Symbolic Verification with Gap-order Constraints

Laurent FRIBOURG
Julian RICHARDSON

LIENS - 96 - 3

Symbolic Verification with Gap-order Constraints

Laurent FRIBOURG
Julian RICHARDSON

LIENS - 96 - 3

February 1996

Laboratoire d'Informatique de l'Ecole Normale Supérieure
45 rue d'Ulm 75230 PARIS Cedex 05

Tel : (33)(1) 44 32 00 00
Adresse électronique : fribourg@dmi.ens.fr

Symbolic verification with gap-order constraints

Laurent Fribourg & Julian Richardson*
Ecole Normale Supérieure/CNRS - 45 rue d'Ulm - 75005 Paris
email: fribourg@dmi.ens.fr

February 23, 1996

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.
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.

Résumé

Les automates finis sont utiles pour modéliser les systèmes avec paramètres discrets. Un outil important pour l'analyse de ces systèmes est le calcul des *invariants des états*. D'autres auteurs ont présenté des techniques pour calculer les invariants à base d'approximations par polyèdres convexes.
Dans cet article nous présentons une nouvelle méthode pour calculer les invariants des automates finis avec compteurs. Cette méthode utilise une représentation des invariants par *contraintes d'ordre avec intervalles entiers*. Cette méthode diffère des approches précédentes par son utilisation de techniques de calcul de plus petits points fixes. L'utilisation des plus petits points fixes permet souvent d'obtenir des résultats exacts (non approximés), et permet également d'engendrer des invariants non-convexes.

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

1 Introduction

The extension of finite state automata with arithmetical variables has received recently considerable attention. Such automata are very useful for modeling systems which use parameters (e.g., time) taking values on an infinite (or very large) domain. The domain of the arithmetic variables is either dense (e.g., reals) as used in the model of timed automata [1, 6], discrete as used in the model of automata with (delay-) counters [5], or mixed (hybrid automata). Here we focus on automata with counters over a discrete domain.

An automaton with counters is essentially 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 may only be made if its guard part is true for the current counter values. In this case, the operation part is executed: this may involve addition of fixed integer constants (incrementation) or direct assignment of specific integer values to counters (reinitialisation).

In order to analyse such systems, for every state s, one tries to find some arithmetic relations that are satisfied by the values of the counters whenever the automaton reaches s. Such relations are called the *invariants* for s. Invariants are generally computed in a bottom-up manner starting from the initial values and collecting the different values of the counters every time one goes by state s. The process terminates when the iteration of cyclic paths does not make the counter take new values (stabilisation). To ensure stabilisation, it is often necessary to replace a set of values by some upper approximation.

Different methods have been proposed for the efficient manipulation and approximation of invariant sets. In [7, 5] invariant sets are approximated by convex polyhedra, and in [2] by "periodic sets". Here we propose a third method of representation based on Revesz's [8] notion of a *gap-order constraint*. 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 (e.g., $X <_0 5 <_3 Y <_0 <_2 Z = W$).[1] Our interest in gap-order constraints as a means of representation comes from their closed-form property: given an input of the gap-constraint form and a set of recursive rules with gap-order constraints, the bottom-up evaluation procedure terminates and yields a gap-order constraint as an output.

2 Sketch of the bottom-up evaluation method

Hereafter we assume that the "internal" transitions of the automaton, i.e. the transitions going from a state s to the same state s, only perform incrementation on the counters, but no assignment. It can be seen that such a requirement does not yield any loss of expressivity. For the subcomponents of the automaton made solely of internal transitions, the invariants can be calculated by applying the bottom-up evaluation procedures of [4, 3]. The output is either equivalent to a gap-order constraint or can be often approximated by one. We call the resulting gap-order constraint the *local invariant* associated with the subcomponent. The *global invariants* associated with the whole automaton are then computed by composing local invariants and performing closure under recursion using Revesz's procedure. This method of computing invariants is thus the combination of two bottom-up evaluation procedures (Fribourg-Olsen's and Revesz's ones). More formally, we assume the division of **transitions** into two kinds:

- Internal transitions: transitions which start and end in the same state s, while incrementing the counters (reinitialisation is forbidden).

- 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**, which are shown in figure 1:

[1] $X <_c Y$ where c is a nonnegative constant means: $X + c \leq Y$.

- Internal cycles. These are paths starting from and ending in a state s using repeatedly a set of internal transitions.

- 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. More generally, a simple circuit consists of a number n of internal cycles connected by n external transitions.

- General cyclic paths: these paths are obtained by composing simple circuits.

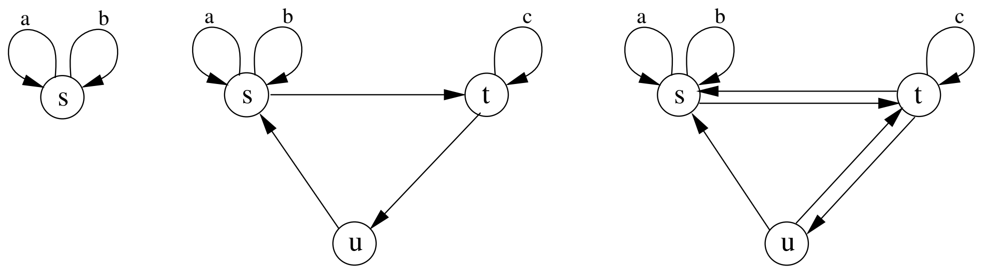


Figure 1: The three levels of cyclic paths. From left to right: internal cycles, simple circuits and general cyclic paths.

Following the hierarchical methodology presented in [2], we determine the **invariants** for a state s at four successive levels:

- Level 1: Invariants are calculated by the method of [3] for the internal cycles.

- Level 2: A recurrence relation is derived linking the values of the counters at the entry of a state s before and after the execution of a simple circuit. The recurrence relations can be seen as meta-transitions from s to s (see [2]). Ideally, the guard of each meta-transition is already a gap-order constraint. If not, we replace it by a gap-order constraint which is an upper approximation.

- Level 3: The meta-transitions found at level 2 are evaluated bottom-up using Revesz's procedure. Once the meta-transitions have been rewritten into a suitable form the resulting invariant is automatically calculated by our Prolog implementation (§6) of Revesz's method. The output is a gap-order constraint which represents the invariant relation satisfied by the counters values at the *entry* of state s.

- level 4: The invariant found for s at level 3 is then propagated through the automaton; the invariant calculated for one state s is used as the input to states t to which s is connected. The closure property of gap-order form ensures that since the invariant at s is in gap-order form, so will be the invariant at t. Propagation may involve further calculations at levels 1–4 for other parts of the automaton. (Consider, for example, two complex automata A and B linked by a single transition. Clearly the calculation of invariants in A and B should be calculated in the same way, using the invariants found for A as inputs to B.)

Figure 2 depicts the replacement of cycles in the automaton with meta-transitions.

In the following sections we illustrate our method on two examples drawn from [5]. We have also applied our techniques by hand to most of the examples from [5, 7, 2].

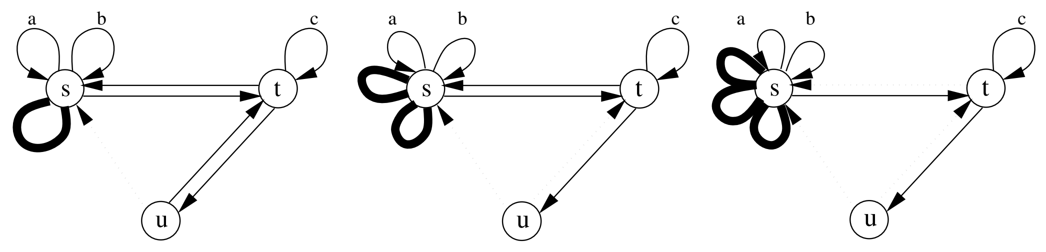


Figure 2: From left to right, simple cycles are removed and replaced by meta-transitions (thick lines).

3 An example

3.1 Informal specification of problem

This example concerns a speed regulation system for a subway trains. Here we reproduce the informal specification from [5]. Each train detects beacons that are placed along the track, and receives the "second" from a central clock. Ideally, a train should encounter one beacon each second. So the space left between beacons regulates the speed of the train. Now, a train adjusts the speed as follows: let b and s respectively be the number of encountered beacons and the number of received seconds.

- when $b \geq s + 10$, the train notices it is early and applies the brake as long as $b > s$. Continuously braking causes the train to stop before encountering 10 beacons.
- when $b \leq s - 10$, the train is late, and will be considered late as long as $b < s$. A late train signals it to the central clock, which does not emit the "second" as long as at least one train is late.

The aim of the analysis is to show formally that the absolute difference $|b - s|$ is bounded.

3.2 State diagram

Figure 3 depicts a finite state automaton which models the above specification. The finite state automaton can be implemented as a Prolog program (see figure 4). Briefly, each transition is represented by a separate clause of a single Prolog[2] predicate, p. The arguments of this predicate signify:

- A list of signals (*beacon, second*) received by the automaton.
- A state (*ontime, late, onbrake* or *stopped*).
- Three counters (B, S, D).

3.3 Level 1: internal cycles

In this section we briefly describe the internal cycles of level 1. There are four cycles associated with states ***ontime***, ***late***, ***onbrake*** and ***stopped***. For each cycle, the method of [3] allows us to calculate the relations linking the values of counters B, S, D before and after the application of an arbitrary number of transitions.

[2]For clarity's sake we allow addition and subtraction in the head and body of the predicate definition.

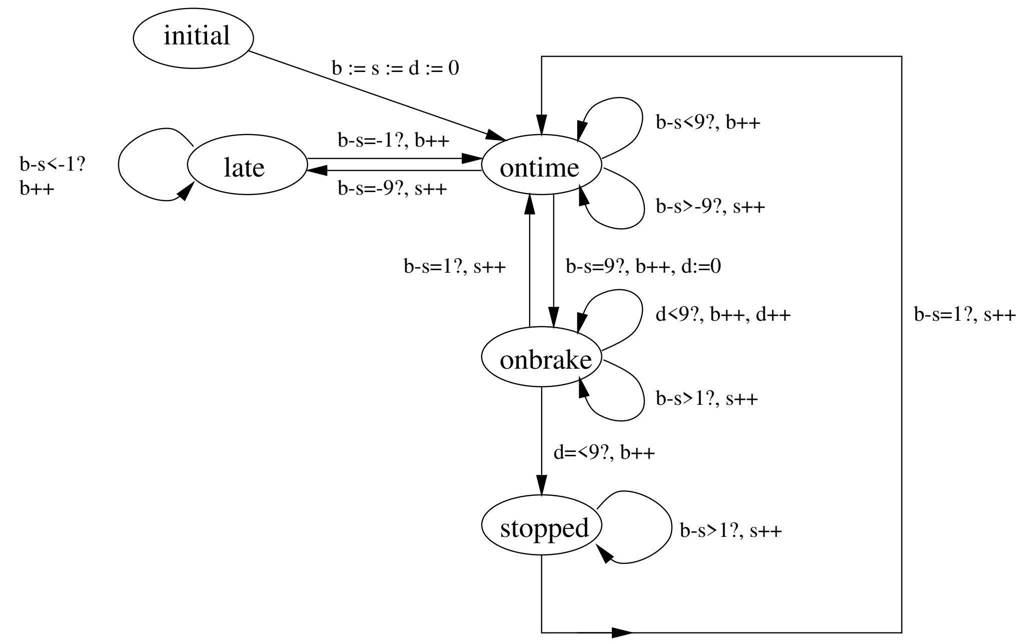


Figure 3: A subway train speed regulation system.

$$p([], ontime, B, S, D) \leftarrow B = 0, S = 0, D = 0. \quad (1)$$
$$p([beacon|L], ontime, B+1, S, D) \leftarrow p(L, ontime, B, S, D), B - S < 9. \quad (2)$$
$$p([second|L], ontime, B, S+1, D) \leftarrow p(L, ontime, B, S, D), B - S > -9. \quad (3)$$
$$p([beacon|L], late, B+1, S, D) \leftarrow p(L, late, B, S, D), B - S < -1. \quad (4)$$
$$p([beacon|L], onbrake, B+1, S, D+1) \leftarrow p(L, onbrake, B, S, D), D < 9. \quad (5)$$
$$p([second|L], onbrake, B, S+1, D) \leftarrow p(L, onbrake, B, S, D), B - S > 1. \quad (6)$$
$$p([second|L], stopped, B, S+1, D) \leftarrow p(L, stopped, B, S, D), B - S > 1. \quad (7)$$
$$p([beacon|L], ontime, B+1, S, D) \leftarrow p(L, late, B, S, D), B - S = -1. \quad (8)$$
$$p([second|L], late, B, S+1, D) \leftarrow p(L, ontime, B, S, D), B - S = -9. \quad (9)$$
$$p([beacon|L], stopped, B+1, S, D) \leftarrow p(L, onbrake, B, S, D), D \leq 9. \quad (10)$$
$$p([beacon|L], onbrake, B+1, S, 0) \leftarrow p(L, ontime, B, S, D), B - S = 9. \quad (11)$$
$$p([second|L], ontime, B, S+1, D) \leftarrow p(L, onbrake, B, S, D), B - S = 1. \quad (12)$$
$$p([second|L], ontime, B, S+1, D) \leftarrow p(L, stopped, B, S, D), B - S = 1. \quad (13)$$

Figure 4: The subway speed regulator automaton encoded as a Prolog program.

Figure 5 shows how the three simple circuits are successively eliminated and replaced by meta-transitions on state ***ontime***.

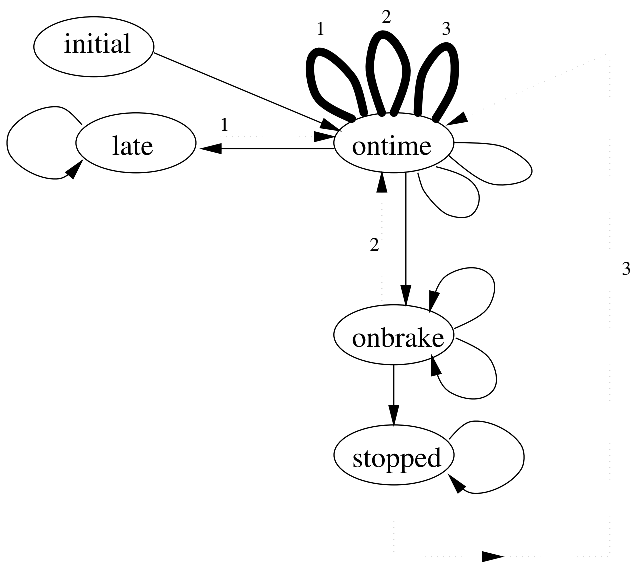


Figure 5: The replacement of simple circuits by meta-transitions. The numbers indicate the order in which simple circuits are replaced by meta-transitions.

3.3.1 state: *late*

There is a single applicable internal transition, (4). After r applications of rule (4), the counters values b', s', d', are linked to the entry counter values b, s, d by the equations:

$$\begin{array}{rcl} b' & = & b + r \\ s' & = & s \\ d' & = & d \end{array}$$

The constraint is: $r \geq 0, b' - s' \leq -1$.

3.3.2 state: *ontime*

There are two internal transitions, (2) and (3). After n applications of rule (2) and m applications of rule (3), we have, using the method of [3], the following equations for the counter values:

$$\begin{array}{rcl} b' & = & b + n \\ s' & = & s + m \\ d' & = & d \end{array}$$

with the constraints $n \geq 0$, $m \geq 0$, $-9 \leq b' - s' \leq 9$.

3.3.3 state: *onbrake*

There are two internal transitions, (5) and (6). After p applications of (5) and q applications of (6), we have, using the method of [3], the following equations for the counter values:

$$\begin{aligned} b' &= b+p \\ s' &= s+q \\ d' &= p \end{aligned}$$

with constraints $p \geq 0, q \geq 0, b'-s' \geq 1, d' \leq 9$.

3.3.4 state: *stopped*

There is a single internal transition given by (7). After l applications of this rule, we have:

$$\begin{aligned} b' &= b \\ s' &= s+l \\ d' &= d \end{aligned}$$

The constraints are $l \geq 0, b'-s' \geq 1$

3.4 Level 2: simple circuits

At level two there are three simple circuits: ***ontime-late-ontime***, ***ontime-onbrake-ontime***, ***ontime-onbrake-stopped-ontime***. We now consider these in turn.

3.4.1 circuit: *ontime-late-ontime*

From §3.3.2, we know that the values of the counters in ***ontime***, after n applications of (2) and m applications of (3), are given by the equations:

$$\begin{aligned} b_1 &= b_0+n \\ s_1 &= s_0+m \\ d_1 &= d_0 \end{aligned}$$

with the constraints $n \geq 0$, $m \geq 0$, $-9 \leq b_1 - s_1 \leq 9$.

Rule (9) specifies the transition from ***ontime*** to ***late***. In order for this rule to be applied, we must have $b_1 - s_1 = -9$, so that $b_0 - s_0 + n - m = -9$, so $m = b_0 - s_0 + n + 9$. After application of (9), we enter ***late*** with:

$$\begin{aligned} b_2 &= b_1 &&= b_0+n \\ s_2 &= s_1+1 &&= b_0+n+10 \\ d_2 &= d_1 &&= d_0 \end{aligned}$$

From §3.3.1, we know that the values of the counters in ***late***, after r applications of (4), are described by the equations:

$$\begin{aligned} b_3 &= b_2+r &&= b_0+n+r \\ s_3 &= b_3 &&= b_0+n+10 \\ d_3 &= d_2 &&= d_0 \end{aligned}$$

Rule (8) allows us to pass back to ***ontime***. Before application of this rule, we have $b_3 - s_3 = -1$, so $r = 9$. After application of this rule, we have $b_4 - s_4 = 0$, $b_4 = b_3 + 1$, $s_4 = s_3$, $d_4 = d_3$. Since we have now arrived back at the entry to ***ontime***, we can write the following recurrence equations describing the values of the counters at the entry to ***ontime***:

$$\begin{aligned} b_4 &= s_4 \\ b_4 &= b_0+n+10 \\ s_4 &= b_0+n+10 \\ d_4 &= d_0 \end{aligned}$$

The sole constraint is that $n \geq 0$. After elimination of n, we find:

$$\begin{array}{rcl} b_4 & = & s_4 \\ b_4 & \geq & b_0 + 10 \\ s_4 & \geq & b_0 + 10 \\ d_4 & = & d_0 \end{array}$$

This can be encoded as the following meta-transition (after renaming b_4, s_4, d_4 as b', s', d' and b_0, s_0, d_0 as b, s, d):

$$p^{entry}_{ontime}(b', s', d') \leftarrow p^{entry}_{ontime}(b, s, d), b' \geq b + 10, s' \geq b + 10, b' = s', d' = d$$

3.4.2 circuit: *ontime-onbrake-ontime*

An analogous analysis yields the following recurrence equations at entry to ***ontime*** for circuit ***ontime-onbrake-ontime***:

$$\begin{array}{rclcl} b_4 & = & s_4 & & \\ b_4 & \geq & b_0 + 10 & & \\ s_4 & \geq & s_0 + 10 & & \\ 9 & \geq & d_4 & \geq & 0 \end{array}$$

This corresponds to the following meta-transition:

$$p^{entry}_{ontime}(b', s', d') \leftarrow p^{entry}_{ontime}(b, s, d), b' \geq b + 10, s' \geq b + 10, b' = s', 0 \leq d' \leq 9.$$

3.4.3 circuit: *ontime-onbrake-stopped-ontime*

Likewise, for circuit ***ontime-onbrake-stopped-ontime***, we get the meta-transition:

$$p^{entry}_{ontime}(b', s', d') \leftarrow p^{entry}_{ontime}(b, s, d), b' \geq b + 11, s' \geq b + 11, b' = s', 0 \leq d' \leq 9.$$

3.5 Level 3: invariant at the entry of *ontime*

Collecting all the meta-transitions of the simple circuits of level 2 and using the fact that the initial values of the counters at the entry of ***ontime*** are all zero, we derive the following program:

$$\begin{array}{l} p^{entry}_{ontime}(b, s, d) \leftarrow b = 0, s = 0, d = 0 \\ p^{entry}_{ontime}(b', s', d') \leftarrow p^{entry}_{ontime}(b, s, d), b' \geq b + 10, s' \geq s + 10, b' = s', d' = d \\ p^{entry}_{ontime}(b', s', d') \leftarrow p^{entry}_{ontime}(b, s, d), b' \geq b + 10, s' \geq s + 10, b' = s', 0 \leq d' \leq 9 \\ p^{entry}_{ontime}(b', s', d') \leftarrow p^{entry}_{ontime}(b, s, d), b' \geq b + 11, s' \geq s + 11, b' = s', 0 \leq d' \leq 9. \end{array}$$

The second recursive clause subsumes the two others. The initial phase of the Revesz's method requires the transformation of the constraints into gap order form, carrying out the following replacements:

1. replace $x < y$ by $x <_0 y$,
2. replace $x > y$ by $y <_0 x$,
3. replace $x >_k y$ by $y <_k x$,
4. replace $x \leq y$ by $x <_0 y \vee x = y$,
5. replace $x \geq y$ by $y <_0 x \vee y = x$,
6. replace $x + k < y$ by $x <_k y$ (as long as $k \geq 0$).

After these replacements, the clauses must be converted to disjunctive normal form.

After applying these transformations, our implementation (§6) of [8] derives the following constraints for B, S, D at the entry of state ***ontime***:

$$b^{entry}_{ontime} = s^{entry}_{ontime} \ \wedge\ d^{entry}_{ontime} = 0 \ \wedge\ 0 <_{19} b^{entry}_{ontime} \ \wedge\ 0 <_{19} s^{entry}_{ontime}$$
$$b^{entry}_{ontime} = s^{entry}_{ontime} \ \wedge\ 0 <_0 d^{entry}_{ontime} <_0 10 \ \wedge\ 0 <_9 b^{entry}_{ontime} \ \wedge\ 0 <_9 s^{entry}_{ontime}$$
$$b^{entry}_{ontime} = 0 \ \wedge\ d^{entry}_{ontime} = 0 \ \wedge\ s^{entry}_{ontime} = 0$$

This can be simplified to:

$$(b^{entry}_{ontime} = 0 \wedge s^{entry}_{ontime} = 0 \wedge d^{entry}_{ontime} = 0)$$
$$\bigvee$$
$$(b^{entry}_{ontime} \geq 10 \wedge s^{entry}_{ontime} \geq 10 \wedge b^{entry}_{ontime} = s^{entry}_{ontime} \wedge 0 \leq d^{entry}_{ontime} \leq 9)$$

3.6 Level 4: general invariants

From the above entry values, it is possible to derive the general values $b_{ontime}, s_{ontime}, d_{ontime}$ of counters B, S, D in state ***ontime*** by applying the transitions of the internal cycle of ***ontime***. So, after n transitions (2) and m transitions (3), we get:

$b_{ontime} = b^{entry}_{ontime} + n$, $s_{ontime} = s^{entry}_{ontime} + m$, $d_{ontime} = d^{entry}_{ontime}$,
with $m \geq 0, n \geq 0, -9 \leq b_{ontime} - s_{ontime} \leq 9$.
After elimination of m and n (using some standard variable elimination techniques), we get:
$b_{ontime} \geq 0 \wedge s_{ontime} \geq 0 \wedge -9 \leq b_{ontime} - s_{ontime} \leq 9 \wedge 0 \leq d_{ontime} \leq 9$
From these values, we can derive in turn the general values $b_{late}, s_{late}, d_{late}$ of the counters on ***late*** by applying transition (9) and composing with the internal cycle of ***late***. After simplification, this gives:
$b_{late} \geq 0 \wedge s_{late} \geq 10 \wedge -10 \leq b_{late} - s_{late} \leq -1 \wedge 0 \leq d_{late} \leq 9$
Likewise for ***onbrake*** and ***stopped***, we get:
$b_{onbrake} \geq d_{onbrake} + 10 \wedge s_{onbrake} \geq 0 \wedge 1 \leq b_{onbrake} - s_{onbrake} \leq 19 \wedge 0 \leq d_{onbrake} \leq 9$
$b_{stopped} \geq d_{stopped} + 10 \wedge s_{stopped} \geq 0 \wedge 1 \leq b_{stopped} - s_{stopped} \leq 19 \wedge 0 \leq d_{stopped} \leq 9$
It follows immediately from these relations, that the values b, s of counters B, S always satisfy: $-9 \leq b - s < 19$. This shows that the absolute difference $|B - S|$ is bounded, as required.

4 Another example: A speed-limited car

4.1 Introduction

This example is also taken from [5]. It describes a car which has a speed regulator. The aim of the study is to show that a certain event "bump" can never occur. This example shows that the guards of the derived meta-transitions may not be gap-order constraints and must sometimes be replaced by upper approximations which are in gap-order form (see §4.5).

The finite state automaton is depicted in figure 6.

4.2 Preparation of automaton

In order to permit analysis by the technique of [3], several transformations are applied to the automaton.

1. Our definition of internal cycles forbids direct assignments, so a simple transformation as applied to cut the state ***q2*** into two states ***q2.0*** and ***q2.1***, which are free of direct assignments and so can be analysed by [3]. Such a transformation can always be made to eliminate assignments.

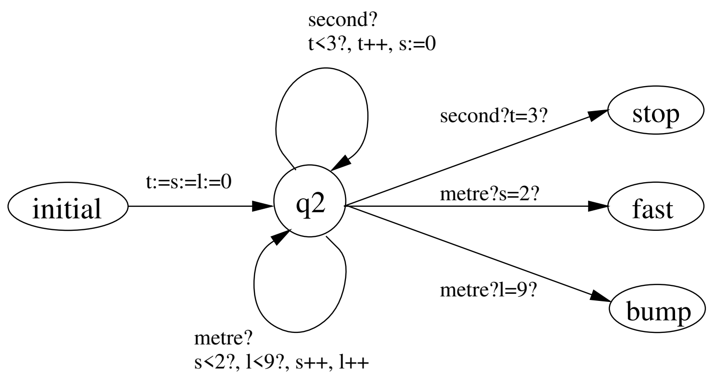


Figure 6: The car automaton.

2. Only single conditions are allowed in internal cycles. Therefore, we must drop some of the conditions when there is more than one. The resulting invariant will be therefore only be an upper approximation. Multiple conditions *are* allowed in the transitions between internal cycles, so we retain them there.

The transformed automaton is shown in figure 7, where, for the sake of conciseness, transitions to the final states have been omitted (both ***q2.0*** and ***q2.1*** take a copy of the three transitions out of ***q2***, making a total of six transitions to the final states).

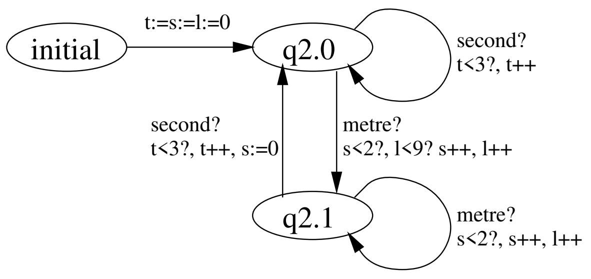


Figure 7: The modified segment of the car automaton.

The Prolog program which models the automaton is listed in figure 8.

4.3 Level 1: internal cycles

This subsection gives a step-by-step analysis of the automaton at level 1.

$$p([], q2.0, S, T, L) \leftarrow S = 0, T = 0, L = 0. \tag{14}$$
$$p([metre|X], q2.1, S+1, T, L+1) \leftarrow p(X, q2.1, S, T, L), S < 2. \tag{15}$$
$$p([second|X], q2.0, 0, T+1, L) \leftarrow p(X, q2.1, S, T, L), T < 3. \tag{16}$$
$$p([second|X], q2.0, S, T+1, L) \leftarrow p(X, q2.0, S, T, L), T < 3. \tag{17}$$
$$p([metre|X], q2.1, S+1, T, L+1) \leftarrow p(X, q2.0, S, T, L), S < 2. \tag{18}$$

Figure 8: Clauses defining the car automaton.

4.3.1 state: *q2.0*

After n applications of rule (17), the counters values s', t', l' are linked to the entry counter values s, t, l by the equations:

$$\begin{array}{rcl} s' & = & s \\ t' & = & t + n \\ l' & = & l \end{array}$$

with $t' \leq 3$, so $0 \leq n \leq 3 - t$.

4.3.2 state: *q2.1*

After m applications of rule (15), the counters values s', t', l' are linked to the entry counter values s, t, l by the equations:

$$\begin{array}{rcl} s' & = & s + m \\ t' & = & t \\ l' & = & l + m \end{array}$$

with $s' \leq 2$, so $0 \leq m \leq 2 - s$.

4.4 Level 2: circuit *q2.0-q2.1-q2.0*

This subsection gives a step-by-step analysis of the automaton for the unique circuit of level 2.

Start from state ***q2.0*** with values s_0, t_0, l_0 for counters S, T, L. From §4.3.1, we have, after n applications of rule (17):

$$\begin{array}{rcl} s_1 & = & s_0 \\ t_1 & = & t_0 + n \\ l_1 & = & l_0 \end{array}$$

with the constraint: $0 \leq n \leq 3 - t_0$.

Pass to ***q2.1*** using rule (18). In order to allow application of this rule, we must have $s_1 < 2$. Therefore, after application, we have:

$$\begin{array}{rcccl} s_2 & = & s_1 + 1 & = & s_0 + 1 \\ t_2 & = & t_1 & = & t_0 + n \\ l_2 & = & l_1 + 1 & = & l_0 + 1 \end{array}$$

Merging the constraint $s_1 < 2$ (i.e. $s_0 < 2$) for this transition with those before we have: $0 \leq n \leq 3 - t_0 \wedge s_0 \leq 1$.

Now loop m times in ***q2.1***, applying rule (15). From §4.3.2 we know that in ***q2.1*** we have:

$$\begin{array}{rcccl} s_3 & = & s_2 + m & = & s_0 + m + 1 \\ t_3 & = & t_2 & = & t_0 + n \\ l_3 & = & l_2 + m & = & l_0 + m + 1 \end{array}$$

with: $s_3 \leq 2$, so $0 \leq m \leq 1 - s_0$. Merging the latter constraints with those before, we have: $0 \leq n \leq 3 - t_0 \wedge 0 \leq \ m \leq 1 - s_0$.

Pass back to ***q2.0*** using rule (16). Before application, we must have $t_3 < 3$, so $0 \leq n < 3 - t_0$. After application, have:

$$\begin{array}{lllll} s_4 & = & 0 & & \\ t_4 & = & t_3 + 1 & = & t_0 + n + 1 \\ l_4 & = & l_3 & = & l_0 + m + 1 \end{array}$$

This gives us the following recurrence relations for the circuit ***q2.0-q2.1-q2.0*** (after renaming s_4, t_4, l_4 as s', t', l' and s_0, t_0, l_0 as s, t, l):

$$\begin{array}{lll} s' & = & 0 \\ t' & = & t + n + 1 \\ l' & = & l + m + 1 \end{array}$$

with : $0 \leq n < 3 - t \wedge 0 \leq m \leq 1 - s$. After elimination of the variables n, m and s we derive the following inequalities:

$$\begin{array}{lllll} s' & & = & 0 & \\ t + 1 & \leq & t' & < & 4 \\ l + 1 & \leq & l' & \leq & l + 2 \end{array}$$

Using the fact that initially, the values of s, t, l are zero, we can represent the evolution of the counters through a general path starting and ending at ***q2.0*** by the following meta-program:

$$\begin{array}{lll} p_{q2.0}^{entry}(S, T, L) & \leftarrow & S = 0, T = 0, L = 0 \\ p_{q2.0}^{entry}(S', T', L') & \leftarrow & p_{q2.0}^{entry}(S, T, L), S' = 0, T < T' < 4, L < L' < L + 3 \end{array}$$

4.5 Level 3: invariant at the entry of *q2.0*

The guard of the recursive meta-transition, $S' = 0, T < T' < 4, L < L' < L + 3$ is not a gap-order constraint. The problem comes from the inequality $L' < L + 3$. i.e. $L' - 2 \leq L$, which is not allowed. (Only inequalities of the form $L' + c \leq L$ with $c \geq 0$ are allowed.) A solution consists of eliminating the constant 2 in $L' - 2 \leq L$, by converting it into a new existentially quantified variable Z, then eliminating Z.

The constraint $T < T' \wedge L' < L + 3$, i.e., $T' - T \geq 1 \wedge 2 \geq L' - L$ is thus rewritten as $\exists Z (T' - T \geq Z \wedge 2Z \geq L' - L)$, i.e. $\exists Z (2T' - 2T \geq 2Z \geq L' - L)$. Elimination of Z gives: $2T' - 2T \geq L' - L$, i.e.: $2T' - L' \geq 2T - L$. Therefore the constraint $T < T' \wedge L' < L + 3$ can be approximated by $T < T' \wedge U' \geq U$, where U (resp. U') is a new variable standing for $2T - L$ (resp. $2T' - L'$).

The former meta-program is thus approximated as:

$$\begin{array}{lll} p'^{\,entry}_{q2.0}(S, T, U) & \leftarrow & S = 0, T = 0, U = 0 \\ p'^{\,entry}_{q2.0}(S', T', U') & \leftarrow & p'^{\,entry}_{q2.0}(S, T, U), S' = 0, T < T' < 4, U \leq U'. \end{array}$$

The program is evaluated bottom-up using Revesz's method. This gives the following solutions for the values of S, T, U at the entry on state ***q2.0***:

$$\begin{array}{c} s_{q2.0}^{entry} = 0 \wedge t_{q2.0}^{entry} = 0 \wedge u_{q2.0}^{entry} = 0 \\ \bigvee \\ s_{q2.0}^{entry} = 0 \wedge 0 < t_{q2.0}^{entry} < 4 \wedge 0 < u_{q2.0}^{entry} \end{array}$$

Using the fact that $u_{q2.0}^{entry} = 2t_{q2.0}^{entry} - l_{q2.0}^{entry}$ we can simplify this to:

$$(s_{q2.0}^{entry} = 0 \wedge t_{q2.0}^{entry} = 0 \wedge l_{q2.0}^{entry} = 0) \ \bigvee \ (s_{q2.0}^{entry} = 0 \wedge 1 \leq t_{q2.0}^{entry} \leq 3 \wedge l_{q2.0}^{entry} < 2t_{q2.0}^{entry})$$

4.6 Level 4: general invariants

From the latter solutions, we derive the general values $s_{q2.0}, t_{q2.0}, l_{q2.0}$ of S, T, L on state ***q2.0*** by applying the transitions of the internal cycle for ***q2.0***. Thus, after n transitions (17), we have:

$$s_{q2.0} = s_{q2.0}^{entry},\ t_{q2.0} = t_{q2.0}^{entry} + n,\ l_{q2.0} = l_{q2.0}^{entry},\ \text{with } 0 \leq t_{q2.0} \leq 3$$

After simplification, this gives $s_{q2.0} = 0 \wedge 0 \leq t_{q2.0} \leq 3 \wedge l_{q2.0} < 6$.

From the latter relations, the values of counters S, L, T at the entry of ***q2.1*** can be derived by applying rule (18). This gives:

$$s_{q2.1}^{entry} = 1 \wedge 0 \leq t_{q2.1} \leq 3 \wedge l_{q2.1} < 7$$

Then, composing with the internal cycle for ***q2.1*** (i.e., applying rule (15) m times) gives the following general values of the counters on state ***q2.1***:

$$s_{q2.1} = s_{q2.1}^{entry} + m,\ t_{q2.1} = t_{q2.1}^{entry},\ l_{q2.1} = l_{q2.1}^{entry} + m,\ \text{with } 0 \leq s_{q2.1} \leq 2$$

Thus, we have $s_{q2.1} = 1 + m \wedge 0 \leq t_{q2.1} \leq 3 \wedge l_{q2.1} < 7 + m \wedge 0 \leq m \leq 1$, i.e.

$$\begin{array}{ll} & (s_{q2.1} = 1 \wedge 0 \leq t_{q2.1} \leq 3 \wedge l_{q2.1} < 7) \\ \vee & (s_{q2.1} = 2 \wedge 0 \leq t_{q2.1} \leq 3 \wedge l_{q2.1} < 8) \end{array}$$

It is now straightforward to check that the general values $l_{q2.0}$ and $l_{q2.1}$ of counter L in states ***q2.0*** and ***q2.1*** are always smaller than 9. This shows that the transition "bump" (which is guarded by the constraint $L = 9$, see [5]) will never occur, as required.

5 A larger example

5.1 Introduction

In this section we outline the analysis of an automaton describing a set of readers and writers, an example from [7].

5.2 The automaton

This 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 then formed by synchronising the readers and the writers with the control process, which can then be seen as a single automaton. Unfortunately, this quickly results in a large number of states. In order to treat this example with our (only partly-automated) methods, it was necessary to simplify 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.

The rules for the automaton are listed in figure 9.

The automaton is displayed in figure 10. In order to permit analysis by [3], state ***resource*** (left of diagram) must be split into two states, neither of which contains an assignment.

5.3 Analysis

The automaton contains two internal cycles (level 1): ***resource**** and ***res1****. There are a total of 9 simple circuits (level 2):

1. ***resource****

$$
\begin{aligned}
&p([], resource, 0, 0, 0). && && (19)\\
&p([e\_write|T], resource, R, D, 0) &:-\ & p(T, resource, R, D, W). && (20)\\
&p([e\_read|T], resource, R-1, D, W) &:-\ & p(T, resource, R, D, W). && (21)\\
&p([write\_demand|T], resource, R, D+1, 0) &:-\ & p(T, resource, R, D, W). && (22)\\
&p(T, not\_writing, R, D, 0) &:-\ & p(T, resource, R, D, 0). && (23)\\
&p([s\_read|T], resource, R+1, 0, W) &:-\ & p(T, not\_writing, R, 0, W). && (24)\\
&p(T, demands, R, D, W) &:-\ & p(T, not\_writing, R, D, W), D > 0 && (25)\\
&p([s\_write|T], resource, 0, D-1, 1) &:-\ & p(T, demands, 0, D, W). && (26)
\end{aligned}
$$

Figure 9: The code for the read/write automaton.

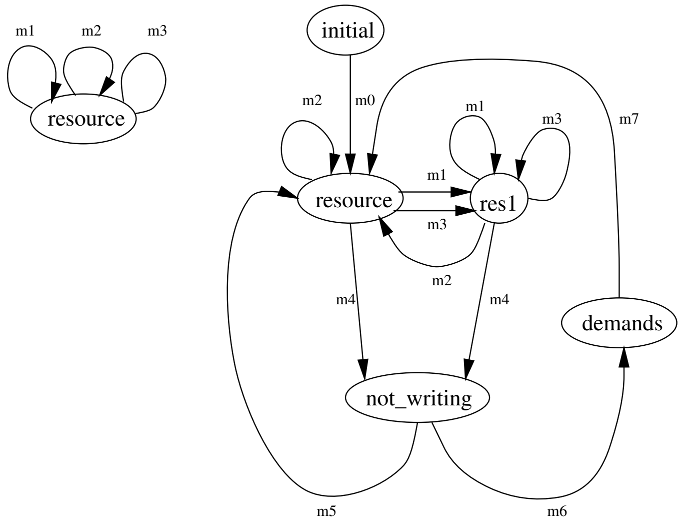


Figure 10: State ***resource*** in the original rules is shown on the left of the diagram. The automaton, modified to split this into two, is shown on the right. Rules m0 – m7 are rules 19 – 26 in the text.

2. ***resource****-(20)-***res1****-***resource****
3. ***resource****-(22)-***res1****-***resource****
4. ***resource****-***not_writing-demands-resource****
5. ***resource****-***not_writing-resource****
6. ***resource****-(20)-***res1****-***not_writing-demands-resource****
7. ***resource****-(22)-***res1****-***not_writing-demands-resource****
8. ***resource****-(20)-***res1****-***not_writing-resource****
9. ***resource****-(22)-***res1****-***not_writing-resource****

Note that the two transitions (given by rules 20 and 22) from ***resource*** to ***res1*** each give rise to separate circuits. Also note that in general, internal circuits are not replaced by meta-transitions because the translation to gap-order form is likely to lose too much accuracy.

The last circuit (number 9 above) does not give rise to a meta-transition because it is not possible. Using the techniques described in this paper, we derive the following meta-transitions for each of these circuits (in the same order as above):

$$\begin{array}{lcl}
p^{entry}_{resource}(r', d', w') & \leftarrow & p^{entry}_{resource}(r, d, w), 0 \leq r', r' \leq r, w' = w, d' = d, r \leq rmax, d \leq dmax \\
p^{entry}_{resource}(r', d', w') & \leftarrow & p^{entry}_{resource}(r, d, w), 0 \leq r', r' < r, d \leq d', d' \leq d_{max}, w' = 0 \\
p^{entry}_{resource}(r', d', w') & \leftarrow & p^{entry}_{resource}(r, d, w), 0 \leq r', r' < r, d < d', d' \leq d_{max}, r \leq rmaxw' = 0 \\
p^{entry}_{resource}(r', d', w') & \leftarrow & p^{entry}_{resource}(d, r, w), r' = 0, 0 \leq d' = d - 1 < d_{max}\dagger, w' = 1, r \leq rmax \\
p^{entry}_{resource}(r', d', w') & \leftarrow & p^{entry}_{resource}(r, d, w), 0 < r', r' \leq r + 1\dagger, r' \leq r_{max}, d' = 0, d \leq dmax, w' = 0 \\
p^{entry}_{resource}(r', d', w') & \leftarrow & p^{entry}_{resource}(d, r, w), r' = 0, 0 \leq d', d' < d_{max}, d' \geq d - 1\dagger, r \leq rmax, w' = 1 \\
p^{entry}_{resource}(r', d', w') & \leftarrow & p^{entry}_{resource}(d, r, w), r' = 0, 0 \leq d', d' < d_{max}, r \leq rmax, w' = 1 \\
p^{entry}_{resource}(r', d', w') & \leftarrow & p^{entry}_{resource}(r, d, w), 0 < r', r' \leq r + 1\dagger, r' \leq r_{max}, \\
 & & \qquad r \leq rmax, d \leq dmax, d' = 0, w' = 0
\end{array}$$

The three constraints marked with a $\dagger$ cannot be translated to gap-order form. These must be replaced by upper approximations:

1. $r' \leq r + 1$. This is approximated by $r' \leq r_{max}$.
2. $d' = d - 1$. This is approximated by $d' < d$.
3. $d' \geq d - 1$. This is equivalent to $d' \geq d \;\vee\; d' = d - 1$, which in view of the above approximation, is replaced by $d' \geq d \;\vee\; d' < d$, i.e. it is deleted.

Making these approximations, converting to gap-order form, setting $d_{max} = 5$ and $r_{max} = 3$, and solving automatically gives:

$$\begin{array}{lcl}
p^{entry}_{resource}(r, d, w) & \leftarrow & d = 0 \;\wedge\; w = 0 \;\wedge\; 0 <_0 r <_0 3 \\
p^{entry}_{resource}(r, d, w) & \leftarrow & d = 0 \;\wedge\; r = 3 \;\wedge\; w = 0 \\
p^{entry}_{resource}(r, d, w) & \leftarrow & w = 0 \;\wedge\; 0 <_0 r <_2 5 \;\wedge\; d <_0 5 \\
p^{entry}_{resource}(r, d, w) & \leftarrow & r = 0 \;\wedge\; w = 0 \;\wedge\; d <_0 5 \\
p^{entry}_{resource}(r, d, w) & \leftarrow & d = 5 \;\wedge\; w = 0 \;\wedge\; 0 <_0 r <_2 5 \\
p^{entry}_{resource}(r, d, w) & \leftarrow & d = 5 \;\wedge\; r = 0 \;\wedge\; w = 0 \\
p^{entry}_{resource}(r, d, w) & \leftarrow & w = 0 \;\wedge\; 0 <_0 d <_0 5 \;\wedge\; 0 <_0 r <_2 5
\end{array}$$

$$\begin{array}{lcl}
p^{entry}_{resource}(r,d,w) & \leftarrow & r = 0 \wedge w = 0 \wedge 0 <_0 d <_0 5 \\
p^{entry}_{resource}(r,d,w) & \leftarrow & d = 0 \wedge w = 0 \wedge 0 <_0 r <_2 5 \\
p^{entry}_{resource}(r,d,w) & \leftarrow & d = 0 \wedge r = 0 \wedge w = 1 \\
p^{entry}_{resource}(r,d,w) & \leftarrow & r = 0 \wedge w = 1 \wedge 0 <_0 d <_0 5 \\
p^{entry}_{resource}(r,d,w) & \leftarrow & d = 0 \wedge r = 0 \wedge w = 0
\end{array}$$

These equations can be simplified to:

$$\begin{array}{lcl}
p^{entry}_{resource}(r,d,w) & \leftarrow & w = 1 \wedge r = 0 \wedge 0 \le d < 5 \\
p^{entry}_{resource}(r,d,w) & \leftarrow & w = 0 \wedge 0 \le r < 3 \wedge 0 \le d < 5 \\
p^{entry}_{resource}(r,d,w) & \leftarrow & w = 0 \wedge r = 0 \wedge d = 5 \\
p^{entry}_{resource}(r,d,w) & \leftarrow & w = 0 \wedge r = 3 \wedge d = 0
\end{array}$$

Now propagate these values at entry to ***resource*** through the automaton:
In ***resource***, we can repeatedly apply m2, but this does not change the invariant, giving:

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

Transferring to ***res1*** using either m1 or m3, then repeatedly applying m1 and m3 gives:

$$p_{res1} \leftarrow w = 0 \wedge 0 \le r \le 3 \wedge 0 \le d \le 5$$

The invariant in ***not_writing*** is the disjunction of the invariants formed by transferring from ***resource*** using m4 and from ***res1*** using m4. This ensures that $w = 0$, giving:

$$p_{not\_writing} \leftarrow w = 0 \wedge 0 \le r \le 3 \wedge 0 \le d \le 5$$

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

$$p_{demands} \leftarrow w = 0 \wedge 0 \le r \le 3 \wedge 0 < d \le 5$$

From these equations, we can readily verify the following, as required:

1. $w = 1 \rightarrow r = 0$.
2. $d + w \le d_{max}$.

5.4 Discussion

The analysis of the previous section required three 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.

From this discussion we see that the approximations we have been forced to make do not affect the final invariant. Therefore, this invariant is exact.

6 Implementation

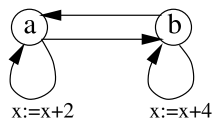
Currently our implementation consists of Prolog predicates for the translation of meta-transitions into gap-order form, and their subsequent solution according to the method of Revesz. Note that the user must make any approximations necessary to put the constraints into gap-order form.

Appendix A provides some brief notes on the implementation. Appendix B lists the main code for bottom-up evaluation of gap-order constraints.

7 A comparison with previous approaches

Both [5, 2] 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. This exploits the efficient algorithms that are known for manipulating convex polyhedra. He also makes use of the operation of *widening* in order to avoid the generation of an infinite sequence of increasing sets of points.

In [2], 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, $x' \geq x + 2$. 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. [3] 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 [2, 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).

In this paper, we have taken the approach first suggested in [3]. Thus we have replaced at the inner level (internal cycles), the computation of approximated invariants by the calculation of exact least fixed points. We have then gone much further by approximating such least fixed points as gap-order constraints, thus allowing for the use of another exact least fixed point procedure: Revesz's one. On the other hand, the method raises the new issue of approximating linear arithmetic formulae by gap-order constraints. A heuristic based on the introduction of new existential variables has been indicated in §4.5, but this issue deserves a more systematic investigation.

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

8 Further work

Most of the examples given in [5, 7, 2] have been successfully treated by hand with our method. An implementation of our method should have much in common with existing work such as [2]. The method for calculating least fixed points for internal cycles has already been systematically described in [3]. The algorithm for the bottom-up evaluation of the resulting gap-order constraints has already been given by [8]. Since the guards of the meta-transitions representing invariants of simple circuits are not always in the form of gap-order constraints, some approximation is inevitable. This should be studied further, as previously pointed out.

9 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 [3, 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.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of 5th IEEE LICS*, pages 414–425. IEEE, 1990.

[2] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proceedings of conference on Computer-Aided Verification, 1994*, pages 55–67, 1994.

[3] 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.

[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] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings 7th LICS Symposium, Santa Cruz*, pages 394 – 406, 1992.

[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.

A Implementation notes

The Prolog code which implements the bottom-up evaluation of the meta-transitions is a fairly direct and declarative implementation of the algorithm described in [8], with some efficiency improvements. Meta-transitions are represented as terms of the form `Head <-- Body, Constraints`, where `Head` is a single literal, `Body` is a list of literals (always a singleton list for meta-transitions), and `Constraints` is a list, representing a conjunction, of constraints, for example:

```
p(r1,d1,w1)<-- [p(r,d,w)],[lt(0,r1), leq(r1,r), eq(w1,w),eq(d1,d),leq(r,3),leq(d,4)]
```

The constraints need to be translated into true gap-order form by for example rewriting $x \leq y$ as $x < y \vee x = y$ and then putting the resulting constraints into disjunctive normal form. This is achieved using the predicate **gap_graph_form(Rule, GapRules)**, e.g.

```
| ?- gap_graph_form((p(r1,d1,w1)<-- [p(r,d,w)],[lt(0,r1), leq(r1,r), eq(w1,w),eq(d1,d),
                                  leq(r,3),leq(d,4)]), Rs).

Rs = [(p(r1,d1,w1)<--[p(r,d,w)],[lt(0,0,r1),lt(0,r1,r),eq(w1,w),eq(d1,d),lt(0,r,3),lt(0,d,4)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[lt(0,0,r1),lt(0,r1,r),eq(w1,w),eq(d1,d),lt(0,r,3),eq(d,4)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[lt(0,0,r1),lt(0,r1,r),eq(w1,w),eq(d1,d),eq(r,3),lt(0,d,4)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[lt(0,0,r1),lt(0,r1,r),eq(w1,w),eq(d1,d),eq(r,3),eq(d,4)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[lt(0,0,r1),eq(r1,r),eq(w1,w),eq(d1,d),lt(0,r,3),lt(0,d,4)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[lt(0,0,r1),eq(r1,r),eq(w1,w),eq(d1,d),lt(0,r,3),eq(d,4)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[lt(0,0,r1),eq(r1,r),eq(w1,w),eq(d1,d),eq(r,3),lt(0,d,4)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[lt(0,0,r1),eq(r1,r),eq(w1,w),eq(d1,d),eq(r,3),eq(d,4)])] ?

yes
```

A list *Rules* of meta-transitions in gap-order form is evaluated bottom-up. The example below (from the analysis of the reader-writer example of §5) shows the form of the predicate call, and the form of the output, which is a list (disjunction) of constraints on the counter values. The initial rules expand to 51 gap graphs. Bottom up evaluation takes about a minute on a Sun SPARCstation-10.

```
?- gap_graph_form(
     [(p(r1,d1,w1)<--[p(r,d,w)],[lt(0,r1),leq(r1,r),eq(w1,w),eq(d1,d),leq(r,3),leq(d,5)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[leq(0,r1),lt(r1,r),leq(d,d1),leq(d1,5),eq(w1,0)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[leq(0,r1),lt(r1,r),lt(d,d1),leq(d1,5),eq(w1,0),leq(r,3)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[eq(r1,0),leq(0,d1),lt(d1,5),leq(1,d),eq(w1,1),leq(r,3)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[lt(0,r1),leq(r1,3),eq(d1,0),eq(w1,0),leq(d,5)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[eq(r1,0),leq(0,d1),lt(d1,5),eq(w1,1),leq(r,3)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[eq(r1,0),leq(0,d1),lt(d1,5),eq(w1,1),leq(r,3)]),
      (p(r1,d1,w1)<--[p(r,d,w)],[lt(0,r1),leq(r1,3),eq(d1,0),eq(w1,0),leq(d,5),leq(r,3)])],
     Rules),
   eval(Rules, Database, [], Solution).
```

... diagnositic output ...

```
Solution =
            [p(r,d,w)=[lt(0,0,r),eq(d,0),lt(0,r,3),eq(w,0)],
             p(r,d,w)=[eq(w,0),eq(d,0),eq(r,3)],
             p(r,d,w)=[lt(0,0,r),lt(0,d,5),lt(2,r,5),eq(w,0)],
             p(r,d,w)=[eq(w,0),lt(0,d,5),eq(0,r)],
             p(r,d,w)=[lt(0,0,r),eq(d,5),lt(2,r,5),eq(w,0)],
             p(r,d,w)=[eq(w,0),eq(d,5),eq(0,r)],
             p(r,d,w)=[lt(0,0,r),lt(0,d,5),lt(0,0,d),lt(2,r,5),eq(w,0)],
             p(r,d,w)=[eq(w,0),lt(0,d,5),eq(0,r),lt(0,0,d)],
             p(r,d,w)=[eq(w,0),eq(d,0),lt(2,r,5),lt(0,0,r)],
             p(r,d,w)=[eq(w,1),eq(0,d),eq(r,0)],
             p(r,d,w)=[eq(w,1),lt(0,d,5),lt(0,0,d),eq(r,0)],
             p(r,d,w)=[eq(r,0),eq(d,0),eq(w,0)]] ?

yes
```

The predicate **latex_pprint(Solution)** can be used to print a solution such as that above in a suitable form for input to LaTeX.

B Code for bottom-up evaluation of gap-order constraints

This section lists the main code for the bottom-up evaluation of gap-order constraints.

```
/*
  gapgraphs.pl          25 January 1996    First working version 0.1.
                        5 February 1996    Optimised merge.

  Julian Richardson

  Predicates for manipulating gap graphs. Eval/4 takes a set of datalog
  rules with gap order constraints and evaluates them bottom-up,
  returning a solution in gap-order form.

  The representation of a gap graph is simple - just a list (conjunction)
  of the relations lt(n,x,y) eq(x,y). lt(n,x,y) is taken to mean that
  x+n<y. Nodes are named by atoms which are not constants. Constants are
  integers or infinity or -infinity.

  This code implements the method described in [Revesz]: Revesz, P.Z.,
  "A closed-form evaluation for Datalog queries with integer
  (gap-)order constraints."

  The code was developed under sicstus2.1 (library(lists) is loaded
  for the setof predicate) and is intended to be declarative rather
  than efficient.

*/
?- use_module(library(lists)).
?- op(900, yfx, '<--').

%
% eval(+Rules, +Database, +InResult, -OutResult)
%
% Evaluate a set of rules with gap order constraints by bottom-up
% evaluation as described in [Revesz].
%
% Database provides initial values for the variables.
% Rules are of the form:
%     head(vars) <-- Body, Constraint
%
% Body is a list of literals, Constraint is a gap order constraint.
% -OutResult is a list representing a disjunction of gap order constraints.
%
eval(Rules, Database, InResult, OutResult) :-
        nmember(RuleNum,(P <-- Plist, C),Rules),
        evalrule((P <-- Plist, C), Database, InResult, NGraph),
        consistent(NGraph),
        verbose('Rule ',RuleNum, ' successfully applied.'),
        (true; print('...new graph subsumed and so discarded.'),nl,fail),
        rename_vars(P=NGraph, InResult, [P1=NGraph2 | NewResult]),
        \+ member(P1=NGraph2, InResult),
        \+ ((member(P1=Graph, InResult), subsumes(NGraph2, Graph))),
        length([P1=NGraph2 | NewResult],N),
        verbose('Database contains ',N,' gap graphs.'),
        eval(Rules, Database, [P1=NGraph2 | NewResult], OutResult).

eval(_Rules, _Database, FinalDB, FinalDB).
```

```
verbose(A,B,C) :- print(A),print(B),print(C),nl.

%
% simplify(+Graph, -SimpleGraph)
%
% Since we know that the graph is consistent, we can remove some trivial
% relations.
%

%
% Remove trivial inequalities
%
simplify(Graph, SGraph) :-
        delmember(lt(K1,K2,K3), Graph, Graph1),
        is_constant(K1), is_constant(K2), is_constant(K3),!,
        simplify(Graph1, SGraph).

%
% Remove trivial equalities
%
simplify(Graph, SGraph) :-
        delmember(eq(X,X), Graph, Graph1),!,
        simplify(Graph1, SGraph).

%
% Remove inequalities x <_k c   or   c <_k x   when there is also an
% equality x = c', for constants c,k,c'.
%
simplify(Graph, SGraph) :-
        delmember(lt(K1,A,B), Graph, Graph1),
        (is_constant(A), \+is_constant(B),
            (member(eq(B,K2), Graph1),is_constant(K2) ;
             member(eq(K2,B), Graph1),is_constant(K2)),!,
         simplify(Graph1, SGraph);
        (is_constant(B), \+is_constant(A),
            (member(eq(A,K2), Graph1),is_constant(K2) ;
             member(eq(K2,A), Graph1),is_constant(K2)),!,
         simplify(Graph1, SGraph))).

simplify(Graph, Graph).

%
% Attempt to apply a rule - first pick gap graphs for the terms in the
% body of the rule, then merge them together and with the gap
% constraint C. Finally, shortcut out the existentially quantified
% variables.
%
evalrule((P <-- Plist, C), Database, InResult, NGraph) :-
        merge_body(Plist, C, InResult, NGraph2),
        existential_vars((P <-- Plist,C), EVars),
        shortcuts(NGraph2, EVars, NGraph),!.

%
% merge_body(+Literals, +Constraint, +InResult, -OutResult)
%
% Merge the gap graphs which are associated with the literals in the body
% of a clause.
%
```

```
% Database is a gap graph which acts as the starting point for ascending
% evaluation.
%
merge_body([], X, _, X).

merge_body([P1 | Ps], Constraint, InResult, NGraph) :-
        member((P1=Gap1), InResult),
        merge_body(Ps, Constraint, InResult, NGraph1),
        merge(Gap1, NGraph1, NGraph).

%
% Find the existentially quantified variables in the rule so we can
% eliminate them later.
%
existential_vars((P <-- Plist,_C),EVars) :-
        vars([P], UVars),
        vars(Plist, UEVars),
        ssetof(Var,(member(Var,UEVars),\+member(Var,UVars)),EVars).

%
% consistent(+Graph) succeeds iff the graph is consistent.
%
consistent(Graph) :-
        compact(Graph, CGraph),
        \+cyclic(CGraph),
        \+constants_too_close(CGraph,_,_),
        respects_infinity(CGraph).

%
% compact(+Graph, -CGraph)
% CGraph is the compact version of Graph when all relations eq(X,Y)
% are eliminated by merging X and Y to X. All
% the relations involving X or Y are transferred to the new node X.
%
% If one is a constant, then make that the new node. If they are both
% constants, then fail if they are unequal.
%
compact(Graph, CGraph) :-
        delmember(eq(X,Y),Graph,Graph1),!,
        \+ ((is_constant(X), is_constant(Y), \+ X=Y)),
        (is_constant(Y), A=Y, B=X ;
         \+is_constant(Y), A=X, B=Y
         ),
        replace_all([A/B],Graph1,Graph2),!,
        compact(Graph2,CGraph).

compact(CGraph, CGraph).

%
% shortcut(+Graph, +NodeToBeDeleted, -ShortGraph)
%
% Remove a node from the graph.
%
shortcut(Graph, Y, ShortGraph) :-
        add_edges(Graph, Y, ShortGraph1),
        remove_subsumed_lt(ShortGraph1, ShortGraph2),
        \+double_edge(ShortGraph2, _Edge1, _Edge2),
        remove_node(ShortGraph2, Y, ShortGraph),!.
```

```
%
% Remove a list of nodes from the graph
%
shortcuts(Ingraph, [Var | Vars], Outgraph) :-
        shortcut(Ingraph, Var, Outgraph1),
        shortcuts(Outgraph1, Vars, Outgraph).

shortcuts(Graph, [], Graph).

%
% Merge two gap graphs. Each of the input graphs must use the same two
% constants l,u.
%
% For efficiency reasons, the merging of graphs has been separated
% into two distinct phases:
% 1) The symmetric operations for which sop(g1,g2)=sop(g2,g1) so we only
%    need to do one, and
% 2) The asymmetric operations for which aop(g1,g2) =/= aop(g2,g1) so
%    we need to do both.
%
% This results in a huge improvement over the previous code.
%
merge(G1,G2,G) :-
        normalise_constants(G1,G2,NG1,NG2),
        single_edges(NG1,NG2,GS1,GRest1),
        single_edges(NG2,GRest1,GS2,GRest2),
        append(GS1,GS2,GG1),
        merge1(GRest1,GRest2,GG2),
        list_union_e(GG1,GG2,[],G).

%
% Gap graph subsumption test
%
subsumes(G1,G2) :-
        delmember(lt(K1,Vi,Vj),G1,G11),!,
        delmember(lt(K2,Vi,Vj),G2,G22),!,
        K1>=K2,!,
        subsumes(G11,G22).

subsumes(G1,G2) :-
        delmember(eq(Vi,Vj),G1,G11),!,
        (delmember(eq(Vi,Vj),G2,G22) ; delmember(eq(Vj,Vi),G2,G22)),!,
        subsumes(G11,G22).

subsumes([], []).


vars([P | Ps], [P | Vars]) :-
        \+compound(P),
        \+integer(P),
        \+member(P,[infinity,-infinity]),
        vars(Ps, Vars).

vars([P | Ps], Vars) :-
        compound(P),
        P =.. [_ | PArgs],
        append(PArgs, Ps, Ps1),
        vars(Ps1, Vars).

vars([_P | Ps], Vars) :-
        vars(Ps, Vars).

vars([], []).


%
% Rename the variables in a a gap graph to agree with those
% already in the list of gap graphs.
%
rename_vars(P=Graph, Results, [Q=RGraph | Results]) :-
        P =.. [Pf | PArgs],
        member(Q=_QGraph, Results),
        Q =.. [Pf | QArgs],
        generate_substlist(QArgs/PArgs, Substlist),
        replace_all(Substlist, Graph, RGraph),!.

rename_vars(X, Results, [X | Results]).

%
% Generate a subsitution list.
%
generate_substlist([]/[], []).

generate_substlist([X | Xs]/[Y | Ys], [X/Y | XYs]) :-
        generate_substlist(Xs/Ys, XYs).

%
% Normalise constants by eliminating all constants between the least
% and greatest in the graph as indicated by the proof of lemma 3.4 in
% [Revesz].  We need to treat +/-infinity specially (this detail is
% omitted from the proof in [Revesz]).
%
normalise_constants(G1,G2,NG1,NG2) :-
        constants(G1,CG1),
        constants(G2,CG2),
        append(CG1,CG2,Constants),
        minlist(Constants,Min),
        maxlist(Constants,Max),
        normalise_constants1(Min,Max,G1,NG1),
        normalise_constants1(Min,Max,G2,NG2),!.

normalise_constants1(_Min,_Max,[], []) :- !.

normalise_constants1(Min,Max,[lt(K,X,Y) | G], [lt(K1,X1,Y1) | NG]) :-
        (
        integer(X), X1=Min, K1 is X - Min + K, Y1=Y ;
        X= (-infinity), X1= (-infinity), K1=0, Y1=Y ;
        integer(Y), Y1=Max, K1 is Max - Y + K, X1=X ;
        Y=infinity, Y1=infinity, K1=0, X1=X
        ),
        normalise_constants1(Min,Max,G,NG).

normalise_constants1(Min,Max,[Rel | G], [Rel | NG]) :-
        normalise_constants1(Min,Max,G,NG).
```

```
constants([Rel | G],C) :-
        (Rel = lt(_,X,Y) ; Rel = eq(X,Y)),
        constants1([X,Y],C1),
        constants(G,C2),
        append(C1,C2,C).

constants([],[]).
constants1(Args, Constants) :-
        ssetof(C,(member(C,Args), is_constant(C)), Constants).

is_constant(infinity).
is_constant(-infinity).
is_constant(X) :- integer(X).

list_union_e([], [], Acc, Acc).

list_union_e([], [X|T], Acc, G) :-
        list_union_e([X|T], [], Acc, G).

list_union_e([lt(X,Y,Z) | G1], G2, Acc, G) :-
        \+member(lt(X,Y,Z),Acc),
        list_union_e(G1,G2,[lt(X,Y,Z) | Acc], G).

list_union_e([eq(X,Y) | G1], G2, Acc, G) :-
        \+member(eq(X,Y),Acc),
        \+member(eq(Y,X),Acc),
        list_union_e(G1,G2,[eq(X,Y) | Acc],G).

list_union_e([_ | G1], G2, Acc, G) :-
        list_union_e(G1, G2, Acc, G).

%
% single_edges(+Graph1, +Graph2, -Singles, -Rest).
% Find the edges which occur in only the first graph, returning a list
% of these edges, plus a list of the other edges in the graph, so that
% Graph1 =_set union(Singles, Rest)
%
single_edges([], G2, [], []).

single_edges([Edge | Edges], G2, [Edge | Singles], Rest) :-
        nodesofrel(Edge, [X,Y]),
        \+((member(Edge1, G2),
              (nodesofrel(Edge1, [X, Y]) ;
               nodesofrel(Edge1, [Y, X])))),
        single_edges(Edges, G2, Singles, Rest).

single_edges([Edge | Edges], G2, Singles, [Edge | Rest]) :-
        single_edges(Edges, G2, Singles, Rest).


merge1([lt(K1,Vi,Vj) | G1],G2,[lt(Kmax,Vi,Vj) | Gout]) :-
        member(lt(K2,Vi,Vj),G2),
        max(K1,K2,Kmax),
        merge1(G1,G2,Gout).

merge1([lt(K1,Vi,Vj) | G1],G2,_lt) :-
        member(lt(K2,Vj,Vi),G2),
        !,
        fail.

merge1([lt(K1,Vi,Vj) | G1],G2,_lt) :-
        member(eq(Vi,Vj),G2),
        !,
        fail.

merge1([lt(K1,Vi,Vj) | G1],G2,_lt) :-
        member(eq(Vj,Vi),G2),
        !,
        fail.

merge1([eq(Vi,Vj) | G1],G2,[eq(Vi,Vj) | Gout]) :-
        merge1(G1,G2,Gout).

merge1([eq(Vi,Vj) | G1],G2,_lt) :-
        member(lt(_K2,Vi,Vj),G2),
        !,
        fail.

merge1([eq(Vi,Vj) | G1],G2,_lt) :-
        member(lt(_K2,Vj,Vi),G2),
        !,
        fail.

merge1([], _, []).



constants_too_close(CGraph,X,Y) :-
        nodesof(CGraph, Nodes),
        member(X, Nodes), integer(X),
        member(Y, Nodes), integer(Y),
        Y>=X,
        Diff is Y - X,
        path(X,Y,CGraph,L), L> 0,
        L >= Diff.

%
% Find the length of a path between two nodes. If the path is
% x_1 <_k1 x_2 <_k3 ... <_k_(n-1) x_n, the the length is k1+...+kn+n-1
%
path(Start, End, CGraph, L) :-
        member(lt(K, Start, Next), CGraph),
        path(Next, End, CGraph, L1),
        L is L1+K+1.

path(Node, Node, _Graph, 0).

%
% respects_infinity(+Compact_Graph)
% Succeeds iff there are no edges x < -infinity or x > infinity in
% Compact_Graph
%
respects_infinity(CGraph) :-
        \+member(lt(_,-infinity),CGraph),
        \+member(lt(infinitym_),CGraph).
```

```
%
% cyclic(Graph) succeeds iff the graph is cyclic.
%
cyclic(Graph) :-
        nodesof(Graph, Nodes),
        member(Node, Nodes),
        cycle([Node], Graph).

cycle([X | Xs], Graph) :-
        member(lt(_K,X,Y),Graph),
        (member(Y,Xs) ;
         cycle([Y,X | Xs],Graph)).

nodesof(Graph, Nodes) :-
        ssetof(X,A^B^Rel^(member(Rel,Graph),nodesofrel(Rel,[A,B]),(X=A; X=B)),
               Nodes),!.

nodesofrel(eq(X,Y),[X,Y]).
nodesofrel(lt(_,X,Y),[X,Y]).

%
% Add all the edges which are implied by edges incident on Y
%
add_edges(Graph, Y, ShortGraph1) :-
        sbagof(Edge, add_edge(Graph, Y, Edge), Edges),
        append(Graph, Edges, ShortGraph1).

%
% If vi <_g vj and vi <_h vj and g<h then remove the first edge.
%
remove_subsumed_lt(ShortGraph, ShortGraph2) :-
        delmember(lt(K1,Vi,Vj),ShortGraph, SG1),
        delmember(lt(K2,Vi,Vj),SG1, SG2),
        (K1=<K2, remove_subsumed_lt([lt(K2,Vi,Vj) | SG2], ShortGraph2) ;
         K2<K1, remove_subsumed_lt([lt(K1,Vi,Vj) | SG2], ShortGraph2)).

remove_subsumed_lt(Graph, Graph).

%
% Are there two distinct edges between nodes X,Y?
%
double_edge(Graph, lt(K,X,Y), Edge2) :-
        delmember(lt(K,X,Y), Graph, G1),
        delmember(Edge2, G1, _G2),
        (Edge2=lt(_,X,Y) ;
         Edge2=lt(_,Y,X) ;
         Edge2=eq(X,Y) ;
         Edge2=eq(Y,X)).

%
% Finally, delete a node from the graph.
%
remove_node([Edge | G], Y, NewG) :-
        remove_edge(Edge, Y, NewEdge),
        remove_node(G, Y, NewG1),!,
        append(NewEdge, NewG1, NewG).

remove_node([], _Y, []).
```

```
remove_edge(lt(_,_,Y), Y, []).
remove_edge(lt(_,Y,_), Y, []).
remove_edge(eq(_,Y), Y, []).
remove_edge(eq(Y,_), Y, []).
remove_edge(Edge, _Y, [Edge]).


%
% Each of the various ways in which the nodes incident on Y affect
% other nodes in the graph.
%
add_edge(G, Y, eq(Vi,Vj)) :-
        delmember(eq(Vi,Y),G,G1),
        delmember(eq(Y,Vj),G1,_).

add_edge(G, Y, lt(K,Vi,Vj)) :-
        equal(Vi,Y,G),
        lessthan(K,Y,Vj,G).

add_edge(G, Y, lt(K,Vi,Vj)) :-
        lessthan(K,Vi,Y,G),
        equal(Y,Vj,G).

add_edge(G, Y, lt(KK,Vi,Vj)) :-
        lessthan(K1,Vi,Y,G),
        lessthan(K2,Y,Vj,G),
        KK is K1+K2+1.


equal(X,Y,G) :-
        member(eq(A,B),G),
        (A=X,B=Y ;
         A=Y,B=X).

lessthan(K,X,Y,G) :-
        member(lt(K,X,Y),G).

%
% replace_all(+Substlist, +TermList, -NTermList)
% Each substitution in Substlist is of the form New/Old. Replace Old in
% terms eq(Old,X), eq(X,Old), lt(K,Old,X), lt(K,X,Old) by New in the
% list TermList with the result NTermList.
%
replace_all([], L, L).
replace_all([New/Old | Subs], Terms, NTerms) :-
        replace1(New/Old, Terms, NTerms1),
        replace_all(Subs, NTerms1, NTerms).

replace1(_, [], []).

replace1(New/Old, [Rel | Terms], NTerms) :-
        replace0(New/Old, Rel, eq(T,T)),!,
        replace1(New/Old, Terms, NTerms).

replace1(New/Old, [Rel | Terms], [NRel | NTerms]) :-
        replace0(New/Old, Rel, NRel),
        replace1(New/Old, Terms, NTerms).
```

```
replace0(New/Old, Rel, NRel) :-
         Rel = lt(K,Old,X), NRel = lt(K,New,X) ;
         Rel = eq(Old,X), NRel = eq(New,X) ;
         Rel = lt(K,X,Old), NRel = lt(K,X,New) ;
         Rel = eq(X,Old), NRel = eq(X,New) ;
         Rel = NRel.

delmember(X,[X|Y],Y).

delmember(X,[A|T],[A|NT]) :-
        delmember(X,T,NT).

%
% Example graphs
%
egraph(1,[eq(y,x3), lt(0,x1,y),lt(3,2,y),lt(2,2,x3),lt(5,y,x2),lt(7,x3,18),lt(2,x2,18)]).

egraph(2,[lt(4,x1,x2),lt(3,x2,18),lt(0,y,x2),eq(y,x3)]).

egraph(3,[eq(y,x3), lt(6,x1,y),lt(3,2,y),lt(4,2,x3),lt(5,y,x2),lt(9,x3,18),lt(3,x2,18)]).

minlist([H | T], Min) :-
        minlist([H | T], H, Min),!.

%
% Tail-recursive minlist(+List, +Accumulator, -Result).
%
minlist([H | T], A, R) :-
        H<A,
        minlist(T, H, R).

minlist([_ | T], A, R) :-
        minlist(T, A, R).

minlist([], R, R).


maxlist([H | T], Max) :-
        maxlist([H | T], H, Max),!.

%
% Tail-recursive maxlist(+List, +Accumulator, -Result).
%
maxlist([H | T], A, R) :-
        H>A,
        maxlist(T, H, R).

maxlist([_ | T], A, R) :-
        maxlist(T, A, R).

maxlist([], R, R).


max(H,L,H) :-
        H>=L.

max(L,H,H) :-
```

```
        H>=L.

ssetof(A,B,C) :- setof(A,B,C),! ; C=[].
sbagof(A,B,C) :- bagof(A,B,C),! ; C=[].

greplace_all(B, A/B, A).

greplace_all(T, A/B, T1) :-
        compound(T),
        T =.. [F | Args],
        greplace_alll(Args, A/B, NArgs),
        T1 =.. [F | NArgs].

greplace_all(T, A/B, T).

greplace_alll([], _, []).
greplace_alll([H | T], Sub, [H1 | T1]) :-
        greplace_all(H, Sub, H1),
        greplace_alll(T, Sub, T1).


nmember(1,H,[H|T]).
nmember(N,X,[_H|T]) :-
        nmember(N1,X,T),
        N is N1+1.

%
% Code for pretty-printing gap constraints
%
latex_pprint([P = G | GGs]) :-
        nl,
        format("\[~k\ \leftarrow\  ",[P]),
        simplify(G,G1),
        latex_pprint1(G1),
        format("\]",[]),nl,
        latex_pprint(GGs).

latex_pprint([]).

latex_pprint1(G) :-
        normalise_eq(G,G1),
        sort(G1,G2),!,
        latex_pprint2(G2).

normalise_eq([],[]).
normalise_eq([eq(X,Y) | T],[eq(A,B) | T1]) :-
        (is_constant(X), A=Y, B=X ;
         A=X, B=Y
        ),
        normalise_eq(T,T1).

normalise_eq([H | T], [H | T1]) :-
        normalise_eq(T,T1).

latex_pprint2([]).
latex_pprint2(G) :-
        delmember(eq(X,Y),G,G1),
        format("~k = ~k",[X,Y]),
        (G1=[_ | _] -> format("\ \wedge\ ",[]) ; true),
        latex_pprint2(G1).

latex_pprint2(G) :-
        delmember(lt(K1,X,Y),G,G1),
        delmember(lt(K2,Y,Z),G1,G2),
        format("~k <_{~k} ~k <_{~k} ~k",[X,K1,Y,K2,Z]),
        (G2=[_ | _] -> format("\ \wedge\ ",[]) ; true),
        latex_pprint2(G2).

latex_pprint2([lt(K,X,Y) | G]) :-
        format("~k <_{~k} ~k",[X,K,Y]),
        (G=[_ | _] -> format("\ \wedge\ ",[]) ; true),
        latex_pprint2(G).
```