I. Introduction
What is PATHG? It’s an application developed over a library of integrated BDD-like diagrams for dense-time model verification. To say how PATHG runs, we first talk a little about REDLIB.

The basic motivation of developing REDLIB is to experiment with the capabilities of BDD-like diagrams that allow the integrated representation and manipulation of state-space characterizations of both discrete and dense variables. The performance of operations on REDLIB is based on an integration of BDD-like diagrams for BDD, MDD, CRD, and HRD. Such integration allows structure-sharing among constraints for discrete and dense spaces. And REDLIB not only allows us to manipulate BDD-like diagrams, but it also allows us to declare behavior structures represented as parameterized communicating automata (PCA).

PATHG is a game-based path explorer for communicating timed automaton (CTA) built on top of REDLIB.

II. Technology of PATHG
The following grammar can generate the model that can be entered into PATHG.

Grammar:
V is a variable declaration.
E is an arithmetic expression.
B is a Boolean condition.
C is a program of IMP commands or “goto name” where name is a mode name.

G ::= P VS [ILS] MS INI [SP]
P ::= process count = E;
VS ::= | V VS
V ::= SCOPE TYPE x [: c .. c ]; // c ∈ N, x : a variable
SCOPE ::= global | local
TYPE ::= discrete | pointer | clock | dense | synchronizer
IL ::= inline TYPE name ( FSL ) { EI }
FSL ::= | FS
FS ::= f | f , FS // f : a formal argument
EI ::= f | x | x[c] // x : a declared discrete variable | ( EI ) | EI+EI | EI-EI | EI*EI |
We give a model that modeling a machine which outputs 1 if the bit string read in so far is divisible by 3; otherwise it outputs 0.

Bit_String_3:
process count = 2;  one for MACHINE, one for STRING

#define MACHINE 1  process number
#define STRING 2

global discrete remain:0..4;  // 2 discrete variables
global discrete output:0..1;

global synchronizer
    one,
    zero;

2 modes for MACHINE
mode a1
shape circle (10)
(true)
{
    when ?one(true)
        may if ((remain + 1) == 3) {
            output = 1;
            remain = 0;
        }
    else {
        output = 0;
        remain = remain + 1;
    }
    goto a2;
    when ?zero(true)
        may
        goto a2;
}

mode a2
shape circle (10)
(true)
{
    when ?one(true)
        may if ((remain + 2) == 3) {
            output = 1;
            remain = 0;

```
else if ((remain + 2) == 4) {
    output = 0;
    remain = 1;
}
else {
    output = 0;
    remain = remain + 2;
}

goto a1;
when ?zero(true)
    may
    goto a1;
}

1 mode for STRING
mode bit_string
shape circle (10)
(true)
{
    when !one(true) may;
    when !zero(true) may;
}
initially
    a1[MACHINE]
    && bit_string[STRING]
    && remain == 0
    && output == 0;

Above is the Bit_String_3 model.
PATHG also uses following procedures to manipulate diagrams:
redgram *red_true()
This procedure returns a diagram for Boolean true.

redgram *red_false()
This procedure returns a diagram for Boolean false.
The procedure returns a BDD-like diagram for constraint format string F with variable numbers of arguments. The format string is like a constraint except that there could be place-holder strings like “%s” and “%d” for strings and integers respectively. The format string F is like those in procedure printf(). The procedure substitutes the i’th arguments in the variable-length argument list for the i’th place-holder strings in F.

The procedure returns a BDD-like diagram for constraint string F interpreted with respect to process pi. The format string is like a constraint except that there could be place-holder strings like “%s” and “%d” for strings and integers respectively. The format string F is like those in procedure printf(). The procedure substitutes the i’th arguments in the variable-length argument list for the i’th place-holder strings in F.

III. PATHG Demo

In each step, PATHG asks the users to select the transitions of the model processes, randomly picks those of the specification and environment processes, and enter necessary data from stream variables.

We give an example, Bit_String_3, and enter a bit string, 10111, to present the demo.

On step 1, we can see that there are four transitions we can choose in the red
rectangle box. The output is zero since variable output_1==0. We choose pi = 2 and xi = 5 to represent entering bit 1 into the machine.

On step 2, the output is zero and we choose pi = 2 and xi = 5 to represent entering bit 1 into the machine. Note that our bit string is 10111, and we actually enter the least significant bit into the machine. When this step is over, the bit string we entered is 11.

On step 3, we can clearly see that the output becomes one! That’s because the bit string we actually entered is 11, and 11 in binary is 3 in decimal. The number 3 can be divided by 3, so the output becomes one. We choose pi = 2 and xi = 5 to represent entering bit 1 into the machine.
Step 4.

On step 6, the final step, we have entered 10111 into the machine, and in decimal, it is 23, so the output is zero finally.
IV. Conclusion

PATHG is an application that can verify models manually. Not as other verification tools, PATHG can be used to do verification in time domain. But there is a disadvantage needs to improve that, when models get larger and larger, running PATHG by hands can be an impossible thing. We can generate testcases and input them to run PATHG automatically. Or instead, we can try to develop a test strategy to automatically executing PATHG and finding bugs.

V. Reference


