psolve : Deciding Satisfiability for Presburger Formulae Using Automata, Rewriting and a Model Checker

Michael Jones, Ganesh Gopalakrishnan, Palaith Narendran

October 13, 1999

Abstract

Presburger formulae are an expressive but decidable language of arithmetic expressions and boolaen connectives with quantification. psolve is a prototype of an automata based tool for deciding the satisfiability of Presburger formulae extended with a predicate that recognizes powers of two. The unique features of psolve are the use of model checkers to examine the automata, the use of concurrent automata and the inclusion of the powers of two predicate. psolve was used to compare the time requirements for implicit and explicit state ennumeration model checkers as the automata animation tools. On our simple examples, we found that implicit state ennumeration model checkers require less time. psolve will eventualy be used in a type system for a hardware description language which uses Presburger formulae as a type language.

1 Introduction

Presburger formula cover linear arithmetic with quantification and inequalities. When conjoined together, several Presburger formulae form a system of linear constraints. While Presburger formulae are decidable, the complexity of deciding systems of Presburger equations is $2^{2^{2^n}}$.

One way to decide satisfiability for Presburger formulae is with automata operating on two's complement representations of the integers. psolve builds and checks the automata representation using term rewriting and existing model checkers to improve efficiency. Term rewriting replaces expressions with an equivalent conjunction of expressions which have a more compact automata representation. By using existing model checkers, we exploit recent advances in model checking algorithms. One purpose of the current psolve prototype is to determine which flavor (explicit or implicit) of model checking is best suited for this application.

The psolve prototype supports a subset of Presburger arithmetic. The actual subset of Presburger equations supported by psolve is an artifact of the current representation

of multiplication. At present, any variables in equations which include linearly multiplied variables can not appear in any other equations. This will be addressed in future releases.

The next section briefly surveys related work. Section 3 outlines the algorithms used in this approach. Section 4 contains the results of using this approach on several simple equations. Section 5 contains the conclusions and future work suggested by this report.

2 Related Work

Boudet and Comon give a technique for handling full Presburger arithmetic on systems of equations in [BC96]. Their method uses divisibility to create a single automata for an equation which solves an entire equation modulo its solution.

Wolper and Boigelet also have a technique for deciding satisfiability for Presburger arithmetic [WB95]. The Wolper method relies on concurrent automata to represent parts of an equation. The concurrent automata are implicitly conjoined and run in parallel. The paper also includes a clever method for dealing with multiplication by a constant. In their method, the constant multiplicand is expanded to its binary representation then computed using addition of powers of two. For example, $3x$ is reduced to $(1*(2^1)+1*(2^0))x$. which is $(2^1)x+(2^0)x$. Multiplication by powers of 2 is then easily represented by repeated addition of the form $x + x = (2^1)x$. An advantage to this approach is that it does not limit the expressions that can be represented. `psolve` does require that multiplied variables appear in no other equations.

Kukala, Shiple and Aziz have also studied the application of automata based satisfiability checker for Presburger arithmetic [SKR98] but in the context of hardware verification. We do not compare `psolve` to [SKR98] in this report.

3 The Method

Our method for deciding satisfiability of a subset of Presburger formulae using automata has three steps:

1. Rewrite the equation into a conjunction of simpler equations each of which can be represented using an automaton.
2. Translate each conjunct into an automaton.
3. Use an model checker (SPIN [Hol91] or VIS [Gro96]) to determine if the accepting state is reachable. If the accepting state is reachable, then the original formula is satisfiable.

The algorithms involved in each of the three steps are presented in the next section. Section 3.2 attempts to tie the algorithms together with a brief concrete example. Section 3.3 briefly discusses the current implementation of each algorithm.

3.1 Algorithms

3.1.1 Rewriting

Our approach relies on using simple automata to represent three arithmetic operations: addition, equality and divisibility. Rewriting the original equation into simpler equations involves two sets of rules: rules for reducing the equation to a normal form and rules for breaking the equation into pieces manageable by the automata. The normal form rules move constants in linear multiplication (the only kind allowed in Presburger arithmetic) to the right, move addition to the left, and move multiplied terms in addition expression to the left. The normalizing rules are given below.

$$n \times C \rightarrow C \times n \quad (1)$$

$$a + n \times C \rightarrow n \times C + a \quad (2)$$

$$(a + b) + c \rightarrow a + (b + c) \quad (3)$$

Where C is a constant and the remaining terms are variables over the integers. The reducing rules introduce auxiliary variables to transform the equation into a conjunction of 2-place additions involving only variables, replace multiplication by multiples of the constant multiplicand and eliminate constants in addition terms. Some of the transformation rules are listed below:

$$(C \times x) + n = z \rightarrow (x' MOD C = 0) \wedge (x' + n = z) \quad (4)$$

$$(M \times x) + (N \times y) = z \rightarrow (x' MOD M = 0) \wedge (y' MOD N = 0) \wedge (x' + y' = z) \quad (5)$$

$$a + (b + c) = z \rightarrow a + b = c' \wedge c' + c = z \quad (6)$$

Once again, C, M and N are constants and the remaining terms are variables. Normalizing the equation reduces the number of simplification rules required and simplifying the equation allows small automata to represent each conjunct.

In the future, we envision a third set of rules for eliminating commonly encountered trivialy true or trivialy false expressions. It is anticipated that sets of rewrite rules targeted to a specific application domains will be most effective in simplifying expressions before the automata representation is created.

3.1.2 Automata Representation

Basic arithmetic operations can be represented by automata operating on words from $0, 1$. If an automata accepts a sequence of words from its input language then the two's complement integers represented by that sequence (with the least significant bit first) satisfy the formula represented by the automata. For example, one of the sequences (or words) accepted by the automata for equality is $(1,1),(1,1),(0,0)$; which when interpreted as a pair of two's compliment numbers means $3 = 3$. The automata for addition takes a triple (a, b, c) as input and recognizes sequences that encode $a + b = c$ in two's complement. The divisibility by a constant c automata recognizes positive or negative numbers that are divisible by c. Powers of two are recognized by another automton and the equality automata takes pairs of numbers and recognizes pairs of equal numbers. If an automata requires a tuple of inputs then each bit stream in the tuple must be the same length. This

can be accomplished by sign-extending shorter bit streams to match the longest stream in the tuple.

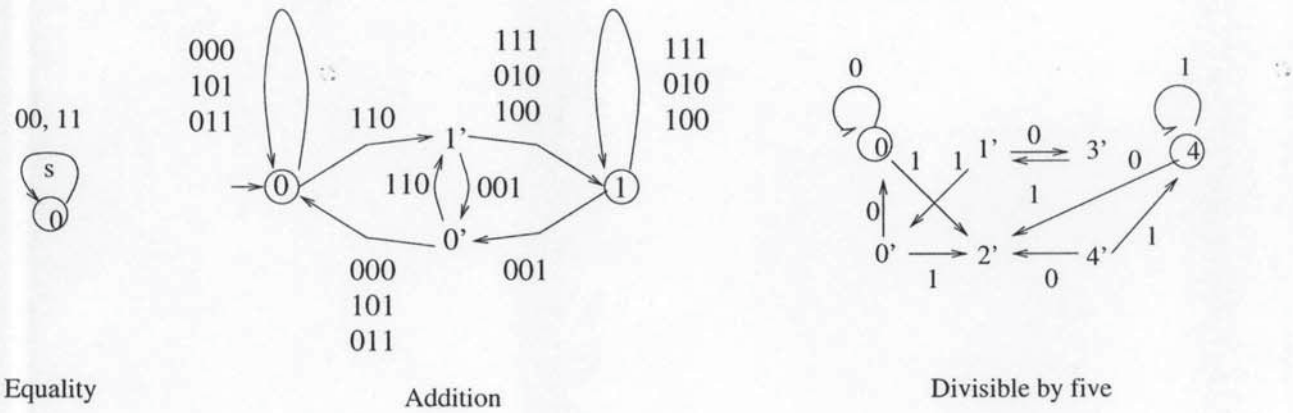

Figure 1: Automata for three simple arithmetic expressions.

Each basic automata is shown in figure 1. A single Presburger formula is represented by one or more of these automata. In the figure, halting states are circled and transitions to the dead state are omitted. Not shown is the automata for recognizing powers of two. The powers of two automata accepts bit streams containing exactly one one (such as "0010").

All of the automata for a single Presburger formula are then executed in parallel on the same inputs. Boolean connectives are represented by boolean operations on the accepting states of the automata in the concurrent automaton representation. For example, the concurrent automata representation of a disjunction of three formluae accepts an input stream if one of the three concurrent automata reaches the accepting state and the concurrent automaton representing a conjuction of formulae accepts only input streams that are accepted by all of the automata.

The construction of the transition function for divisibility automata is particularly interesting. If a constant M divides a number n then $nMOD\ M = 0$ or $Mx + 0 = n$ for some x. But this is just $x + x + ...x + 0 = n$ which can be recognized by an automaton which takes x and n as inputs. However, since we aren't interested in the value of x, we throw x away and are left with a single-input automata on n. Constructing the transition function for an automata which recognizes numbers divisible by a particular M can be done by observing a pattern in the transition tables. Here is the transition table for an automata that recognizes numbers divisible by 5:

x	n	0	0'	1'	2'	3'	4'	4
00000	0	0	0	x	1'	x	2'	2'
00000	1	x	x	0'	x	1'	x	x
11111	0	x	x	3'	x	4'	x	x
11111	1	2'	2'	x	3'	x	4	4

The states are numbered across the top of the table and states 0 and 4 and the halting states, state 0 is the start state and state x is the dead state. The pattern becomes more

obvious when the values for x are ignored and both rows for $n = 0$ and both rows for $n = 1$ are combined:

n	0	0'	1'	2'	3'	4'	4
0	0	0	3'	1'	4'	2'	2'
1	2'	2'	0'	3'	1'	4	4

For odd numbers, such as 5, the next state for the primed states, can be determined by a function on the current state q, the current input n and the divisor m ($m = 5$ in the previous example).

$$q'(q, n, m) = ((q + n) MOD\ 2 \times m + q - n) DIV 2$$

And the unprimed states are simply copies of their primed counterparts. The situation for even numbers is slightly more complicated. Here is the folded transition table for an automata the recognizes numbers divisible by 4:

n	0	0'	1'	2'	3'	3
0	(0,2')	(0,2')	x	(1',3')	x	x
1	x	x	(0',2')	x	(1',3)	(1',3)

The transition function for even numbers is non-deterministic. The non-determinism can be eliminated however using a standard algorithm (the construction of automata for recognizing numbers divisible by a constant is given as an exercise on page 89 of [Sip97].

3.1.3 Model Checker Representation

There are two code generators for translating the internal automata representation into input for the VIS and Spin model checkers. In both cases, the automata are modeled using if,then,else statements which model the transition functions of each automaton.

We then use the model checker to determine if the concurrent automaton ever reaches its accepting state. This is done by asserting that the accept state is unreachable. If the accept state is reachable, then formulae represented by the automaton is satisfiable and a solution is given by the counter example returned by the model checker.

The entire PROMELA model for the simple example equation in section 3.2 is given and explained in more detail in appendix A.

3.2 Example

A simple example will illustrate the method. Suppose we begin with the equation

$$3 * x + y = z$$

We want to determine if this equation is satisfiable.

First, we apply the rewrite rules to transform the equation into a series of conjuncts in the form required for the automata representation. Rewriting results in the equation:

$$(fvb\ MOD\ 3 = 0) \wedge (fvb + y = z)$$

fvb =	0	1	0	0	1	1	1	0	1	1	1	0	1	0	0
y =	1	0	1	1	0	1	1	0	0	1	1	0	1	0	0
z =	1	1	1	1	1	0	1	1	1	0	1	1	0	1	0

Table 1: Satisfying variable assignment for $3x + y = z$ found by SPIN

fvb =	0	0
y =	0	0
z =	0	0

Table 2: Satisfying variable assignment for $3x + y = z$ found by VIS

The variable fvb was introduced by the rewriter and $fvb\ MOD\ 3 = 0$ is equivalent to "fvb is divisible by 3."

Each of these terms are then represented with a single automaton. Two automata are required to represent the original equation. The automata are then combined to form a single PROMELA or Verilog model. A fragment of the PROMELA model is given in figure 3.2 In the figure, the line below label A defines the acceptance condition. The never claim for this model states that the acceptance condition can not happen. The `if` statement below label B non-deterministically generates values of `z`. And the `if` statement following label C gives the transition function for the automata representing $fvb\ MOD\ 3 = 0$. The flag `a1` is set if and only if the automata is in an accepting state.

The entire process of rewriting the equation, generating the automata, making the PROMELA or Verilog model and checking for satisfiability have been combined and can be executed with an SML function call. The function call for using SPIN from within SML is:

```
spin --"((((3 * x) + y) = z)"--;
```

For this equation, SPIN finds a violation of the never claim in 0.4 seconds–indicating that the equation is satisfiable. The satisfying assignment can be found by performing a guided simulation on the resulting trace file. SPIN found the assignment shown in table 3.2. Which when read as a two's complement number with the LSB at the left, gives the assignments $fva = 12,018, y = 11,501$ and $z = 23,519$ but $3x = fva$ so $x = 4,006$. Note that $3(4006) + 11501 = 23519$. This particular solution required traversing 1,368 states. This solution is correct, but certainly not minimal.

The equation can also be checked using the BDD based VIS model checker using a different SML function call:

```
vis --"((((3 * x) + y) = z)"--;
```

VIS finds a violation in 0.1 seconds and the counter example yields the assignment shown in table 3.2. Which assigns each variable to zero. This solution is both correct and minimal.

3.3 Implementation

The equation rewriting was done using an SML rewriter based loosely on Michael Gordon's simple prover from [Gor93]. Another SML program transformed each conjunct of the

```
    /* LABEL A */
#define p     (a1 && a2 && all_run)

/*  global variables */

bit all_run, a1, a2, z, y, fvb;

proctype equation ()
{

do :: atomic {

/* generate input values non-deterministicaly */
    /* LABEL B */
if
    :: 1 -> z = 1
    :: 1 -> z = 0
fi;
    /* LABEL C */

if
  :: (state1 == 1) && (fvb == 0) -> state1 = 1; a1 = 1
  :: (state1 == 2) && (fvb == 0) -> state1 = 1; a1 = 1
  :: (state1 == 1) && (fvb == 1) -> state1 = 3; a1 = 0
  :: (state1 == 2) && (fvb == 1) -> state1 = 3; a1 = 0
  :: (state1 == 3) && (fvb == 0) -> state1 = 5; a1 = 0
  :: (state1 == 3) && (fvb == 1) -> state1 = 2; a1 = 0
  :: (state1 == 4) && (fvb == 0) -> state1 = 3; a1 = 0
  :: (state1 == 5) && (fvb == 0) -> state1 = 3; a1 = 0
  :: (state1 == 4) && (fvb == 1) -> state1 = 4; a1 = 1
  :: (state1 == 5) && (fvb == 1) -> state1 = 4; a1 = 1
  :: (1) -> state1 = 0 ; a1 = 0;
  :: (state1 == 0) ->  state1 = 0 ; a1 = 0;
fi;

...
od
```

Figure 2: Partial PROMELA model for $3x + y = z$.

Equation	SPIN	VIS	satisfiable?
((a + a) = 121)	0.5	1.21	no
((a + (b + (c + d))) = 121)	5:47.3	4.4	yes
(((3 * d) + (5 * c)) = 120)	6.4	1.6	yes
(a + (b + c) = d)	1.3	1.2	yes
(a + (b + (c + (e + f))) = d)	33:59.3	2.8	yes
(a + (b + (c + (e + (f + g)))) = d)	3 hrs (DNF)	3.8	yes
sum of 17 variables	-	22.2	yes
((x + ((2 * y) + (3 * z))) = 2)	1:29.3	1.8	yes
(a + (b + ((2^x) + 4)) = 20)	-	6.7	yes

Table 3: Results of checking satisfiability for several simple equations.

simplified equation into a finite automata data structure. These automata were then combined and used to generate a PROMELA or a Verilog model of the automata running synchronized on the same non-deterministically generated input streams. The PROMELA program was then executed using SPIN v.3.0.4 on a ULTRASPARC-I with 64MB of RAM. And the Verilog model was checked using VIS on the same machine.

4 Results

Table 4 contains the SPIN and VIS results for checking each equation for satisfiability using automata. The SPIN verification for each equation was run in exhaustive search mode with a maximum depth of 10,000 states. Memory usage is reported in MB and the the depth at which the satisfying assignment was found is reported in parenthesis next to the depth reached in the state exploration. In both cases, the time required refers to real time spent model checking as returned by the Unix `time` command and does not include translation or file reading time.

The first set of equations explores the time required to test equality with a constant for simple addition and multiplication. In all but the first case, VIS requires less time than SPIN. The second set of equations tests the affect of adding more variables to a simple expression. VIS easily outperforms VIS on these equations. In the second equation of this set, SPIN did not finish after three hours. The first equation in the final set is the example from the Boudet paper [BC96]. The final equation uses the "power of two" automata.

5 Conclusions and Future Work

We have discussed a preliminary implementation of a method for deciding satisfiability of a limited subset of Presburger arithmetic. The equations tested suggest that BDD based symbolic model checking requires less time to find solutions to arithmetic equations represented as automata in a model checker. This could be because SPIN uses a depth first

state exploration algorithm. The test cases also suggest that this method is more sensitive to the number of variables than the size of constants in the equations.

Having built a working prototype and used it to compare implicit and explicit state enumeration model checking, we now turn our attention to two areas. First, we will improve the efficiency and expressiveness of the `psolve` tool. Efficiency will be improved by studying variable orderings for BDD representations of the concurrent automata and improving the rewriter. Expressiveness will be added by changing our representation of multiplication to allow multiplied variables to appear in other expressions. Although none of the test expressions included boolean connectives and quantification, `psolve` does support them. Second, we plan to develop a type checker for a type language based on Presburger formulae and investigate its use in hardware description languages. Predicate subtyping has enjoyed some success in hardware verification using the PVS theorem prover. The type language of PVS is undecidable thus requiring interactive proofs in some type checking. A type language based on Presburger formulae will give a type language that is expressive but still decidable.

References

[BC96] A. Boudet and H. Comon. Diophantine equations, presburger arithmetic and finite automata. In H. Kirchner, editor, *Trees and Algebra in Programming-CAAP*, volume 1059 of *LNCS*, pages 30–43. Springer-Verlag, 1996.

[Gor93] Michael Gordon. *Programming Language Theory and Implementation.* Prentice-Hall, 1993.

[Gro96] The VIS Group. VIS: A system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[SKR98] T.R. Shiple, J.H. Kukala, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In A. Hu and M. Vardi, editors, *Proc. Computer Aided Verification*, LNCS. Springer-Verlag, 1998.

[WB95] P. Wolper and B. Boigelot. An automata-theoretic approach to presburger arithmetic constraints. In *Proc. Static Analysis Symposium*, volume 983 of *LNCS*, pages 21–32, Glasgow, September 1995. Springer-Verlag.

A SML transcript and PROMELA model for the Example

Recall that the original equation from section 3.2 was $3x + y = z$, was rewritten to

$$(fvb\ MOD\ 3 = 0) \wedge (fvb + y = z)$$

and represented with an automaton for each conjunct. First, we give a transcript for the SML session in which the equation was tested for satisfiability. In this transcript, commands entered by the user (there's only one) are proceeded by an SML>

```
SML> --"(((3 * x) + y) = z)"--;

after reduction...
~"(((fvx MOD 3) = 0) and ((fvx + y) = z))"~~
generating promela code...
compiling and verifying promela code...
/home/cs/grad/mjones/cs611/project/sml/spin.ksh
working... a big spin dump will happen soon...
val it = () : unit
- warning: for p.o. reduction to be valid the never claim must be
stutter-closed
(never claims generated from LTL formulae are stutter-closed)
pan: claim violated! (at depth 285)
pan: wrote pan_in.trail
(Spin Version 3.0.4 -- 10 September 1997)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never-claim         +
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness disabled)
invalid endstates - (disabled by never-claim)

State-vector 24 byte, depth reached 379, errors: 1
      64 states, stored
    1304 states, matched
    1368 transitions (= stored+matched)
    9105 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

2.542  memory usage (Mbyte)
```

```
real        0.4
user        0.1
sys         0.0
```

The PROMELA model for the example equation follows. First, we define the acceptance condition p. The acceptance condition states that all automata have reached their accepting states and all input has been consumed (`a1` is the acceptance flag for automata 1 and `a2` is the acceptance flag for automata 2). Within the `proctype equation ()`, first the state variables are declared and initialized. Then the `all-run` variable, which indicates that all the automata have run on the current inputs, is set to zero and the new input values are generated. The first large `if` statement is the transition function for recognizing divisibility by 3 and the second large `if` statement recognizes addition of $fvx + y = x$. The final two branches of each transition function is the implicit transition to the dead state and a loop to remain in the dead state. This is repeated until the never claim is either violated or validated. The `init` process clears all acceptance flags and runs the equation. Finally, the `never` claim is the Buchi automata generated by SPIN to represent the LTL formula "eventually p" where p was defined at the top of the file.

```
/* the acceptance condition.  Denotes that all automata are accepting
   and have consumed this round of input */

#define p     (a1 && a2 && all_run)

/*  global variables */

bit all_run, a1, a2, z, y, fvx;

proctype equation ()
{
byte state1, state2;
state1 = 1; state2 = 1;

do :: atomic {
all_run = 0;
/* generate input values non-deterministicaly */

if
    :: 1 -> z = 1
    :: 1 -> z = 0
fi;
if
    :: 1 -> y = 1
    :: 1 -> y = 0
fi;
if
    :: 1 -> fvx = 1
```

```
    :: 1 -> fvx = 0
fi;


if
  :: (state1 == 1) && (fvx == 0) -> state1 = 1; a1 = 1
  :: (state1 == 2) && (fvx == 0) -> state1 = 1; a1 = 1
  :: (state1 == 1) && (fvx == 1) -> state1 = 3; a1 = 0
  :: (state1 == 2) && (fvx == 1) -> state1 = 3; a1 = 0
  :: (state1 == 3) && (fvx == 0) -> state1 = 5; a1 = 0
  :: (state1 == 3) && (fvx == 1) -> state1 = 2; a1 = 0
  :: (state1 == 4) && (fvx == 0) -> state1 = 3; a1 = 0
  :: (state1 == 5) && (fvx == 0) -> state1 = 3; a1 = 0
  :: (state1 == 4) && (fvx == 1) -> state1 = 4; a1 = 1
  :: (state1 == 5) && (fvx == 1) -> state1 = 4; a1 = 1
  :: (1) -> state1 = 0 ; a1 = 0;
  :: (state1 == 0) ->  state1 = 0 ; a1 = 0;
fi;

if
  :: (state2 == 1) && (fvx == 0) && (y == 0) && (z == 0) -> state2 = 1; a2
= 1
  :: (state2 == 1) && (fvx == 0) && (y == 1) && (z == 1) -> state2 = 1; a2
= 1
  :: (state2 == 1) && (fvx == 1) && (y == 0) && (z == 1) -> state2 = 1; a2
= 1
  :: (state2 == 1) && (fvx == 1) && (y == 1) && (z == 0) -> state2 = 2; a2
= 0
  :: (state2 == 4) && (fvx == 0) && (y == 0) && (z == 0) -> state2 = 1; a2
= 1
  :: (state2 == 4) && (fvx == 0) && (y == 1) && (z == 1) -> state2 = 1; a2
= 1
  :: (state2 == 4) && (fvx == 1) && (y == 0) && (z == 1) -> state2 = 1; a2
= 1
  :: (state2 == 4) && (fvx == 1) && (y == 1) && (z == 0) -> state2 = 2; a2
= 0
  :: (state2 == 3) && (fvx == 1) && (y == 1) && (z == 1) -> state2 = 3; a2
= 1
  :: (state2 == 3) && (fvx == 1) && (y == 0) && (z == 0) -> state2 = 3; a2
= 1
  :: (state2 == 3) && (fvx == 0) && (y == 1) && (z == 0) -> state2 = 3; a2
= 1
  :: (state2 == 3) && (fvx == 0) && (y == 0) && (z == 1) -> state2 = 4; a2
= 0
  :: (state2 == 2) && (fvx == 1) && (y == 1) && (z == 1) -> state2 = 3; a2
```

```
= 1
  :: (state2 == 2) && (fvx == 1) && (y == 0) && (z == 0) -> state2 = 3; a2
= 1
  :: (state2 == 2) && (fvx == 0) && (y == 1) && (z == 0) -> state2 = 3; a2
= 1
  :: (state2 == 2) && (fvx == 0) && (y == 0) && (z == 1) -> state2 = 4; a2
= 0
  :: (1) -> state2 = 0 ; a2 = 0;
  :: (state2 == 0) ->  state2 = 0 ; a2 = 0;
fi;all_run = 1;
}
od
}


/* the init proccess */

init {
all_run = 0;
a1 = 0;
a2 = 0;

atomic{run equation ();
      }
}


/* the never automaton.  a violation implies the equation
   is satisfiable */

never {    /*  <>  p */
T0_init:
if
  :: (1) -> goto T0_init
  :: ((p)) -> goto accept_all
fi;
accept_all:
skip
}
```