Symbolic Verification With Gap-Order Constraints

Laurent Fribourg[1] & Julian Richardson[1*]

[1]Ecole Normale Supérieure/CNRS - 45 rue d'Ulm - 75005 Paris
email: fribourg,richard@dmi.ens.fr

Abstract. Finite state automata with counters are useful for modeling systems with discrete parameters. The calculation of *state invariants* is an important tool in the analysis of such systems. Previous authors have presented techniques for the calculation of state invariants based on their approximation by convex polyhedra or periodic sets.
In this paper we present a new method for the calculation of invariants for finite state automata with counters, based on their representation by *gap-order constraints*. This method differs from previous approaches by exploiting existing techniques for the calculation of least fixed points. The use of least fixed points reduces the need for approximation and allows the generation of non-convex invariants. We do not need to specify the initial inputs to the automaton, but can leave them as uninstantiated parameters, or partially specify them using gap-order constraints.

1 Introduction

In this paper we present a new method for determining state invariants for finite state automata with counters. Existing methods (e.g. [5, 7]) make use of an approximation operation called *widening*. One of the drawbacks of widening is that it decreases the accuracy of the resulting invariant. Our new method reduces the need for widening by exploiting two techniques for calculating precise fixed points.

As in [5], the automaton is divided into progressively larger cycles. The major difference from [5] is, however, that we can calculate least fixed points (i.e. *exact* invariants) for some of the cycles. The method of [2, 4] is used to calculate least fixed points for the innermost loops, and the method of [8] is used to calculate least fixed points for the outermost loops. This leads us to choose a different representation scheme to that used by Halbwachs.

A significant advantage of our method is that we do not need to specify the initial inputs to the automaton, but can leave them as uninstantiated parameters, or partially specify them using gap-order constraints.

After a few preliminary definitions (§2), we present the method in its basic form (§3). §4 goes through an example. Some approximation is needed, and this is discussed in §5. After a brief discussion of our implementation and a presentation of results (§6), we compare our technique with previous work in §8. We discuss the representation of invariants by periodic sets, and by convex hulls. Analysis of an example from [6] suggests that the methods and representations are complementary. The paper finishes by indicating our plans for further work, and drawing the conclusions.

2 Preliminaries

An automaton with counters is a finite-state automaton augmented by a certain number of integer variables: the counters. Transitions consist of a guard part and an operation part. The guard part is a linear relation over the counters. A transition $s_1 \xrightarrow{?\phi;\psi} s_2$ may be made from state s_1 with counter values $\overline{x}$ to state s_2 with counter values $\overline{x}'$ if the guard part evaluates to true

[1] The second author is supported by EC HCM grants Compulog ERBCHBGCT 930365 and Logic Program Synthesis and Transformation CHRX.CT.93 0414.

when applied to $\overline{x}$, i.e. $\phi(\overline{x})$ holds. In this case, the operation part is executed, yielding new counter values $\overline{x}'$ such that $\psi(\overline{x}, \overline{x}')$ is satisfied. Such a transition can be encoded as a Prolog clause (with argument a tuple of counters, $\overline{x}$):

$$p(s_2, \overline{x}') \text{ :- } p(s_1, \overline{x}), \phi(\overline{x}), \psi(\overline{x}, \overline{x}').$$

Operations $x' = x + k$ and $x' := k$, where k is a positive or negative integer, are called incrementation and reinitialisation respectively.

In our analysis, we distinguish between two kinds of transition:

1. Internal transitions: transitions which start and end in the same state s, whose operation parts may contain incrementation, but not reinitialisation.
2. External transitions: transitions which start in a state s and end in a distinct state t, incrementing or reinitialising the counters.

We distinguish accordingly three levels of *cyclic paths*, depicted in figure 1:

1. Internal cycles. These are paths starting from and ending in a state s using repeatedly a set of internal transitions. We use the notation s^* to signify a state s together with a set of internal transitions on s.
2. Simple circuits. A simple circuit (with two states) is a loop consisting of an internal cycle on a state s, an external transition from s to a state t, an internal cycle on t and another external transition back to s. Such a circuit is denoted $s^* \rightarrow t^* \rightarrow s^*$. More generally, a simple circuit consists of a number $n+1$ of internal cycles connected by n external transitions. Such a circuit is denoted $s^* \rightarrow t_1^* \rightarrow \ldots \rightarrow t_n^* \rightarrow s^*$
3. General cyclic paths: these paths are obtained by composing simple circuits.

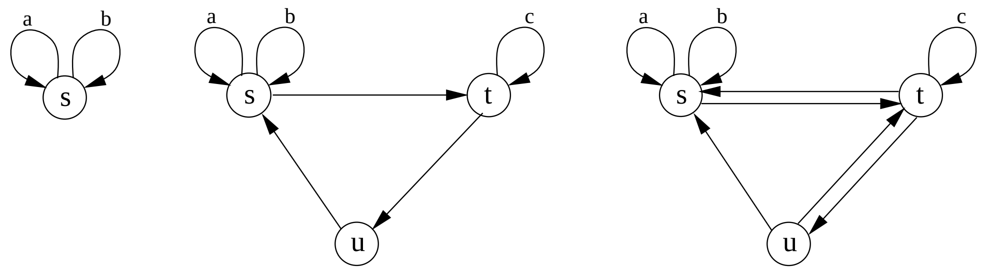


Fig. 1. The three levels of cyclic paths. From left to right: internal cycles, simple circuits and general cyclic paths.

Before analysis can proceed, the automaton must be checked (manually) for transitions of the form $s \xrightarrow{?\phi;\psi} s$, where ψ contains a reinitialisation. These 'illegal' internal transitions are removed by replacing the state s by two states, s_1 and s_2, linked by the external transition $s_1 \xrightarrow{?\phi;\psi} s_2$. Transitions into and out of s may then need to be duplicated, with one copy for each of the new states. In practice, many of these extra transitions can never be made and so can be eliminated.

Our method combines two methods for computing invariants, one for computing the invariants of internal cycles [2], and another for simple circuits [8].

3 Method

The basis of the method (see figure 2) is the replacement of simple circuits in the automaton with *meta-transitions* on a single state, and then using bottom-up evaluation to calculate the state invariant.

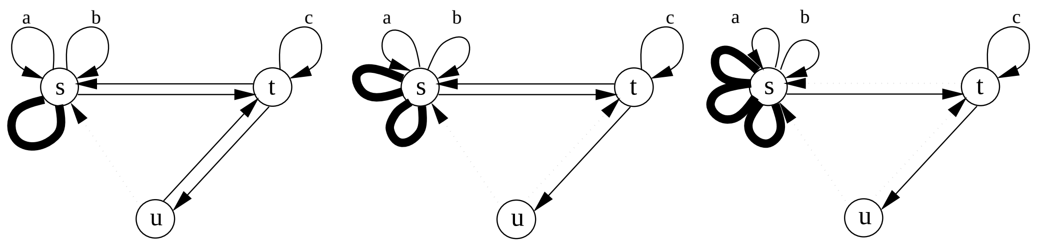


Fig. 2. From left to right, simple cycles are removed and replaced by meta-transitions (thick lines).

More precisely, there are five steps to the method:

1. Identify the internal cycles in the automaton.
   We exploit the method of [2] to calculate least fixed points for the internal cycles. This allows us to calculate more accurate invariants than if, for example, we just treated internal cycles in the same way as we treat simple circuits below. The least fixed points can be seen as recurrence relations, relating values of the counters in a state before and after iterated application of its internal cycles.[2]
2. Identify the simple circuits in the automaton.
   For each simple circuit, we derive a recurrence relation linking the values of the counters at the entry of a state s before and after the execution of the simple circuit.
   The recurrence relations are added as *meta-transitions* from s to s (see [1]), and the final transition ($t_n \rightarrow s$) of each simple circuit is deleted. Meta-transitions can be expressed as Prolog clauses in the same way as ordinary transitions.
3. Approximate the meta-transitions as *gap-order constraints*. A gap-order constraint is a total ordering of variables together with a "gap assignment" to each order relation where each gap assignment is a nonnegative integer subscript that denotes the minimal difference between two elements. The notation $X <_c Y$ means that $X + c < Y$. For example, the gap-order constraint $X <_0 5 <_3 Y <_2 Z = W$ is equivalent to $X < 5 \wedge 5{+}3 < Y \wedge Y{+}2 < Z \wedge Z = W$. Approximation is not always necessary, and need not result in a loss of accuracy in the invariant which is finally calculated (see §5).
4. Find a fixed point for the system of meta-transitions. Revesz's method [8] calculates the least fixed point of such a system of meta-transitions by bottom-up evaluation. The output constitutes the invariant at s.
5. Use the remaining transitions in the automaton to *propagate* the invariant at s to the other states.

4 Example: A Read/Write File System

In this section we outline the analysis of an automaton describing a set of readers and writers. The behaviour to be verified is that no reads can take place when the system is writing.

[2] These differ from the meta-transitions derived for the simple circuits in that they are *fixed* points, μ such that $\mu = \mu^*$.

The example is presented in [7] as a LOTOS specification in three parts: the reader, the writer, and the control process. A system with r_{max} readers and d_{max} writers is formed by synchronising the readers and the writers with the control process, which can then be seen as a single automaton. Unless the number of readers and writers is small, this quickly results in a large number of states. Here, we have simplified the problem somewhat. Instead of forming the parallel composition of r_{max} readers, d_{max} writers and a control process, we form the parallel composition of three processes: a reader which can perform up to r_{max} reads simultaneously, a writer which can buffer up to d_{max} writes simultaneously, and a control process.

Kerbrat performs the analysis for a number of specific positive values of r_{max}, d_{max}. Here, we are able to compute the invariant parameterised by d_{max}, r_{max}.

The automaton is displayed in figure 4. In order to permit analysis by [2], state *resource* (left of diagram) must be split into two states, $res0$ and $res1$, whose internal cycles contain no reinitialisation Once this has been done, we arrive at the rules in figure 3.

$$
\begin{array}{rlr}
p(res0, 0, 0, 0). & & (m0) \\
p(res1, R, D, 0) :- & p(res0, R, D, W). & (m1.1) \\
p(res1, R, D, W) :- & p(res1, R, D, W). & (m1.2) \\
p(res0, R-1, D, W) :- & p(res0, R, D, W), R > 0. & (m2.1) \\
p(res0, R-1, D, W) :- & p(res1, R, D, W), R > 0. & (m2.2) \\
p(res1, R, D+1, 0) :- & p(res0, R, D, W), D < d_{max}. & (m3.1) \\
p(res1, R, D+1, W) :- & p(res1, R, D, W), D < d_{max}. & (m3.2) \\
p(not\_writing, R, D, W) :- & p(res0, R, D, W), W = 0. & (m4.1) \\
p(not\_writing, R, D, W) :- & p(res1, R, D, W), W = 0. & (m4.2) \\
p(res0, R+1, 0, W) :- & p(not\_writing, R, 0, W), R < r_{max}. & (m5) \\
p(demands, R, D, W) :- & p(not\_writing, R, D, W), D > 0. & (m6) \\
p(res0, 0, D-1, 1) :- & p(demands, 0, D, W), D > 0. & (m7)
\end{array}
$$

Fig. 3. The code for the read/write automaton, after splitting state *resource* into two.

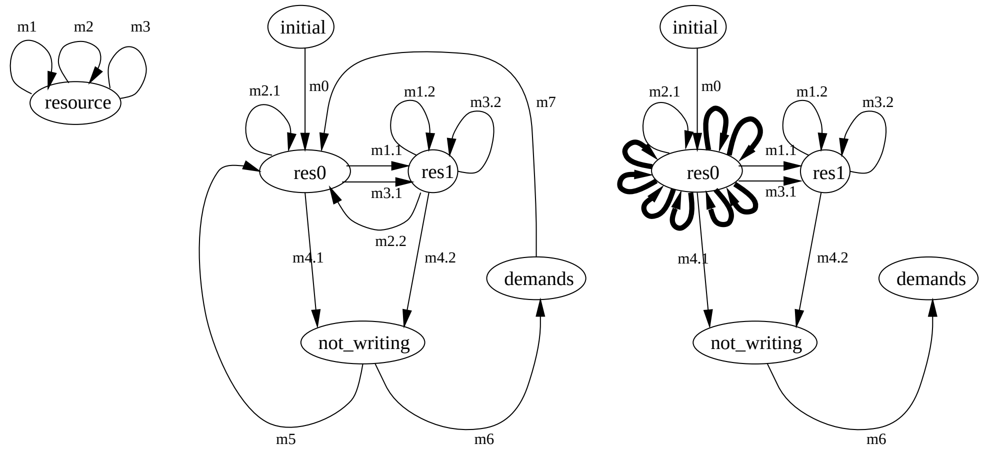


Fig. 4. State *resource* in the original rules is shown on the left of the diagram. The automaton, modified to split this into two, is shown in the middle. On the right, the simple circuits have been replaced by meta-transitions on $res0$.

4.1 Analysis

Internal Cycles The automaton contains two internal cycles:

1. $res0^*$. If the values at entry to $res0$ are r,d,w, then the values after n applications of rule (m2.1) are: $w' = w$, $r' = r - n$, $d' = d$. Applying (manually) the algorithm of [2] gives a least fixed point with bounds $0 \leq n \leq r$, which in turn gives constraints $w' = w$,$0 \leq r' \leq r$,$d' = d$. This least fixed point can be encoded as a Prolog clause:
$$p(res0, R_1, D_1, W_1) : -p(res0, R, D, W), 0 \leq R_1, R_1 \leq R, D_1 = D, W_1 = W. \quad (1)$$
2. $res1^*$. In state $res1$, we apply rule (m1.2) k times and rule (m3.2) l times to get the following: $r' = r$,$d' = d + l$,$w' = w$. Again, we use the algorithm of [2] to manually produce a least fixed point with bounds $0 \leq l \leq d_{max} - d$, giving $r' = r$, $d \leq d' \leq d_{max}$,$w' = w$. This least fixed point can be encoded as a Prolog clause:
$$p(res1, R_1, D_1, W_1) : -p(res1, R, D, W), R_1 = R, D \leq D_1, D_1 \leq dmax, W_1 = W. \quad (2)$$

Simple Circuits There are a total of 8 simple circuits:

1. $res0^*$-(m1.1)-$res1^*$-$res0^*$
2. $res0^*$-(m3.1)-$res1^*$-$res0^*$
3. $res0^*$-$not\_writing$-$demands$- $res0^*$
4. $res0^*$-$not\_writing$-$res0^*$
5. $res0^*$-(m1.1)-$res1^*$- $not\_writing$-$demands$-$res0^*$
6. $res0^*$-(m3.1)-$res1^*$- $not\_writing$-$demands$-$res0^*$
7. $res0^*$-(m1.1)-$res1^*$- $not\_writing$-$res0^*$
8. $res0^*$-(m3.1)-$res1^*$- $not\_writing$-$res0^*$

Note that the two transitions (given by rules (m1.1) and (m3.1)) from $res0$ to $res1$ each give rise to separate circuits.

Here, we only give the detailed analysis of one circuit. The other analyses are very similar.

Simple circuit $res0^*$-(m1.1)-$res1^*$- $res0^*$

Assume the counters have values r_0,d_0,w_0 at entry to $res0$. Clause (1) describes how the counter values change after an arbitrary number of applications of the internal cycle $res0^*$:
$$p(res0, R_1, D_1, W_1) : -p(res0, R_0, D_0, W_0), 0 \leq R_1, R_1 \leq R_0, D_1 = D_0, W_1 = W_0. \quad (3)$$

Now a transition to $res1$ is made using rule (m1.1). The effect of this transitions is calculated by composing (3) above with (m1.1), giving:
$$p(res1, R_2, D_2, W_2) : -p(res0, R_0, D_0, W_0), 0 \leq R_2, R_2 \leq R_0, D_2 = D_0, W_2 = 0. \quad (4)$$
The effect of applying internal cycle $res1^*$ is calculated by composing (4) above with (2), giving:
$$p(res1, R_3, D_3, W_3) : -p(res0, R_0, D_0, W_0), 0 \leq R_3, R_3 \leq R_0, D_0 \leq D_3, D_3 \leq dmax, W_3 = 0. \quad (5)$$
Finally, we return to state $res0$ by composing (5) above with (m2.2), to give:
$$p(res1, R_4, D_4, W_4) : -p(res0, R_0, D_0, W_0), -1 \leq R_4, R_4 < R_0, D_0 \leq D_4, D_4 \leq dmax, W_4 = 0. \quad (6)$$
We conjoin this with the domain constraint $r \geq 0$ and rename the variables to get the meta-transition:
$$p^{entry}_{res0}(r', d', w') \leftarrow p^{entry}_{res0}(r, d, w), 0 \leq r', r' < r, d \leq d', d' \leq d_{max}, w' = 0$$

The process of composing transitions to form meta-transitions can be automated. Sequential composition of two transitions is equivalent to partially evaluating the composition of the two Prolog clauses which encode them, followed by solving for the resulting inequalities. This has been implemented (see §6).

Similarly, we derive the following meta-transitions for each of these circuits (in the same order as above). Note that when the transitions and internal cycles which make up 8 are composed, the resulting meta-transition has unsatisfiable (i.e. $false$) constraints. It is therefore omitted.

$$
\begin{aligned}
&p^{entry}_{res0}(r', d', w') \leftarrow p^{entry}_{res0}(r, d, w), 0 \leq r', r' < r, d \leq d', d' \leq d_{max}, w' = 0\\
&p^{entry}_{res0}(r', d', w') \leftarrow p^{entry}_{res0}(r, d, w), 0 \leq r', r' < r, d < d', d' \leq d_{max}, r \leq r_{max}, w' = 0\\
&p^{entry}_{res0}(r', d', w') \leftarrow p^{entry}_{res0}(d, r, w), r' = 0, 0 \leq d' = d - 1 < d_{max}\dagger, w' = 1, r \leq r_{max}\\
&p^{entry}_{res0}(r', d', w') \leftarrow p^{entry}_{res0}(r, d, w), 0 < r', r' \leq r + 1\dagger, r' \leq r_{max}, d' = 0, d \leq d_{max}, w' = 0\\
&p^{entry}_{res0}(r', d', w') \leftarrow p^{entry}_{res0}(d, r, w), r' = 0, 0 \leq d', d' < d_{max}, d' \geq d - 1\dagger, r \leq r_{max}, w' = 1\\
&p^{entry}_{res0}(r', d', w') \leftarrow p^{entry}_{res0}(d, r, w), r' = 0, 0 \leq d', d' < d_{max}, r \leq r_{max}, w' = 1\\
&p^{entry}_{res0}(r', d', w') \leftarrow p^{entry}_{res0}(r, d, w), 0 < r', r' \leq r + 1\dagger, r' \leq r_{max}, r \leq r_{max}, d \leq d_{max}, d' = 0, w' = 0
\end{aligned}
$$

Replacing the simple circuits by their corresponding meta-transitions gives the rightmost part of figure 4.

Approximation The four constraints marked with a † cannot be translated to an equivalent gap-order form. They must be replaced by upper approximations: $r' \leq r+1$ is approximated by $r' \leq r_{max}$, $d' = d-1$ is approximated by $d' < d$. $d' \geq d-1$ is equivalent to $d' \geq d \vee d' = d-1$, which in view of the above approximation, is replaced by $d' \geq d \vee d' < d$, which is a tautology, and so omitted. We show in §5 that in this case, approximation does not affect the resulting invariant.

After these approximations, conversion to gap-order form is automatic.

Solution of Meta-transitions After making the necessary approximations described in §5 and converting to gap-order form, we can either substitute particular positive integers for r_{max} and d_{max}, or we can leave them as variables, with constraints $r_{max} > 0$ and $d_{max} > 0$. A great strength of Revesz's method is that the initial values of the counters need not be given specific values, but can be specified by a set of gap-order constraints.

We find an invariant describing the counter values at entry to state $s1$ which is a disjunction of 14 gap-order constraints:

$$
\begin{aligned}
&p^{entry}_{res0}(r, d, w) \leftarrow d = 0 \;\wedge\; r = 0 \;\wedge\; w = 0 \;\wedge\; 0 <_0 dmax \wedge\; 0 <_0 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = 0 \;\wedge\; r = 0 \;\wedge\; w = 0 \;\wedge\; 0 <_0 dmax \wedge\; 0 <_0 rmax \;\wedge\; d <_0 dmax \;\wedge\; r <_0 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = 0 \;\wedge\; r = 0 \;\wedge\; w = 1 \;\wedge\; 0 <_0 dmax \wedge\; 0 <_0 rmax \;\wedge\; d <_0 dmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = 0 \;\wedge\; r = rmax \;\wedge\; w = 0 \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_0 r \;\wedge\; 0 <_0 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = 0 \;\wedge\; r = rmax \;\wedge\; w = 0 \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_0 r \;\wedge\; 0 <_0 rmax \;\wedge\; d <_0 dmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = 0 \;\wedge\; w = 0 \;\wedge\; 0 <_0 r <_0 rmax \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_0 rmax \;\wedge\; d <_0 dmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = 0 \;\wedge\; w = 0 \;\wedge\; 0 <_0 r <_0 rmax \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_0 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = dmax \;\wedge\; r = 0 \;\wedge\; w = 0 \;\wedge\; 0 <_0 d \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_0 rmax \;\wedge\; r <_0 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = dmax \;\wedge\; r = 0 \;\wedge\; w = 0 \;\wedge\; 0 <_0 d \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_1 rmax \;\wedge\; r <_1 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = dmax \;\wedge\; w = 0 \;\wedge\; 0 <_0 r <_0 rmax \;\wedge\; 0 <_0 d \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_0 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow d = dmax \;\wedge\; w = 0 \;\wedge\; 0 <_0 r <_1 rmax \;\wedge\; 0 <_0 d \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_1 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow r = 0 \;\wedge\; w = 0 \;\wedge\; 0 <_0 d <_0 dmax \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_0 rmax \;\wedge\; r <_0 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow r = 0 \;\wedge\; w = 1 \;\wedge\; 0 <_0 d <_0 dmax \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_0 rmax\\
&p^{entry}_{res0}(r, d, w) \leftarrow w = 0 \;\wedge\; 0 <_0 d <_0 dmax \;\wedge\; 0 <_0 r <_0 rmax \;\wedge\; 0 <_0 dmax \;\wedge\; 0 <_0 rmax
\end{aligned}
$$

These can be manually simplified to:

$$p^{entry}_{res0}(r,d,w) \leftarrow \begin{array}{c} w = 0 \wedge \begin{cases} 0 \le d \le d_{max} \wedge 0 \le r < r_{max} \\ \vee \\ d = 0 \wedge r = r_{max} \end{cases} \\ \bigvee \\ w = 1 \wedge r = 0 \wedge 0 \le d < d_{max} \end{array} \tag{7}$$

Propagation The invariant in a single state s can be encoded as a set of clauses:

$$p_s(\overline{x}) \leftarrow \Phi_i(\overline{x})$$

Propagation is carried out by composing these clauses with the clauses for the remaining transitions in the automaton.

Here, we propagate the values at entry to $res0$ (equation (7)) to the other states in the automaton.

In $res0$, we can repeatedly apply (m2.1), but this does not change the invariant, giving:

$$p_{res0}(r,d,w) \leftarrow p^{entry}_{res0}(r,d,w)$$

Transferring to $res1$ using either (m1.1) or (m3.1), then repeatedly applying (m1.2) and (m3.2) gives:

$$p_{res1} \leftarrow w = 0 \;\wedge\; 0 \le r \le r_{max} \;\wedge\; 0 \le d \le d_{max}$$

The invariant in $not\_writing$ is the disjunction of the invariants formed by transferring from $res0$ using (m4.1) and from $res1$ using (m4.2). This ensures that $w = 0$, giving the same invariant as above:

$$p_{not\_writing} \leftarrow p_{res1}$$

Transfer to $demands$ using (m6), which ensures that $d > 0$, giving:

$$p_{demands} \leftarrow w = 0 \;\wedge\; 0 \le r \le r_{max} \;\wedge\; 0 < d \le d_{max}$$

From these equations, we can readily verify, as required, that $w = 1 \rightarrow\; r = 0$, and $d + w \le d_{max}$. Note that this invariant holds for *all values* of d_{max} and r_{max}, not just specific values.

5 Approximation by Gap-Order Constraints

The analysis of the previous section required four constraints to be replaced by upper approximations in order to ensure that the meta-transitions were in gap-order form and so could be solved by the method of Revesz. What are the effects of these approximations?

1. $r' \le r + 1$. This is approximated by $r' \le r_{max}$. This approximation can be safely made, because the extra values allowed by the approximation (i.e. those with $r+1 < r' \le r_{max}$) can all be achieved by multiple applications of the unapproximated rule, i.e. the approximated rule is equivalent to an iterated version of the unapproximated rule.
2. $d' = d - 1$. This is approximated by $d' < d$. Iteration of the original rule gives all values $d' < d$, so again, the approximated rule is equivalent to an iterated version of the unapproximated rule.
3. $d' \ge d - 1$. This is deleted entirely. The range of values allowed in the approximated rule can, however, be achieved by iterating the original rule: values of d' such that $d' < d$ can be achieved by repeatedly applying the original rule with $d' = d - 1$. The other values of d' are allowable by both rules. Therefore again we see that the approximated rule is equivalent to an iterated version of the unapproximated rule.

6 Revesz's Algorithm

Once the simple circuits (compositions of transitions) have been replaced by equivalent meta-transitions, an invariant is found by constructing a fixed point of the meta-transitions. This is a difficult process, and to obtain a practical algorithm, some kind of approximation is necessary.

Revesz's algorithm [8] provides us with a terminating algorithm for calculating the *least fixed point* of a set of datalog clauses, as long as they are in gap-order form. The approximation in our method takes place in the translation to gap-order form.

The essence of Revesz's algorithm is the following. At any point during the bottom-up evaluation process, the current solution set will be represented by a set (disjunction) $i = 0...n$ of constraints:

$$p(\overline{x}) \leftarrow \phi_i(\overline{x}).$$

The set of constraints is extended by applying the input gap-order constraints (rules) in every possible way to the existing set of constraints.[3] In order to apply an input rule (8) to an existing solution (9), the variables in $\overline{x}$ are renamed to match those in $\overline{x}_0$, $\phi(\overline{x}_0)$ and $\psi(\overline{x}_0, \overline{x}_1)$ are conjoined, the variables $\overline{x}_0$ are eliminated, and finally the new constraint is added to the solution set if it does not subsume an existing constraint.

$$p(\overline{x}_1) \leftarrow p(\overline{x}_0), \psi(\overline{x}_0, \overline{x}_1) \tag{8}$$

$$p(\overline{x}) \leftarrow \phi(\overline{x}) \tag{9}$$

In order for the algorithm to yield a least fixed point, elimination of the variables $\overline{x}_0$ from a conjunction $\phi(\overline{x}_0) \wedge \psi(\overline{x}_0, \overline{x})$ must yield an equivalent formula. Revesz's algorithm provides a way (the shortcut operation described below) of achieving this for gap-order constraints.

In Revesz's algorithm, gap-order constraints are represented as *gap graphs*. Each gap-graph represents a conjunction of gap-order constraints. Three operations are defined on gap-graphs:

1. Merge. Merging two gap-graphs produces a gap-graph which corresponds to the conjunction of the inputs. If the inputs are inconsistent, then the merge operation fails.
2. Shortcut. The shortcut operation is used to eliminate a variable from a gap-graph.
3. Subsumption checking.

These operations are depicted in figure 5.

Our implementation consists of Prolog predicates for the translation of meta-transitions into gap-order form, and their subsequent solution using Revesz's algorithm. Occasionally, the user is required to supply an appropriate approximation of a meta-transition as a gap-order constraint.

Some optimisations have been made to Revesz's algorithm for efficiency reasons: an agenda keeps track of combinations of rules and database members which have yet to be processed, and the *merge* operation has been split into symmetric and asymmetric parts. Some simplification of the database is carried out during the execution of Revesz's procedure, by application of a number of equivalence-preserving normalisation rules, followed by removal of duplicate gap-graphs from the database. This is quite easily proved to preserve the correctness of the algorithm. Altogether, the optimisations have made the code roughly ten times faster. More sophisticated representations for gap-graphs (such as hash tables) would be expected to produce a further ten-fold improvement in code speed.

The procedures for composing transitions to form meta-transitions, translating the meta-transitions into gap-order form, solving them using Revesz's algorithm, and subsequently translating them back to meta-transitions, have been integrated into a workbench.

[3] The clauses derived from meta-transitions define a single recursive predicate, p. Revesz's algorithm allows clauses with more than one recursive predicate.

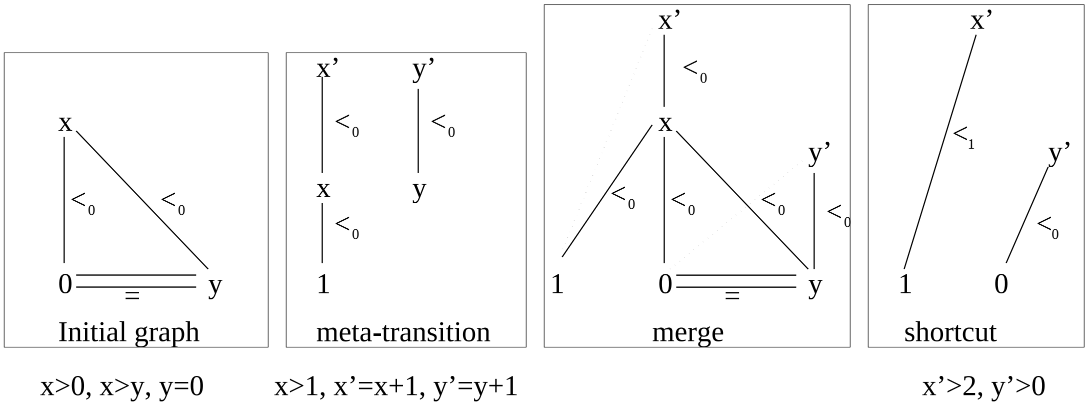


Fig. 5. The gap-graph operations.

7 Results

This section provides a summary of the results. The fields in the table are respectively, the name of the problem, the number of states in the automaton, the number of transitions in the automaton, the number (M0) of meta-transitions before conversion to their gap-graphs, the number (M1) of meta-transitions after conversion to gap-graphs, the CPU time taken to evaluate the meta-transitions using Revesz's procedure in Sicstus Prolog on a Sparcstation-10 with 48M of memory, whether approximation was required to convert the meta-transitions to gap-order form (A?), the number (C) of gap-order constraints in the invariant found, and whether the desired invariants were found (Invts?).

Example	States	Counters	Transitions	M0	M1	CPU time	A?	C	Invts?
Subway[a]	5	3	13	4	10	4.1s	No	7	✓
Car[a]	3	3	5	2	3	0.8s	✓	5	✓
Readwriter(5,3)[b]	3	5	12	8	53	38s	✓	12	✓
Readwriter(n,m)	3	5	12	8	53	30s	✓	14	✓
Comparison §8.2	2	3	5	3	5	0.5s	✓	3	No
Comparison §8.2[c]	2	3	5	3	5	0.8s	✓	4	✓
Boolean circuit[d]	6	9	12	7	11	3.2s	✓	1-4	✓

[a] These examples are also treated in [5]. We present a detailed treatment of them in [3].
[b] The reader-writer example is discussed in §4. The parameters are d_{max} and r_{max} in that order.
[c] After addition of a new variable $z = x - y$ to the automaton. See §8.2.
[d] This automaton is a part of a larger example passed to us by an industrial collaborator. The invariants were calculated for several sets of initial values.

The results suggest that approximation is often necessary.

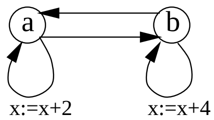
It is surprising that the readwriter problem takes less time to solve when parameters d_{max} and r_{max} are set to specific values than in the abstract case. The order in which gap graphs are produced during bottom-up evaluation can effect the runtime, and that could be the cause of this phenomenon. It is not surprising, however, that the solution contains less gap-graphs, because some gap-graphs which are different when the values of d_{max} and r_{max} are unknown, become identical when these parameters are assigned specific values, which allows one to be eliminated.

8 Comparison With Previous Work

Both [5, 1] apply abstract interpretation to the determination of state invariants. The primary difference lies in how the abstract domains are represented. Halbwachs represents domains as convex polyhedra, exploiting the efficient algorithms that are known for manipulating convex polyhedra, and making use of the operation of *widening* in order to avoid the generation of an infinite sequence of increasing sets of points.

8.1 Comparison With Periodic Sets

In [1], domains are expressed as *periodic sets*. This representation is chosen because if the state transitions are of a certain form, the invariants which result naturally have a periodic structure. Such periodic structures cannot be represented by gap-order constraints, as shown by the following example (the transitions from state a to b and b to a leave the counter x unchanged):



The simple circuit a-b-a can be represented by the following meta-transition:

$$p_a^{entry}(x') \leftarrow p_a^{entry}(x), x' = x + 2$$

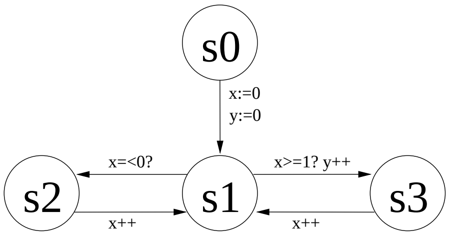
The guard is not, however, a gap-order constraint. In order to make it so, we have to replace the equality on the right hand side with an inequality, $x' > x$. Now, the exact state invariant for state a is given by $x = x_0 + 2\lambda$ for $\lambda \in \mathbf{N}$, whereas that which we derive is simply $x \geq x_0$. Thus, we have lost the periodicity inherent in the system. Note, however, that if we replace $x := x + 4$ by $x := x + 3$ in the internal transition on b, then the system is no longer periodic and $x \geq x_0$ is almost the exact solution. [4] Commonly, periodic subsystems do not fit together, with the result that no periodicity appears when they are combined. The approximation by gap-order constraints seems appropriate in such cases. Furthermore, the use of gap-order constraints, rather than periodic sets or convex polyhedra, provides us with a formal explanation of two phenomena observed in [1, p.60]: first, the bottom-up evaluation procedure with a subsumption test often terminates in practice (without use of widening), second, the output may be not convex (due to the presence of disjunctions).

8.2 Comparison With Convex Hulls: An Example

Figure 6 shows an example used by Kerbrat [6, pp142–143] to illustrate the effect of different widening strategies on the invariants produced by his method. In this section, we shall see that using our techniques we arrive at quite different invariants to Kerbrat's. This suggests the techniques are complementary. Note that the exact invariant at $s1$ is: $\Omega_1 = \{x = y = 0 \vee x > 0 \wedge y = x - 1\}$.

Kerbrat's Invariants Kerbrat determines the invariant in state $s1$ using three different widening strategies. The most accurate invariant, produced by "delayed widening", is $\Phi_1 = \{x \leq y + 1, 0 \leq y\}$.

[4] The exact solution is $x = x_0 \vee x \geq x_0 + 2$.



$$p(s1,0,0) \leftarrow p(s0,x,y). \tag{10}$$
$$p(s2,x,y) \leftarrow p(s1,x,y), x \leq 0. \tag{11}$$
$$p(s1,x+1,y) \leftarrow p(s2,x,y). \tag{12}$$
$$p(s3,x,y+1) \leftarrow p(s1,x,y), x \geq 1. \tag{13}$$
$$p(s1,x+1,y) \leftarrow p(s3,x,y). \tag{14}$$

Fig. 6. Top: the automaton and its transition rules encoded in Prolog.

Our Invariants Using the techniques we have described, we arrive at the following invariant for state $s1$.

$$\Psi_1 = \{x \geq 0 \wedge y \geq 0\}$$

This invariant can be improved if we introduce a new counter $z = x - y$, and amend the automaton rules correspondingly, so that z is incremented when x is incremented or y is decremented, and decremented whenever x is decremented or y is incremented.

The new invariant is:

$$\Psi_1' = \{x \geq 0 \wedge y \geq 0 \wedge z = 0 \ \vee \ x > 1 \wedge y \geq 0 \wedge z > 0\}$$

Since $z = x - y$, we can incorporate its constraints into the diagrammatical representation of the state invariant Ψ_1'.

Figure 7 depicts Kerbrat's invariant calculated using delayed widening, and the invariant calculated by our methods after introduction of the new counter.

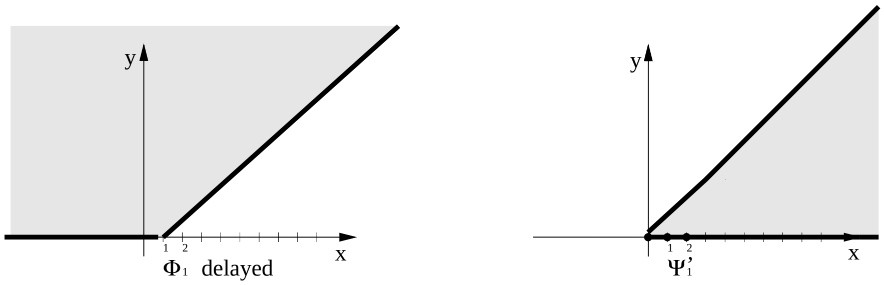


Fig. 7. Kerbrat's invariant, Φ_1, (left), and ours, Ψ_1 (right).

The invariant produced by our method is substantially different from that produced by the convex hull technique. This suggests that the two methods are complementary.

9 Further Work

Most of the examples given in [5, 7, 1] have been successfully treated semi-automatically with our method. Composition of transitions to form meta-transitions, translation of meta-transitions to gap-order form and their subsequent solution by Revesz's method have been almost completely automated in Prolog. Once automation is complete, it is important to treat some large examples. At the same time, it will be necessary to carefully consider the complexity of our algorithms, and determine whether additional approximation operations are required.

10 Conclusion

In this paper we have presented a new method for the calculation of invariants for finite state automata with counters. This differs from previous methods by exploiting existing techniques for the calculation of least fixed points [2, 8]. The use of gap-order constraints allows the representation of non-convex invariants and often makes bottom-up evaluation terminate without the need for widening.

A significant advantage of our approach is that we can treat parameterised problems in which the initial values of the counters are not specified, or are partially specified by gap-order constraints.

The technique has been implemented in Prolog.

A A Short Session With the Workbench

```
| ?- workbench.
Welcome to the Invariant Workbench.
All user inputs and commands are appended to the file ./tmp_gap_script.
Type help. for help.

>> load(kerbrat_extended).
Loaded rules from file lib/kerbrat_extended.
>> print.
Current ruleset is:

  [
      r6:(p1(s1,x_,y,z_)<--[p1(s3,x,y,z),x_=x+1,z_=z+1]),
      r5:(p1(s3,x,y_,z_)<--[p1(s1,x,y,z),y_=y+1,z_=z-1,x>=1]),
      r4:(p1(s1,x_,y,z_)<--[p1(s2,x,y,z),x_=x+1,z_=z+1]),
      r3:(p1(s2,x,y,z)<--[p1(s1,x,y,z),x=<0]),
      r2:(p1(s1,x,y,z)<--[x=0,y=0,z=0])
   ]
>> compose([r5,r6]).

Added rule v0:(p1(s1,x_,y_,z_)<--[p1(s1,x,y,z),x_=x+1,z_=z-
1+1,y_=y+1,x>=1]).
>> compose([r3,r4]).

Added rule v13:(p1(s1,x_,y_,z_)<--
[p1(s1,x,y,z),x_=x+1,z_=z+1,y_=y,x=<0]).
>> solve([r2,v0,v13]).
Meta transitions:
 [(p1(x,y,z)<--[],[eq(x,0),eq(y,0),eq(z,0)]),
  (p1(x_,y_,z_)<--[p1(x,y,z)],[lt(0,x,x_),eq(z_,z),lt(0,y,y_),lt(0,0,x)]),
  (p1(x_,y_,z_)<--[p1(x,y,z)],[lt(0,x,x_),lt(0,z,z_),eq(y_,y),lt(0,x,0)]),
  (p1(x_,y_,z_)<--[p1(x,y,z)],[lt(0,x,x_),lt(0,z,z_),eq(y_,y),eq(x,0)])]

Rules contain the following free variables: [].

Agenda now contains 3 items... 5 items... 7 items... 5 items.

Result contains 4 gap graphs
```

$$p1(x,y,z) \leftarrow x = 0 \ \wedge \ y = 0 \ \wedge \ z = 0$$

$$p1(x,y,z) \leftarrow y = 0 \ \wedge \ 0 <_0 x \ \wedge \ 0 <_0 z$$
$$p1(x,y,z) \leftarrow z = 0 \ \wedge \ 0 <_0 x \ \wedge \ 0 <_0 y$$
$$p1(x,y,z) \leftarrow 0 <_0 y \ \wedge \ 0 <_0 z \ \wedge \ 0 <_1 x$$

```
>> quit.
Quitting. Saving rules in tmp_rules_.
Saved rules in file lib/tmp_rules_.
```

References

1. B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proceedings of conference on Computer-Aided Verification, 1994*, pages 55–67, 1994.
2. L. Fribourg and H. Olsén. Datalog programs with arithmetical constraints: hierarchic, periodic and spiralling least fixed points. Research report 95-26, Laboratoire Informatique, École Normale Supérieure, Paris, November 1995.
3. L. Fribourg and J. D. C. Richardson. Symbolic verification with gap-order constraints. Report LIENS - 96 - 3, Laboratoire d'Informatique, École Normale Supérieure (LIENS), Paris, February 1996. Available by ftp from host ftp.ens.fr in directory /pub/reports/liens.
4. L. Fribourg and M. Veloso Peixoto. Bottom-up evaluation of Datalog programs with arithmetic constraints. In Alan Bundy, editor, *12th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, pages 311–325, Nancy, France, 1994. Springer-Verlag.
5. N. Halbwachs. Delay analysis in synchronous programs. In *Proceedings of conference on Computer-Aided Verification, 1993*, pages 333–346, 1993.
6. A. Kerbrat. *Méthodes Symboliques pour la Vérification de Processus Communicants: étude et mise en oeuvre*. PhD thesis, L'Université Joseph Fourier - Grenoble I, November 1994. In French.
7. A. Kerbrat. Reachable state space analysis of LOTOS specifications. In *Proceedings of the 7th international conference on formal description techniques*, pages 161–176, 1994.
8. P. Z. Revesz. A closed-form evaluation for Datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 1993. vol. 116, pages 117–149.

This article was processed using the LaTeX macro package with LLNCS style