Computer Science at Kent

Inferring Congruence Equations with SAT

Andy King and Harald Søndergaard

Technical Report No. 1-08
January 2008

Abstract We propose a new approach for deriving invariants that are systems of congruence equations where the modulo is a power of 2. The technique is an amalgam of SAT-solving, where a propositional formula is used to encode the semantics of a basic block, and abstraction, where the solutions to the formula are systematically combined and summarised as a system of congruence equations. The resulting technique is more precise than existing congruence analyses since a single optimal transfer function is derived for a basic block as a whole.

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

1 Introduction

Applications in compilation, optimisation and verification have motivated analyses that infer linear equality relationships [7, 9, 13] or linear congruence relationships [1, 6, 14] that hold between the variables of a program. For each point in a program, the former analyses discover systems of affine constraints of the form $\sum_{i=1}^{n} c_i x_i = d$ where $c_1, \ldots, c_n, d \in \mathbb{Z}$ and $x_1, \ldots, x_n$ are the program variables. The latter infer systems of congruence constraints of the form $(\sum_{i=1}^{n} c_i x_i) \mod m = d$ where $m \in \mathbb{Z}$ is some modulus. These analyses accurately trace relationships between variables when the assignments that arise in a program can be modeled with a linear transformation. But this precludes meaningful analysis of programs that use bitwise operators; whether written in Java, C, or assembly language. The extreme approach of treating all operands of such operators as sequences of named bits, to track all bit interrelations, does not appear attractive, owing to the large number of Boolean variables involved. However, we show that a mixture of congruence analysis and Boolean reasoning does appear to be both feasible and able to generate bit-level invariants of great precision.

We draw inspiration from the domain of congruent equations modulo 2^w [14] and the affinity between this domain and the finite-nature of the underlying computer arithmetic, to propose an extreme-precision analysis which produces tight invariants for programs with non-linear, including bitwise, operations. The idea is to express the relationship between the bits in input variables and the bits of output variables for each basic block. This technique is not new within itself [8] and programs are now routinely reduced to very large systems of propositional constraints in bounded model checking [3, 19]. Our main novel contribution is in the use of a SAT solver to incrementally compute a summary (affine relaxation) of the output variables, given a summary for the input variables. This new approach is capable of discovering invariants even for programs that apply bit-twiddling; programs that have thus far thwarted automatic analysis. Summaries that are systems of congruence equations modulo 2^w naturally fit into this mix of model checking and abstract interpretation because (technically) their ascending chain property constrains the number of times the SAT-solver can be reapplied and (philosophically) the propositional encoding also makes assumptions about the finite, modulo-nature of computer arithmetic. As in conventional abstract interpretation, the summaries enable all paths through the program to be considered systematically, without enforcing a bound on depth to which loops are explored. The approach to analysis is attractive because, quite apart from providing a bridging result between SAT solving

and analysis, it can compute an optimal transfer function for a whole basic block, even when the block contains non-linear assignments. This is the key to the improved precision.

Since we draw together a number of threads in program analysis, for clarity, we summarise the contributions as follows:

- We show how to repeatedly apply a SAT solver to summarise all the solutions of a propositional formula as a system of congruence equations;
- We show how this approach to congruence analysis derives maximal information from each basic block (provided that the block is exactly translated into a system of propositional constraints);
- Computing a summary involves, among other things, computing the join of two systems of congruence equations. We show how the join can be computed without recourse to the generator representation [1, 6, 14].

The paper is structured as follows: Section 2 illustrates the key ideas of the analysis using a worked example, demonstrating how a SAT-solver can be interleaved with a relaxation technique to compute summaries of basic blocks. The outline is quite detailed and hence quite lengthy—the aim is to provide a good understanding of the method without providing all the formal detail. Section 3 defines the programming model in more detail, giving syntax and semantics. Section 4 provides details left out in the worked example, explaining how bit-precise transfer functions are calculated. Section 5 shows how the join of two congruence systems can be summarised merely by syntactically rearranging matrices and computing a triangular form, an important contribution of the paper. Section 6 discusses the relationship with the wider literature, and Section 7 concludes.

2 Outline of the Method

In 1960 Peter Wegner [18] reported a fast bit counting algorithm. Expressed in the language C, the method counts the number of 1-bits in a word `x`, leaving the result in a variable `c`, as follows:

```
y = x; for (c = 0; y; c++) y &= y - 1;
```

Since the method is rather devious (and the explanation pre-dates the invariant assertion principle), one may want to derive an invariant that aids

understanding of the code. This is challenging because the bit-twiddling cannot be modeled with a conventional affine assignment [14], that is, `y` is not updated with a value that is a linear combination of the values of the program variables. Furthermore, modeling the update as an assignment to an arbitrary value (a so-called non-deterministic assignment [14]) is too crude to derive a useful loop invariant.

One might think that these problems are insurmountable but we show that an invariant can be derived by modeling non-linear assignments as exact operations on sequences of bits and then computing optimal congruence abstractions for a composition of bit operations. The last point warrants elaboration: numeric analyses are usually presented in terms of programs which have already been abstracted through the use of assignments that are either affine or non-deterministic. This is adequate when working at the granularity of whole numbers but the best congruent abstraction of a bit-level operation, let alone a composition of them, is somewhat less obvious. We systematise the computation of an optimal abstraction and integrate this into the analysis itself.

In the rest of this section we sketch the approach:

1. A local, bit-precise transfer function is established for each basic block.
2. These Boolean functions are then used to build a set of recursive dataflow equations, expressing the program's overall runtime behaviour.
3. In the context of a finite set of w-bit variables, a closed form of the dataflow equations can be derived using Kleene iteration. In practice, however, this iteration may need to be interleaved with steps to relax constraints, and we propose a suitable relaxation to congruence equations.

2.1 Representing bit-level semantics without abstraction

It is possible to express the semantics of the basic blocks of a program, even to the bit-level, using Boolean constraints that relate the bits of its inputs to those of its outputs. But the problem is how to do so, retain tractability, and derive *loop invariants*, that is, not just explore loops to a fixed depth [8].

Figure 1 recasts Wegner's code as a flow diagram. There are three basic blocks in the program: the initial code '$y := x$; $c := 0$', the body of the loop '`assume` $y \neq 0$; $y := y\&(y-1)$; $c := c+1$' and the loop exit '`assume` $y = 0$'. The exact semantics of these blocks can be described relationally, as systems of propositional constraints. The idea is to represent the input and output

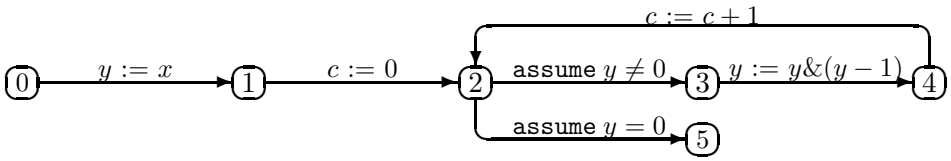

Figure 1: Bit counting using Wegner's method

relationships across a basic block using two systems of propositional variables $x_0, \ldots, x_{w-1}$ and $x'_0, \ldots, x'_{w-1}$ (abbreviated to $\vec{x}$ and $\vec{x}'$) that encode the input and output state of each integer variable x (x_0 is the least significant bit). We assume a twos complement integer representation and let w denote the number of bits that make up an integer. The constraints generated for the example are:

$$\begin{aligned}
&[\![y := x; c := 0]\!] \\
&\quad = (\textstyle\bigwedge_{i=0}^{w-1} y'_i \leftrightarrow x_i) \wedge (\bigwedge_{i=0}^{w-1} \neg c'_i) \wedge (\bigwedge_{i=0}^{w-1} x'_i \leftrightarrow x_i) \\
&[\![\texttt{assume } y \neq 0; y := y\&(y-1); c := c+1]\!] \\
&\quad = (\textstyle\bigvee_{i=0}^{w-1} y_i) \wedge (\bigwedge_{i=0}^{w-1} y'_i \leftrightarrow (y_i \wedge \bigvee_{j=0}^{i-1} y_j)) \\
&\qquad \wedge (\textstyle\bigwedge_{i=0}^{w-1} c'_i \leftrightarrow (c_i \oplus \bigwedge_{j=0}^{i-1} c_j)) \wedge (\bigwedge_{i=0}^{w-1} x'_i \leftrightarrow x_i) \\
&[\![\texttt{assume } y = 0]\!] \\
&\quad = (\textstyle\bigwedge_{i=0}^{w-1} \neg y_i) \wedge (\bigwedge_{i=0}^{w-1} x'_i \leftrightarrow x_i) \wedge (\bigwedge_{i=0}^{w-1} y'_i \leftrightarrow y_i) \wedge (\bigwedge_{i=0}^{w-1} c'_i \leftrightarrow c_i)
\end{aligned}$$

where $\oplus$ denotes exclusive-or. In Section 4 we explain how these constraints can be generated automatically from the program.

2.2 Setting up the dataflow equations

The relational semantics for the basic blocks allows us to derive the states that are reachable at program points 0, 2 and 5. They are obtained as the least (strongest) solution to the following recursive equations:

$$\begin{aligned}
f_0 = \;& 1 \\
f_2 = \;& \rho_{\vec{v}',\vec{v}}(\pi_{\vec{v}'}(f_0 \wedge [\![y := x; c := 0]\!])) \vee \\
& \rho_{\vec{v}',\vec{v}}(\pi_{\vec{v}'}(f_2 \wedge [\![\texttt{assume } y \neq 0; y := y\&(y-1); c := c+1]\!])) \\
f_5 = \;& \rho_{\vec{v}',\vec{v}}(\pi_{\vec{v}'}(f_2 \wedge [\![\texttt{assume } y = 0]\!]))
\end{aligned}$$

where $\vec{v}$ and $\vec{v}'$ are the input and output variables, that is, $\vec{v} = \vec{c} \cdot \vec{x} \cdot \vec{y}$ and $\vec{v}' = \vec{c}' \cdot \vec{x}' \cdot \vec{y}'$. The Boolean functions f_0, f_2 and f_5 represent

sets of reachable states, for example, a state $\sigma = \{c_0 \mapsto 0, \ldots, c_{w-1} \mapsto 0, x_0 \mapsto 1, \ldots, x_{w-1} \mapsto 0, y_0 \mapsto 1, \ldots, y_{w-1} \mapsto 0\}$ is considered to be reachable at program point 2 iff σ satisfies f_2. The projection operation $\pi_{\vec{v}'}(f) = f'$ computes the Boolean function f' by eliminating, by existential quantification, any propositional variable y from f that does not occur in the system $\vec{v}'$. For instance, $\pi_{\langle x_1, x_2\rangle}(f) = \exists_y(f) = x_1 \leftrightarrow x_2$ when $f = (x_1 \leftrightarrow y) \wedge (y \leftrightarrow x_2)$. Projection is used to derive a function that only expresses relations between the output variables $\vec{v}'$. The renaming operation $\rho_{\vec{v}',\vec{v}}(f) = f'$ constructs a function f' by replacing each output variable y' in f with its counterpart input variable y, for example, $\rho_{\vec{v}',\vec{v}}(c_0' \wedge (x_1' \oplus y_1')) = c_0 \wedge (x_1 \oplus y_1)$. Iteration can be used to compute f_2 from the predetermined $f_0 = 1$ and once f_2 is known, f_5 can be derived. For f_2 the iterates start:

$$\begin{array}{lllll} f_2^0 = & 0 & & \\ f_2^1 = & f_2^0 & \vee & (\bigwedge_{i=0}^{w-1} x_i \leftrightarrow y_i) \wedge (\bigwedge_{i=0}^{w-1} \neg c_i) \\ f_2^2 = & f_2^1 & \vee & (\bigvee_{i=0}^{w-1} x_i) \wedge (\bigwedge_{i=0}^{w-1} y_i \leftrightarrow (x_i \wedge \bigvee_{j=0}^{i-1} x_j)) \wedge c_0 \wedge (\bigwedge_{i=1}^{w-1} \neg c_i) \\ f_2^3 = & f_2^2 & \vee & \ldots \end{array}$$

This sequence will eventually stabilise because a bounded number (2^{3w}) of Boolean functions are definable over $\vec{v}$. However, the $\vec{c}$ variables will enumerate all 2^w bit patterns and therefore at least 2^w iterates will be computed. This will take an impractically long time, even for $w = 16$. There is also an issue of space. A Boolean function can be represented as an ROBDD but the size of an ROBDD can be exponentially large in the number of variables (even with dynamic variable reordering [2]), and this is a pressing issue when w propositional variables are needed to represent each integer variable. Tractability can be recovered by approximating [16] or widening [10] an ROBDD when it becomes intolerably large. This would replace an ROBDD with one that could be stored more compactly and yet represented a larger set of states. The problem with this approach is that the ROBDD widenings that have been proposed thus far [10, 16], do not preserve sufficient information to infer useful loop invariants.

2.3 Abstracting bit-level inputs and outputs with congruences

The considerations of tractability suggest that we look for principled ways of over-approximating solutions to systems of equations of the form

$$f = \bigvee_{m=1}^{n} \rho_{\vec{y}',\vec{y}}(\pi_{\vec{y}'}(f_m \wedge f_m')) \tag{1}$$

without giving up too much bit-level information. We suggest that this can be achieved by restricting f, as well as each f_m, to a class of functions that can be expressed as conjunctions of congruence equations modulo m, where m is a power of 2. Each function f_m summarises the inputs to one of n basic blocks and the function f summarises all (the join of) the outputs of the n blocks. *No constraint is placed on the generality of any of the f'_m formulae.* This is crucial and means that no abstraction needs to be applied to a function that describes the relational semantics of a basic blocks—this description can be kept precise to the bit-level.

How to solve equation systems of the form (1) under the restrictions just mentioned? The rest of this section explains the idea, based on the Wegner example.

Let $x \equiv_{2^w} y$ abbreviate $x = y + k2^w$ for some integer multiplier k. Observe that each of the equations $t \equiv_{2^w} t' + t''$, $t \equiv_{2^w} ny$, $t \equiv_{2^w} n$ and the disequation $t \not\equiv_{2^w} n$ can be expressed as propositional constraints when the t variables are w-bit, y is 1-bit and n is an integer. This is a consequence of the modulus being a power of 2. For instance, $t \equiv_{2^w} ny$ and $t \not\equiv_{2^w} ny$ can be expressed as $\bigwedge_{i=0}^{w-1} t_i \leftrightarrow (n_i \wedge y)$ and $\bigvee_{i=0}^{w-1} t_i \oplus (n_i \wedge y)$ where $n \equiv_{2^w} \sum_{i=0}^{w-1} 2^i n_i$ and $t \equiv_{2^w} \sum_{i=0}^{w-1} 2^i t_i$. Moreover, an equation $\sum_{i=1}^{k} n_i y_i \equiv_{2^w} n$ can be reduced by $\sum_{i=1}^{j} n_i y_i \equiv_{2^w} t$, $\sum_{i=j+1}^{k} n_i y_i \equiv_{2^w} t'$, $t + t' \equiv_{2^w} t''$ and $t'' \equiv_{2^w} n$ using fresh w-bit variables t, t', and t'', and hence also reduced to a propositional system. Any disequation $\sum_{i=1}^{k} m_i y_i \not\equiv_{2^w} n$ can similarly be described propositionally. Henceforth let $[\![\sum_{i=1}^{k} n_i y_i \equiv_{2^w} n]\!]$ and $[\![\sum_{i=1}^{k} n_i y_i \not\equiv_{2^w} n]\!]$ denote such propositional encodings.

To illustrate the value of these encodings, let $w = 8$ and consider computing $f_2^2 = f_2^1 \vee \rho_{\vec{y}',\vec{y}}(\pi_{\vec{y}'}(f_2^1 \wedge g))$ where

$$\begin{array}{lcl} f_2^1 & = & \rho_{\vec{y}',\vec{y}}(\pi_{\vec{y}'}(1 \wedge [\![y := x; c := 0]\!])) \\ g & = & [\![\texttt{assume}\ y \neq 0; y := y\&(y-1); c := c+1]\!] \end{array}$$

The encodings are used to direct the generation of truth assignments for the function $f_2^1 \wedge g$. Truth assignments are generated so as to incrementally derive the most precise congruence system describing both f_2^1 and $\rho_{\vec{y}',\vec{y}}(\pi_{\vec{y}'}(f_2^1 \wedge g))$. This system is used as the iterate, rather than the function f_2^1 itself. Since the function $f_2^0 = 0$ can be represented as a congruence system, namely $0 \equiv_{256} 1$, this construction ensures that all iterates are congruences. The number of iterates is bounded: the length of an increasing chain of congruences is at most 192, that is, the product of the power $w = 8$ and the maximum number, 24 ($3w$), of variables in each system [14].

The function f_2^1 falls into the class of formulae that can be represented congruently. This is because the satisfying assignments of f_2^1 coincide with

those of the Boolean formula $(\bigwedge_{i=0}^{7}[\![c_i \equiv_{256} 0]\!]) \wedge (\bigwedge_{i=0}^{7}[\![x_i \equiv_{256} y_i]\!])$ on $\vec{c}$, $\vec{x}$ and $\vec{y}$. Hence computing $\rho_{\vec{y}',\vec{y}}(\pi_{\vec{y}'}(f_2^1 \wedge g))$ is equivalent to computing $\rho_{\vec{y}',\vec{y}}(\pi_{\vec{y}'}(g'))$ where

$$g' = g \wedge (\bigwedge_{i=0}^{7}[\![c_i \equiv_{256} 0]\!]) \wedge (\bigwedge_{i=0}^{7}[\![x_i \equiv_{256} y_i]\!])$$

This is convenient, because it permits the encodings to be demonstrated on a representative, non-trivial example, namely the derivation of f_2^2. To see how the congruence system for f_2^2 is incrementally constructed, observe that any model of the Boolean function

$$g' \wedge ((\bigvee_{i=0}^{7}[\![c_i' \not\equiv_{256} 0]\!]) \vee (\bigvee_{i=0}^{7}[\![x_i' \not\equiv_{256} y_i']\!])) \tag{2}$$

defines a run of the block with an entry state that is described by f_2^1 but whose exit state is *not* characterised by f_2^1. For example, the truth assignment

$$\left\{\begin{array}{lll} c_0 \mapsto 0, c_1 \mapsto 0, \ldots, c_7 \mapsto 0, & x_0 \mapsto 0, \ldots, x_6 \mapsto 0, x_7 \mapsto 1, & y_0 \mapsto 0, \ldots, y_7 \mapsto 1, \\ c_0' \mapsto 1, c_1' \mapsto 0, \ldots, c_7' \mapsto 0, & x_0' \mapsto 0, \ldots, x_6' \mapsto 0, x_7' \mapsto 1, & y_0' \mapsto 0, \ldots, y_7' \mapsto 0 \end{array}\right\}$$

satisfies (2) and demonstrates that when `c`, `x`, and `y` assume values of 0, 128 and 128 on entry to the block, they can take values of 1, 128 and 0 on exit from the block (assuming an unsigned representation). By construction, the output state is not summarised by f_2^1 and therefore f_2^1 needs to be enlarged to accommodate this state. Since the output state can be represented in congruence form as

$$c_0 \equiv_{256} 1 \wedge (\bigwedge_{i=1}^{7} c_i \equiv_{256} 0) \wedge (\bigwedge_{i=0}^{6} x_i \equiv_{256} 0) \wedge x_7 \equiv_{256} 1 \wedge (\bigwedge_{i=0}^{7} y_i \equiv_{256} 0) \tag{3}$$

this system and that for f_2^1 can be joined (represented by a single system) to obtain the summary

$$(\bigwedge_{i=1}^{7} c_i \equiv_{256} 0) \wedge (\bigwedge_{i=0}^{6} x_i \equiv_{256} y_i) \wedge x_7 \equiv_{256} c_0 + y_7 \tag{4}$$

A model for the formula (2) can be found using standard techniques [15], translating the formula into an equi-satisfiable conjunctive normal form (CNF) representation, and presenting the CNF formula to any SAT-solver.

The join can be computed by translating the two congruence systems to their sets of generators, taking the union of the two sets, then converting the union to a new congruence system [1, 6, 14]. Alternatively, the join can be obtained by relaxing a system of congruences constructed syntactically from the two input systems (see Section 5). Either way, whether the join describes all possible output states can be determined by solving the Boolean formula

$$g' \wedge ((\bigvee_{i=1}^{7} [\![c_i \not\equiv_{256} 0]\!]) \vee (\bigvee_{i=0}^{6} [\![x_i \not\equiv_{256} y_i]\!]) \vee [\![x_7 \not\equiv_{256} c_0 + y_7]\!]) \tag{5}$$

This formula is satisfied, for example, by a truth assignment $\{\ldots, c'_0 \mapsto 1, c'_1 \mapsto 0, \ldots c'_7 \mapsto 0, x'_0 \mapsto 0, \ldots x'_5 \mapsto 0, x'_6 \mapsto 1, x'_7 \mapsto 0, y'_0 \mapsto 0, \ldots y'_7 \mapsto 0\}$ from which the following congruence system can be derived:

$$c_0 \equiv_{256} 1 \wedge (\bigwedge_{i=1}^{7} c_i \equiv_{256} 0) \wedge (\bigwedge_{i=0, i\neq 6}^{7} x_i \equiv_{256} 0) \wedge x_6 \equiv_{256} 1 \wedge (\bigwedge_{i=0}^{7} y_i \equiv_{256} 0) \tag{6}$$

Note that the assignments to the input variables, as well as any temporary variables introduced in CNF conversion [15], are inconsequential for constructing the congruence system. Disregarding these assignments amounts to projecting onto the output variables. Notice too that the system is expressed in terms of unprimed variables, even though it encodes an output state. Constructing the congruence system thus involves renaming as well as projection, though both operations are performed on truth assignments, at which level they collapse to computationally trivial operations.

Merging the previous summary (4) with the new system (6) gives the new summary

$$(\bigwedge_{i=1}^{7} c_i \equiv_{256} 0) \wedge (\bigwedge_{i=1}^{5} x_i \equiv_{256} y_i) \wedge x_6 + x_7 \equiv_{256} c_0 + y_6 + y_7 \tag{7}$$

Continuing this way, we obtain a sequence of Boolean formulae $h_0, h_1, h_2, \ldots$, the first two of which are (2) and (5), and where, more generally, h_j is

$$g' \wedge ((\bigvee_{i=1}^{7} [\![c_i \not\equiv_{256} 0]\!]) \vee (\bigvee_{i=0}^{7-j} [\![x_i \not\equiv_{256} y_i]\!]) \vee [\![\textstyle\sum_{i=7-j+1}^{7} x_i \not\equiv_{256} c_0 + \sum_{i=7-j+1}^{7} y_i]\!])$$

Of these, $h_1, \ldots, h_6$ are satisfiable, but h_7 is not, from which we conclude that the congruence system

$$(\bigwedge_{i=1}^{7} c_i \equiv_{256} 0) \wedge \sum_{i=0}^{7} x_i \equiv_{256} c_0 + \sum_{i=0}^{7} y_i \tag{8}$$

summarises all reachable output states when the input states are described by f_2^1. The next iterate f_2^2 is then assigned to this system which is the most precise congruence system that describes the set of output states given an input state drawn from f_2^1.

Note that the method is not sensitive to the way the relational semantics is presented. This contrasts with previous analyses which critically depend on how the statements of a program are translated into, say, affine assignments. This is particularly pertinent when deriving invariants for assembler or obfuscated code [17].

Of course, thus far, only f_2^2 has been derived. By repeating the above process with an updated input state we obtain the sequence of iterates:

$$\begin{array}{rcl} f_2^3 & = & (\bigwedge_{i=2}^{7} c_i \equiv_{256} 0) \wedge \sum_{i=0}^{7} x_i \equiv_{256} c_0 + 2c_1 + \sum_{i=0}^{7} y_i \\ f_2^4 & = & (\bigwedge_{i=3}^{7} c_i \equiv_{256} 0) \wedge \sum_{i=0}^{7} x_i \equiv_{256} c_0 + 2c_1 + 4c_2 + \sum_{i=0}^{7} y_i \\ f_2^6 = f_2^5 & = & (\bigwedge_{i=4}^{7} c_i \equiv_{256} 0) \wedge \sum_{i=0}^{7} x_i \equiv_{256} c_0 + 2c_1 + 4c_2 + 8c_3 + \sum_{i=0}^{7} y_i \end{array}$$

Interestingly, although the derivation of f_2^2 requires 8 calls to a SAT-solver and 7 join computations, the iterates f_2^3, f_2^4 and f_2^5 each require just two calls to a solver and one join. To check stability, that is, deduce $f_2^6 = f_2^5$, requires one call to a solver, hence 15 invocations are required in total, the largest of which involves 4507 variables and 11648 clauses (though more compact CNF conversion is possible [15]). Nevertheless, the longest time that it takes to solve any instance is 0.61 ms (wall-time) using SAT4J version 1.5 [11] on a 2.4 GHz MacBook Pro. Even without deriving $f_5 = \sum_{i=0}^{7} x_i \equiv_{256} c_0 + 2c_1 + 4c_2 + 8c_3 \wedge (\bigwedge_{i=4}^{7} c_i \equiv_{256} 0)$, it is now evident that Wegner's bit-twiddling algorithm assigns to the variable `c` the number of bits which are set in the variable `x`. As far as we aware, no other analysis is capable of deriving such an invariant. Furthermore, observe that the invariant includes coefficients of 4 and 8, and thus precision would be degraded if a modulo of 2 was employed rather than the word-level modulo of 2^w.

3 The Programming Model Formally

While the outline has presented programs in a C-like language, it is convenient to work instead with their translation to flowchart programs. The flowchart nodes are considered program points, and edges are labeled with statements—an example was given in Figure 1. Formally, a program is a set of edges, together with an indication of the entry point. Thus, letting Pp

denote the set of program points, we define

$$\begin{aligned} Pgm &= 2^{Edge} \times Pp \\ Edge &= Pp \times Stmt \times Pp \end{aligned}$$

The well-formed statements *Stmt* are given by this grammar:

$$\begin{aligned} Stmt &\rightarrow var := Exp \mid \texttt{assume}\ Bexp \\ Exp &\rightarrow var \mid num \mid Exp\ binop\ Exp \mid unop\ Exp \mid (\ Exp\ ) \\ Bexp &\rightarrow \texttt{false} \mid \texttt{true} \mid \texttt{not}\ Bexp \mid Bexp\ \texttt{and}\ Bexp \mid Bexp\ \texttt{or}\ Bexp \mid Exp\ relop\ Exp \end{aligned}$$

We assume that arithmetic operators include the binary (*binop*) `+`, `-`, `*`, and `/`, bitwise operators (C-style) `&`, `|`, `^`, for conjunction, disjunction, and exclusive or, respectively, and shift operators. Unary arithmetic operators include `~` for negation and `-` (unary minus). Relational operators (*relop*) include `==`, `!=`, `<`, `<=`, `>`, and `>=`. We shall use natural numbers as program point identifiers (*Pp*).

In this syntax the program from Figure 1 consists of six edges (with 0 being the entry point):

$$\begin{aligned} &\langle 0, y := x, 1\rangle \\ &\langle 1, c := 0, 2\rangle \\ &\langle 2, \texttt{assume}\ y\ != 0, 3\rangle \\ &\langle 3, y := y\ \&\ (\ y - 1\ ), 4\rangle \\ &\langle 4, c := c + 1, 2\rangle \\ &\langle 2, \texttt{assume}\ y == 0, 5\rangle \end{aligned}$$

All variables are of limited-precision integer type, based on some word length w. As mentioned, we assume a twos complement integer representation and interpret operations accordingly.

The semantics of the language can be given a standard denotational definition, by providing semantic *functions*

$$\begin{aligned} \mathcal{S} &: Stmt \rightarrow \Sigma \rightarrow \Sigma \\ \mathcal{E} &: Exp \rightarrow \Sigma \rightarrow \mathcal{D} \end{aligned}$$

in the usual manner, with $\mathcal{D} = \mathbb{Z}_m$, the residue class of integers modulo m. We let V denote the set of program variables, $V = \{v_1, \ldots, v_k\}$, and so $\Sigma = V \rightarrow \mathcal{D}$, that is, a state is a mapping from the set of program variables to the set of values. However, for a number of reasons we prefer to give a relational semantics. First, we consider programs to take input via the

program variables (hence no 'read' statement is needed), so the semantics needs to express how, at different program 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 that captures the values of program variables before and after the statement is executed. Given a word-length w, let $m = 2^w$ and let

$$\mathcal{S} = \mathcal{P}(\mathcal{D}^{2k})$$

where $\mathcal{P}$ is the powerset operator and $\mathcal{D} = \mathbb{Z}_m$. The meaning of a program (or of a statement) is then an input-output relation $r \in \mathcal{S}$. The relation r expresses how the program variables' values before and after execution of the program (or statement) are related, using the first k components of a tuple for the "before" values and the last k components for the "after" values.

Compared to the more traditional denotational approach outlined above, our approach replaces a state transformer $\tau = \mathcal{S}[\![s]\!]$ by a relation whose tuples are

$$\{\langle \sigma(v_1), \ldots, \sigma(v_k), \tau(\sigma)(v_1), \ldots, \tau(\sigma)(v_k)\rangle \mid \sigma \in \Sigma\}$$

Let $\mathsf{Id} = \{\langle a_1, \ldots, a_k, a_1, \ldots, a_k\rangle \mid a_1, \ldots, a_k \in \mathcal{D}\}$. Given a k-tuple t, a value v, and an index $i \in \mathbb{N}$, let $t[i \mapsto v]$ denote the k-tuple which is identical to t, except it has v as its ith component. Then the effect of a statement can be defined as follows.

$$\begin{array}{rcl} \mathcal{S}[\![v_i := e]\!] & = & \{t[k+i \mapsto \mathcal{E}[\![e]\!]t] \mid t \in \mathsf{Id}\} \\ \mathcal{S}[\![\texttt{assume } c]\!] & = & \{t \mid t \in \mathsf{Id} \wedge \mathcal{C}[\![c]\!]t\} \end{array}$$

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

$$\begin{array}{rcl}
\mathcal{E}[\![v_i]\!]t & = & t[i] \\
\mathcal{E}[\![n]\!]t & = & n \\
\mathcal{E}[\![e_1 + e_2]\!]t & = & (\mathcal{E}[\![e_1]\!]t) + (\mathcal{E}[\![e_2]\!]t) \\
\mathcal{E}[\![e_1 \;\&\; e_2]\!]t & = & (\mathcal{E}[\![e_1]\!]t) \wedge (\mathcal{E}[\![e_2]\!]t) \\
\mathcal{E}[\![e_1 \mid e_2]\!]t & = & (\mathcal{E}[\![e_1]\!]t) \vee (\mathcal{E}[\![e_2]\!]t) \\
\mathcal{E}[\![e_1 \;\hat{}\; e_2]\!]t & = & (\mathcal{E}[\![e_1]\!]t) \oplus (\mathcal{E}[\![e_2]\!]t) \\
\mathcal{E}[\![-e]\!]t & = & -\mathcal{E}[\![e]\!]t
\end{array}$$

$$\begin{array}{rcl}
\mathcal{C}[\![\texttt{false}]\!]t & = & 0 \\
\mathcal{C}[\![\texttt{true}]\!]t & = & 1 \\
\mathcal{C}[\![\texttt{not } c]\!]t & = & \neg(\mathcal{C}[\![c]\!]t) \\
\mathcal{C}[\![c_1 \texttt{ and } c_2]\!]t & = & (\mathcal{C}[\![c_1]\!]t) \wedge (\mathcal{C}[\![c_2]\!]t) \\
\mathcal{C}[\![c_1 \texttt{ or } c_2]\!]t & = & (\mathcal{C}[\![c_1]\!]t) \vee (\mathcal{C}[\![c_2]\!]t) \\
\mathcal{C}[\![e_1 == e_2]\!]t & = & (\mathcal{E}[\![e_1]\!]t) = (\mathcal{E}[\![e_2]\!]t) \\
\mathcal{C}[\![e_1 \leq e_2]\!]t & = & (\mathcal{E}[\![e_1]\!]t) \leq (\mathcal{E}[\![e_2]\!]t)
\end{array}$$

and similarly for other relaltional operators. Additionally we define

$$r_1 \circ r_2 = \left\{ \langle a_1, \ldots, a_k, c_1, \ldots, c_k \rangle \;\middle|\; \begin{array}{l} \langle a_1, \ldots, a_k, b_1, \ldots, b_k \rangle \in r_1 \\ \langle b_1, \ldots, b_k, c_1, \ldots, c_k \rangle \in r_2 \end{array} \right\}$$

Alternatively we may express the relations symbolically. Let V' be a set of "fresh" variables, one for each variable in V. We denote the V'-variable that corresponds to $v \in V$ by v'. Similarly let V'' be a set of fresh variables, disjoint from V', one for each variable in V. We denote the V''-variable that corresponds to $v \in V$ by v''. We can then express the composition $\psi \circ \varphi$ as

$$\exists V''(\exists V(\psi \wedge \bigwedge_V (v \leftrightarrow v'')) \wedge \exists V'(\varphi \wedge \bigwedge_V (v' \leftrightarrow v'')))$$

In other words, we rename each output value in φ and we rename the corresponding input value in ψ identically. The conjunction of the results expresses (in terms of V and V' as usual) the combined effect. The intermediate values from V'' can then be eliminated.

Note that, in this view, $\mathsf{Id} = \bigwedge_V (v \leftrightarrow v')$.

We can then give a fixed point characterization of a relational semantics for the small language. Let the program (P, p_0) be given (recall that P is a set of edges and p_0 is the entry program point). Its semantics is the smallest solution to the equations

$$\begin{array}{rcll}
\mathbf{C}_{p_0} & \supseteq & \mathsf{Id} & \\
\mathbf{C}_{p'} & \supseteq & \mathcal{S}[\![s]\!] \circ \mathbf{C}_p & \text{for } (p, s, p') \in P
\end{array}$$

Since values are in $\mathbb{Z}_m$, (and the various operations defined accordingly), the meaning of a program is a finite relation.

4 Bit-Precise Transfer Functions

Our analysis includes one component that translates basic blocks into Boolean formulae. This component preserves all the information within a basic block, which is key to reasoning about non-linear operations such as bit-twiddling. In this section we describe the generation of bit-precise transfer functions.

The other important component of our system joins two congruence systems as a single system. This component is complementary to what was described in Section 4, where the goal was to achieve maximal precision in the summary of each block. The join, on the other hand, discards information, which is key to retaining tractability. The join ensures that the summaries reside in a finite ascending chain so they cannot be weakened forever; the maximal chain length is w^2n [14], as wn (propositional) variables are needed to represent the state of n (integer) variables of width w. In this section we present an efficient approach to calculating the join.

For an integer n we use $\langle\langle n \rangle\rangle$ to denote the bit-vector representing n's signed value. For a non-negative integer n, $\langle n \rangle$ is the bit-vector representing n's unsigned value. We use $\cdot[\cdot]$ to access elements of bit-vectors, numbering, as usual, the bits from right to left, starting from index 0.

For statements, we define the bit-precise semantics $\mathcal{S}_b$:

$$\begin{array}{rcl} \mathcal{S}_b[\![v := e]\!] & = & \bigwedge_{i=0}^{w-1}(v'_i \leftrightarrow \mathcal{E}[\![e]\!][i]) \wedge \bigwedge_{\vec{u}\in V\setminus\{v\}} \bigwedge_{i=0}^{w-1}(u'_i \leftrightarrow u_i) \\ \mathcal{S}_b[\![\texttt{assume } c]\!] & = & \mathcal{C}_b[\![c]\!] \end{array}$$

The second conjunct expresses that variables other than the assignment's target remain unchanged.

The bit-precise semantic function $\mathcal{E}_b : \mathit{Exp} \to \mathbb{N} \to \mathcal{B}$ takes an expression and a bit position between 0 and $w-1$ inclusive and returns the value of that bit. The bit-precise function $\mathcal{C}_b : \mathit{Bexp} \to \mathcal{B}$ takes a condition and returns its (Boolean) value. These functions are defined as follows. We let r_1 and r_2 abbreviate $\mathcal{E}_b[\![e_1]\!]$ and $\mathcal{E}_b[\![e_2]\!]$, respectively.

$$
\begin{array}{rcl}
\mathcal{E}_b[\![v]\!][i] & = & v_i \\
\mathcal{E}_b[\![n]\!][i] & = & \langle\!\langle n \rangle\!\rangle[i] \\
\mathcal{E}_b[\![e_1 + e_2]\!][i] & = & r_1[i] \oplus r_2[i] \oplus \bigoplus_{j=0}^{i-1}(r_1[j] \wedge r_2[j] \wedge \bigwedge_{k=j+1}^{i-1}(r_1[k] \oplus r_2[k])) \\
\mathcal{E}_b[\![e_1 \ \&\ e_2]\!][i] & = & r_1[i] \wedge r_2[i] \\
\mathcal{E}_b[\![e_1 \mid e_2]\!][i] & = & r_1[i] \vee r_2[i] \\
\mathcal{E}_b[\![e_1 \ \hat{}\ e_2]\!][i] & = & r_1[i] \oplus r_2[i] \\
\mathcal{E}_b[\![-e]\!][i] & = & \mathcal{E}_b[\![e]\!][i] \oplus \bigvee_{j=0}^{i-1} \mathcal{E}_b[\![e]\!][j]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_b[\![\texttt{false}]\!] & = & 0 \\
\mathcal{C}_b[\![\texttt{true}]\!] & = & 1 \\
\mathcal{C}_b[\![\texttt{not } c]\!] & = & \neg(\mathcal{C}_b[\![c]\!]) \\
\mathcal{C}_b[\![c_1 \texttt{ and } c_2]\!] & = & (\mathcal{C}_b[\![c_1]\!]) \wedge (\mathcal{C}_b[\![c_2]\!]) \\
\mathcal{C}_b[\![c_1 \texttt{ or } c_2]\!] & = & (\mathcal{C}_b[\![c_1]\!]) \vee (\mathcal{C}_b[\![c_2]\!]) \\
\mathcal{C}_b[\![e_1 == e_2]\!] & = & \bigwedge_{i=0}^{w-1}(r_1[i] \leftrightarrow r_2[i])
\end{array}
$$

Other equations can be derived from these. For example, $\mathcal{E}_b[\![e_1 - e_2]\!]$ can be derived from the equations for $e_1 + e_2$ and $-e$. The equation for a relational operator can similarly be derived [17], section 2-11. The constraint for an expression such as `x + y` has been derived by considering a cascade of full adders using intermediate carry bits $\vec{b}$. Namely, $\langle\!\langle x + y \rangle\!\rangle[i]$ is

$$(x_i \oplus y_i \oplus b_i) \wedge \neg b_0 \wedge \bigwedge_{j=1}^{w-1} b_j \leftrightarrow ((x_{j-1} \wedge y_{j-1}) \vee (x_{j-1} \wedge b_{j-1}) \vee (y_{j-1} \wedge b_{j-1}))$$

The variables $\vec{b}$ are existentially quantified, and the formula can be simplified using standard Boolean quantifier elimination. This yields $x_i \oplus y_i \oplus \bigoplus_{j=0}^{i-1}(x_j \wedge y_j \wedge \bigwedge_{k=j+1}^{i-1}(x_k \oplus y_k))$.

Note that $\mathcal{E}_b[\![e_1 - e_2]\!]$ can be obtained as $\mathcal{E}_b[\![e_1 + (-e_2)]\!]$. The equations for other relational operators can be obtained in ways similar to those for $\leq$ and $<$.

Example 4.1 *Various special cases are easily derived, for example,*

$$
\begin{array}{rcl}
\mathcal{S}_b[\![x := y]\!] & = & \bigwedge_{i=0}^{w-1}(x'_i \leftrightarrow y_i) \wedge \bigwedge_{\vec{u} \in V \setminus \{x\}} \bigwedge_{i=0}^{w-1}(u'_i \leftrightarrow u_i) \\
\mathcal{S}_b[\![x := x + 1]\!] & = & \bigwedge_{i=0}^{w-1}(x'_i \leftrightarrow x_i \oplus \bigwedge_{j=0}^{i-1} x_j) \\
\mathcal{S}_b[\![x := x - 1]\!] & = & \bigwedge_{i=0}^{w-1}(x'_i \oplus x_i \oplus \bigvee_{j=0}^{i-1} x_j)
\end{array}
$$

Composition of transfer relations is as before (Section 3), and, without any change, the semantic equations remain:

$$\begin{array}{lcll} \mathbf{C}_{p_0} & \supseteq & \mathsf{Id} & \\ \mathbf{C}_{p'} & \supseteq & \mathcal{S}[\![s]\!] \circ \mathbf{C}_p & \text{for } (p, s, p') \in P \end{array}$$

Again, the bit-precise semantics is the smallest solution to these equations.

It should be noted that the correctness and accuracy of the analysis is unaffected by the presence of intermediate variables. Thus, although their removal may (sometimes) improve the efficiency of SAT-solving and aid human understanding, this simplification step is not strictly necessary.

5 Joining Congruence Equations

The other important component of our system joins two congruence systems as a single system. This component is complementary to what was described in Section 4, where the goal was to achieve maximal precision in the summary of each block. The join, on the other hand, discards information, which is key to retaining tractability. The join ensures that the summaries reside in a finite ascending chain so they cannot be weakened forever; the maximal chain length is w^2n [14], as wn (propositional) variables are needed to represent the state of n (integer) variables of width w. In this section we present an efficient approach to calculating the join.

Recent work on congruence domains [14] has exploited how congruence equation systems can be represented by sets of generators that span the solution space of the congruence system. This representation is useful because it reduces the join operation to set union. However, our refined form of analysis relies on a translation mechanism from an equation $\sum_{i=1}^{k} n_i y_i \equiv_{2^w} n$ to a formula $[\![\sum_{i=1}^{k} n_i y_i \equiv_{2^w} n]\!]$ that becomes more convoluted when the generator representation is adopted. Thus it is convenient to compute the join whilst representing the input and output systems as a conjunction of equations. This can be achieved by reformulating the join of two systems as a projection operation which can, in turn, be computed by calculating a triangular form. This section explains these steps.

5.1 Basics

To state the algorithmic results of this section, it is necessary to recall some mathematical concepts and notation. The set of congruence classes modulo m is defined $\mathbb{Z}_m = \{[n] \mid n \in \mathbb{Z}\}$ where $[n] = \{n' \in \mathbb{Z} \mid n \equiv_m n'\}$ and

$\equiv_m$ denotes equivalence modulo m. Henceforth, we blur the distinction between a class $[n]$ and its representative element n. The 2-fold Cartesian product $\mathbb{Z}_m^2$ is defined $\mathbb{Z}_m^2 = \mathbb{Z}_m \times \mathbb{Z}_m$ and the k-fold product $\mathbb{Z}_m^k$ is likewise defined. If $S_1, S_2 \subseteq \mathbb{Z}_m^k$ then their (Minkowski) sum is $S_1 + S_2 = \{\vec{x} \in \mathbb{Z}_m^k \mid \vec{x}_i \in S_i \wedge \vec{x} \equiv_m \vec{x}_1 + \vec{x}_2\}$. If $\lambda \in \mathbb{Z}$ and $S \subseteq \mathbb{Z}_m^k$, then $\lambda S = \{\vec{x} \in \mathbb{Z}_m^k \mid \vec{x}' \in S \wedge \vec{x} \equiv_m \lambda\vec{x}'\}$. Moreover, the linear closure of S is $\mathsf{linear}(S) = \{\sum_{i=1}^{\ell} \lambda_i \vec{x}_i \mid \lambda_i \in \mathbb{Z} \wedge \vec{x}_1, \ldots, \vec{x}_\ell \in S\}$. The modules of $\mathbb{Z}_m^k$ are those subsets of $\mathbb{Z}_m^k$ that are closed under linear combination, that is, $\mathsf{Module}_m^k = \{M \subseteq \mathbb{Z}_m^k \mid \mathsf{linear}(M) = M\}$. The affine subsets of $\mathbb{Z}_m^k$ are translated modules, that is, $\mathsf{Affine}_m^k = \{\{\vec{x}\} + M \mid \vec{x} \in \mathbb{Z}_m^k \wedge M \in \mathsf{Module}_m^k\}$. The affine closure of S is the smallest affine space that encloses S and is thus defined $\mathsf{affine}(S) = \bigcap\{S' \in \mathsf{Affine}_m^k \mid S \subseteq S'\}$.

Example 5.1 *Observe that* $\varnothing$ *and* $\{\vec{0}\}$ *are closed under linear combination, whence* $\varnothing \in \mathsf{Module}_m^k$ *and* $\{\vec{0}\} \in \mathsf{Module}_m^k$. *As* $\varnothing \in \mathsf{Module}_m^k$, *we have* $\varnothing \in \mathsf{Affine}_m^k$. *Moreover, since* $\{\vec{0}\} \in \mathsf{Module}_m^k$, *it follows that* $\{\vec{x}\} \in \mathsf{Affine}_m^k$ *for all* $\vec{x} \in \mathbb{Z}_m^k$.

5.2 Triangularisation

While congruence equations appear as good companions for a bit-level relational semantics, Gaussian elimination cannot be immediately applied to compute a triangular form for such equations because of the need to deal with zero divisors [14]. Müller-Olm and Seidl have thus devised a triangularisation algorithm that, given an input system $A\vec{x} \equiv_{2^w} \vec{b}$, computes output system $A'\vec{x} \equiv_{2^w} \vec{b}'$ where $A' = [a_{i,j}]$ and $a_{i,j} = 0$ whenever $i > j$. Figure 2 gives the algorithm and Example 5.2 illustrates what the algorithm will compute for an example. (This example will, in turn, support Example 5.4 which demonstrates how join can be straightforwardly realised.) In the description of the algorithm, $\mathsf{leading}(\vec{a})$ returns -1 if $\vec{a} = \vec{0}$ and otherwise the position of the first non-zero element of the vector $\vec{a}$; $\pi_\ell(\vec{a})$ extracts the ℓ'th element from $\vec{a}$; and $\mathsf{power}(n)$ returns the largest integer p such that 2^p divides n.

Example 5.2 The input and output to triangularisation procedure are

```
1:  procedure triangular(in: S, out: t)
2:      t := λℓ.⊥
3:      let {a⃗_1, ... a⃗_s} = S
4:      for i := 1 to s do
5:          ℓ := leading(a⃗_i)
6:          while (ℓ > 0 ∧ ℓ ∈ dom(t))
7:              a⃗' := t(ℓ)
8:              p := power(π_ℓ(a⃗_i))
9:              p' := power(π_ℓ(a⃗'))
10:             if p ≥ p' then
11:                 a⃗_i := (π_ℓ(a⃗')/2^p')a⃗_i − 2^(p−p') a⃗'
12:             else
13:                 t := t[ℓ ↦ a⃗_i]
14:                 a⃗_i := (π_ℓ(a⃗_i)/2^p)a⃗' − 2^(p'−p) a⃗_i
15:             endif
16:             ℓ := leading(a⃗_i)
17:         endwhile
18:         if ℓ > 0 then t := t[ℓ ↦ a⃗_i]
19:     endfor
20: endprocedure
```

Figure 2: The triangularisation method of Müller-Olm and Seidl [14]

$A\vec{x} \equiv_2 \vec{b}$ and $A'\vec{x} \equiv_2 \vec{b}'$ respectively:

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \quad \vec{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A' = \begin{bmatrix} 1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,1\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,1\,0\,0\,1\,0\,0\,1 \\ 0\,0\,0\,0\,0\,1\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,1\,1\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,1\,1\,0 \end{bmatrix} \quad \vec{b}' = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

By reasoning about upper triangular form, one can argue that any subset of $\mathbb{Z}_m^k$ that is closed under affine combination, can be represented congruently:

Proposition 5.1 $S \in \mathsf{Affine}_m^k$ iff there exists a congruence system $A\vec{x} \equiv_m \vec{b}$ such that $S = \{\vec{x} \in \mathbb{Z}_m^k \mid A\vec{x} \equiv_m \vec{b}\}$.

5.3 Projection

Quite apart from establishing this result, upper triangular form provides a way of computing arbitrary projections. Projection onto the i'th element of a k-ary vector is defined $\pi_i(\langle x_1, \ldots, x_k \rangle) = x_i$. Single element projections can be composed so that if $1 \leq i_1 < ... < i_j \leq k$ then the j-ary vector $\langle \pi_{i_1}(\vec{x}), \ldots, \pi_{i_j}(\vec{x}) \rangle$ is also a projection in that it also discards information pertaining to certain dimensions. Projection of an affine space is also affine:

Proposition 5.2 Let $S \in \mathsf{Affine}_m^k$ and $1 \leq i_1 < ... < i_j \leq k$. Then $T \in \mathsf{Affine}_m^j$ where $T = \{\langle \pi_{i_1}(\vec{x}), \ldots, \pi_{i_j}(\vec{x}) \rangle \mid \vec{x} \in S\}$.

If $A = [a_{i,j}]$ is in upper triangular form, the projection of $A\vec{x} \equiv_m \vec{b}$ onto a suffix $\vec{y} = \langle x_i, \ldots, x_k \rangle$ of $\vec{x}$ is found very easily. Suppose row j is the top-most row of A in which $\langle a_{j,1}, \ldots, a_{j,i-1} \rangle = \vec{0}$. Then the projection onto $\vec{y}$ is

$$\begin{bmatrix} a_{j,i} & \cdots & a_{j,k} \\ \vdots & & \vdots \\ a_{s,i} & \cdots & a_{s,k} \end{bmatrix} \vec{y} \equiv_m \begin{bmatrix} \pi_j(\vec{b}) \\ \vdots \\ \pi_s(\vec{b}) \end{bmatrix}$$

Example 5.3 Projecting $A\vec{x} \equiv_2 \vec{b}$ of Example 5.2, or equivalently the system $A(t)\vec{x} \equiv_2 \vec{b}(t)$, onto $\vec{y}_i = \langle x_i, \ldots, x_{11} \rangle$ for $i = 7, 9$ and 10 yields:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \vec{y}_7 \equiv_2 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \vec{y}_9 \equiv_2 \begin{bmatrix} 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \end{bmatrix} \vec{y}_{10} \equiv_2 \begin{bmatrix} 0 \end{bmatrix} \text{ (or 1)}$$

Given a congruence system $A\vec{x} \equiv_m \vec{b}$, it is possible to project onto any subset of $\vec{x}$ merely by reordering the rows of A in synchronicity with the elements of $\vec{b}$, prior to computing the triangular form.

5.4 Join

We finally show how the join can be reduced to computing a projection (a relaxation) which, in turn, amounts to deriving an upper triangular form.

Proposition 5.3 Let $S_i = \{\vec{x} \in \mathbb{Z}_m^k \mid A_i\vec{x} \equiv_m \vec{b}_i\}$ and

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ -\vec{b}_1 & 0 & A_1 & 0 & 0 \\ 0 & -\vec{b}_2 & 0 & A_2 & 0 \\ 0 & 0 & -I & -I & I \end{bmatrix} \quad S = \left\{ \vec{x} \in \mathbb{Z}_m^k \;\middle|\; \exists \sigma_i \in \mathbb{Z}_m . \exists \vec{x}_i \in \mathbb{Z}_m^k . A \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vec{x}_1 \\ \vec{x}_2 \\ \vec{x} \end{bmatrix} \equiv_m \begin{bmatrix} 1 \\ \vec{0} \\ \vec{0} \\ \vec{0} \end{bmatrix} \right\}$$

Then $S = \mathsf{affine}(S_1 \cup S_2)$ if $S_1 \neq \varnothing$ and $S_2 \neq \varnothing$.

If a system $A_i\vec{x} \equiv_{2^w} \vec{b}_i$ has a solution set S_i, then the join of $A_1\vec{x} \equiv_{2^w} \vec{b}_1$ and $A_2\vec{x} \equiv_{2^w} \vec{b}_2$ is a system whose solutions coincide with $\mathsf{affine}(S_1 \cup S_2)$. Proposition 5.3 states that such a system can be obtained by rearranging A_1 and A_2 to form a new matrix A and then eliminating variables.

Example 5.4 Consider the join of $A_1\vec{x} \equiv_2 \vec{b}_1$ and $A_2\vec{x} \equiv_2 \vec{b}_2$ where $\vec{x} = \langle x, y, z \rangle$

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \vec{b}_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \vec{b}_2 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

As well as minimising the size of coefficients and thereby making the presentation of large matrices manageable, a by-product of $2^w = 2$ is that $-I \equiv_2 I$ and $\vec{b}_i \equiv_2 -\vec{b}_i$. Using this, the combined system of Proposition 5.3 is formed—it is given below, on the left. On the right is the triangular system derived in Example 5.2, and we conclude that the join is $x + y \equiv_2 1$.

$$\left[\begin{array}{cc|ccc|ccc|ccc}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1
\end{array}\right]
\begin{bmatrix}
\sigma_1 \\ \sigma_2 \\ x_1 \\ y_1 \\ z_1 \\ x_2 \\ y_2 \\ z_2 \\ x \\ y \\ z
\end{bmatrix}
\equiv_2
\begin{bmatrix}
1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0
\end{bmatrix}$$

$$\begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
\sigma_1 \\ \sigma_2 \\ x_1 \\ y_1 \\ z_1 \\ x_2 \\ y_2 \\ z_2 \\ x \\ y \\ z
\end{bmatrix}
\equiv_2
\begin{bmatrix}
1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1
\end{bmatrix}$$

6 Discussion

Work on deriving systems of equalities [9] and inequalities [4] between program variables dates back to the very early days of abstract interpretation. Congruence domains were pioneered by Granger [5, 6] who proposed, among other things, using sets of generators for representing congruence equations and showed that congruence equations satisfied the ascending chain condition.

Recently there has been a resurgence of interest in inferring both linear [7, 13] and congruence relationships [1, 14], mainly from the perspective of improving efficiency, for instance, by applying randomisation [7], or fusing the domain operations with the fixed-point calculations [13], or refining the conversion between equations and generators [1], or bounding the size of the coefficients in the representation [1, 14]. An interesting twist to linear

equalities was given by Leroux [12] who has applied the disjunctive closure of this domain in model checking.

Our work revisits congruence analysis, not to enhance efficiency, but to improve precision. Precision is refined by capturing the semantics of a basic block accurately as a system of propositional constraints. These are combined with formulae that express congruence equations that hold upon entry to the block. The constraints that hold at the end of the block are then abstracted as a system of optimal congruence equations. This avoids the need to construct specialised transfer functions for affine assignment, nondeterministic assignment, etc. Instead all primitives, linear and non-linear, can be handled uniformly by translating them into systems of propositional constraints using transformations devised for bounded model checking [3, 8, 19].

An issue for future work is extending the intra-procedural analysis to inter-procedural analysis and systematic benchmarking. In its present form, the analyser consists of a Prolog and a Java component that are linked with temporary files. The Prolog component translates basic blocks, congruence equalities and congruence disequalities into propositional formulae and then applies CNF conversion [15] to construct a DIMACS file for SAT4J. The Java component implements triangularisation and join. The fixed-point is under manual control since it is both useful and pleasing to watch as the summaries converge onto a loop invariant. However, the Prolog component needs to be extended to translate other operations into formulae in order to deploy the analysis on other code and particularly programs that apply bit-level programming tricks [17].

7 Conclusion

This paper shows how congruence equations, with a modulo that is a power of two, fit elegantly with SAT solving and a relational bit-level encoding of the behaviour of the program, to derive invariants for programs that contain non-linear operations such as bit twiddling. The work calls for further research into methods in which SAT solvers are applied repeatedly to infer abstractions drawn from abstract domains that satisfy the ascending chain condition.

Acknowledgments

This work was funded by EPSRC projects EP/C015517, EP/E033105 and EP/F012896. We thank Paul Docherty for motivating discussions on reverse

engineering, Neil Kettle and Axel Simon for their comments on SAT-solving and Gift Nuka for his help with Floyd-style assertions.

References

[1] R. Bagnara, Katy Dobson, Patricia M. Hill, Matthew Mundell, and Enea 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, 2006.

[2] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[3] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176. Springer-Verlag, 2004.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Symposium on Principles of Programming Languages*, pages 84–97. ACM, 1978.

[5] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[6] Philippe 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.

[7] Sumit Gulwani and George C. Necula. Discovering affine equalities using random interpretation. In *Principles of Programming Languages*, pages 74–84. ACM, 2003.

[8] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis*, pages 14–25. ACM, 2000.

[9] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[10] N. Kettle, A. King, and T. Strzemecki. Widening ROBDDs with prime implicants. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 105–119. Springer-Verlag, 2006.

[11] Daniel Le Berre. A satisfiability library for Java. http://www.sat4j.org.

[12] J. Leroux. Disjunctive invariants for numerical systems. In Farn Wang, editor, *International Symposium on Automated Technology for Verification and Analysis*, volume 3299 of *LNCS*, pages 99–107. Springer-Verlag, 2004.

[13] Markus Müller-Olm and Helmut Seidl. A note on Karr's algorithm. In *International Colloquim on Automata, Languages and Programming*, volume 3142 of *LNCS*, pages 1016–1028. Springer-Verlag, 2004.

[14] Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. *ACM Transactions on Programming Languages and Systems*, 29(5), August 2007. Article 29.

[15] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Compututation*, 2(3):293–304, 1986.

[16] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *Design Automation Conference*, pages 445–450. IEEE Press, 1998.

[17] H. S. Warren Jr. *Hacker's Delight*. Addison Wesley, 2003.

[18] P. Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5):322, 1960.

[19] Yichen Xie and Alex Aiken. SATURN: A scalable framework for error detection using Boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29(3), 2007. Article 16.

A Proof Appendix

Lemma A.1 Let $\{S_i\}_{i\in I} \subseteq \mathsf{Affine}_m^k$. Then $\bigcap_{i\in I} S_i \in \mathsf{Affine}_m^k$.

Proof A.1 Put $S = \bigcap_{i\in I} S_i$. If $S = \varnothing$ then observe $S = \{\vec{0}\} + \varnothing$ and $\varnothing \in \mathsf{Module}_m^k$ as required. Otherwise there exists $\vec{x} \in S$ whence $\vec{x} \in S_i$ for all $i \in I$. There exists $M_i \in \mathsf{Module}_m^k$ such that $S_i = \{\vec{x}\} + M_i$. Put $M = \bigcap_{i\in I} M_i$ and observe $M \in \mathsf{Module}_m^k$. Finally $S = \bigcap_{i\in I}(\{\vec{x}\} + M_i) = \{\vec{x}\} + \bigcap_{i\in I} M_i = \{\vec{x}\} + M$. Therefore $S \in \mathsf{Affine}_m^k$ as required.

Lemma A.2 Let $S \subseteq \mathbb{Z}_m^k$. Then

$$\mathsf{affine}(S) = \bigcup \left\{ \sum_{i=1}^{\ell} \lambda_i S_i \;\middle|\; \lambda_i \in \mathbb{Z} \wedge S_1, \ldots, S_\ell \subseteq S \wedge \sum_{i=1}^{\ell} \lambda_i \equiv_m 1 \right\}$$

Proof A.2 If $S = \varnothing$ then $\mathsf{affine}(S) = \varnothing$ as required. Now put

$$S' = \bigcup \{ \sum_{i=1}^{\ell} \lambda_i S_i \mid \lambda_i \in \mathbb{Z} \wedge S_i \subseteq S \wedge \sum_{i=1}^{\ell} \lambda_i \equiv_m 1 \}$$

and suppose $S \neq \varnothing$.

- To show $\mathsf{affine}(S) \subseteq S'$. Observe that this follows if $S' \in \mathsf{Affine}_m^k$. Hence let $\vec{x} \in S \subseteq S'$ and put $M' = \{-\vec{x}\} + S'$. Let $S_i' \subseteq M'$ and $\lambda_i \in \mathbb{Z}$ for some $1 \leq i \leq \ell$. Put $S_i = \{\vec{x}\} + S_i'$ for all $1 \leq i \leq \ell$. Then consider $S_i \subseteq S'$ for all $1 \leq i \leq \ell$.

  Moreover put $S_0 = \{\vec{x}\}$ and $\lambda_0 \equiv_m 1 - \sum_{i=1}^{\ell} \lambda_i$ so that $\sum_{i=0}^{\ell} \lambda_i \equiv_m 1$. Then $\sum_{i=1}^{\ell} \lambda_i S_i' = (\sum_{i=1}^{\ell} \lambda_i S_i) - (\sum_{i=1}^{\ell} \lambda_i \{\vec{x}\}) = (\sum_{i=0}^{\ell} \lambda_i S_i) - (\sum_{i=1}^{\ell} \lambda_i)\{\vec{x}\} - \lambda_0\{\vec{x}\} = (\sum_{i=0}^{\ell} \lambda_i S_i) - \{\vec{x}\}$. Hence $(\sum_{i=1}^{\ell} \lambda_i S_i') + \{\vec{x}\} = \sum_{i=0}^{\ell} \lambda_i S_i$. Since $\sum_{i=0}^{\ell} \lambda_i \equiv_m 1$ and $S_i \subseteq S'$ for all $1 \leq i \leq \ell$ it follows that $\sum_{i=0}^{\ell} \lambda_i S_i \subseteq S'$, hence $\sum_{i=1}^{\ell} \lambda_i S_i' \subseteq M'$. Therefore $M' \in \mathsf{Module}_m^k$ and $S' \in \mathsf{Affine}_m^k$ as required.

- To show $S' \subseteq \mathsf{affine}(S)$. It is sufficient to show $\sum_{i=1}^{\ell} \lambda_i S_i \subseteq \mathsf{affine}(S)$ for any $S_i \subseteq \mathsf{affine}(S)$, $\lambda_i \in \mathbb{Z}$ with $\sum_{i=1}^{\ell} \lambda_i \equiv_m 1$. Let $\vec{y} \in \mathbb{Z}^k$ and $M'' \in \mathsf{Module}_m^k$ such that $\mathsf{affine}(S) = \{\vec{y}\} + M''$. Put $S_i' = \{-\vec{y}\} + S_i$ for all $1 \leq i \leq k$. Then $\sum_{i=1}^{\ell} \lambda_i S_i = \{\vec{y}\} + (\sum_{i=1}^{\ell} \lambda_i S_i')$, hence $\{-\vec{y}\} + (\sum_{i=1}^{\ell} \lambda_i S_i) = \sum_{i=1}^{\ell} \lambda_i S_i' \subseteq M''$. Thus $\sum_{i=1}^{\ell} \lambda_i S_i \subseteq \{y\} + M'' = \mathsf{affine}(S)$ as required.

Lemma A.3 Let $[A|\vec{b}]$ be an integer matrix. Then there exists an integer matrix $[A'|\vec{b'}]$ in upper triangular form such that $A\vec{x} \equiv_m \vec{b}$ iff $A'\vec{x} \equiv_m \vec{b'}$.

Proof A.3 Follows by the triangularisation algorithm of Müller-Olm and Seidl [14].

Proposition A.1 $S \in \mathsf{Affine}^k_m$ iff there exists a congruence system $A\vec{x} \equiv_m \vec{b}$ such that $S = \{\vec{x} \in \mathbb{Z}^k_m \mid A\vec{x} \equiv_m \vec{b}\}$.

Proof A.4

- Suppose $S \in \mathsf{Affine}^k_m$. Then $S = \{\vec{y}_0\} + M$ for some $\vec{y}_0 \in \mathbb{Z}^k_m$ and we have $M \in \mathsf{Module}^k_m$. Let $\{\vec{y}_i\}^\ell_{i=1} \subseteq M$ such that $\mathsf{linear}(\{\vec{y}_i\}^\ell_{i=1}) = M$ and let $\vec{\lambda} = \langle \lambda_1, \ldots, \lambda_\ell \rangle$. Then

$$S = \left\{ \vec{x} \in \mathbb{Z}^k_m \;\middle|\; [-\vec{y}_1 \,|\, \cdots \,|\, -\vec{y}_\ell \,|\, I] \begin{bmatrix} \vec{\lambda} \\ \vec{x} \end{bmatrix} \equiv_m \vec{y}_0 \right\}$$

  The above system can be put into the following upper triangular form

$$\begin{bmatrix} C & B \\ 0 & A \end{bmatrix} \begin{bmatrix} \vec{\lambda} \\ \vec{x} \end{bmatrix} \equiv_m \begin{bmatrix} \vec{d} \\ \vec{b} \end{bmatrix}$$

  whence it follows that $S = \{\vec{x} \in \mathbb{Z}^k_m \mid A\vec{x} \equiv_m \vec{b}\}$ as required.

- Suppose $S = \{\vec{x} \in \mathbb{Z}^k_m \mid A\vec{x} \equiv_m \vec{b}\}$. Let $S_i \subseteq S$ and $\lambda_i \in \mathbb{Z}$ where $\sum^\ell_{i=1} \lambda_i \equiv_m 1$. Let $\vec{x} \in \sum^\ell_{i=1} \lambda_i S_i$. Then $\vec{x} \equiv_m \sum^\ell_{i=1} \lambda_i \vec{x}_i$ for some $\vec{x}_i \in S_i$. Then $A\vec{x} \equiv_m A(\sum^\ell_{i=1} \lambda_i \vec{x}_i) = \sum^\ell_{i=1} \lambda_i A\vec{x}_i \equiv_m \sum^\ell_{i=1} \lambda_i \vec{b} \equiv_m \vec{b}$. Hence $\vec{x} \in S$ and $\sum^\ell_{i=1} \lambda_i S_i \subseteq S$. By Lemma A.2 and Lemma A.1, $S = \mathsf{affine}(S) \in \mathsf{Affine}^k_m$ as required.

Proposition A.2 Let $S \in \mathsf{Affine}^k_m$ and $1 \leq i_1 < \ldots < i_j \leq k$. Then $T \in \mathsf{Affine}^j_m$ where $T = \{\langle \pi_{i_1}(\vec{x}), \ldots, \pi_{i_j}(\vec{x}) \rangle \mid \vec{x} \in S\}$.

Proof A.5 Suppose that $\{\langle \pi_{i_1}(\vec{x}_n), \ldots, \pi_{i_j}(\vec{x}_n) \rangle\}^\ell_{n=1} \subseteq T$ and let $\sum^\ell_{n=1} \lambda_n \equiv_m 1$. Then $\{\vec{x}_n\}^\ell_{n=1} \subseteq S$ and $\vec{x} \in S$ where $\vec{x} \equiv_m \sum^\ell_{n=1} \lambda_n \vec{x}_n$. Hence we have $\sum^\ell_{n=1} \lambda_n \langle \pi_{i_1}(\vec{x}_n), \ldots, \pi_{i_j}(\vec{x}_n) \rangle \equiv_m \langle \pi_{i_1}(\vec{x}), \ldots, \pi_{i_j}(\vec{x}) \rangle \in T$ as required.

Lemma A.4 Suppose $S_i = \{\vec{y}_i\} + M_i$ where $\vec{y}_i \in \mathbb{Z}^k_m$, $M_i \in \mathsf{Module}^k_m$ and $M_i \neq \varnothing$ for $i \in \{1, 2\}$. Then $\mathsf{affine}(S_1 \cup S_2) = \{\vec{y}_1\} + M + M_1 + M_2$ where M is defined $M = \{\vec{x} \in \mathbb{Z}^k_m \mid \lambda \in \mathbb{Z} \wedge \vec{x} \equiv_m \lambda(\vec{y}_2 - \vec{y}_1)\}$

Proof A.6 Write $M' = \{\vec{y}_1\} + M + M_1 + M_2$.

- To show $\mathsf{affine}(S_1 \cup S_2) \subseteq M'$. Observe $S_1 = \{\vec{y}_1\} + M_1 \subseteq M'$. Since $\vec{y}_2 \in \{\vec{y}_1\} + M$ it follows that $S_2 \subseteq (\{\vec{y}_1\} + M) + M_2 \subseteq M'$. Thus $S_1 \cup S_2 \subseteq M'$ hence $\mathsf{affine}(S_1 \cup S_2) \subseteq \mathsf{affine}(M') = M'$ as required.

- Since $S_1 \neq \varnothing$, $\vec{y}_1 \in S_1 \subseteq \mathsf{affine}(S_1 \cup S_2)$ hence there exists $M'' \in \mathsf{Module}_n^k$ such that $\mathsf{affine}(S_1 \cup S_2) = \{\vec{y}_1\} + M''$. It remains to show $M + M_1 + M_2 \subseteq M''$. Because $S_2 \neq \varnothing$, $\vec{y}_2 \in S_2 \subseteq \mathsf{affine}(S_1 \cup S_2) \subseteq \{\vec{y}_1\} + M''$ it follows that $\{\vec{y}_1\} + M'' = \{\vec{y}_2\} + M''$ hence $\vec{y}_2 - \vec{y}_1 \in M''$ and $M \subseteq M''$. Moreover $\{\vec{y}_1\} + M_1 = S_1 \subseteq \mathsf{affine}(S_1 \cup S_2) = \{\vec{y}_1\} + M''$. Therefore $M_1 \subseteq M''$. Likewise $M_2 \subseteq M''$ since $\{\vec{y}_2\} + M_2 = S_2 \subseteq \mathsf{affine}(S_1 \cup S_2) = \{\vec{y}_1\} + M'' = \{\vec{y}_2\} + M''$. Therefore $M + M_1 + M_2 \subseteq M''$ whence $\{\vec{y}_1\} + M + M_1 + M_2 \subseteq \{\vec{y}_1\} + M'' = \mathsf{affine}(S_1 \cup S_2)$ as required.

Proposition A.3 Let $S_i = \{\vec{x} \in \mathbb{Z}_m^k \mid A_i\vec{x} \equiv_m \vec{b}_i\}$ and

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ -\vec{b}_1 & 0 & A_1 & 0 & 0 \\ 0 & -\vec{b}_2 & 0 & A_2 & 0 \\ 0 & 0 & -I & -I & I \end{bmatrix} \quad S = \left\{ \vec{x} \in \mathbb{Z}_m^k \;\middle|\; A \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vec{x}_1 \\ \vec{x}_2 \\ \vec{x} \end{bmatrix} \equiv_m \begin{bmatrix} 1 \\ \vec{0} \\ \vec{0} \\ \vec{0} \end{bmatrix} \right\}$$

Then $S = \mathsf{affine}(S_1 \cup S_2)$ if $S_1 \neq \varnothing$ and $S_2 \neq \varnothing$.

Proof A.7 Let $S_i = \{\vec{y}_i\} + M_i$ where $S_1 \neq \varnothing$ and $S_2 \neq \varnothing$. Therefore $\vec{y}_i \in S_i$ and $M_i = \{\vec{x} \in \mathbb{Z}_m^k \mid A_i\vec{x} \equiv_m \vec{0}\}$ where $M_i \neq \varnothing$. Put $M = \{\vec{x} \in \mathbb{Z}_m^k \mid \lambda \in \mathbb{Z} \wedge \vec{x} \equiv_m \lambda(\vec{y}_2 - \vec{y}_1)\}$. By Lemma A.4 it suffices to show that $\{\vec{y}_1\} + M + M_1 + M_2 = S$.

- To show $\{\vec{y}_1\} + M + M_1 + M_2 \subseteq S$.
  - Observe $A[1, 0, \vec{y}_1, \vec{0}, \vec{x}]^T \equiv_m [1, \vec{0}, \vec{0}, \vec{0}]^T$ for $\vec{x} = \vec{y}_1$. Therefore $\vec{y}_1 \in S$.
  - Observe for any $\lambda \in \mathbb{Z}$ and $\vec{x} = \lambda(\vec{y}_2 - \vec{y}_1)$, $A[-\lambda, \lambda, -\lambda\vec{y}_1, \lambda\vec{y}_2, \vec{x}]^T \equiv_m \vec{0}$. Therefore $\{\vec{y}_1\} + M \subseteq S$.
  - Observe $A[0, 0, \vec{x}, \vec{0}, \vec{x}]^T \equiv_m \vec{0}$ for any $\vec{x}$ with $A_1\vec{x} = \vec{0}$. Thus $\{\vec{y}_1\} + M_1 \subseteq S$ and by a similar argument $\{\vec{y}_1\} + M_2 \subseteq S$.

  Therefore $\{\vec{y}_1\} + M + M_1 + M_2 \subseteq S$ as required.

- To show $S \subseteq \{\vec{y}_1\} + M + M_1 + M_2$. Let $A[\sigma_1, \sigma_2, \vec{x}_1, \vec{x}_2, \vec{x}]^T \equiv_m \vec{0}$ since $\vec{y}_i \in S$. Then $\vec{x} \equiv_m \vec{x}_1 + \vec{x}_2$ where $A_1\vec{x}_1 \equiv_m -\lambda\vec{b}_1$ and $A_2\vec{x}_2 \equiv_m \lambda\vec{b}_2$ for some $\lambda \in \mathbb{Z}$. Observe $\vec{x}_1 \equiv_m \mu_1\vec{z}_1 - \lambda\vec{y}_1$ and $\vec{x}_2 \equiv_m \mu_2\vec{z}_2 + \lambda\vec{y}_2$ for some $\mu_i \in \mathbb{Z}$ and $A_i\vec{z}_i \equiv_m \vec{0}$. Therefore $\vec{x} \equiv_m \lambda(\vec{y}_2 - \vec{y}_1) + \mu_1\vec{z}_1 + \mu_2\vec{z}_2$ whence $S \subseteq \{\vec{y}_1\} + M + M_1 + M_2$ as required.