Partial Projection of Sets Represented by Finite Automata, with Application to State-Space Visualization*

Bernard Boigelot and Jean-François Degbomont

Institut Montefiore, B28
Université de Liège
B-4000 Liège, Belgium
{boigelot,degbomont}@montefiore.ulg.ac.be

Abstract. This work studies automata-based symbolic data structures for representing infinite sets. Such structures are used in particular by verification tools in order to represent the sets of configurations handled during symbolic exploration of infinite state spaces. Our goal is to develop an efficient projection operator for these representations. There are several needs for such an operator during state-space exploration; we focus here on projecting the set of reachable configurations obtained at the end of exploration. An interesting application is the state-space visualization problem, which consists in providing the user with a graphical picture of a relevant fragment of the reachable state space.
For theoretical reasons, the projection of automata-represented sets is inherently costly. The contribution of this paper is to introduce an improved automata-based data structure that makes it possible to reduce in several cases the effective cost of projection. The idea is twofold. First, our structure allows to apply projection to only a part of an automaton, in cases where a full computation is not necessary. Second, the structure is able to store the results of past projection operations, and to reuse them in order to speed up subsequent computations. We show how our structure can be applied to the state-space visualization problem, and discuss some experimental results.

1 Introduction

State-space exploration is a powerful technique for analyzing the properties of computerized systems. It is not restricted to finite models: infinite state spaces can be explored symbolically, with the help of suitable data structures for representing the sets of configurations that have to be handled [Boi98,BJNT00]. Concretely, if a system undergoing symbolic state-space exploration is controlled by n variables defined over the respective domains $D_1, D_2, \ldots, D_n$, then one needs a data structure suited for representing subsets of $D_1 \times D_2 \times \cdots \times D_n$. This structure must be closed under all operations to be performed during exploration.

* This work is supported by the *Interuniversity Attraction Poles* program *MoVES* of the Belgian Federal Science Policy Office, and by the grant 2.4530.02 of the Belgian Fund for Scientific Research (F.R.S.-FNRS).

1.1 Automata-Based Representations

A simple approach consists in representing sets as finite automata: given a fixed alphabet Σ, one considers an encoding relation that maps every value in a domain D onto words over Σ. Such a relation thus encodes subsets S of D as languages. Whenever such languages are regular, they can be accepted by finite automata, which then provide symbolic representations of the sets S. The advantages are that automata are easily manipulated algorithmically, and that most usual operations on sets (such as intersection, union, ...) reduce to carrying out the same operations on the languages accepted by automata [Boi98].

Consider for instance the important case of programs relying on integer variables. Using the positional notation, an integer number can be encoded as a finite word over the alphabet $\{0, 1, \ldots, r-1\}$, where $r > 1$ is an arbitrarily chosen *base*. It has been established that, in this setting, every subset of $\mathbb{Z}$ that is definable in Presburger arithmetic, i.e., the first-order theory $\langle \mathbb{Z}, +, \leq \rangle$, is encoded by a regular language, and is thus automata-representable [BHMV94]. Interestingly, Presburger-definable sets are those that can be expressed as combinations of linear constraints and discrete periodicities [Pre29], which matches quite well the requirements of infinite state-space exploration applications [WB95,SKR98].

In order to represent multidimensional sets, i.e., subsets of $D = D_1 \times D_2 \times \cdots \times D_n$, one needs an encoding relation suited for the global domain D. Such a relation can be obtained by combining together encoding relations suited for each individual domain D_i. Several types of combinations are possible. A first strategy consists in encoding a value $(v_1, v_2, \ldots, v_n) \in D$ by concatenating the individual encodings of $v_1, v_2, \ldots, v_n$, expressed over distinct alphabets. This method is more suited for domains such as communication channel contents [BG96] than for integer variables. Indeed, with this scheme, the Presburger-definable subsets of $\mathbb{Z}^n$ are generally not encoded by regular languages[1].

Another approach is to interleave the encodings of the individual values $v_1, v_2, \ldots, v_n$, by reading repeatedly and successively one symbol in words $w_1, w_2, \ldots, w_n$ encoding those values. This requires these words to share the same length, which can always been achieved by appending a suitable number of padding symbols. In the case of integer numbers encoded positionally, padding is not necessary, since every integer admits encodings of any sufficiently large length. It is known that this composite encoding relation maps every Presburger-definable subset of $\mathbb{Z}^n$ onto a regular language [BHMV94,WB95,Boi98].

In this work, we restrict ourselves to multidimensional encoding relations obtained by the interleaving method. Our motivation stems from the case of programs manipulating integer variables, which is probably the most relevant one in actual applications. Nevertheless, the techniques developed in this paper extend naturally to other domains for which interleaved encodings are also applicable.

[1] A simple example is given by the set $\{(x_1, x_2) \in \mathbb{Z}^2 \mid x_1 = x_2\}$.

1.2 Set Projection and State-Space Visualization

This paper studies the *projection* operation, which intuitively consists in discarding a given subset of variables from a multidimensional set. Formally, given a set $S \subseteq D_1 \times D_2 \times \cdots \times D_n$ and a set $C = \{i_1, i_2, \ldots\} \subseteq \{1, 2, \ldots, n\}$ of components, the projection of S over C is given by the set $S' = \{(u_{i_1}, u_{i_2}, \ldots) \in D_{i_1} \times D_{i_2} \times \cdots \mid \exists (v_1, v_2, \ldots, v_n) \in S : \forall j : u_{i_j} = v_{i_j}\}$.

During symbolic state-space exploration, the projection operation has to be carried out in several forms. The first one, *local projection*, is needed when one needs to compute the image of a subset of configurations by an operation that discards the current value of some variables. For instance, the effect of an assignment instruction such as $x_1 := 2$ amounts to first projecting the current set of configurations onto all variables but x_1, and then inserting the constant 2 in the first component of all tuples in the resulting set. A different application, *global projection*, corresponds to projecting the whole set of reachable configurations obtained at the end of state-space exploration. This makes it possible to reason on the properties of the reachable set without being hampered by the presence of non-relevant variables.

The aim of this work is to develop an efficient implementation of global projection operations. Our main motivation is the *state-space visualization* problem for programs manipulating integer variables, defined as follows. The goal of state-space exploration is to compute the set of reachable configurations of the system under analysis, in the form of a symbolically-represented set of vectors with integer components. The visualization problem then consists in producing a two-dimensional image of the values taken by a pair of specified variables. Such an image can be obtained by projecting the original set over the selected variables, and then enumerating the values in the resulting set, within given bounds. The aim of visualization is to provide the user with a synthetic and global view of the reachable configurations, so as to draw quickly attention towards erroneous behaviors (which can then be the subject of more focused investigations), or modeling errors.

In order for state-space visualization to be helpful during the software development process, it has to be reasonably efficient. Of course, state-space exploration in itself is usually a quite costly procedure, but that has only to be carried out once for a given system. Having obtained a (typically large) symbolic representation of the reachable set, the problem is thus to visualize it as efficiently as possible, with respect to different choices of variables or viewing parameters (in particular, one should be able to move at will the visualization window, as well as change the zoom factor). This requires an efficient implementation of the projection operator.

1.3 Projecting Sets Represented by Automata

In the case of automata-based representations of multidimensional sets, projection is seemingly a simple operation. Indeed, assuming an interleaved encoding scheme, one can locate in linear time the automaton transitions associated

with the variables discarded by the projection, and simply relabel them with the empty word. The drawback of this approach is that it gives out non-deterministic automata.

For state-space exploration however, using non-deterministic representations is problematic. First, during exploration, working systematically with deterministic automata makes it possible to minimize them into a canonical form [Hop71]. This makes the representation of sets independent from their construction, which often helps to keep the size of the representations under control. Another problem is that testing inclusion between sets, which is needed for checking that a fixed point has been reached during exploration, can only be implemented with a reasonable cost on deterministic automata. Furthermore, visualizing a set with respect to a given pair of variables requires to check for each pixel of the display window whether it has to be lit or not. For a given pixel, this amounts to checking whether the projection of the underlying automaton over the selected variables accepts or not an encoding of the coordinates of the corresponding point. With non-deterministic automata, this procedure requires to check a prohibitively large number of paths. Finally, it is worth mentioning that the worst-case exponential cost of the determinization operation is seldom observed in practice; for automata produced by state-space exploration tools, determinization usually remains an efficient operation [BW02].

The implementation of a usable state-space visualization tool is thus faced with the problem of computing as efficiently as possible a deterministic automaton representing the projection of a given set. This problem is inherently difficult. Indeed, one can easily build families of deterministic automata whose determinized projection is exponentially larger. A possible workaround could be to limit the expressiveness of the symbolic representations. Assuming that only Presburger-definable sets have to be represented, a potential strategy could be to exploit the known structure of the automata representing such sets [Lat05,Ler05]. Unfortunately, this approach is not feasible, for a polynomial algorithm for the projection operator would lead to a polynomial decision procedure for Presburger arithmetic, which does not exist [Opp78].

In spite of these theoretical limitations, it is nevertheless possible to reduce the effective cost of projection in some applications. The approach we propose is based on two ideas. First, projection does not always have to be applied to whole automata. In particular, we show that state-space visualization can be speeded up by only projecting subsets of the transition graph of the original automaton (we name this operation *partial projection*). Second, some projection computations can reuse the results of previous computations. For instance, projecting a set over $\{x_1\}$ becomes simpler if the projection of that set over $\{x_1, x_2\}$ is already available.

The contributions of this paper are the definition of an original data structure allowing the computation of partial projections as well as the efficient reuse of the results of past computations. We also show how this data structure can be exploited for implementing state-space visualization, and then discuss some experimental results.

2 Basic Notions

2.1 Automata-Based Representations of Sets

In order to represent subsets of a domain D by finite automata, one needs an *encoding relation* $E \subseteq D \times \Sigma^*$ mapping the elements of D onto finite words over a finite alphabet Σ. A word can only encode one value, hence the relation E must be such that $\forall (v_1, w_1), (v_2, w_2) \in E : v_1 \neq v_2 \Rightarrow w_1 \neq w_2$. Moreover, each element of D must be encoded by at least one word.

For a set $S \subseteq D$, we define its *encoding* as the language $E(S) = \{w \in \Sigma^* \mid \exists v \in S : (v, w) \in E\}$. If this language is regular, then any finite automaton that accepts $E(S)$ is a *representation* of S. We denote finite automata by tuples $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ the alphabet, δ a transition relation, with $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ for non-deterministic automata, and $\delta : (Q \times \Sigma) \to Q$ for deterministic ones, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of accepting states.

2.2 Multidimensional Domains

A domain D is *multidimensional* if it can be expressed as the Cartesian product $D = D_1 \times D_2 \times \cdots D_n$ of simpler domains D_i, where $n \geq 0$ is the *dimension*. Assuming that encoding relations E_i have been defined for the domains D_i, for $i = 1, 2, \ldots, n$, we build an encoding relation E suited for D in the following way. First, for simplicity's sake, we assume that the relations E_i are defined over the same alphabet, i.e., $E_i \in D_i \times \Sigma^*$. Then, we require each E_i to be such that, for every value $v_i \in D_i$ and sufficiently large integer N, there exists w_i such that $|w_i| = N$ and $(v_i, w_i) \in E_i$. In other words, every value in D_i must admit encodings of any sufficiently large length. Note that any relation can be turned into one that satisfies this requirement, by appending appropriate *padding symbols* to the encodings of values.

In order to encode a multidimensional value $v = (v_1, v_2, \ldots, v_n) \in D$, we first consider individual encodings $w_1, w_2, \ldots, w_n \in \Sigma^*$ of $v_1, v_2, \ldots, v_n$, such that $|w_1| = |w_2| = \cdots = |w_n|$. Then, we build the word $w = a_1 a_2 \ldots a_n\, b_1 b_2 \ldots b_n \ldots$, where $w_i = a_i b_i \ldots$ for each $i = 1, 2, \ldots, n$. In other words, w is constructed by reading repeatedly one symbol in $w_1, w_2, \ldots, w_n$, in fixed order. By mapping every value $v \in D$ onto the words w obtained in this way, we get an encoding relation $E \subseteq D \times \Sigma^*$ suited for D. Note that a value may admit several distinct encodings, and that the length of encodings is restricted to integer multiples of the domain dimension n. Hence, we have $E \subseteq D \times (\Sigma^n)^*$ [Boi98].

2.3 Projection

Let S be a subset of a multidimensional domain $D = D_1 \times D_2 \times \cdots D_n$, and $C \subseteq \{1, 2, \ldots, n\}$ be a set of *components*. The *projection* of S over C is defined as the set $\pi_C(S) = \{(u_{i_1}, u_{i_2}, u_{i_3}, \ldots) \in D_{i_1} \times D_{i_2} \times D_{i_3} \times \cdots \mid \exists (v_1, v_2, \ldots, v_n) \in S : \forall j : u_{i_j} = v_{i_j}\}$, where $C = \{i_1, i_2, i_3, \ldots\}$ with $i_1 < i_2 < i_3, \cdots$. In other

words, projecting S over C amounts to removing from every element of S the components that do not belong to C.

A similar operator can also be defined over words that encode multidimensional values. Given a dimension n and a nonempty set of components $C \subseteq \{1, 2, \ldots, n\}$, the projection of a word $a = a_1 a_2 \cdots a_n$ over C, with $\forall i : a_i \in \Sigma$, is given by $\pi_C(a) = a_{i_1} a_{i_2} a_{i_3} \ldots$, where $C = \{i_1, i_2, i_3, \ldots\}$ with $i_1 < i_2 < i_3, \cdots$. For an empty set of components $C = \{\,\}$, we define $\pi_C(a) = \alpha$, where α is a distinguished symbol[2]. Then, the projection of a word $w = w_1 w_2 \cdots w_q$, with $|w_i| = n$ for each $i \in \{1, \ldots, q\}$, over C is then defined as $\pi_C(w) = \pi_C(w_1)\pi_C(w_2)\cdots\pi_C(w_q)$. Finally, the projection of a language of encodings $L \subseteq (\Sigma^n)^*$ over C is given by $\pi_C(L) = \{\pi_C(w) \mid w \in L\}$.

Note that the projection operator preserves the regularity of languages: an automaton accepting $\pi_C(L)$ can easily be built from one accepting L. This is done by first unrolling the transition graph of the automaton so that each transition corresponds to one individual component. Then, the transitions associated with the components that do not belong to C are relabeled with the empty word. This procedure gives out a non-deterministic automaton.

Let $S \subseteq D$ be a set represented by an automaton $\mathcal{A}$. The projection $\pi_C(S)$ of S over some set of components C cannot be simply computed by constructing an automaton accepting $\pi_C(L(\mathcal{A}))$, where $L(\mathcal{A})$ denotes the language accepted by $\mathcal{A}$. Indeed, although each word in $\pi_C(L(\mathcal{A}))$ encodes correctly the projection of some element of S, this language may not necessarily contain all such encodings. The solution to this problem depends on the encoding relation used, and consists in first applying the language projection operator, and then performing a domain-specific *saturation* operation that adds to the resulting language the missing encodings of the represented values [Boi98].

2.4 Application to Sets of Integers

Consider the domain $\mathbb{Z}$ of integer numbers. Choosing a *base* $r \in \mathbb{N} \setminus \{0\}$, the positional notation encodes a number $z \in \mathbb{N}$ as a word $d_{p-1}d_{p-2}\ldots d_1 d_0$ over the finite alphabet $\{0, 1, \ldots, r-1\}$, such that $z = \sum_{0 \leq i < p} d_i r^i$. This encoding relation generalizes to numbers in $\mathbb{Z}$ by encoding negative numbers by their *r-complement* [WB95]. The length p of encodings is not fixed. This implies that every number has encodings of any sufficiently large length. One can then apply the technique outlined in Section 2.2 so as to obtain a positional encoding relation suited for subsets of $\mathbb{Z}^n$, for any dimension $n \geq 0$.

The resulting automata-based representation for sets of vectors in $\mathbb{Z}^n$ is called the *Number Decision Diagram (NDD)* [WB95,Boi98]. It is known that all subsets of $\mathbb{Z}^n$ that are definable in Presburger arithmetic, i.e., the first-order theory $\langle \mathbb{Z}, +, \leq \rangle$, can be represented by NDDs [Büc62,BHMV94]. In order to project a set represented by an NDD $\mathcal{A}$, one simply projects the language $L(\mathcal{A})$ accepted by $\mathcal{A}$, and then applies the saturation algorithm developed in [BL04].

[2] The motivation behind this definition is to keep track of the length of a word in all its projections, even those over the empty set of components.

Automata-based representations of numbers have been extended to sets of vectors with mixed integer and real components, by moving to infinite-word automata [BJW05].

3 State-Space Visualization

3.1 Problem Statement

Let $n \geq 2$ be a dimension, and $S \subseteq \mathbb{Z}^n$ be a set of vectors represented by a NDD $\mathcal{A}$ in a base $r > 1$. In our intended application, S is the set of reachable configurations of a program controlled by n integer variables, and its representation $\mathcal{A}$ is produced by a symbolic state-space exploration algorithm [Boi98]. In this setting, r is typically equal to 2.

The *visualization* problem consists in extracting from $\mathcal{A}$ a two-dimensional picture of some fragment of S that is relevant to the user. More precisely, the user provides two *components* $i_1, i_2 \in \{1, 2, \ldots, n\}$, with $i_1 \neq i_2$, a *center position* $(x, y) \in \mathbb{Z}^2$, a *zoom factor* $f \in \mathbb{N}$, with $f > 0$, and a *window size* $(W, H) \in \mathbb{N}^2$ with $W > 0$ and $H > 0$. The idea is that each pixel in the visualization window corresponds to a square region of size $f \times f$ in the domain $\mathbb{Z}^2$, with the window centered on the point of coordinates (x, y). The goal of the visualization procedure is to light up the pixels corresponding to regions that contain at least one value in $\pi_{\{i_1,i_2\}}(S)$.

3.2 Visualization and Projection

Visualization can thus be achieved by first computing a NDD representing the set $\pi_{\{i_1,i_2\}}(S)$, and then scanning its accepting paths for values that fit in the visualization window. In order to speed up the latter operation, which has to be carried out for each pixel in the window, a good strategy is to determinize the automaton representing $\pi_{\{i_1,i_2\}}(S)$. Then, whether a value (v_1, v_2) belongs or not to this set can be checked by examining a single automaton path.

Even with a deterministic automaton, the number of paths to be checked can become prohibitive for large zoom factors f, since each pixel covers f^2 distinct points of $\mathbb{Z}^2$. A solution is to restrict the allowable values of f to be equal to powers r^k, with $k \in \mathbb{N}$, of the representation base r, and to align the pixel boundaries on coordinates that are exact multiples of f (this is not problematic in actual applications). With those restrictions, the values covered by a single pixel correspond to a square region of the form $[v_1 r^k, (v_1+1)r^k - 1] \times [v_2 r^k, (v_2+1)r^k - 1]$, with $v_1, v_2 \in \mathbb{Z}$ and $k \in \mathbb{N}$. It is then sufficient to check whether the automaton admits a accepting path that reads an encoding of (v_1, v_2) followed by $2k$ arbitrary symbols.

In a deterministic automaton, following a given prefix is a simple operation. Checking whether, from a given state q, there exists an accepting path of given length l is more problematic, since it may require to examine a large number of paths. Remark that this operation is actually equivalent to projecting the

language $L(q)$ accepted from q over the empty set of components, determinizing the resulting automaton, and then checking whether one has $\alpha^l \in \pi_{\{\}}(L(q))$, which can then be done efficiently.

3.3 Avoiding Redundant Computations

Explicitly carrying out a projection followed by a determinization for each considered pixel would however lead to redundant computations. First, some automata states are reached by different prefixes, hence the number of distinct states from which a projection needs to be performed can actually be smaller than the total number of pixels. Second, when the user moves the visualization window, some pixels may correspond to coordinates that have already been checked in previous computations.

Our solution for curbing redundant computations is based on an original data structure, the *Partially Projected Automaton (PPA)*, in which projection and determinization operations can be applied to sub-automata, with their results stored in order to be subsequently reusable. For visualization, the idea is thus to use PPA mainly in order to project efficiently the languages accepted from automaton states over the empty set of components. Interestingly, it turns out that the computation of $\pi_{\{i_1,i_2\}}(S)$, i.e., the projection of the whole reachable set over the pair of selected components, can also benefit from such a data structure.

Indeed, for a regular language $L \subseteq (\Sigma^n)^*$ and a set of components $C \in \{1, 2, \ldots, n\}$, there are several ways of computing a deterministic automaton accepting $\pi_C(L)$ from one accepting L. A first technique is to build a nondeterministic automaton accepting $\pi_C(L)$, and then determinize it at once. An alternative approach is to extract the components $i_1, i_2, \ldots, i_m$ that do not belong to C, i.e., such that $\{i_1, i_2, \ldots, i_m\} = \{1, 2, \ldots, n\} \setminus C$, and to project them out one by one, in some given order, determinizing the resulting automaton after each projection. In the context of symbolic state-space exploration of programs relying on integer variables, we have observed experimentally that the latter solution is more efficient in practice (see Section 5).

With this solution, the results of past projection operations can in some situations be reused in order to speed up new computations. Consider for instance a 5-dimensional domain. Assuming that components are projected out individually in increasing order, the computations of $\pi_{\{2,3\}}(L)$ and $\pi_{\{2,5\}}(L)$ will respectively be decomposed into $\pi_{\{1,2\}}\,\pi_{\{1,2,4\}}\,\pi_{\{2,3,4,5\}}(L)$ and $\pi_{\{1,3\}}\,\pi_{\{1,2,4\}}\,\pi_{\{2,3,4,5\}}(L)$. In this case, the intermediate result $\pi_{\{1,2,4\}}\,\pi_{\{2,3,4,5\}}(L)$ can be reused across both computations.

4 Partially Projected Automata

4.1 Definition

A *Partially Projected Automaton (PPA)* $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, \delta^D, dim, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : (Q \times \Sigma) \to Q$ is

a (deterministic) transition relation, $\delta^D : (Q \times 2^{\mathbb{N}}) \to Q$ is a *decomposition relation*, $dim : Q \to \mathbb{N}$ is a *dimension function*, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states.

In order for a PPA to be well formed, its components have to satisfy some constraints. The language $L(\mathcal{A})$ accepted by $\mathcal{A}$ is defined as being equal to the language accepted by the finite automaton $(Q, \Sigma, \delta, q_0, F)$. This language is supposed to encode some set of vectors from a n-dimensional domain D. This implies that all words in $L(\mathcal{A})$ have a length that is an integer multiple of n. In order to enforce this constraint, the function dim associates each state of $\mathcal{A}$ with a *dimension*, which intuitively corresponds to the dimension of the vectors recognized from that state. For all states q visited by the paths of $(Q, \Sigma, \delta, q_0, F)$, one thus has $dim(q) = dim(q_0) = n$.

The purpose of the decomposition relation δ^D is to provide redundant information, corresponding to the results of language projection operations that have previously been carried out from automaton states. For states $q, q' \in Q$ and a subset of components $C \subseteq \mathbb{N}$ such that $q' = \delta^D(q, C)$, we must have $C \subset \{1, \ldots, dim(q)\}$ and $dim(q') = |C|$. The language accepted from the state q' is then equal to $\pi_C(L(q))$, where $L(q)$ denotes the language accepted from q. The paths of (Q, δ) that can be followed from q' must thus only visit states q'' such that $dim(q'') = dim(q') = |C|$. Each such path σ reads a word w that belongs to $\pi_{(C,|w|)}(L(q))$ iff σ ends in an accepting state.

In summary, a PPA accepting a language encoding a set of n-dimensional vectors can be seen as n distinct finite-state automata, each of them recognizing sets with a specific dimension in $\{1, \ldots, n\}$, linked together by the decomposition relation. Decompositions can be nested, in the sense that from the destination of a decomposition transition, one may reach states from which other decomposition transitions are defined. The advantage of PPA is that they provide a way of storing and reusing the results of previous projection operations carried out from automaton states. These stored results do not have to be kept indefinitely, since decomposition transitions can always be removed from a PPA without affecting its accepted language. Procedures for constructing well-formed PPA and manipulating them are discussed in the next section.

4.2 Algorithms

Let $L \subseteq \Sigma^*$ be a regular language encoding the n-dimensional set $S \subseteq D$. A well-formed PPA $\mathcal{A}$ representing S can simply be constructed by associating a deterministic finite-state automaton $(Q, \Sigma, \delta, q_0, F)$ accepting L with an empty decomposition relation, i.e., by defining $\mathcal{A} = (Q, \Sigma, \delta, \{\,\}, dim, q_0, F)$, with $dim(q) = n$ for all $q \in Q$.

After obtaining such a PPA, projecting the language accepted from one of its states may result in adding decomposition transitions, in order to remember the result of this operation so as to be able to reuse it later. The size of a PPA thus grows with projection operations. At any time, one may choose to curb this growth by removing arbitrary decomposition transitions, as well as the states and transitions that then become unreachable. Different heuristics

can be used for selecting the decomposition transitions to be removed: bounding the total memory footprint of the structure, discarding the last recently or the least frequently followed decomposition transition, ... In all cases, removing decomposition transitions has no influence on the language accepted by the PPA.

We now sketch the algorithm computing the projection $\pi_C(L(q))$ of the language accepted from some state q of a PPA $\mathcal{A} = (Q, \Sigma, \delta, \delta^D, dim, q_0, F)$, with respect to a set of components $C \subset \{1, \ldots, dim(q)\}$.

If $\delta^D = \{\}$, then the projection is carried out by composing the automaton $(Q, \Sigma, \delta, q, F)$ with a finite-state transducer, constructed with a transition relation that takes the form of a single cycle of length $dim(q)$. Each transition in this cycle thus corresponds to one vector component in $\{1, \ldots, dim(q)\}$. The transitions corresponding to components in C are designed so as to give out a copy of their input symbol, the other transitions producing an empty word (or, in a case of a projection over $\{\}$, one occurrence of the distinguished symbol α). The result takes the form of a non-deterministic automaton $\mathcal{A}'$, that can be determinized into an automaton $\mathcal{A}''$ using the classical subset construction. Finally, a decomposition transition $\delta^D(q, C) = q''$ is added to $\mathcal{A}$, where q'' is the initial state of $\mathcal{A}''$. Note that each state q' of the determinized projected automaton thus corresponds to a subset $Q_{q'}$ of states of $\mathcal{A}$.

If, on the other hand, $\delta^D \neq \{\}$, the construction is similar but it may then become possible to reuse the results of earlier projections. This happens when a state q' of the projected automaton is such that all the states in $Q_{q'}$ admit outgoing decomposition transitions with the same destination q'' and subset C' of components, such that $C \subseteq C'$. In this case, the computation of the projection can be continued by exploring the successors of q'' instead of those of q.

Finally, the automaton obtained after a projection operation are minimized, by merging states that are known to accept identical languages. This is done by partitioning the states of the automaton according to their dimension, and then applying classical finite-state machine minimization [Hop71] to each part.

5 Experimental Results

The data structure and the algorithms outlined in Section 4 have been implemented in a prototype tool for visualizing NDDs. Our evaluation considers random NDDs generated by the same method as the one used in [BW02], and measures the time needed for displaying them in a 512×512 window, changing the zoom factor from 1 to 256, and moving the view from $(0, 0)$ to $(2^{16}+1, 2^{16}+1)$. The results are given in Figure 1. Those results demonstrate that, although the inherent cost of projection is not avoided, reusing previous results is useful, especially for translating efficiently the visualization window.

We also evaluated experimentally the benefits of projecting and determinizing a NDD incrementally instead of at once, as discussed in Section 3.3. Figure 2 gives the time spent by both methods for projecting the family of sets described in [Lat05] over two components. The incremental approach clearly stands out.

	NDD size	without PPA			with PPA		
	$\|Q\|$	t_{disp}	$t$zoom	$t$move	t_{disp}	$t$zoom	$t$move
N_1	621	0.01	0.85	2.33	0.01	0.07	0.04
N_2	755	0.03	2.56	7.63	0.03	0.12	0.12
N_3	1938	0.14	15.98	12.06	0.14	0.64	0.28
N_4	13944	0.15	15.26	11.58	0.15	11.44	0.84

Fig. 1. Time cost of visualization operations (in seconds).

	n	$\|Q\|$	$t_{\text{at-once}}$	t_{incr}		n	$\|Q\|$	$t_{\text{at-once}}$	t_{incr}
S_1	4	434	0.02	0.03	S_7	7	50135	240.86	9.30
S_2	5	224	0.04	0.01	S_8	5	4474	1.70	0.50
S_3	4	15272	154.15	0.80	S_9	7	99169	176.29	100.07
S_4	5	915	0.80	0.06	S_{10}	7	132709	243.52	109.94
S_5	4	3446	0.30	0.30	S_{11}	6	63410	71.50	2.90
S_6	3	1279	0.08	0.10	S_{12}	6	66330	> 1000	5.50

Fig. 2. At-once vs incremental projection (in seconds).

6 Conclusions

In this paper, we have introduced a simple yet powerful data structure for storing and reusing the results of *partial projections* of finite automata, i.e., projection operations carried out from a given state. We have applied our results to the problem of visualizing a part of the set of reachable configurations produced by a state-space exploration tool. In this setting, the advantage of our approach is not to reduce the inherent cost of projection, but rather to avoid performing redundant computations. This makes the procedure efficient when only slight modifications are applied to the value of parameters, such as the zoom factor or the coordinates of the visualization window. A prototype implementation of the proposed method has been developed, showing that the approach provides clear benefits. Although the focus of this paper was on sets on integers represented by finite-word automata, it is worth mentioning that our technique straightforwardly generalizes to mixed integer and real sets represented by weak infinite-word automata [BJW05]. Future work will address the problem of making PPA compatible with efficient representations of automata, such as [Cou04].

7 Acknowledgments

We would like to thank Louis Latour and Laetitia Smisdom for their contribution to the investigation of the visualization problem.

References

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proc. 8th CAV*, volume 1102, pages 1–12. Springer, 1996.

[BHMV94] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. *Bulletin of the Belgian Mathematical Society*, 1(2):191–238, March 1994.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and Tayssir Touili. Regular model checking. In *Proc. 12th CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000.

[BJW05] B. Boigelot, S. Jodogne, and P. Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Trans. Comput. Logic*, 6(3):614–633, 2005.

[BL04] B. Boigelot and L. Latour. Counting the solutions of Presburger equations without enumerating them. *Theoretical Computer Science*, 313:17–29, 2004.

[Boi98] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces.* PhD thesis, University of Liège, Belgium, 1998.

[BW02] B. Boigelot and P. Wolper. Representing arithmetic constraints with finite automata: An overview. In *Proc. 18th ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[Büc62] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. International Congress on Logic, Methodoloy and Philosophy of Science*, pages 1–12, Stanford, 1962. Stanford University Press.

[Cou04] J.-M. Couvreur. A BDD-like implementation of an automata package. In *Proc. 9th CIAA*, volume 3317 of *Lecture Notes in Computer Science*, pages 310–311. Springer, 2004.

[Hop71] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. *Theory of Machines and Computation*, pages 189–196, 1971.

[Lat05] L. Latour. *Presburger Arithmetic: From Automata to Formulas.* PhD thesis, University of Liège, Belgium, 2005.

[Ler05] J. Leroux. A polynomial time Presburger criterion and synthesis for number decision diagrams. In *Proc. 20th LICS*, pages 147–156. IEEE Computer Society, 2005.

[Opp78] D. C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16:323–332, 1978.

[Pre29] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves*, pages 92–101, Warsaw, 1929.

[SKR98] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of presburger engines for EFSM reachability. In *Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427, pages 280–292. Springer, 1998.

[WB95] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proc. 2nd SAS*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 1995.