Automatic Abstraction for Congruences

Andy King[1] and Harald Søndergaard[2]

[1] Portcullis Computer Security Limited, Pinner, HA5 2EX, UK**
[2] The University of Melbourne, Victoria 3010, Australia

Abstract. One approach to verifying bit-twiddling algorithms is to derive invariants between the bits that constitute the variables of a program. Such invariants can often be described with systems of congruences where in each equation $\boldsymbol{c} \cdot \boldsymbol{x} = d \mod m$, m is a power of two, $\boldsymbol{c}$ is a vector of integer coefficients, and $\boldsymbol{x}$ is a vector of propositional variables (bits). Because of the low-level nature of these invariants and the large number of bits that are involved, it is important that the transfer functions can be derived automatically. We address this problem, showing how an analysis for bit-level congruence relationships can be decoupled into two parts: (1) a SAT-based abstraction (compilation) step which can be automated, and (2) an interpretation step that requires no SAT-solving. We exploit triangular matrix forms to derive transfer functions efficiently, even in the presence of large numbers of bits. Finally we propose program transformations that improve the analysis results.

1 Introduction

Recently there has been a resurgence of interest in inferring numeric relations between program variables, most notably with congruences [1,8,11]. In this abstract domain, each description is a system of congruence equations (over n variables), each taking the form $\boldsymbol{c} \cdot \boldsymbol{x} = d \mod m$, with $c \in \mathbb{Z}^n$, $d, m \in \mathbb{Z}$ and $\boldsymbol{x}$ an n-ary vector of variables. The congruence $\boldsymbol{c} \cdot \boldsymbol{x} = d \mod m$, henceforth abbreviated to $\boldsymbol{c} \cdot \boldsymbol{x} \equiv_m d$, expresses that there exists a multiplier $k \in \mathbb{Z}$ of m such that $\boldsymbol{c} \cdot \boldsymbol{x} = d + km$. Quite apart from their expressiveness [5], such systems are attractive computationally since, if the values in $[0, m-1]$ can be represented with machine integers then arbitrary precision arithmetic can be avoided in abstract operations, and at the same time, polynomial performance guarantees are obtained [11]. This compares favourably with systems of inequalities that present, among other problems, the issue of how to curb the growth of coefficients [6,9,15].

Of particular interest are congruences where m is a power of two, since these can express invariants that hold at the level of machine words [11] or bits [8]. The central idea of [8] is *congruent closure* which computes a system of congruences c to describe all the solutions of a given Boolean function f. To see the motivation for this, consider bit-twiddling programs such as those in Figure 1 (we return to the two programs in Section 3). Such programs often establish important but

** Andy King is on secondment from the University of Kent, CT2 7NF, UK

```
$\ell_0$:  $p := 0$; $y := x$;
$\ell_1$:  while $(y \neq 0)$
               $y := y \,\&\, (y - 1)$;
               $p := 1 - p$;
$\ell_2$:  skip
```

(a)

```
$\ell_0$:  $y := x$;
           $y := ((y \gg 1)$  & 0x5555) | ((y  & 0x5555)  $\ll 1)$;
           $y := ((y \gg 2)$  & 0x3333) | ((y  & 0x3333)  $\ll 2)$;
           $y := ((y \gg 4)$  & 0x0F0F) | ((y  & 0x0F0F)  $\ll 4)$;
           $y := (y \gg 8)$             | $(y \ll 8)$;
$\ell_1$:  skip
```

(b)

Fig. 1. Computing the parity of x and reversing the 16-bit word x

obscure invariants. Performing a complete bit-precise analysis is infeasible for all but the simplest loop-free programs. At the same time, the invariants can often be captured succinctly as a system of congruence equations. However, as the assignments involved are not linear, traditional congruence analyses will not work. An alternative is to summarise basic program blocks bit-precisely and apply congruent closure judiciously. This allows us to reveal "numeric" invariants amongst bits, even for flowchart programs with loops, such as in Figure 1(a). Congruences satisfy the ascending chain condition: no infinite chain $c_1, c_2, \ldots$ with $[\![c_i]\!] \subset [\![c_{i+1}]\!]$ exists. We exploit this to compute congruent closure symbolically, by solving a finite number of SAT instances [8].

Congruent closure connects with work on how to compute most precise transfer functions for a given abstract domain. A transfer function simulates the effect of executing an operation where the possible input values are summarised by an element in the abstract domain. The problem is how to find, in the domain, the most precise element that summarises all outputs that can result from the summarised inputs. In predicate abstraction, when the abstract domain is a product of Boolean values, decision procedures have been used to solve this problem [4]. More generally, a decision procedure can also be applied to compute the most precise transfer function when the domain satisfies the ascending chain condition [13]. The idea is to translate the input summary into a formula which is conjoined with another that expresses the semantics of the operation as a relationship between input values and output values. An output summary is then extracted from the conjoined formula by repeatedly calling the decision procedure. Reps et al [13] illustrate this construction for constant propagation, and the technique is equally applicable to congruences. In this context, the semantics of an operation can be expressed propositionally [8]. The state of each integer variable is represented by a vector of propositional variables, one propositional variable for each bit. A formula is then derived [2, 7], that is propositional, which specifies how the output bits depend on the input bits. Given an input summary that is congruent, a congruent output summary can be derived by: (1) converting the input summary to a propositional formula; (2) conjoining it with the input-output formula; (3) applying congruent closure to the conjunction. The advantage of this formulation is that it can derive invariants down to the level of bits, which enables the correctness of bit-twiddling code to be verified [8].

Congruent closure may require many calls to a SAT solver. As sketched, it is computed each time a transfer function is applied. A critical observation in this paper is that it is possible, and simpler, to summarise the input-output formula as a linear system that prescribes a transfer function. Once all transfer functions have been derived, it is only necessary to manipulate linear systems. In this new scheme, the application of a SAT solver is limited to the compilation step: the derivation of the transfer function. With this paper we:

- Consider an unrestricted flowchart language with non-linear, bit-manipulating operations and provide it with a relational semantics. The semantic definition lets us dovetail bit-blasting with congruent closure, and avoids the need for a separate collecting semantics.
- Show that congruent closure is only needed in the derivation, from the bit-blasted relational semantics, of a certain transition system; thereafter congruence invariants can be inferred by repeatedly applying linear operations to the transition system. As well as allowing separation of concerns, this avoids the overhead of repeated closure calculation.
- Present a new algorithm for congruent closure. Its use of (upper triangular) matrices for congruence systems makes it considerably faster than a previous algorithm [8].
- Show how an input program can be transformed so that range information can be inferred for variables occurring in loops. This is possible since bit-level (rather than word-level) congruences can express the non-negativity of a variable, which is sufficient to verify that inequalities hold.

Analyses using congruences modulo 2^k have previously been designed [8, 11]. Our main contribution here is the automated derivation of transfer functions for these analyses. This complements recent work [10] on automatically deriving transfer functions for linear template domains [14] (which can realise octagons and intervals) where the semantics of instructions is modelled with piecewise linear functions. However, our approach does not impose this semantic restriction and is not based on quantifier elimination.

The paper is structured as follows: The new algorithm for congruent closure is given in Section 2. Section 3 presents a relational semantics for flowchart programs over machine integers and Section 4 develops a bit-level relational semantics that encapsulates the spirit of bit-blasting. Section 5 shows how these semantics can be abstracted to derive transition systems over congruences. Section 6 explains how programs can be transformed to derive range information. Section 7 concludes.

2 Congruent Closure

This section introduces a new algorithm for computing the congruent closure of a Boolean function. Let $\mathbb{B}_m = \{0, 1\}$ and let $\mathbb{Z}_m = [0, m-1]$. If $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{Z}^k$ then we write $\boldsymbol{x} \equiv_m \boldsymbol{y}$ for $\bigwedge_{i=1}^{k} x_i \equiv_m y_i$ where $\boldsymbol{x} = \langle x_1, \ldots, x_k \rangle$ and $\boldsymbol{y} = \langle y_1, \ldots, y_k \rangle$.

Definition 1. The (modulo m) *affine hull* of $S \subseteq \mathbb{Z}_m^k$ is defined:

$$\mathsf{aff}_m^k(S) = \left\{ \boldsymbol{x} \in \mathbb{Z}_m^k \;\middle|\; \begin{array}{l} \boldsymbol{x}_1, \ldots, \boldsymbol{x}_\ell \in S \quad \wedge \quad \lambda_1, \ldots, \lambda_\ell \in \mathbb{Z} \quad \wedge \\ \sum_{i=1}^{\ell} \lambda_i \equiv_m 1 \quad \wedge \quad \boldsymbol{x} \equiv_m \sum_{i=1}^{\ell} \lambda_i \boldsymbol{x}_i \end{array} \right\}$$

Example 1. If $S = \emptyset$, $S = \mathbb{Z}_m^k$ or $S = \{\boldsymbol{x}\}$ for some $\boldsymbol{x} \in \mathbb{Z}_m^k$ then $\mathsf{aff}_m^k(S) = S$. Now consider $S = \{\langle 0, 3\rangle, \langle 1, 5\rangle\}$. We have

$$\begin{aligned} \mathsf{aff}_8^2(S) &= \{\boldsymbol{x} \in \mathbb{Z}_8^2 \mid \lambda_1 + \lambda_2 \equiv_8 1 \wedge \boldsymbol{x} \equiv_8 \lambda_1\langle 0,3\rangle + \lambda_2\langle 1,5\rangle\} \\ &= \{\boldsymbol{x} \in \mathbb{Z}_8^2 \mid \boldsymbol{x} \equiv_8 \langle k, 3+2k\rangle \wedge k \in \mathbb{Z}\} \end{aligned}$$

Let $\mathsf{Aff}_m^k = \{S \subseteq \mathbb{Z}_m^k \mid \mathsf{aff}_m^k(S) = S\}$. Suppose $S_i \in \mathsf{Aff}_m^k$ for all $i \in I$ where I is some index set. Put $S = \bigcap_{i \in I} S_i$. It is not difficult to see that $\mathsf{aff}_m^k(S) = S$. In other words, $\langle \mathsf{Aff}_m^k, \subseteq, \bigcap\rangle$ is a Moore family [3], and we obtain a complete lattice $\langle \mathsf{Aff}_m^k, \subseteq, \bigcap, \bigsqcup\rangle$ by defining $\bigsqcup_{i \in I} S_i = \bigcap\{S' \in \mathsf{Aff}_m^k \mid \forall i \in I.S_i \subseteq S'\}$. This gives rise to a notion of abstraction in the following sense:

Definition 2. The abstraction map $\alpha_m^k : \wp(\mathbb{B}^k) \to \mathsf{Aff}_m^k$ and concretisation map $\gamma_m^k : \mathsf{Aff}_m^k \to \wp(\mathbb{B}^k)$ are defined: $\alpha_m^k(S) = \mathsf{aff}_m^k(S)$ and $\gamma_m^k(S) = S \cap \mathbb{B}^k$.

For any k and m we call α_m^k the modulo m *congruent closure*[3] of its argument.

Example 2. Let us denote the set of solutions (models) of a Boolean function f by $[\![f]\!]$ thus, for example, $[\![x_1 \wedge x_2]\!] = \{\langle 1,1\rangle\}$ and $[\![x_1 \oplus x_2]\!] = \{\langle 0,1\rangle, \langle 1,0\rangle\}$ where $\oplus$ denotes exclusive-or. Likewise, let us denote the set of solutions of a system of congruences c by $[\![c]\!]$. For instance, if $c = (x_1 + x_2 \equiv_4 3 \wedge 3x_2 \equiv_4 2)$ then $[\![c]\!] = \{\langle 4k_1 + 1, 4k_2 + 2\rangle \in \mathbb{Z}_4^2 \mid k_1, k_2 \in \mathbb{Z}\}$ where $\mathbb{Z}_m = [0, m-1]$. Given f over n (propositional) variables $\boldsymbol{x}$ and a modulus m, congruent closure computes the strongest congruence system c over n (integer) variables such that $[\![f]\!] \subseteq [\![c]\!]$, or equivalently, $[\![f]\!] \subseteq [\![c]\!] \cap \mathbb{B}^n$ where $\mathbb{B} = \{0,1\}$. For example, given $m = 4$, $f_1 = (\neg x_1) \wedge (x_1 \oplus x_2 \oplus x_3)$, and $f_2 = x_1 \wedge (x_2 \vee x_3)$, congruent closure computes $c_1 = (x_1 \equiv_4 0 \wedge x_2 + x_3 \equiv_4 1)$ and $c_2 = (x_1 \equiv_4 1)$ respectively. The congruences c_1 and c_2 describe all solutions of f_1 and f_2, as

$$\begin{aligned} [\![f_1]\!] = \{\langle 0,0,1\rangle, \langle 0,1,0\rangle\} &= [\![c_1]\!] \cap \mathbb{B}^3 \\ [\![f_2]\!] = \{\langle 1,0,1\rangle, \langle 1,1,0\rangle, \langle 1,1,1\rangle\} \subseteq \{\langle 1, x_2, x_3\rangle \mid x_2, x_3 \in \mathbb{B}\} &= [\![c_2]\!] \cap \mathbb{B}^3 \end{aligned}$$

Note that c_2 additionally contains a non-solution $\langle 1,0,0\rangle$ of f_2 and hence, in general, congruent closure upper-approximates the set of models of a Boolean function.

It is straightforward to verify that α_m^k and γ_m^k form a Galois connection between the complete lattices $\langle \wp(\mathbb{B}^k), \subseteq, \bigcap, \bigcup\rangle$ and $\langle \mathsf{Aff}_m^k, \subseteq, \bigcap, \bigsqcup\rangle$.

[3] The notion should not be confused with congruence closure as used in the automated deduction community for the computation of equivalence relations over the set of nodes of a graph a la Nelson and Oppen [12].

```
function closure(input: $S \subseteq \mathbb{B}^k$ and modulus $m \in \mathbb{N}$)
    $[A|\boldsymbol{b}] := [0, \ldots, 0, 1]$;                       – the unsatisfiable system
    i := 0; r := 1;
    while $(i < r)$
        $\langle a_1, \ldots, a_k, b\rangle := \text{row}([A|\boldsymbol{b}], r - i)$;       – last non-stable row
        $S' := \{\boldsymbol{x} \in S \mid \langle a_1, \ldots, a_k\rangle \cdot \boldsymbol{x} \not\equiv_m b\}$; – impose disequality
        if (there exists $\boldsymbol{x} \in S'$) then                – solve new SAT instance
            $[A'|\boldsymbol{b}'] := [A|\boldsymbol{b}] \sqcup [Id|\boldsymbol{x}]$;           – merge with new solution $\boldsymbol{x}$
            $[A|\boldsymbol{b}] := \text{triangular}([A'|\boldsymbol{b}'])$;
            $r := \text{rows}([A|\boldsymbol{b}])$;
        else
            $i := i + 1$;                                – $\boldsymbol{a} \cdot \boldsymbol{x} \equiv_m b$ is invariant so move on
    return $[A|\boldsymbol{b}]$;
```

Fig. 2. Calculating Congruent Closure Based on Triangularisation

Example 3. Suppose $S_{b_0b_1b_2b_3} = \{\langle 0,0\rangle \mid b_0 = 1\} \cup \{\langle 0,1\rangle \mid b_1 = 1\} \cup \{\langle 1,0\rangle \mid b_2 = 1\} \cup \{\langle 1,1\rangle \mid b_3 = 1\}$ where $b_0, b_1, b_2, b_3 \in \mathbb{B}$. Then

$\alpha^2_{16}(S_{0000}) = \emptyset$	$\alpha^2_{16}(S_{1000}) = \{\langle 0,0\rangle\}$
$\alpha^2_{16}(S_{0001}) = \{\langle 1,1\rangle\}$	$\alpha^2_{16}(S_{1001}) = \{\langle k,k\rangle \mid k \in [0,15]\}$
$\alpha^2_{16}(S_{0010}) = \{\langle 1,0\rangle\}$	$\alpha^2_{16}(S_{1010}) = \{\langle k,0\rangle \mid k \in [0,15]\}$
$\alpha^2_{16}(S_{0011}) = \{\langle 1,k\rangle \mid k \in [0,15]\}$	$\alpha^2_{16}(S_{1011}) = \mathbb{Z}^2_{16}$
$\alpha^2_{16}(S_{0100}) = \{\langle 0,1\rangle\}$	$\alpha^2_{16}(S_{1100}) = \{\langle 0,k\rangle \mid k \in [0,15]\}$
$\alpha^2_{16}(S_{0101}) = \{\langle k,1\rangle \mid k \in [0,15]\}$	$\alpha^2_{16}(S_{1101}) = \mathbb{Z}^2_{16}$
$\alpha^2_{16}(S_{0110}) = \{\langle k_1,k_2\rangle \in \mathbb{Z}^2_{16} \mid k_1 + k_2 \equiv_{16} 1\}$	$\alpha^2_{16}(S_{1110}) = \mathbb{Z}^2_{16}$
$\alpha^2_{16}(S_{0111}) = \mathbb{Z}^2_{16}$	$\alpha^2_{16}(S_{1111}) = \mathbb{Z}^2_{16}$

From this we conclude that, in general, α^k_m is not surjective and therefore α^k_m and γ^k_m do not form a Galois insertion.

Example 4. Let f be the Boolean function $c'_0 \leftrightarrow (c_0 \oplus 1) \wedge c'_1 \leftrightarrow (c_1 \oplus c_0) \wedge c'_2 \leftrightarrow (c_2 \oplus (c_0 \wedge c_1)) \wedge c'_3 \leftrightarrow (c_3 \oplus (c_0 \wedge c_1 \wedge c_2))$. Then $\alpha^8_{16}([\![f]\!]) = [\![c]\!]$, where c is the conjunction of two equations $c_0 + 2c_1 + 4c_2 + 8c_3 + 1 \equiv_{16} c'_0 + 2c'_1 + 4c'_2 + 8c'_3$ and $c_0 + c'_0 \equiv_{16} 1$. This illustrates how congruent closure can extract numeric relationships from a Boolean function.

Figure 2 presents a new algorithm for finding the congruent closure of a Boolean function. For the purpose of presentation, it is convenient to pretend the function is given as a set S of models, although we assume it given in conjunctive normal form. If A is an $m \times n$ matrix and $\boldsymbol{b} = (b_1, \ldots, b_m)$ is a vector, we denote by $[A|\boldsymbol{b}]$ the $m \times (n + 1)$ matrix B defined by

$$B_{ij} = \begin{cases} A_{ij} & \text{if } 1 \leq i \leq m \text{ and } 1 \leq j \leq n \\ b_i & \text{if } 1 \leq i \leq m \text{ and } j = n + 1 \end{cases}$$

Given a matrix A, we write 'row(A, i)' for its ith row, and 'rows(A)' for the number of rows. We use 'triangular(A)' for the result of bringing A into upper triangular form—Müller-Olm and Seidl [11] provide an algorithm for this. The join operation $\sqcup$ can be implemented in terms of projection which in turn has a simple implementation utilising the maintenance of upper-triangular form [8]. Space constraints prevent us from repeating the join algorithm here.

It is important to observe that S' can be expressed propositionally by augmenting S with a propositional encoding of the *single* disequality constraint $\langle a_1, \ldots, a_k\rangle \cdot \boldsymbol{x} \not\equiv_m b$. This ensures that the propositional encoding of S' does not grow without bound, which is vital for tractability. A chain length result for congruences [11] ensures that the total number of calls to the SAT solver is $O(wk)$ when $m = 2^w$.

Example 5. Suppose $f = (\neg x_3 \wedge \neg x_2 \wedge \neg x_1 \wedge \neg x_0 \wedge \neg x'_3 \wedge \neg x'_2 \wedge \neg x'_1 \wedge \neg x'_0) \vee (x_3 \wedge x'_3 \wedge x'_2 \wedge x'_1 \wedge x'_0) \vee (\neg x_3 \wedge (x_2 \vee x_1 \vee x_0) \wedge \neg x'_3 \wedge \neg x'_2 \wedge \neg x'_1 \wedge x'_0)$. (This function could appear in an attempt to reason about an assignment $x :=$ sign(x) for a machine with 4-bit words.) The table given in Figure 3 shows how the algorithm proceeds when computing the congruent closure of f, assuming a particular sequence of results being returned from a SAT solver. The responses from the solver are shown. In step 0, a single model of f produces the equation system s_1. This, and the subsequent congruence systems, are also shown. Each system s_i is produced from its predecessor s_{i-1} by identifying some model $\boldsymbol{x}$ of f that is not covered by s_{i-1} and calculating the strongest congruence system covering both, that is, s_i is the join of s_{i-1} and the system expressing the fact that $\boldsymbol{x}$ is a model. The congruent closure of f is finally given by s_6.

The following proposition states the correctness of the algorithm: the result is independent of the order in which a SAT/SMT solver finds solutions. A proof sketch has been relegated to the appendix.

Proposition 1. Let $S \subseteq \mathbb{B}^k$ and $m \in \mathbb{N}$, and let $[A|\boldsymbol{b}] = \text{closure}(S, m)$. Then $\mathsf{aff}^k_m(S) = \{\boldsymbol{x} \in \mathbb{Z}^k_m \mid A\boldsymbol{x} \equiv_m \boldsymbol{b}\}$.

3 Relational Semantics

Flowchart programs are defined over a finite set of labels L and a set of variables $X = \{x_1, \ldots, x_k\}$ that range over values drawn from $R = [-2^{w-1}, 2^{w-1} - 1]$. A flowchart program P is a quadruple $P = \langle L, X, \ell_0, T\rangle$ where $\ell_0 \in L$ indicates the program entry point and $T \subseteq L \times L \times \mathsf{Guard} \times \mathsf{Stmt}$ is a finite set of transitions.

3.1 Syntax of flowchart programs

The classes of well-formed expressions, guards and statements are defined by:

$$\begin{aligned}
\mathsf{Expr} &::= X \mid R \mid -\mathsf{Expr} \mid \mathsf{Expr\ bop\ Expr}\\
\mathsf{Guard} &::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{Expr\ rop\ Expr} \mid \mathsf{Guard\ lop\ Guard}\\
\mathsf{Stmt} &::= \mathsf{skip} \mid X := \mathsf{Expr} \mid \mathsf{Stmt}; \mathsf{Stmt}
\end{aligned}$$

Step	i	Response from SAT solver	$A\boldsymbol{x} \equiv_m \boldsymbol{b}$
0	0	$x_0 = 0, x_1 = 0, x_2 = 1, x_3 = 0, x'_0 = 1, x'_1 = 0, x'_2 = 0, x'_3 = 0$	s_1
1	0	$x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 1, x'_0 = 1, x'_1 = 1, x'_2 = 1, x'_3 = 1$	s_2
2	0	UNSATISFIABLE	s_2
3	1	UNSATISFIABLE	s_2
4	2	$x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 0, x'_0 = 0, x'_1 = 0, x'_2 = 0, x'_3 = 0$	s_3
5	2	UNSATISFIABLE	s_3
6	3	$x_0 = 0, x_1 = 1, x_2 = 0, x_3 = 0, x'_0 = 1, x'_1 = 0, x'_2 = 0, x'_3 = 0$	s_4
7	3	$x_0 = 0, x_1 = 1, x_2 = 1, x_3 = 0, x'_0 = 1, x'_1 = 0, x'_2 = 0, x'_3 = 0$	s_5
8	3	$x_0 = 1, x_1 = 0, x_2 = 0, x_3 = 0, x'_0 = 1, x'_1 = 0, x'_2 = 0, x'_3 = 0$	s_6

s_1 :
$$\left\{\begin{array}{ccccccccl}
x_0 &&&&&&&& \equiv_4 0 \\
& x_1 &&&&&&& \equiv_4 0 \\
&& x_2 &&&&&& \equiv_4 1 \\
&&& x_3 &&&&& \equiv_4 0 \\
&&&& x'_0 &&&& \equiv_4 1 \\
&&&&& x'_1 &&& \equiv_4 0 \\
&&&&&& x'_2 && \equiv_4 0 \\
&&&&&&& x'_3 & \equiv_4 0
\end{array}\right\}$$

s_2 :
$$\left\{\begin{array}{ccccccccl}
x_0 &&&&&&&& \equiv_4 0 \\
& x_1 &&&&&&& \equiv_4 0 \\
&& x_2 & x_3 &&&&& \equiv_4 1 \\
&&& x_3 && -x'_1 &&& \equiv_4 0 \\
&&&& x'_0 &&&& \equiv_4 1 \\
&&&&& x'_1 & -x'_2 && \equiv_4 0 \\
&&&&&& x'_2 & -x'_3 & \equiv_4 0
\end{array}\right\}$$

s_3 :
$$\left\{\begin{array}{ccccccccl}
x_0 &&&&&&&& \equiv_4 0 \\
& x_1 &&&&&&& \equiv_4 0 \\
&& x_2 & x_3 & -x'_0 &&&& \equiv_4 0 \\
&&& x_3 && -x'_1 &&& \equiv_4 0 \\
&&&&& x'_1 & -x'_2 && \equiv_4 0 \\
&&&&&& x'_2 & -x'_3 & \equiv_4 0
\end{array}\right\}$$

s_4 :
$$\left\{\begin{array}{ccccccccl}
x_0 &&&&&&&& \equiv_4 0 \\
& x_1 & x_2 & x_3 & -x'_0 &&&& \equiv_4 0 \\
&&& x_3 && -x'_1 &&& \equiv_4 0 \\
&&&&& x'_1 & -x'_2 && \equiv_4 0 \\
&&&&&& x'_2 & -x'_3 & \equiv_4 0
\end{array}\right\}$$

s_5 :
$$\left\{\begin{array}{cccccl}
x_0 &&&&& \equiv_4 0 \\
& x_3 & -x'_1 &&& \equiv_4 0 \\
&& x'_1 & -x'_2 && \equiv_4 0 \\
&&& x'_2 & -x'_3 & \equiv_4 0
\end{array}\right\}$$

s_6 :
$$\left\{\begin{array}{ccccl}
x_3 & -x'_1 &&& \equiv_4 0 \\
& x'_1 & -x'_2 && \equiv_4 0 \\
&& x'_2 & -x'_3 & \equiv_4 0
\end{array}\right\}$$

Fig. 3. SAT responses and the six congruence systems that arise for Example 5

where the sets of binary operators bop, logical operators lop and relational operators rop are defined thus rop $= \{=, \neq, <, \leq\}$, bop $= \{+, -, \,\&\,, \,|\,, \ll, \gg\}$, lop $= \{\wedge, \vee\}$ and the $\,\&\,, \,|\,, \ll, \gg$ symbols denote C-style bitwise operations.

Example 6. The program in Figure 1(a) can be expressed as the flowchart $\langle\{\ell_0, \ell_1, \ell_2\}, \{p, x, y\}, \ell_0, T\rangle$ where $T = \{t_1, t_2, t_3\}$ and $t_1 = \langle \ell_0, \ell_1, \mathit{true}, p = 0; y = x\rangle$, $t_2 = \langle \ell_1, \ell_1, y \neq 0, y := y \,\&\, (y-1); p := 1 - p\rangle$, $t_3 = \langle \ell_1, \ell_2, y = 0, \mathsf{skip}\rangle$.

Example 7. The program in Figure 1(b) is expressed as

$$\langle\{\ell_0, \ell_1\}, \{x, y\}, \ell_0, \{\langle \ell_0, \ell_1, \mathit{true}, y := x; y := e_1; y := e_2; y := e_3; y := e_4\rangle\}\rangle$$

where e_1, e_2, e_3 and e_4 are the RHSs of the assignments that follow $y := x$.

3.2 Semantics of flowchart programs

All variables are of limited-precision integer signed type, based on some word length w. The semantics can be formulated denotationally in terms of functions: The set of states is the function space $\Sigma = X \rightarrow R$ and each state $\sigma \in \Sigma$ maps a variable to a value of R; the function $\mathcal{E} : \mathsf{Expr} \rightarrow \Sigma \rightarrow R$ evaluates an expression; and the function $\mathcal{S} : \mathsf{Stmt} \rightarrow \Sigma \rightarrow \Sigma$ transforms one state to another. However, we prefer to give a relational semantics, for a number of reasons. First, we consider programs to take input via the program variables, so the semantics needs to express how, at different points, program states are related to initial states. Second, the relational semantics can be bit-blasted in a natural way, and this is essential to the program analysis that we discuss. Third, we avoid a need to lift a standard semantics to a so-called collecting semantics. Hence we wish to express the effect of a program statement as a relation $r \subseteq R^{2k}$ that captures the values of the k variables before and after the statement is executed. Compared to the denotational approach, in our relational viewpoint a state transformer $\mathcal{S}[\![s]\!] : \Sigma \rightarrow \Sigma$ is replaced by a relation $r = \{\langle \sigma(x_1), \ldots, \sigma(x_k), \tau(x_1), \ldots, \tau(x_k) \rangle \mid \sigma \in \Sigma \wedge \tau = \mathcal{S}[\![s]\!](\sigma)\}$. Henceforth $\mathcal{S}[\![s]\!]$ will denote a relation $\mathcal{S}[\![s]\!] \subseteq R^{2k}$.

3.3 Semantic machinery: composition and bit manipulation

To formulate a relational semantics, if $\boldsymbol{a}, \boldsymbol{b} \in R^k$ then let $\boldsymbol{a} \cdot \boldsymbol{b} \in R^{2k}$ denote the concatenation of $\boldsymbol{a}$ and $\boldsymbol{b}$. The identity relation is then $\mathsf{Id} = \{\boldsymbol{a} \cdot \boldsymbol{a} \mid \boldsymbol{a} \in R^k\}$. If $r_1, r_2 \subseteq R^{2k}$ then the composition of r_1 and r_2 is defined $r_1 \circ r_2 = \{\boldsymbol{a} \cdot \boldsymbol{c} \mid \boldsymbol{b} \in R^k \wedge \boldsymbol{a} \cdot \boldsymbol{b} \in r_1 \wedge \boldsymbol{b} \cdot \boldsymbol{c} \in r_2\}$. Furthermore, if $r_1 \subseteq R^k$ and $r_2 \subseteq R^{2k}$ then let $r_1 \circ r_2 = \{\boldsymbol{b} \mid \boldsymbol{a} \in r_1 \wedge \boldsymbol{a} \cdot \boldsymbol{b} \in r_2\}$. If $\boldsymbol{a} = \langle a_1, \ldots, a_k \rangle \in R^k$ let $\boldsymbol{a}[i] = a_i$ and if $b \in R$ let $\boldsymbol{a}[i \mapsto b] = \langle a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_k \rangle$.

To specify bit-twiddling operations, let $\langle\!\langle . \rangle\!\rangle : [-2^{w-1}, 2^{w-1} - 1] \rightarrow \mathbb{B}^w$ and $\langle . \rangle : [0, 2^w - 1] \rightarrow \mathbb{B}^w$ denote the signed and unsigned w-bit representation of an integer. Thus let $\langle\!\langle n \rangle\!\rangle = \langle x_0, \ldots, x_{w-1} \rangle$ where $n = (\sum_{i=0}^{w-2} 2^i x_i) - 2^{w-1} x_{w-1}$ and let $\langle m \rangle = \langle x_0, \ldots, x_{w-1} \rangle$ where $m = \sum_{i=0}^{w-1} 2^i x_i$. Let $n_1, n_2 \in R$. To define $n_1 \mid n_2 = n$ let $\langle\!\langle n \rangle\!\rangle = \langle x_0^1 \vee x_0^2, \ldots, x_{w-1}^1 \vee x_{w-1}^2 \rangle$ where $\langle\!\langle n_i \rangle\!\rangle = \langle x_0^i, \ldots, x_{w-1}^i \rangle$. To define $n_1 \ll n_2 = n$ let $\langle\!\langle n \rangle\!\rangle = \langle 0, \ldots, 0, x_0^1, \ldots, x_{w-1-n_2}^1 \rangle$ if $n_2 \in [0, w-1]$ otherwise $n = 0$ (which handles the normally unspecified case of when $n_2 < 0$). To define $n_1 + n_2 = n$ let $n \in R$ such that $n_1 + n_2 \equiv_{2^w} n$. Bitwise conjunction, rightshift and subtraction are analogously defined.

3.4 Semantic equations

The relational semantics of a guard $g \in \mathsf{Guard}$ is then given by $\mathcal{S}[\![g]\!] = \{\boldsymbol{a} \cdot \boldsymbol{a} \mid \boldsymbol{a} \in R^k \wedge \mathcal{G}[\![g]\!]\boldsymbol{a}\}$. The effect of a statement $s \in \mathsf{Stmt}$ is defined thus:

$$\begin{aligned}
\mathcal{S}[\![\mathsf{skip}]\!] &= \mathsf{Id} \\
\mathcal{S}[\![x_i := e]\!] &= \{\boldsymbol{a} \cdot \boldsymbol{a}[i \mapsto \mathcal{E}[\![e]\!]\boldsymbol{a}] \mid \boldsymbol{a} \in R^k\} \\
\mathcal{S}[\![s_1; s_2]\!] &= \mathcal{S}[\![s_1]\!] \circ \mathcal{S}[\![s_2]\!]
\end{aligned}$$

where $\mathcal{E}$ and $\mathcal{C}$ are defined:

$$\begin{aligned}
\mathcal{E}[\![x_i]\!]\boldsymbol{a} &= \boldsymbol{a}[i] \\
\mathcal{E}[\![n]\!]\boldsymbol{a} &= n \\
\mathcal{E}[\![-e]\!]\boldsymbol{a} &= r \in R && \text{where } r \equiv_{2^w} -(\mathcal{E}[\![e]\!]\boldsymbol{a}) \\
\mathcal{E}[\![e_1 \odot e_2]\!]\boldsymbol{a} &= (\mathcal{E}[\![e_1]\!]\boldsymbol{a}) \odot (\mathcal{E}[\![e_2]\!]\boldsymbol{a}) && \text{where } \odot \in \mathsf{bop}
\end{aligned}$$

$$\begin{aligned}
\mathcal{G}[\![e_1 \otimes e_2]\!]\boldsymbol{a} &= (\mathcal{E}[\![e_1]\!]\boldsymbol{a}) \otimes (\mathcal{E}[\![e_2]\!]\boldsymbol{a}) && \text{where } \otimes \in \mathsf{rop} && \mathcal{G}[\![\mathsf{true}]\!]\boldsymbol{a} = 1 \\
\mathcal{G}[\![g_1 \ominus g_2]\!]\boldsymbol{a} &= (\mathcal{G}[\![g_1]\!]\boldsymbol{a}) \ominus (\mathcal{G}[\![g_2]\!]\boldsymbol{a}) && \text{where } \ominus \in \mathsf{lop} && \mathcal{G}[\![\mathsf{false}]\!]\boldsymbol{a} = 0
\end{aligned}$$

The semantics of a program $P = \langle L, X, \ell_0, T\rangle$ is then defined as the set of smallest relations $\{r_\ell \in R^{2k} \mid \ell \in L\}$ such that $\mathsf{Id} \subseteq r_{\ell_0}$ and $r_{\ell_i} \circ \mathcal{S}[\![g]\!] \circ \mathcal{S}[\![s]\!] \subseteq r_{\ell_j}$ for all $\langle \ell_i, \ell_j, g, s\rangle \in T$. Each relation r_ℓ is finite and relates states at ℓ_0 to states at ℓ. The set of reachable states at ℓ is given by the composition $R^k \circ r_\ell$.

4 Symbolic Relational Semantics over Boolean Functions

This section shows how a flowchart program can be bit-blasted, that is, described symbolically with Boolean formulae. First, two disjoint sets of propositional variables are introduced: $\mathbf{X} = \{x_{i,j} \mid x_i \in X \wedge j \in [0, w-1]\}$ and $\mathbf{X}' = \{x'_{i,j} \mid x_i \in X \wedge j \in [0, w-1]\}$. Second, each relation $r_\ell \subseteq R^{2k}$ for $\ell \in L$, is encoded symbolically as a formula $f_\ell \in \mathbb{B}_{\mathbf{X}\cup\mathbf{X}'}$, where $\mathbb{B}_Y$ denotes the class of formulae that can be defined over the propositional variables Y. Third, operations over relations are simulated by operations over formulae.

4.1 Semantic machinery: encoding and composition

We introduce a map $\mathsf{sym} : \wp(R^{2k}) \to \mathbb{B}_{\mathbf{X}\cup\mathbf{X}'}$ that specifies the symbolic encoding:

$$\mathsf{sym}(r) = \bigvee\{ \bigwedge_{x_i \in X, j \in [0,w-1]} (x_{i,j} \leftrightarrow \langle\!\langle \boldsymbol{a}[i]\rangle\!\rangle[j] \wedge x'_{i,j} \leftrightarrow \langle\!\langle \boldsymbol{b}[i]\rangle\!\rangle[j]) \mid \boldsymbol{a}\cdot\boldsymbol{b} \in r\}$$

For example, $\mathsf{sym}(\mathsf{Id}) = \bigwedge_{x_i \in X, j\in[0,w-1]} x_{i,j} \leftrightarrow x'_{i,j}$. To handle expressions and guards, we introduce a variant of the encoding map $\mathsf{sym} : \wp(R^k) \to \mathbb{B}_{\mathbf{X}}$ defined thus $\mathsf{sym}(r) = \bigvee\{\bigwedge_{x_i\in X, j\in[0,w-1]} x_{i,j} \leftrightarrow \langle\!\langle \boldsymbol{a}[i]\rangle\!\rangle[j] \mid \boldsymbol{a} \in r\}$.

Different formulae can represent the same Boolean function, but if we identify equivalent formulae (implicitly working with equivalent classes of formulae), then functions and formulae can be used interchangeably. With this understanding, sym is bijective so that a relation $r \subseteq R^{2k}$ uniquely defines a function $f \in \mathbb{B}_{\mathbf{X}\cup\mathbf{X}'}$, and vice versa. Moreover, if $r_1, r_2 \in R^{2k}$ then $r_1 \subseteq r_2$ iff $\mathsf{sym}(r_1) \models \mathsf{sym}(r_2)$ where $\models$ denotes logical consequence.

To simulate composition with operations on formulae, let $r_1, r_2 \subseteq R^{2k}$ and suppose $\mathsf{sym}(r_1) = f_1$ and $\mathsf{sym}(r_2) = f_2$, hence $f_1, f_2 \in \mathbb{B}_{\mathbf{X}\cup\mathbf{X}'}$. A formula $f \in \mathbb{B}_{\mathbf{X}\cup\mathbf{X}'}$ such that $\mathsf{sym}(r_1 \circ r_2) = f$ can be derived as follows: Let $\mathbf{X}'' = \{x''_{i,j} \mid x_i \in X \wedge j \in [0, w-1]\}$ so that $\mathbf{X} \cap \mathbf{X}'' = \mathbf{X}' \cap \mathbf{X}'' = \emptyset$. Put $f'_1 = f_1 \wedge \bigwedge_{x_i\in X, j\in[0,w-1]} x'_{i,j} \leftrightarrow x''_{i,j}$ and $f'_2 = f_2 \wedge \bigwedge_{x_i\in X,j\in[0,w-1]} x_{i,j} \leftrightarrow x''_{i,j}$.

Define $f' = \exists_{\mathbf{X}'}(f'_1) \wedge \exists_{\mathbf{X}}(f'_2)$ and then put $f = \exists_{\mathbf{X}''}(f')$ where the operations $\exists_{\mathbf{X}'}(f'_1)$, $\exists_{\mathbf{X}}(f'_2)$ and $\exists_{\mathbf{X}''}(f')$ eliminate the variables $\mathbf{X}'$, $\mathbf{X}$ and $\mathbf{X}''$ from f'_1, f'_2 and f' respectively. Henceforth, denote $f_1 \circ_b f_2 = f$.

4.2 Semantic equations

Analogues of $\mathcal{S}[\![s]\!] \subseteq R^{2k}$, $\mathcal{E}[\![e]\!] \subseteq R^{2k}$ and $\mathcal{G}[\![g]\!] \subseteq R^{2k}$ over Boolean formulae, namely, $\mathcal{S}_b[\![s]\!] \in \mathbb{B}_{\mathbf{X} \cup \mathbf{X}'}$, $\mathcal{E}_b[\![e]\!] \in \mathbb{B}_{\mathbf{X} \cup \mathbf{X}'}$ and $\mathcal{G}_b[\![g]\!] \in \mathbb{B}_{\mathbf{X} \cup \mathbf{X}'}$ can now be constructed. The symbolic bit-level semantics for a guard $g \in \mathsf{Guard}$ is given by $\mathcal{S}_b[\![g]\!] = \mathcal{G}_b[\![g]\!] \wedge \bigwedge_{x_i \in X, j \in [0,w-1]}(x'_{i,j} \leftrightarrow x_{i,j})$ whereas the semantics for a statement $s \in \mathsf{Stmt}$ is given as follows:

$$\begin{aligned}
\mathcal{S}_b[\![\mathsf{skip}]\!] &= \textstyle\bigwedge_{x_i \in X, j \in [0,w-1]}(x'_{i,j} \leftrightarrow x_{i,j}) \\
\mathcal{S}_b[\![x_i := e]\!] &= \textstyle\bigwedge_{j \in [0,w-1]}(x'_{i,j} \leftrightarrow \mathcal{E}_b[\![e]\!][j]) \wedge \bigwedge_{x_k \in X \setminus \{x_i\}, j \in [0,w-1]}(x'_{k,j} \leftrightarrow x_{k,j}) \\
\mathcal{S}_b[\![s_1; s_2]\!] &= \mathcal{S}_b[\![s_1]\!] \circ_b \mathcal{S}_b[\![s_2]\!]
\end{aligned}$$

The second conjunct of $\mathcal{S}_b[\![g]\!]$ expresses that variables remain unchanged. Like before, $\mathcal{S}_b$ is defined in terms of $\mathcal{G}_b$ and $\mathcal{E}_b$. The semantic function $\mathcal{E}_b : \mathsf{Expr} \to [0, w-1] \to \mathbb{B}_{\mathbf{X}}$ takes an expression and a bit position and returns the value of that bit, expressed in terms of a Boolean formula. The function $\mathcal{G}_b : \mathsf{Guard} \to \mathbb{B}_{\mathbf{X}}$ takes a guard and returns its (Boolean) value. In what follows, $\boldsymbol{f}_1 \in \mathbb{B}^w_{\mathbf{X}}$ and $\boldsymbol{f}_2 \in \mathbb{B}^w_{\mathbf{X}}$ abbreviate $\mathcal{E}_b[\![e_1]\!]$ and $\mathcal{E}_b[\![e_2]\!]$, respectively.

$$\begin{aligned}
\mathcal{E}_b[\![x_i]\!][j] &= x_{i,j} \\
\mathcal{E}_b[\![n]\!][j] &= \langle\!\langle n \rangle\!\rangle[j] \\
\mathcal{E}_b[\![-e]\!][j] &= \mathcal{E}_b[\![e]\!][j] \oplus \vee_{j=0}^{i-1} \mathcal{E}_b[\![e]\!][j] \\
\mathcal{E}_b[\![e_1 + e_2]\!][j] &= \boldsymbol{f}_1[j] \oplus \boldsymbol{f}_2[j] \oplus \textstyle\bigoplus_{k=0}^{j-1}(\boldsymbol{f}_1[k] \wedge \boldsymbol{f}_2[k] \wedge \bigwedge_{m=k+1}^{j-1}(\boldsymbol{f}_1[m] \oplus \boldsymbol{f}_2[m])) \\
\mathcal{E}_b[\![e_1 - e_2]\!][j] &= \mathcal{E}_b[\![e_1 + (-e_2)]\!][j] \\
\mathcal{E}_b[\![e_1 \,\&\, e_2]\!][j] &= \boldsymbol{f}_1[j] \wedge \boldsymbol{f}_2[j] \\
\mathcal{E}_b[\![e_1 \mid e_2]\!][j] &= \boldsymbol{f}_1[j] \vee \boldsymbol{f}_2[j]
\end{aligned}$$

$$\begin{aligned}
\mathcal{G}_b[\![g_1 = g_2]\!] &= \textstyle\bigwedge_{i=0}^{w-1}(\boldsymbol{f}_1[j] \leftrightarrow \boldsymbol{f}_2[j]) & \mathcal{G}_b[\![\mathsf{true}]\!] &= 1 \\
\mathcal{G}_b[\![g_1 \neq g_2]\!] &= \neg(\mathcal{G}_b[\![g_1 = g_2]\!]) & \mathcal{G}_b[\![\mathsf{false}]\!] &= 0 \\
\mathcal{G}_b[\![g_1 < g_2]\!] &= \neg(\mathcal{G}_b[\![g_2 \leq g_1]\!]) \\
\mathcal{G}_b[\![g_1 \leq g_2]\!] &= (\boldsymbol{f}_1[w-1] \wedge \neg \boldsymbol{f}_2[w-1]) \qquad\qquad \vee \\
&\quad \textstyle\bigvee_{j=0}^{w-2}(\neg \boldsymbol{f}_1[j] \wedge \boldsymbol{f}_2[j] \wedge \bigwedge_{k=j+1}^{w-1} \boldsymbol{f}_1[k] \leftrightarrow \boldsymbol{f}_2[k]) \\
\mathcal{G}_b[\![g_1 \wedge g_2]\!] &= (\mathcal{G}_b[\![g_1]\!]) \wedge (\mathcal{G}_b[\![g_2]\!]) \\
\mathcal{G}_b[\![g_1 \vee g_2]\!] &= (\mathcal{G}_b[\![g_1]\!]) \vee (\mathcal{G}_b[\![g_2]\!])
\end{aligned}$$

The formula for $e_1 + e_2$ is derived by considering a cascade of full adders with w carry bits $\boldsymbol{c}$. Then $\mathcal{G}_b[\![e_1 + e_2]\!][j] = (\boldsymbol{f}_1[j] \oplus \boldsymbol{f}_2[j] \oplus \boldsymbol{c}[j]) \wedge (\neg \boldsymbol{c}[0]) \wedge f$ where $f = \bigwedge_{j=1}^{w-1} \boldsymbol{c}[j] \leftrightarrow ((\boldsymbol{f}_1[j-1] \wedge \boldsymbol{f}_2[j-1]) \vee (\boldsymbol{f}_1[j-1] \wedge \boldsymbol{c}[j-1]) \vee (\boldsymbol{f}_2[j-1] \wedge \boldsymbol{c}[j-1]))$. By the eliminating the $\boldsymbol{b}$ variables and simplifying, the above formula is obtained. The equation for $\mathcal{E}_b[\![e_1 \ll e_2]\!][j]$ can be straightforwardly defined with $w + 2$ cases that handle the various classes of shift. Likewise for $\mathcal{E}_b[\![e_1 \gg e_2]\!][j]$. Both equations are omitted for brevity.

4.3 Semantic equivalence

The semantics of a program $P = \langle L, X, \ell_0, T\rangle$ can then be prescribed as the set of least Boolean functions $\{f_\ell \in \mathbb{B}_{\mathbf{X}\cup\mathbf{X}'} \mid \ell \in L\}$ such that $\mathsf{sym}(\mathsf{Id}) \models f_{\ell_0}$ and $f_{\ell_i} \circ_b \mathcal{S}[\![g]\!] \circ_b \mathcal{S}[\![s]\!] \models f_{\ell_j}$ for all $\langle \ell_i, \ell_j, g, s\rangle \in T$. The semantics can be equivalently stated as the least fixed point of a system of equations of the form $f_{\ell_j} = \vee\{f_{\ell_i} \circ_b \mathcal{S}[\![g]\!] \circ_b \mathcal{S}[\![s]\!] \mid \langle \ell_i, \ell_j, g, s\rangle \in T\}$, where the equation for f_{ℓ_0} includes the additional disjunction $\mathsf{sym}(\mathsf{Id})$. The semantics of the previous section can likewise be expressed as a fixed point. This allows induction to be applied to argue $\mathsf{sym}(r_\ell) = f_\ell$ for all $\ell \in L$. However this itself requires the use of induction to show $\mathsf{sym}(\mathcal{G}[\![g]\!]) = \mathcal{G}_b[\![g]\!]$ for all $g \in \mathsf{Guard}$ and $\mathsf{sym}(\mathcal{E}[\![e]\!][j]) = \mathcal{E}_b[\![e]\!][j]$ for all $s \in \mathsf{Stmt}$ and $j \in [0, w-1]$. The key point is that the semantics of this section is equivalent to that introduced previously in that $\mathsf{sym}(r_\ell) = f_\ell$ for all $\ell \in L$. The difference is the latter semantics provides a basis suitable for deriving transition systems over congruences.

5 Abstract Relational Semantics over Congruences

Abstract interpretation [3] is a systematic way of deriving invariants by considering all paths through a program. Each atomic operation over the concrete data values is simulated with an abstract version manipulating abstract data values drawn from an abstract domain. The semantics of a transition $t = \langle \ell_i, \ell_i, g, s\rangle$ is expressed by the Boolean function $f = \mathcal{S}[\![g]\!] \circ_b \mathcal{S}[\![s]\!] \in \mathbb{B}_{\mathbf{X}\cup\mathbf{X}'}$, which permits t to be viewed as a single atomic operation. Once the modulo m is chosen, congruent closure provides a way to map f to a system of congruence equations that define an abstract version of t.

5.1 Deriving abstract transitions

Since f is Boolean formula on $\mathbf{X}\cup\mathbf{X}'$, we let $\mathsf{Aff}_m^{\mathbf{X}\cup\mathbf{X}'}$ denote the set of systems of equations modulo m that can be defined over $\mathbf{X}\cup\mathbf{X}'$. Thus if $c \in \mathsf{Aff}_m^{\mathbf{X}\cup\mathbf{X}'}$ then c is a system of implicitly conjoined equations (rather than a single equation). Then the abstraction map $\alpha_m^{2kw} : \wp(\mathbb{B}^{2kw}) \to \mathsf{Aff}_m^{2kw}$ can be extended to $\alpha_m : \mathbb{B}_{\mathbf{X}\cup\mathbf{X}'} \to \mathsf{Aff}_m^{\mathbf{X}\cup\mathbf{X}'}$ in the natural way. This leads to the notion of an abstract flowchart program $\langle L, X, \ell_0, T'\rangle$ where $T' = \{\langle \ell_i, \ell_j, \alpha_m(\mathcal{S}[\![g]\!] \circ_b \mathcal{S}[\![s]\!])\rangle \mid \langle \ell_i, \ell_j, g, s\rangle \in T\}$. Enlarging m preserves more of f at the expense of a more complicated abstract program. Note how $\alpha_m(\mathcal{S}[\![g]\!] \circ_b \mathcal{S}[\![s]\!])$ summarises both g and s (even when s is itself compound) with a *single* system of congruences.

Example 8. Observe $\alpha_m(\mathsf{sym}(\mathsf{Id})) = \bigwedge_{x_i \in X, j \in [0,w-1]} x_{i,j} \equiv_m x'_{i,j}$.

Example 9. Consider again the parity program and suppose $w = 16$. Then computing $\alpha_m(\mathcal{S}[\![g]\!] \circ_b \mathcal{S}[\![s]\!])$ for $m = 2$ and each transition $\langle \ell_i, \ell_j, g, s\rangle \in T$ given in Example 6, we derive the abstract transitions $t'_1 = \langle \ell_0, \ell_1, c_1\rangle$, $t'_2 = \langle \ell_1, \ell_1, c_2\rangle$

and $t'_3 = \langle \ell_1, \ell_2, c_3 \rangle$ where

$$c_1 = \begin{cases} (\wedge_{i=0}^{15} p'_i \equiv_2 0) \ \wedge \\ (\wedge_{i=0}^{15} y'_i \equiv_2 x_i) \wedge \\ (\wedge_{i=0}^{15} x'_i \equiv_2 x_i) \end{cases} c_2 = \begin{cases} p_0 + p'_0 \equiv_2 1 \quad \wedge \\ (\wedge_{i=1}^{15} p_i \equiv_2 p'_i) \wedge \\ (\wedge_{i=0}^{15} x_i \equiv_2 x'_i) \wedge \\ y'_0 \equiv_2 0 \qquad \wedge \\ 1 + \sum_{i=1}^{15} y'_i \equiv_2 \sum_{i=0}^{15} y_i \end{cases} \quad c_3 = \begin{cases} (\wedge_{i=0}^{15} p'_i \equiv_2 p_i) \wedge \\ (\wedge_{i=0}^{15} x'_i \equiv_2 x_i) \wedge \\ (\wedge_{i=0}^{15} y_i \equiv_2 0) \ \wedge \\ (\wedge_{i=0}^{15} y'_i \equiv_2 0) \end{cases}$$

Of course, such translation cannot be performed manually and therefore we have written a Java application that derives abstract transition systems. It applies the congruent closure algorithm presented in section 2 and uses the MiniSat solver through the Kodkod Java bindings. The most complicated system, c_2, requires 97 SAT instances to be derived taking 3s overall. Such a modulus is sufficient to verify the correctness of parity, but in general, the behaviour of the program is unknown and then it is more appropriate to use a modulo that reflects the size of machine words. Using a modulo of, say, 2^{32} does not increase the number of SAT instances but does double the time required to compute c_2. Interestingly, if c_2 is derived without the guard $y \neq 0$ (which accidently happened when this experiment was conducted), then the equation $1 + \sum_{i=0}^{15} y'_i \equiv_2 \sum_{i=0}^{15} y_i$ cannot be inferred. Note too that $y'_0 \equiv_2 0$ asserts that the low bit of y is reset.

Example 10. Let us revisit the word reversal program of Example 7 where $w = 16$. Thus put $m = 2^{16}$. Then the abstract flowchart has a single transition $\langle \ell_0, \ell_1, c \rangle$ where $c = \bigwedge_{i=0}^{15} (x'_i \equiv_{2^{16}} x_i \wedge y'_{15-i} \equiv_{2^{16}} x_i)$. This can be derived in 0.8s and requires 33 calls to the SAT solver. Note how c precisely summarises program behaviour, despite the use of devious bit-twiddling operations.

5.2 Applying abstract transitions

Once an abstract transition system has been derived, existing techniques can be used to compute congruences that hold at each $\ell \in L$. Efficient algorithms have been reported elsewhere [1, 8, 11] for checking entailment $c_1 \models c_2$, calculating join $c_1 \sqcup c_2$, and eliminating variables $\exists_Y(c_1)$ for $c_1, c_2 \in \mathsf{Aff}_m^{\mathbf{X} \cup \mathbf{X}'}$. We make no contribution in this area, but to keep the paper self-contained, we present a semantics for abstract flowchart programs which specifies a program analysis. The semantics is formulated in terms of a composition operator, $\circ_c$, that mirrors $\circ_b$. To define this operator, let $c_1, c_1 \subseteq \mathsf{Aff}_m^{\mathbf{X} \cup \mathbf{X}'}$. Put $c'_1 = c_1 \wedge \bigwedge_{x_i \in X, j \in [0, w-1]} x'_{i,j} \equiv_m x''_{i,j}$ and $c'_2 = c_2 \wedge \bigwedge_{x_i \in X, j \in [0, w-1]} x_{i,j} \equiv_m x''_{i,j}$, and then proceed by analogy with the $\circ_b$ construction to define $c_1 \circ_c c_2 = c$.

The semantics of an abstract flowchart program $\langle L, X, \ell_0, T' \rangle$ can then be defined as the set of least congruence systems $\{c_\ell \in \mathsf{Aff}_m^{\mathbf{X} \cup \mathbf{X}'} \mid \ell \in L\}$ such that $\alpha_m(\mathsf{sym}(\mathsf{Id})) \models c_{\ell_0}$ and $c_{\ell_i} \circ_c c \models c_{\ell_j}$ for all $\langle \ell_i, \ell_j, c \rangle \in T'$. As before, the semantics can be equivalently stated as the least fixed point of a system of equations, which leads to an iterative approach for computing congruence invariants:

```
$\ell_0$: assume($0 \leq n$);
     $x := 0$; $y := 0$;
$\ell_1$: while ($x < n$)
         $y := y + 2$;
         $x := x + 1$;
$\ell_2$: skip
```

```
$\ell_0$: assume($0 \leq n$);
     $x := 0$; $y := 0$; $\delta := n - x$;
$\ell_1$: while ($0 < \delta$)
         $y := y + 2$;
         $x := x + 1$;
         $\delta := n - x$;
$\ell_2$: skip
```

Fig. 4. Inferring $x \leq n$ by using a witness variable δ

Example 11. Returning to Example 9, the invariants $c_{\ell_0}, c_{\ell_1}, c_{\ell_2}$ can be computed iteratively since they are the least solutions to the equations: $c_{\ell_0} = \alpha_m(\mathsf{sym}(\mathsf{Id}))$, $c_{\ell_1} = (c_{\ell_0} \circ_c c_1) \sqcup (c_{\ell_1} \circ_c c_2)$ and $c_{\ell_2} = c_{\ell_1} \circ_c c_3$. To solve these equations, first assign $c_{\ell_0} = c_{\ell_1} = c_{\ell_2} = \mathsf{false}$ where false is the unsatisfiable congruence system. Application of the first equation then yields $c_{\ell_0} = (\wedge_{j\in[0,15]} p'_j \equiv_2 p_j) \wedge (\wedge_{j\in[0,15]} x'_j \equiv_2 x_j) \wedge (\wedge_{j\in[0,15]} y'_j \equiv_2 y_j)$. Thereafter c_{ℓ_0} is stable. For brevity, let $c = (\wedge_{j\in[1,15]} p'_j \equiv_2 p_j) \wedge (\wedge_{j\in[0,15]} x'_j \equiv_2 x_j)$. An application of the second equation gives $c_{\ell_1} = c \wedge p'_0 \equiv_2 0 \wedge (\wedge_{j\in[0,15]} y'_i \equiv_2 x_i)$. Then $c_{\ell_1} \circ_c c_2 = c \wedge (p'_0 \equiv_2 1) \wedge (y_0 \equiv_2 0) \wedge (1 \equiv_2 \sum_{j\in[0,15]} x_i - \sum_{j\in[1,15]} y'_i)$, thus reapplying the second equation gives $c_{\ell_1} = c \wedge (y_0 \equiv_2 0) \wedge (p'_0 \equiv_2 \sum_{j\in[0,15]} x_i - \sum_{j\in[1,15]} y'_i)$. Thereafter c_{ℓ_1} is also stable. Finally, the third equation then gives $c_{\ell_2} = c \wedge (\wedge_{j\in[0,15]} y_0 \equiv_2 0) \wedge (\wedge_{j\in[0,15]} y'_0 \equiv_2 0) \wedge (p'_0 \equiv_2 \sum_{j\in[0,15]} x_i)$. Then c_{ℓ_2} is stable too, and the fixed point has been reached. Correctness of parity follows from the invariant $p'_0 \equiv_2 \sum_{j\in[0,15]} x_i$ that holds at ℓ_2.

6 Transformation for Range Information

Consider the program in the left-hand-side of Figure 4 where n, x and y are signed w bit variables. Bit-level congruences cannot directly represent the inequality $(\sum_{i=0}^{w-2} 2^i x_i) - 2^{w-1} x_{w-1} \leq (\sum_{i=0}^{w-2} 2^i n_i) - 2^{w-1} n_{w-1}$ that holds at ℓ_2, which is crucial for inferring that x and n are bit-wise equivalent at ℓ_2.

6.1 Adding witness variables

However, bit-level congruences can express the non-negativity of a variable, which suggests augmenting the program with a variable δ that witnesses the non-negativity of $n - x$. The program in the right-hand-side of the figure illustrates the tactic, and the flow-graph for this program is given below:

$$\begin{array}{l} \langle \ell_0, \ell_1, 0 \leq n, x := 0; y := 0; \delta := n - x \rangle \\ \langle \ell_1, \ell_1, 0 < \delta, \ y := y + 2; x := x + 1; \delta := n - x \rangle \\ \langle \ell_1, \ell_2, \delta \leq 0, \ \mathsf{skip} \rangle \end{array}$$

Generating the abstract transitions as previously described gives $t'_1 = \langle \ell_0, \ell_1, c_1 \rangle$ and $t'_2 = \langle \ell_1, \ell_1, c_2 \rangle$ where

$$c_1 = \bigwedge \left\{ \begin{array}{llll} x'_0 \equiv_{16} 0, & x'_1 \equiv_{16} 0, & x'_2 \equiv_{16} 0, & x'_3 \equiv_{16} 0, \\ y'_0 \equiv_{16} 0, & y'_1 \equiv_{16} 0, & y'_2 \equiv_{16} 0, & y'_3 \equiv_{16} 0, \\ n'_0 \equiv_{16} n_0, & n'_1 \equiv_{16} n_1, & n'_2 \equiv_{16} n_2, & n'_3 \equiv_{16} n_3, \\ \delta'_0 \equiv_{16} n_0, & \delta'_1 \equiv_{16} n_1, & \delta'_2 \equiv_{16} n_2, & \delta'_3 \equiv_{16} n_3 \end{array} \right\}$$

$$c_2 = \bigwedge \left\{ \begin{array}{lll} \delta_0 + \delta'_0 \equiv_{16} 1, & \delta_1 + 2\delta_2 \equiv_{16} \delta'_0 + \delta'_1 + 2\delta'_2, & 0 \equiv_{16} \delta_3, \\ 0 \equiv_{16} \delta'_3, & n'_0 \equiv_{16} n_0, & n'_1 \equiv_{16} n_1, \\ n'_2 \equiv_{16} n_2, & n_3 \equiv_{16} n'_3, & 1 \equiv_{16} x_0 + x'_0, \\ y_0 \equiv_{16} y'_0, & 1 \equiv_{16} y_1 + y'_1, & \\ \multicolumn{3}{l}{\delta'_0 + 8(n_3 + x_3 + x'_2) + 1 \equiv_{16}} \\ \multicolumn{3}{l}{\qquad n_0 + 2(n_1 + x_1 - \delta'_1) + 4(n_2 + x_2 - \delta'_2 - x'_1) + 3x_0,} \\ \multicolumn{3}{l}{2x'_0 + 2x'_1 + 4x'_2 \equiv_{16} 2x_1 + 4x_2 + 8x_3 + 8x'_3 + 2,} \\ \multicolumn{3}{l}{4y_2 + 4 \equiv_{16} 8y_3 + 4y'_1 + 4y'_2 + 8y'_3} \end{array} \right\}$$

The width is set to $w = 4$ merely for presentational purposes. The key point is that c_2 asserts that $0 \equiv_{16} \delta'_3$ indicating that δ is non-negative at the end of the loop, as required. In general, a loop condition that is a single inequality $e_1 < e_2$ (resp. $e_1 \leq e_2$) can be replaced with $0 < \delta$ where $\delta = e_2 + (-e_1)$ (resp. $\delta = e_2 + (-e_1) + 1$) so the transformation can be automated.

6.2 Decomposing guards

Interestingly, introducing a witness variable is not by itself sufficient to deduce $x_i \equiv_{2^w} n_i$ for all $i \in [0, w-1]$ at ℓ_2. The semantics of abstract transition systems can be applied to derive:

$$c_{\ell_1} = \bigwedge \left\{ \begin{array}{l} \delta_3 \equiv_{16} 0, \\ y_0 \equiv_{16} 0, \quad y_1 \equiv_{16} x_0, \quad y_2 \equiv_{16} x_1, \quad y_3 \equiv_{16} x_2, \\ \sum_{i=0}^{3} 2^i \delta_i + \sum_{i=0}^{3} 2^i x_i \equiv_{16} \sum_{i=0}^{3} 2^i n_i \end{array} \right\}$$

which, although unexpected, is not in error since $8\delta_3 \equiv_{16} -8\delta_3$ and likewise for x_3 and n_3. But to infer the equivalence of x and n at ℓ_2 it is necessary to additionally impose the constraint $\delta \leq 0$. However, such a constraint is not captured in the abstract transition $t_3 = \langle \ell_1, \ell_2, c \rangle$ where

$$c = \bigwedge \left\{ \begin{array}{llll} n_0 \equiv_{16} n'_0, & n_1 \equiv_{16} n'_1, & n_2 \equiv_{16} n'_2, & n_3 \equiv_{16} n'_3, \\ x_0 \equiv_{16} x'_0, & x_1 \equiv_{16} x'_1, & x_2 \equiv_{16} x'_2, & x_3 \equiv_{16} x'_3, \\ y_0 \equiv_{16} y'_0, & y_1 \equiv_{16} y'_1, & y_2 \equiv_{16} y'_2, & y_3 \equiv_{16} y'_3, \\ \delta_0 \equiv_{16} \delta'_0, & \delta_1 \equiv_{16} \delta'_1, & \delta_2 \equiv_{16} \delta'_2, & \delta_3 \equiv_{16} \delta'_3 \end{array} \right\}$$

since c does not preserve any information pertaining to the $\delta \leq 0$ constraint. However, observe that $\delta \leq 0$ holds iff $\delta < 0 \vee \delta = 0$ and both $\delta < 0$ and $\delta = 0$ can represented with bit-level congruences. This suggests transforming the third transition into $\langle \ell_1, \ell_2, \delta < 0, \mathsf{skip} \rangle$ and $\langle \ell_1, \ell_2, \delta = 0, \mathsf{skip} \rangle$. Then these rules respectively yield the abstract transitions $\langle \ell_1, \ell_2, c_1 \rangle$ and $\langle \ell_1, \ell_2, c_2 \rangle$ where $c_1 = c \wedge (\delta_3 \equiv_{16} 1)$ and $c_2 = c \wedge (\wedge_{j \in [0,3]} \delta_j \equiv_{16} 0)$. Only the second transition is applicable since c_{ℓ_1} asserts $\delta_3 \equiv_{16} 0$. Thus the following constraints hold at ℓ_2:

$$c_{\ell_2} = \bigwedge \left\{ \begin{array}{llll} \delta_0 \equiv_{16} 0, & \delta_1 \equiv_{16} 0, & \delta_2 \equiv_{16} 0, & \delta_3 \equiv_{16} 0, \\ y_0 \equiv_{16} 0, & y_1 \equiv_{16} x_0, & y_2 \equiv_{16} x_1, & y_3 \equiv_{16} x_2, \\ x_0 \equiv_{16} n_0, & x_1 \equiv_{16} n_1, & x_2 \equiv_{16} n_2, & x_3 \equiv_{16} n_3 \end{array} \right\}$$

Thus, even if a guard is not amenable to an exact bit-level representation, then it does not imply its transition cannot be decomposed to finesse the problem.

7 Concluding Discussion

We have shown how a SAT/SMT solver can be employed to derive abstract transition systems over linear congruences. The resulting invariants can express congruence relationships amongst the individual bits that comprise variables and, as a consequence, the abstract transition systems can calculate relationships even at the granularity of bit-twiddling. The advantage of the scheme presented in this paper is that SAT solving is confined to the derivation of the abstract transition system; only linear operations of polynomial complexity are required thereafter. We also proposed program transformations to improve the analysis through the use of witness variables that can help observe range information.

One may wonder how the efficiency of congruence closure depends on the SAT (or SMT) engine. Thus, as an experiment, MiniSat was replaced with ZChaff. This had little discernible impact on the overall time to compute the closure (transfer functions) for transition relations that formalised a number of bit-twiddling algorithms given in Warren's book [16]. Rather surprisingly, only a modest slow-down was found when MiniSat was replaced by SAT4J which is a solver that is implemented in Java itself. To understand why, consider Wegner's fast bit counting algorithm [16]. Deriving the transfer functions for the 16 bit version involves 98 SAT instances. Solving the instances takes 0.5s overall but 5.6s is spent computing all the joins. In this example, many joins involve matrices of size 129×129 arising from a relation over 129 bits. The effect is more pronounced for a 32 bit version of the algorithm which has one relation over 259 bits. To solve the 195 SAT instances requires 11.4s overall, but the join operation takes up 149.5s, partly because it manipulates matrices of size 259×259. The bias towards join is least in an example that computes the sign operation by bit-twiddling. For the 32 bit version of the algorithm, the timings are 0.1s and 0.4s for the SAT and join components. We conclude that SAT is not the bottleneck, and that future effort should focus on how to exploit the sparsity of the matrices that arise.

Acknowledgements This work was funded by the EPSRC project EP/F012896, a Royal Society Industrial Fellowship, and a Tewkesbury Fellowship that enabled the first author to visit the University of Melbourne. We thank Tom Reps for helpful discussions on the notion of a best symbolic transformer.

References

1. R. Bagnara, K. Dobson, P. M. Hill, M. Mundell, and E. Zaffanella. Grids: A domain for analyzing the distribution of numerical values. In G. Puebla, editor, *International Symposium on Logic-based Program Synthesis and Transformation*, volume 4407 of *LNCS*, pages 219–235. Springer-Verlag, 2006.
2. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176. Springer-Verlag, 2004.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM, 1977.
4. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, LNCS, pages 72–83. Springer-Verlag, 1997.
5. P. Granger. Static analyses of linear congruence equalities among variables of a program. In *Joint International Conference on Theory and Practice of Software Development*, volume 493 of *LNCS*, pages 167–192. Springer-Verlag, 1991.
6. J. M. Howe and A. King. Logahedra: a new weakly relational domain. In Z. Liu and A. P. Ravn, editors, *Automated Technology for Verification and Analysis*, LNCS. Springer-Verlag, 2009.
7. D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis*, pages 14–25. ACM, 2000.
8. A. King and H. Søndergaard. Inferring congruence equations using SAT. In A. Gupta and S. Malik, editors, *Computer Aided Verification*, volume 5123 of *LNCS*, pages 281–293. Springer-Verlag, 2008.
9. V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 229–244. Springer-Verlag, 2009.
10. D. Monniaux. Automatic modular abstractions for linear constraints. In *Principles of Programming Languages*, pages 140–151. ACM, 2009.
11. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *ACM Transactions on Programming Languages and Systems*, 29(5), Aug. 2007. Article 29.
12. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.
13. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Verification, Model-Checking and Abstract Interpretation*, volume 2937 of *LNCS*, pages 3–25. Springer-Verlag, 2004.
14. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In R. Cousot, editor, *Verification, Model Checking and Abstract Interpretation*, volume 3385 of *LNCS*, pages 21–47. Springer-Verlag, 2005.
15. A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In C. Hankin, editor, *Static Analysis*, volume 3672 of *LNCS*, pages 336–351. Springer-Verlag, 2005.
16. H. S. Warren Jr. *Hacker's Delight.* Addison Wesley, 2003.

A Proof Appendix

Proposition 1. Let $S \subseteq \mathbb{B}^k$ and $m \in \mathbb{N}$, and let $[A|\boldsymbol{b}] = \text{closure}(S, m)$. Then $\mathsf{aff}^k_m(S) = \{\boldsymbol{x} \in \mathbb{Z}^k_m \mid A\boldsymbol{x} \equiv_m \boldsymbol{b}\}$.

Proof. (Sketch.) Let $r = \text{rows}([A|\boldsymbol{b}])$. Let e_j be the equation $\Sigma^k_{i=1} a_{j,i}x_i \equiv_m b_i$, where $a_{j,1}, a_{j,2}, \ldots, a_{j,k}, b_i = \text{row}([A|\boldsymbol{b}], j)$. The following invariant holds at the entry and exit of the body of the while loop, together with the fact that $[A|\boldsymbol{b}]$ is in upper triangular form:

$$\mathsf{aff}^k_m(S) = (\mathsf{aff}^k_m(S) \sqcup \{e_j \mid 1 \leq j \leq r - i\}) \cup \{e_j \mid r - i < j \leq r\} \qquad (1)$$

This is clearly satisfied immediately before the while loop.

To see that the loop body preserves (1), note that by the invariant, every $\boldsymbol{y} \in S$ satisfies $\bigwedge\{e_j \mid r - i < j \leq r\}$. There are two cases:

1. Assume some $\boldsymbol{x} \in S$ fails to satisfy e_{r-i}. The strongest set of congruence equations that will satisfy $\boldsymbol{x}$ as well as every solution to $[A|\boldsymbol{b}]$ is $(A\boldsymbol{x} \equiv_m \boldsymbol{b} \sqcup e_{r-i})$. This join on matrices can be calculated as in [8], which exploits the fact that the input matrices are in upper triangular form. The result can be brought into upper triangular form as well—and algorithm for this is provided by Müller-Olm and Seidl [11]. It remains to show that the last i equations remain unchanged by the join operation.
   Let $[C|c]$ be the matrix corresponding to $e_1, \ldots, e_{r-i}$, and let $[E|e]$ be the matrix corresponding to $e_{r-i+1}, \ldots, e_r$. Then $\langle a_1, \ldots, a_k, b\rangle$ is the last row of $[C|c]$, and the chosen $\boldsymbol{x}$ satisfies, by the invariant, $E\boldsymbol{x} = \boldsymbol{e}$, but not e_{r-i}. Note that both $[C|c]$, $[E|e]$ and $\left[\begin{array}{c|c} C & \boldsymbol{c} \\ E & \boldsymbol{e} \end{array}\right] \in \mathbb{Z}_{r\times(k+1)}$ are all in upper triangular form. The effect of the join is to leave $[E|e]$ unchanged and to remove the last row of $[C|c]$. For each remaining row of $[C|c]$, the index of the leading entry remains unchanged. In other words, the upper triangular form of $\left[\begin{array}{c|c} C & \boldsymbol{c} \\ E & \boldsymbol{e} \end{array}\right] \sqcup [I \mid \boldsymbol{x}]$ is of the form $\left[\begin{array}{c|c} D & \boldsymbol{d} \\ E & \boldsymbol{e} \end{array}\right]$. Hence (1) is preserved.
2. Otherwise, each $\boldsymbol{x} \in S$ satisfies e_{r-i}, and i is incremented. This clearly preserves (1).

Finally, upon exit from the while loop, (1), together with $i \geq r$, entails $\mathsf{aff}^k_m(S) = \{\boldsymbol{x} \in \mathbb{Z}^k_m \mid A\boldsymbol{x} \equiv_m \boldsymbol{b}\}$, as desired.