarrow \{ (x,z) \rightarrow (y,w) \mid \exists \alpha_0, \alpha_1, k, f : k \geq 1 \wedge y = x + f \wedge w = z + k \wedge 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -k + x \wedge x \geq 6k \}.$$

Projecting out the final coordinates encoding the length of the paths, results in the exact transitive closure

$$R^+ = n \rightarrow \{ x \rightarrow y \mid \exists \alpha_0, \alpha_1 : 7\alpha_1 = -2 + n \wedge 6\alpha_0 \geq -x + y \wedge 5\alpha_0 \leq -1 - x + y \}.$$

5.3 Multiple Disjuncts

When the set of distances Δ is a proper union of basic sets $\Delta = \cup_i \Delta_i$, we apply the technique of Section 5.2 to each Δ_i separately, yielding approximations $\mathcal{D}_i^k$ of $k_i\,\Delta_i$ and corresponding paths P_i^k from (4). 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 $S_i = \mathrm{Id}_{\mathbb{Z}^d} \cap \{ \mathbf{x} \rightarrow \mathbf{y} \mid k_i = 0 \}$ and compute the set of global paths as

$$P^k = \left((P_m^{k_m} \cup S_m) \circ \cdots \circ (P_2^{k_2} \cup S_2) \circ (P_1^{k_1} \cup S_1) \right) \cap \{ \mathbf{x} \rightarrow \mathbf{y} \mid k = \sum_i k_i > 0 \}. \quad (12)$$

The final constraint ensures that at least one step is taken. The approximation of the power is then again computed according to (5). As explained in Section 5.1, $\mathcal{P}_k(R)$ can be represented as $\mathcal{T}(C)$, with $C = R \times \{ i \rightarrow i + 1 \}$. 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 (12) 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 [7]. 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 an significant loss in accuracy. Finally, if neither of these optimizations apply, then after each composition in (12) we "coalesce" the resulting relation. Coalescing detects pairs of disjuncts that can be replaced by a single disjunct without introducing any spurious elements [36].

5.4 Properties

By construction (Section 5.2 and Section 5.3), we have the following lemma.

Lemma 5.6 *$\mathcal{P}_k(R)$ 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. $\mathcal{P}_k(R)$ can be represented as $\mathcal{T}(C)$, with $C = R \times \{ i \to i + 1 \}$. $\mathcal{T}(R)$ is obtained from $\mathcal{T}(C)$ by projecting out the final coordinates. We immediately have the following lemma.

Lemma 5.7 *$\mathcal{T}(R)$ 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 5.6. 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 [28, 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.

In some applications, notably those of [18], [4] and [8], it is not sufficient that the computed approximation of the transitive closure be an overapproximation, it should also be transitively closed. 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 (6) and therefore also their sum. Constraint (9) is also satisfied for $\mathbf{x}_1 + \mathbf{x}_2$ and

so $\mathbf{x}_1 + \mathbf{x}_2 \in \mathcal{D}^{k_1+k_2}$. We conclude that in the single disjunct case, P^k in (4) is transitively closed, which in turn implies that also P^k in (12) 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 5.8 *$\mathcal{T}(R)$ is a transitively closed overapproximation of R^+.*

6 Decomposition Methods

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 [28]: we apply the modified Floyd-Warshall algorithm internally after partitioning the domain and we apply an incremental computation method.

6.1 Strongly Connected Components

Computations in Section 5.2 and Section 5.3 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 (5). 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 where the vertices are the disjuncts in R and the 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 [33]. 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 we can 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

$$R_i \circ R_j \not\subseteq R_j \circ R_i. \tag{13}$$

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 (12). The combination must be performed in the correct order: the results of the components should be combined according to a topological ordering of the components. This topological ordering is a byproduct of Tarjan's algorithm. The decomposition preserves the validity of Lemma 5.6. The exactness check of Section 5.4 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 (5), for the powers of the components, implying that the final result is also transitively closed. If (13) is ever

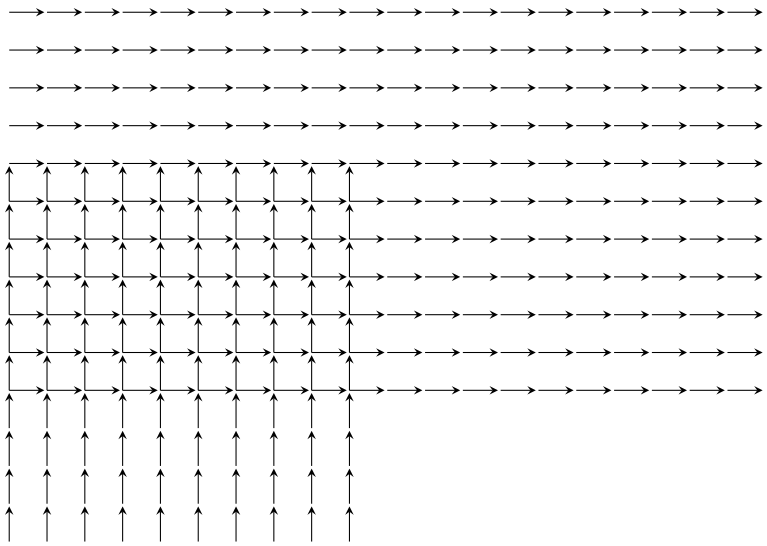

Figure 5: The relation from Example 6.1

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 (13) has been used. If the check fails, we recompute the result without a decomposition into SCCs.

Example 6.1 *Consider the relation in example* `closure4` *that comes with the Omega calculator [26],* $R = R_1 \cup R_2$*, with*

$$R_1 = \{ (x,y) \to (x,y+1) \mid 1 \le x,y \le 10 \}$$
$$R_2 = \{ (x,y) \to (x+1,y) \mid 1 \le x \le 20 \wedge 5 \le y \le 15 \}.$$

This relation is shown graphically in Figure 5. The basic technique of Section 5 would not be able to compute the exact transitive closure for this relation since the computed approximation would allow any path from a domain element to a range element that moves right and up. However, a decomposition into strongly connected components does lead to the exact result. We have

$$R_1 \circ R_2 = \{ (x,y) \to (x+1,y+1) \mid 1 \le x \le 9 \wedge 5 \le y \le 10 \}$$
$$R_2 \circ R_1 = \{ (x,y) \to (x+1,y+1) \mid 1 \le x \le 10 \wedge 4 \le y \le 10 \}.$$

Clearly, $R_1 \circ R_2 \subseteq R_2 \circ R_1$ *and so*

$$(R_1 \cup R_2)^+ = (R_2^+ \circ R_1^+) \cup R_1^+ \cup R_2^+.$$

Example 6.2 *Consider the relation on the right of [7, Figure 2], reproduced in Figure 6. Note that one of the arrows is missing in the original figure. The relation can be described as* $R = R_1 \cup R_2 \cup R_3$*, with*

$$R_1 = n \mapsto \{ (i,j) \to (i+3,j) \mid i \le 2j-4 \wedge i \le n-3 \wedge j \le 2i-1 \wedge j \le n \}$$
$$R_2 = n \mapsto \{ (i,j) \to (i,j+3) \mid i \le 2j-1 \wedge i \le n \wedge j \le 2i-4 \wedge j \le n-3 \}$$
$$R_3 = n \mapsto \{ (i,j) \to (i+1,j+1) \mid i \le 2j-1 \wedge i \le n-1 \wedge j \le 2i-1 \wedge j \le n-1 \}.$$

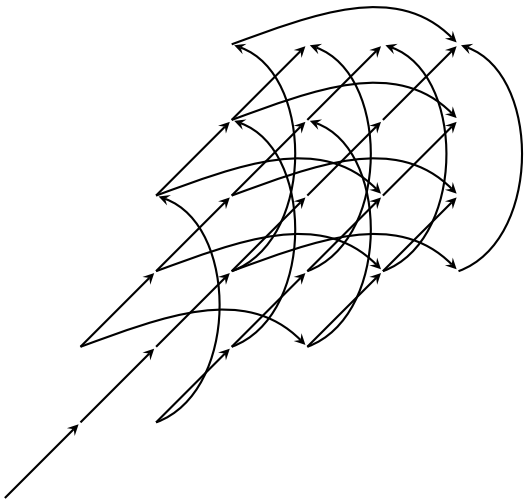

Figure 6: The relation from Example 6.2

The figure shows this relation for $n = 7$. Beletska et al. [7] explain that they are not able to compute the exact parametric power of this relation using their technique (a special case of the technique in Section 5), but that they can compute the exact transitive closure. Using the decomposition into strongly connected components, we can also compute the parametric power exactly. Both $R_3 \circ R_1 \subseteq R_1 \circ R_3$ and $R_3 \circ R_2 \subseteq R_2 \circ R_3$. R_3 can therefore be moved forward in any path. For the other two basic relations, we have both $R_2 \circ R_1 \not\subseteq R_1 \circ R_2$ and $R_1 \circ R_2 \not\subseteq R_2 \circ R_1$ and so R_1 and R_2 form a strongly connected component. By computing the power of R_3 and $R_1 \cup R_2$ separately and composing the results, the power of R can be computed exactly.

6.2 Domain Partitioning

We have just seen how to split off disjuncts that can be serialized with respect to other disjuncts. Within a strongly connected component of disjuncts, however, we may still be able to group their domains and ranges into disjoint sets. This typically happens when an entire dependence graph is encoded in a single relation, as is done in, e.g., [4, Section 6.1]: the original iteration domains are encoded as disjoint subsets of $\mathbb{Z}^d$, with the domain and range of each individual disjunct in the input relation entirely contained in a single of these disjoint subsets. For disjuncts mapping across different subsets, i.e., encodings of dependences between different iteration domains, it makes little sense to compute the difference set. Since the algorithm of Section 5 is based on nothing but this difference set, we cannot expect to obtain accurate results on such inputs. The solution is to detect disjoint groups of domains and ranges and to apply the modified Floyd-Warshall algorithm of [28], reproduced in Algorithm 1, on relations between these groups. Let the input relation R be a union of m basic relations R_i. Let D_{2i} be the domains of R_i and D_{2i+1} the ranges of R_i. The first step groups overlapping D_js until a partition is obtained. If the partition consists of a single part, then we continue with the standard algorithm. Otherwise, we apply Floyd-Warshall on the graph whose vertices are the parts of the partition and whose edges are the R_i attached to the appropriate

Algorithm 1: The modified Floyd-Warshall algorithm of [28]

Input: Relations R_{pq}, $0 \le p, q < n$
Output: Updated relations R_{pq} such that each relation R_{pq} contains all indirect paths from p to q in the input graph

```
1 for r ∈ [0, n − 1] do
2     R_rr := T(R_rr)
3     for p ∈ [0, n − 1] do
4         for q ∈ [0, n − 1] do
5             if p ≠ r or q ≠ r then
6                 R_pq := R_pq ∪ (R_rq ∘ R_pr) ∪ (R_rq ∘ R_rr ∘ R_pr)
```

pairs of vertices. Consider a partition of n parts G_k. We construct n^2 relations

$$R_{pq} := \bigcup_{i \text{ s.t. } \operatorname{dom} R_i \subseteq G_p \wedge \operatorname{ran} R_i \subseteq G_q} R_i,$$

apply Algorithm 1 and return the union of all resulting R_{pq} as $\mathcal{T}(R)$. Each iteration of the r-loop in Algorithm 1 updates all relations R_{pq} to include paths that go from p to r, possibly stay there for a while, and then go from r to q. Note that paths that "stay in r" include all paths that pass through earlier vertices since R_{rr} itself has been updated accordingly in previous iterations of the outer loop. In principle, it would be sufficient to use the R_{pr} and R_{rq} computed in the previous iteration of the r-loop in Line 6. However, from an implementation perspective, it is easier to allow either or both of these to have been updated in the same iteration of the r-loop. This may result in duplicate paths, but these can usually be removed by coalescing the result of the union in Line 6, which should be done in any case. The transitive closure in Line 2 is performed using a recursive call. This recursive call includes the partitioning step, but the resulting partition will usually be a singleton. The result of the recursive call will either be exact or an overapproximation. The final result of Floyd-Warshall is considered exact only if each recursive call produces an exact result.

To see that the Floyd-Warshall algorithm preserves closedness, let $\mathbf{x} \to \mathbf{y} \in R_{ij}$ and $\mathbf{y} \to \mathbf{z} \in R_{jk}$. Let r_1 and r_2 be the iterations of the outermost loop of the algorithm in which these elements were introduced, or -1 if they are elements of the input relation. Let r be the largest of r_1 and r_2. If $j > r$, then $\mathbf{x} \to \mathbf{z}$ is introduced in iteration j. If $j \le r$, then $\mathbf{x} \to \mathbf{z}$ is introduced in iteration r.

As explained at the start of this section, applying Floyd-Warshall should produce more accurate results on certain types of relations. This expectation is confirmed by the experiments of Section 9. However, the algorithm comes with the cost of having to apply recursive transitive closures. We would therefore like to make sure that applying Floyd-Warshall will never produce less accurate results on *any* relation. Unfortunately, this may not be the case in general due to our handling of existentially quantified variables. Nevertheless, we have not been able to construct a counterexample and we expect them to be rare if they exist at all.

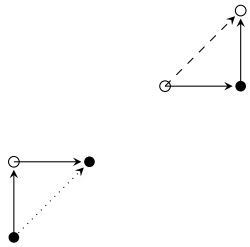

Figure 7: The relation (solid arrows) on the right of Figure 1 of [7] and its transitive closure

Example 6.3 *Consider the relation on the right of Figure 1 of [7], reproduced in Figure 7. This relation can be described as*

$$\{(x,y) \to (x_2,y_2) \mid (3y = 2x \wedge x_2 = x \wedge 3y_2 = 3 + 2x \wedge x \geq 0 \wedge x \leq 3) \vee \\ (x_2 = 1 + x \wedge y_2 = y \wedge x \geq 0 \wedge 3y \geq 2 + 2x \wedge x \leq 2 \wedge 3y \leq 3 + 2x)\}.$$

The basic approach of Section 5 is unable to find the exact transitive closure for this map. However, the exact transitive closure can be computed using either the domain partitioning of this section or the incremental computation of Section 6.3. Let us consider domain partitioning. Note that the domain of the upward relation overlaps with the range of the rightward relation and vice versa, but that the domain of neither relation overlaps with its own range or the domain of the other relation. The domains and ranges can therefore be partitioned into two parts, P_0 and P_1, shown as the white and black dots in Figure 7, respectively. Applying the Floyd-Warshall algorithm of Algorithm 1, we initially have

$$R_{00} = \emptyset$$
$$R_{01} = \{(x,y) \to (x+1,y) \mid (x \geq 0 \wedge 3y \geq 2 + 2x \wedge x \leq 2 \wedge 3y \leq 3 + 2x)\}$$
$$R_{10} = \{(x,y) \to (x_2,y_2) \mid (3y = 2x \wedge x_2 = x \wedge 3y_2 = 3 + 2x \wedge x \geq 0 \wedge x \leq 3)\}$$
$$R_{11} = \emptyset.$$

In the first iteration, R_{00} remains the same ($\emptyset^+ = \emptyset$). R_{01} and R_{10} are therefore also unaffected, but R_{11} is updated to include $R_{01} \circ R_{10}$, i.e., the dashed arrow in the figure. This new R_{11} is obviously transitively closed, so it is not changed in the second iteration and it does not have an effect on R_{01} or R_{10}. However, R_{00} is updated to include $R_{10} \circ R_{01}$, i.e., the dotted arrow in the figure. The transitive closure of the original relation is then equal to $R_{00} \cup R_{01} \cup R_{10} \cup R_{11}$.

6.3 Incremental Computation

In some cases it is possible and useful to compute the transitive closure of union of basic relations incrementally. In particular, if R is a union of m basic maps,

$$R = \bigcup_j R_j,$$

then we can pick some R_i and compute the transitive closure of R as

$$R^+ = R_i^+ \cup \left(\bigcup_{j \neq i} R_i^* \circ R_j \circ R_i^* \right)^+, \tag{14}$$

assuming both of the transitive closures on the right hand side can be computed exactly. The reflexive closure $R_i^* = R_i^+ \cup \mathrm{Id}_D$ is taken over some set D that covers the union of domain and range of R. ([28] uses the notation $R_i^?$.) For this approach to be successful, it is crucial that each of the disjuncts in the argument of the second transitive closure in (14) be representable as a single basic relation, i.e., without a union. If this condition holds, then by using (14), the number of disjuncts in the argument of the transitive closure can be reduced by one. Now, $R_i^* = R_i^+ \cup \mathrm{Id}_D$ is a union, but, as in Section 5.3 with $P_i \cup S_i$, it is sometimes possible to relax the constraints of R_i^+ to include the identity relation on some appropriate D. As before, we relax the constraint $k \geq 1$ to $k \geq 0$, but we also use $P \cap (D \to D)$. instead of (5) and check that the result does not contain any spurious elements, i.e., that projecting out k results in exactly $R_i^+ \cup \mathrm{Id}_D$. As to the choice of D, we compute the "simple hull" of $\mathrm{dom}\, R \cup \mathrm{ran}\, R$, where the simple hull of a set S is defined as the smallest basic set that covers S and is described by only translates of the constraints describing S. It is not clear which D is used in [28]. Presumably, they use either the convex hull of $\mathrm{dom}\, R \cup \mathrm{ran}\, R$ or some approximation of this convex hull.

It is also possible to use a domain D that does *not* include $\mathrm{dom}\, R \cup \mathrm{ran}\, R$, but then we have to compose with R^* more selectively. In particular, if we have

$$\text{for each } j \neq i, \text{ either } \mathrm{dom}\, R_j \subseteq D \text{ or } \mathrm{dom}\, R_j \cap \mathrm{ran}\, R_i = \emptyset \tag{15}$$

$$\text{for each } j \neq i, \text{ either } \mathrm{ran}\, R_j \subseteq D \text{ or } \mathrm{ran}\, R_j \cap \mathrm{dom}\, R_i = \emptyset \tag{16}$$

then we can refine (14) to

$$R_i^+ \cup \Bigg(\Bigg(\bigcup_{\substack{\mathrm{dom}\, R_j \subseteq D \\ \mathrm{ran}\, R_j \subseteq D}} R^* \circ R_j \circ R^* \Bigg) \cup \Bigg(\bigcup_{\substack{\mathrm{dom}\, R_j \cap \mathrm{ran}\, R_i = \emptyset \\ \mathrm{ran}\, R_j \subseteq D}} R^* \circ R_j \Bigg) \cup \Bigg(\bigcup_{\substack{\mathrm{dom}\, R_j \subseteq D \\ \mathrm{ran}\, R_j \cap \mathrm{dom}\, R_i = \emptyset}} R_j \circ R^* \Bigg) \cup \Bigg(\bigcup_{\substack{\mathrm{dom}\, R_j \cap \mathrm{ran}\, R_i = \emptyset \\ \mathrm{ran}\, R_j \cap \mathrm{dom}\, R_i = \emptyset}} R_j \Bigg) \Bigg)^+ .$$

If only property (15) holds, we can use

$$R_i^+ \cup \Bigg(\Big(R_i^+ \cup \mathrm{Id}_{\mathbb{Z}^d} \Big) \circ \Bigg(\Bigg(\bigcup_{\mathrm{dom}\, R_j \subseteq D} R_j \circ R^* \Bigg) \cup \Bigg(\bigcup_{\mathrm{dom}\, R_j \cap \mathrm{ran}\, R_i = \emptyset} R_j \Bigg) \Bigg)^+ \Bigg),$$

while if only property (16) holds, we can use

$$R_i^+ \cup \Bigg(\Bigg(\Bigg(\bigcup_{\mathrm{ran}\, R_j \subseteq D} R^* \circ R_j \Bigg) \cup \Bigg(\bigcup_{\mathrm{ran}\, R_j \cap \mathrm{dom}\, R_i = \emptyset} R_j \Bigg) \Bigg)^+ \circ \Big(R_i^+ \cup \mathrm{Id}_{\mathbb{Z}^d} \Big) \Bigg).$$

Incremental computation is only applied when the result is exact, therefore it is transitively closed.

7 Implementation Details

The algorithms described in the previous sections have been implemented in the `isl` library, available from `http://freshmeat.net/projects/isl/`. The 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, assuming there are at least two disjuncts. 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.

In an attempt to make a meaningful experimental comparison with the approach of Kelly et al. [28], we have also implemented a "box" implementation based on their ideas. Their own implementation in the Omega library only computes underapproximations [27, Section 6.4], so it is impossible to compare the accurateness of our approach with their implementation in those cases where the results are not exact. The base case of their approach is a simplified version of the algorithm in Section 5.2. 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 (6). This algorithm clearly results in an overapproximation of R^k and therefore, after projection, of R^+. As the rest of their paper focuses on underapproximations, we use the above "box-closure" in our box implementation. To improve accuracy, however, we also try the incremental approach, but using their algorithm and only in those cases where the result is exact. Since the domain on which the reflexive closure is taken is not clear from their description, we use the same domain as in our approach, namely the simple hull [36] of domain and range of the input relation. 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.

8 Reachability Analysis

There is a strong relationship between transitive closures and reachability analysis. In fact, one of our target applications, the equivalence checking problem, is essentially a reachability problem. Consider a *transition relation* $T \subseteq \bigcup_{d_1,d_2\geq 0}(\Sigma\times\mathbb{Z}^{d_1}) \to (\Sigma\times\mathbb{Z}^{d_2})$, with Σ a finite set of labels or *control points* and a set of initial states $S_0 \subseteq \bigcup_{d\geq 0} \Sigma\times\mathbb{Z}^d$, the *reachable set* is the set $S \subseteq \bigcup_{d\geq 0} \Sigma \times \mathbb{Z}^d$ of states that can be reached from an initial state through zero or more applications of the transition relation. Note that most researchers in reachability analysis consider integer tuples of a fixed dimension.

It is clear that transitive closures can be used to compute reachable sets. Simply apply the transitive closure of the transition relation to the set of initial sets and take the union with this initial set, i.e.,

$$S = S_0 \cup T^+(S_0).$$

An overapproximation of the transitive closure will lead to an overapproximation of the reachable set, which can be used to obtain invariants on the set of reachable states. Such an approach is taken in, e.g., [2].

```
double x[2][10];
int old = 0, new = 1, i, t;
for (t = 0; t < 1000; t++) {
    for (i = 0; i < 10; i++)
        x[new][i] = g(x[old][i]);
    new = (new+1) % 2; old = (old+1) % 2;
}
```

Figure 8: Flip-flop example from [2, Fig. 3]

Example 8.1 *Consider the example from [2], reproduced in Figure 8. The authors consider several variations of essentially interchanging the values of* `new` *and* `old` *and the main objective is to show that* `new` *and* `old` *always have different values. The variation shown in Figure 8 is one for which the authors are unable to prove this invariant, mainly because they do not support existentially quantified variables. The effect of the loop on the two variables can be represented as*

$$T = \{ (n,o) \to (n',o') \mid \exists \alpha_0, \alpha_1 : 2\alpha_0 = -1-n+n' \wedge 2\alpha_1 = -1-o+o' \wedge 0 \le n', o' \le 1 \}.$$

The (exact) transitive closure of this relation is

$$T^+ = \{ (n,o) \to (n',o') \mid \exists \alpha_0 : 2\alpha_0 = -n-o-n'+o' \wedge n' \ge 0 \wedge n' \le 1 \wedge o' \ge 0 \wedge o' \le 1 \}.$$

The set of reachable states is

$$S = \{ (0,1) \} \cup T^+(\{ (0,1) \}) = \{ (n, 1-n) \mid n \ge 0 \wedge n \le 1 \}.$$

Conversely, by introducing an extra set of variables that is initialized to be equal to the main set of variables and that remains constant in the entire transition system, an overapproximation for the transitive closure of a relation can be obtained from an invariant analysis. To ensure that the transition relation is taken at least once, each control point is duplicated into an "initial" copy and a "final" copy and the transition relation is applied both between initial and final copy and between two final copies. The invariant on the final copies then yields an overapproximation of the transitive closure. Note that many tools only support functional transitions, but general relations can still be represented by adding an extra control point [1]. Existentially quantified variables can be handled in an entirely similar way. Parameters can be handled as variables that remain constant.

Example 8.2 *Let us first consider a translation relation, e.g.,*

$$\{ x \to 1 + x \mid x \ge 0 \}. \tag{17}$$

An Aspic [25] model corresponding to this relation is shown in Figure 9. The initial state equates `in_0` *to* `x0` *at control point* `s0`*. The relation is applied both from the initial control point to the final control point* `s1` *and from the final control point to itself. The invariant on control point* `s1` *provides an overapproximation of the transitive closure of the input relation.*

```
model m1 {
var x0, in_0;
states s0, s1;
transition t0 := {
        from := s0;
        to := s1;
        guard := x0 >= 0;
        action := x0' = 1+x0;
};
transition t1 := {
        from := s1;
        to := s1;
        guard := x0 >= 0;
        action := x0' = 1+x0;
};
}
strategy s1 {
Region init := { state = s0 && in_0 = x0 };
}
```

Figure 9: Aspic model for relation (17)

Example 8.3 *Let us now consider a proper relation, e.g.,*

$$\{ x \to y \mid y \geq 3 + x \wedge y \leq 4 + x \wedge x \geq 0 \wedge y \geq 0 \}. \tag{18}$$

An Aspic model corresponding to this relation is shown in Figure 10. An extra variable `x1` *is used to represent the value of y. The relation is then represented using two transitions. The first resets the values of* `x1` *to an arbitrary value (*`?`*) and the second picks out those pairs of x and y that satisfy relation, taking the value of* `x1` *to be the new value of* `x0`*. The Fast tool [3] does not support an assignment with* `?`*. Instead, the value of* `x1` *is allowed to be decremented and incremented by any amount in steps of one. The resulting model is shown in Figure 11.*

9 Experiments

In all our experiments, we have used `isl` version isl-0.05.1-125-ga88daa9, `Omega+` version 2.1.6 [17], `Fast` version 2.1, `Aspic` version 3.2 and the latest version of `StInG` [30]. The `Fast` and `Aspic` tests are based on the encoding described in Section 8. Version 2.1.6 of `Omega+` provides three transitive closure operations: the original implementation, called `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 [10] 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 [28, 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+`

```
model m1 {
var x0, x1, in_0;
states s0, s1, s2;
transition t0 := {
        from := s0;
        to := s2;
        guard := true;
        action := x1' = ?;
};
transition t1 := {
        from := s1;
        to := s2;
        guard := true;
        action := x1' = ?;
};
transition t2 := {
        from := s2;
        to := s1;
        guard := -3-x0+x1 >= 0 && 4+x0-x1 >= 0 &&
                 x0 >= 0 && x1 >= 0;
        action := x0' = x1, x1' = 0;
};
}
strategy s1 {
Region init := { state = s0 && in_0 = x0 };
}
```

Figure 10: Aspic model for relation (18)

```
model m1 {
var x0, x1, in_0;
states s0, s1, s2;
transition t0 := {
        from := s0;
        to := s2;
        guard := true;
        action := x0' = x0;
};
transition t1 := {
        from := s1;
        to := s2;
        guard := true;
        action := x0' = x0;
};
transition t2 := {
        from := s2;
        to := s2;
        guard := true;
        action := x1' = x1 + 1;
};
transition t3 := {
        from := s2;
        to := s2;
        guard := true;
        action := x1' = x1 - 1;
};
transition t4 := {
        from := s2;
        to := s1;
        guard := -3-x0+x1 >= 0 && 4+x0-x1 >= 0 &&
                x0 >= 0 && x1 >= 0;
        action := x0' = x1;
};
}
strategy s1 {
setMaxState(0);
setMaxAcc(100);
Region init := { state = s0 && in_0 = x0 };
Region final := { state = s1 };
Transitions t := { t0, t1, t2, t3, t4 };
Region reach := post*(init, t) && final;
print(reach);
}
```

Figure 11: Fast model for relation (18)

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 [28, 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 5.4 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 support non-negative variables, we split all variables into a pair of non-negative variables whenever the input relation contains any negative value.

9.1 Sized Types

Chin and Khoo [18] 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:

$$\{(i,j) \to (o_0, o_1) \mid o_0 \geq 0 \wedge o_0 \leq -1 + i \wedge j \geq 0 \wedge o_0 \leq -2 + i + j\} \cup \\ \{(i,j) \to (o_0, 1) \mid o_0 \leq -1 + i \wedge j \geq 0 \wedge o_0 \geq 0\} \cup \\ \{(i,j) \to (i, o_1) \mid i \geq 1 \wedge o_1 \geq 0 \wedge o_1 \leq -1 + j\} \cup \\ \{(i,j) \to (o_0, 0) \mid o_0 \leq -1 + i \wedge j \geq 0 \wedge o_0 \geq 1\}.$$

9.2 Equivalence Checking

Our most extensive set of experiments is based on the algorithm of [5] 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` [29] 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 7 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.

	isl	box	Omega+ TC+AC	Omega+ 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

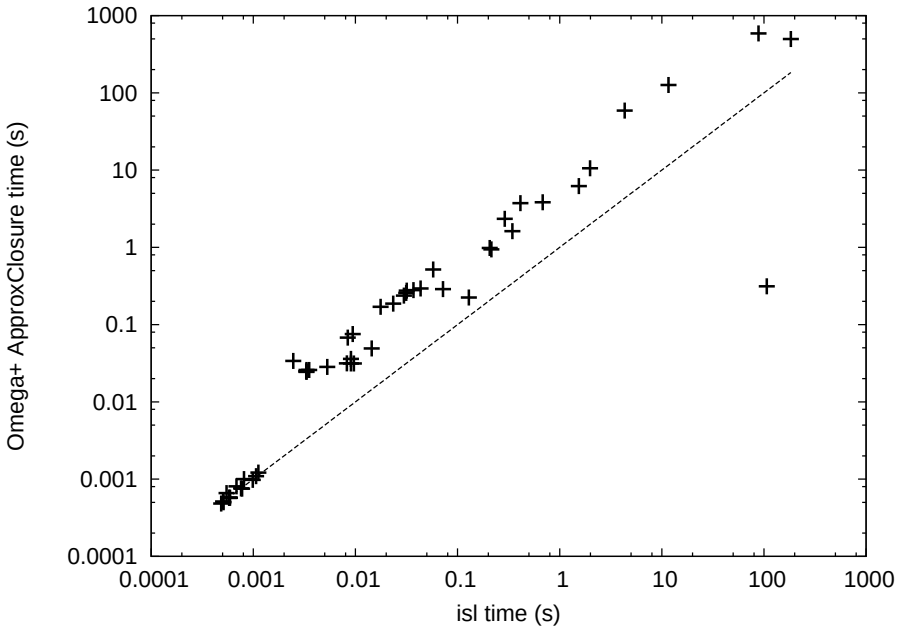

Figure 12: Comparison of equivalence checking times in successful cases

The equivalence checking procedure was applied to the output of `CLooG` [6] 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 [37]. 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,

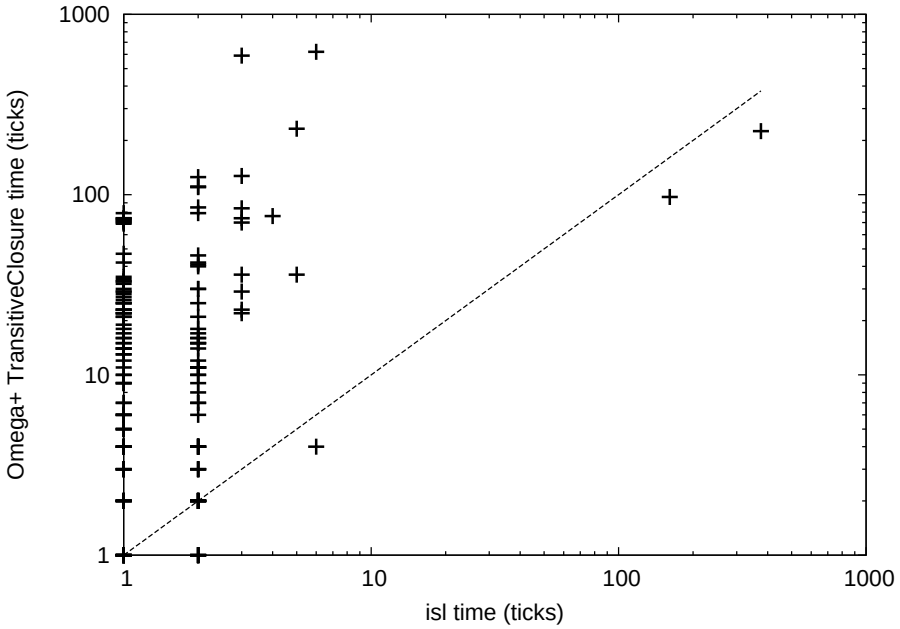

Figure 13: Comparison of transitive closure computation times in successful cases

`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 12 compares the running times. In all but one case, `isl` is faster than `Omega+`'s `ApproxClosure`. This result is somewhat surprising. 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+`, maxing 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. The results show that our box implementation is not very good at mimicking the original `Omega+` implementation, since `TransitiveClosure` is more accurate. The likely reason is that `TransitiveClosure` also computes small powers of the input relation and then checks whether these small powers reach a fixed point. Or, the implementation may have evolved since the publication of the paper. What is more surprising is that on this test set our "box" implementation is more accurate than both `ApproxClosure` and `calculateTransitiveClosure`. 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 where 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 13 shows that for those transitive closures that both `isl` and `Omega+`'s `TransitiveClosure` 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.

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

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

9.3 Iteration Space Slicing

The ISS experiments were performed on the test set of loops previously used in [8] and extracted from version 3.2 of NAS Parallel Benchmarks [38] consisting of five kernels and three pseudo-applications derived from computational fluid dynamics (CFD) applications. These loops were represented in a format required by our dependence analysis tool. From 431 studied loops of the NAS benchmark, it was possible to extract dependences from 257 loops (it is not possible to analyze loops containing "break", "goto", "continue", "exit" statements, functions and when array indexes are elements of other arrays). Of these 257 loops, 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. 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 it 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.

9.4 Reachability Analysis

As explained in Section 8, our transitive closure algorithm can also be used to compute overapproximations of reachable sets. In practice, however, the results are rather disappointing, mainly because the set of initial states is not taken into account during the computation of the transitive closure. Let us first consider the results of applying our algorithm to the `Aspic` [25] test cases. The main objective is to find invariants on the variables for each control point. Of the 21 test cases available from the `Aspic` web site, we found that one could not be handled by the latest version (3.2) of `Aspic`, while two test cases took more than 10 minutes to handle by `isl`. For two test cases, `Aspic` and `isl` produced identical results. `Aspic` produced more accurate results on three test cases, while `isl` produced more accurate results on seven cases. Note that these more accurate results are mainly due to the fact that `isl` has support for existentially quantified variables. For the remaining six cases, neither was equally or more accurate than the other, meaning that the intersection of the results would be strictly more accurate.

The results on the `Lever` [34] test cases were much less encouraging. In particular, we considered the safety analysis problems, where we need to check that a specified set of bad states is unreachable. Of the 32 test cases, `isl` only managed to prove safety (within a reasonable amount of time) for 7 cases. Furthermore, `isl` was only faster than `Lever` in two cases.

10 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 alternative 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 have 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.

11 Acknowledgments

We would like to thank Louis-Noel 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] Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Bounding the computational complexity of flowchart programs with multi-dimensional rankings. Tech. Rep. 7235, INRIA (Mar 2010)

[2] 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), http://dx.doi.org/10.1016/j.entcs.2010.09.002

[3] Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. STTT 10(5), 401–424 (2008)

[4] Barthou, D., Cohen, A., Collard, J.F.: Maximal static expansion. Int. J. Parallel Programming 28(3), 213–243 (2000)

[5] 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)

[6] 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)

[7] 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)

[8] 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)

[9] 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)

[10] 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)

[11] Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces. Ph.D. thesis, Université de Liège (1998)

[12] 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)

[13] 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), http://hal.inria.fr/inria-00335905/en/

[14] 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), http://dx.doi.org/10.1007/978-3-642-00768-2_29

[15] 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)

[16] 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)

[17] Chen, C.: Omega+ library (2009), http://www.chunchen.info/omega/

[18] Chin, W.N., Khoo, S.C.: Calculating sized types. Higher Order Symbol. Comput. 14(2-3), 261–300 (2001)

[19] Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: CAV'98, LNCS 1427. pp. 268–279. Springer (1998)

[20] Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser Boston (2000)

[21] 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)

[22] Feautrier, P.: Parametric integer programming. Operationnelle/Operations Research 22(3), 243–268 (1988)

[23] Feautrier, P.: The Data Parallel Programming Model, LNCS, vol. 1132, chap. Automatic Parallelization in the Polytope Model, pp. 79–100. Springer-Verlag (1996)

[24] Feautrier, P., Griebl, M., Lengauer, C.: On index set splitting. In: Parallel Architectures and Compilation Techniques (PACT'99). Newport Beach, CA (Oct 1999)

[25] Feautrier, P., Gonnord, L.: Accelerated invariant generation for c programs with aspic and c2fsm. Electron. Notes Theor. Comput. Sci. 267, 3–13 (October 2010), http://dx.doi.org/10.1016/j.entcs.2010.09.014

[26] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega calculator and library. Tech. rep., University of Maryland (Nov 1996)

[27] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega library. Tech. rep., University of Maryland (Nov 1996)

[28] 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)

[29] Lombardy, S., Régis-Gianas, Y., Sakarovitch, J.: Introducing VAUCANSON. Theor. Comput. Sci. 328(1-2), 77–96 (2004)

[30] Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: SAS'04. pp. 53–68 (2004)

[31] Saouter, Y., Quinton, P.: Computability of recurrence equations. Theor. Comput. Sci. 116(2), 317–337 (1993)

[32] Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons (1986)

[33] Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)

[34] 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)

[35] Verdoolaege, S.: `barvinok`, a library for counting the number of integer points in parametrized and non-parametrized polytopes (2010), `http://freshmeat.net/projects/barvinok`

[36] 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)

[37] 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)

[38] NAS benchmarks suite. `http://www.nas.nasa.gov`

Contents

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399