Experiments on Blade Model Checker

Experiments are conducted to show the performance of Blade model checker. Performance has two aspects: the runtime and the storage usage. Experimental data show that Blade has shorter runtime and less storage usage than BLAST, the Lazy Abstraction model checker developed at Berkeley. All the experiments on Blade and BLAST are executed on a Linux box with Pentium III 860M CPU and 1G memory.

Currently we are short of a front-end to translate C program to a control flow automata (essentially a control flow diagram). Blade accepts a text file as input, which lists the edges, the transitions (statements) on each edge, the ERROR nodes, and the entry node of the control flow automata. Due to the manual translation from C code to control flow automata, all the experiments presented here are of limited lines of code. Also, we do not support pointer yet.

Runtime

Runtime is measured by the number of calls to a theorem-prover, because the runtime of the theorem-prover dominates that of abstraction-based model checker, such as Blade and BLAST. Blade adopts Simplify, the same theorem-prover adopted by BLAST. The complexity (the number of predicates in a formula) of the formulae in each call to the theorem-prover affects its runtime severely, because theorem proving subsumes the SAT (Satisfiability) problem. The maximum complexity of the formulae is also recorded. The experiment data is listed in Table 1.

Table 1. Runtime

C Program	Lines of Code	Number of Calls to Theorem-Prover			Maximum Complexity of Formulae		
		Blade	BLAST	Craig	Blade	BLAST	Craig
exp1.c	14	2	24	44	1	10	10
exp2.c	63	62	930	916	8	23	18
exp3.c	26	4	958	1028	2	24	23
exp4.c	25	7	76	112	2	8	8
tut2.c	79	24	56	64	5	7	7
inf1.c	12	22	-	-	14	-	-
inf2.c	12	6	-	76	4	-	12
inf3.c	11	6	-	72	4	-	11
inf4.c	11	13	90	108	7	16	15
inf5.c	11	5	-	-	3	-	-

inf6.c	10	5	48	54	3	10	10
inf7.c	8	113	-	20390	56	-	58

In Table 1, Craig means BLAST plus Craig interpolation, i.e., with "`-craig 1`" option. The `tut2.c` is from BLAST package. The "-" means that BLAST threw exceptions during the checking, since it cannot discover new predicates to continue, which probability means BLAST will not terminate for the C program in question. The theoretical explanation is that the problem to determine a proper set of predicates that witness the abstraction is undecidable. Pure BLAST will only try the set of predicates appearing in the program, which may not be in fine enough resolution. Craig interpolation will try to "guess" new predicates, so it has better chance to terminate. However the "guess" is not guaranteed to be accurate and hence the termination is not guaranteed. Blade implements a simple heuristics to resolve monotone loops as presented in `inf1.c` through `inf7.c`. For all experiments, Blade produce the same correct results with BLAST when BLAST terminates normally, i.e., it either returns no counterexample for safe programs or lists a counterexample for unsafe ones, though the counterexamples could be different from what produced by BLAST.

Table 1 shows that Blade incurs much less number of calls (usually less in magnitude) to the theorem-prover than BLAST. The maximum complexity of formulae is always less than that of BLAST. BLAST may push on to stack several predicates as premises before it incurs a call to the theorem-prover, the complexity of which is counted by adding them to that of forth coming formulae.

Another interesting experiment is illustrated by Figure 1. We alter the upper limit of the `for` loop in `inf7.c` between 10 and 60 and observe the runtime in wall-clock-time. BLAST exhibits an exponential growth on the runtime, while the data suggests a low degree polynomial growth on the runtime of Blade.

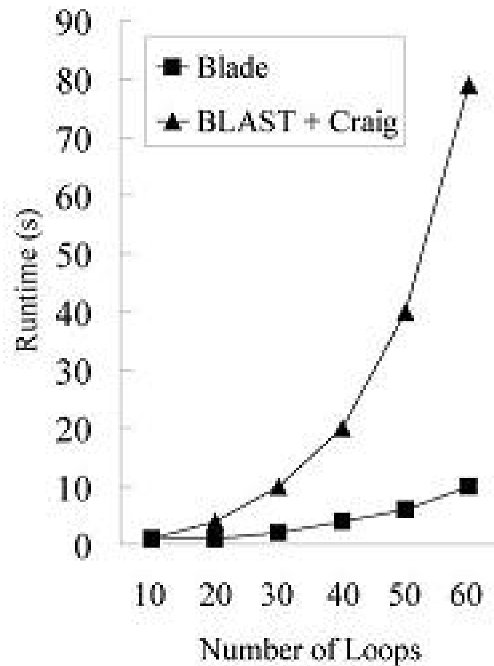

Figure 1. Exponential growth *vs.* low degree polynomial

Storage usage

Storage usage is measured by the number of predicates stored per node in the reachability tree. A reachability tree is generated during the model checking process, to record whether the statements are checked or not. Statements inside a loop or below a branch may occur multiple times in the reachability tree to account for their different status. The number of predicates per node of BLAST is estimated by a half of the maximum number of predicates it discovers, which is a fair estimation since usually the number of predicates grows monotonically along each path in the reachability tree. However, Craig interpolation will break the monotone increase, so we only list the maximum number of predicates. The experimental data is listed in Table 2.

Table 2. Storage usage

C Program	Lines of Code	Blade	BLAST	Craig
exp1.c	14	0.571	3 / 2 = 1.5	3
exp2.c	63	1.694	13 / 2 = 6.5	7
exp3.c	26	0.938	15 / 2 = 7.5	14

exp4.c	25	1.2	3 / 2 = 1.5	4
tut2.c	79	0.714	3 / 2 = 1.5	3
inf1.c	12	0.667	-	-
inf2.c	12	0.222	-	3
inf3.c	11	0.25	-	3
inf4.c	11	0.625	4 / 2 = 2	3
inf5.c	11	0.25	-	-
inf6.c	10	0.286	4 / 2 = 2	3
inf7.c	8	0.392	-	52

Table 2 shows that the storage usage of Blade is more parsimonious than BLAST. It also shows that Craig interpolation is not guaranteed to produce less number of predicates, as in exp4.c.

Experimental C code

exp1.c

```
#include <assert.h>

int main()
{
    int a, b, c;

    a = 1;
    b = 1;
    c = 1;
    assert( a == 1 );
    assert( b == 1 );
    assert( c == 1 );
    return 0;
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) a := 1
ASSIGN (1,2) b := 1
ASSIGN (2,3) c := 1
ASSUME (3,4) (NEQ a 1)
```

```
ASSUME (3,5) (NEQ b 1)
ASSUME (3,6) (NEQ c 1)

ERROR 4
ERROR 5
ERROR 6

ENTRY 0
```

exp2.c

```
#include <assert.h>

int main()
{
    int a, b, c, d, e, f, g, h;

    if ( a == b )
    {
        if ( b == c )
        {
            if ( c == d )
            {
                if ( d == e )
                {
                    if ( e == f )
                    {
                        if ( f == g )
                        {
                            if ( g == h )
                            {
                                assert( a == b );
                                assert( a == c );
                                assert( a == d );
                                assert( a == e );
                                assert( a == f );
                                assert( a == g );
                                assert( a == h );

                                assert( b == c );
                                assert( b == d );
                                assert( b == e );
                                assert( b == f );
                                assert( b == g );
                                assert( b == h );

                                assert( c == d );
```

```
                                        assert( c == e );
                                        assert( c == f );
                                        assert( c == g );
                                        assert( c == h );

                                        assert( d == e );
                                        assert( d == f );
                                        assert( d == g );
                                        assert( d == h );

                                        assert( e == f );
                                        assert( e == g );
                                        assert( e == h );

                                        assert( f == g );
                                        assert( f == h );

                                        assert( g == h );
                                    }
                                }
                            }
                        }
                    }
                }
            }
        return 0;
    }
```

The corresponding control flow automata:

```
ASSUME (0,1) (EQ a b)
ASSUME (1,2) (EQ b c)
ASSUME (2,3) (EQ c d)
ASSUME (3,4) (EQ d e)
ASSUME (4,5) (EQ e f)
ASSUME (5,6) (EQ f g)
ASSUME (6,7) (EQ g h)
ASSUME (7,8) (NOT (EQ a b))
ASSUME (7,9) (NOT (EQ a c))
ASSUME (7,10) (NOT (EQ a d))
ASSUME (7,11) (NOT (EQ a e))
ASSUME (7,12) (NOT (EQ a f))
ASSUME (7,13) (NOT (EQ a g))
ASSUME (7,14) (NOT (EQ a h))
ASSUME (7,15) (NOT (EQ b c))
ASSUME (7,16) (NOT (EQ b d))
ASSUME (7,17) (NOT (EQ b e))
```

```
ASSUME (7,18) (NOT (EQ b f))
ASSUME (7,19) (NOT (EQ b g))
ASSUME (7,20) (NOT (EQ b h))
ASSUME (7,21) (NOT (EQ c d))
ASSUME (7,22) (NOT (EQ c e))
ASSUME (7,23) (NOT (EQ c f))
ASSUME (7,24) (NOT (EQ c g))
ASSUME (7,25) (NOT (EQ c h))
ASSUME (7,26) (NOT (EQ d e))
ASSUME (7,27) (NOT (EQ d f))
ASSUME (7,28) (NOT (EQ d g))
ASSUME (7,29) (NOT (EQ d h))
ASSUME (7,30) (NOT (EQ e f))
ASSUME (7,31) (NOT (EQ e g))
ASSUME (7,32) (NOT (EQ e h))
ASSUME (7,33) (NOT (EQ f g))
ASSUME (7,34) (NOT (EQ f h))
ASSUME (7,35) (NOT (EQ g h))

ERROR 8
ERROR 9
ERROR 10
ERROR 12
ERROR 13
ERROR 14
ERROR 15
ERROR 16
ERROR 17
ERROR 18
ERROR 19
ERROR 20
ERROR 21
ERROR 22
ERROR 23
ERROR 24
ERROR 25
ERROR 26
ERROR 27
ERROR 28
ERROR 29
ERROR 30
ERROR 31
ERROR 32
ERROR 33
ERROR 34
ERROR 35
```

```
ENTRY 0
```

exp3.c

```
#include <assert.h>

int main()
{
    int a, b, c, d, e, f, g, h, i;

    g = h;
    f = g;
    e = f;
    d = e;
    c = d;
    b = c;
    a = b;
    if ( h == i )
    {
        assert( g == i );
        assert( f == i );
        assert( e == i );
        assert( d == i );
        assert( c == i );
        assert( b == i );
        assert( a == i );
    }

    return 0;
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) g := h
ASSIGN (1,2) f := g
ASSIGN (2,3) e := f
ASSIGN (3,4) d := e
ASSIGN (4,5) c := d
ASSIGN (5,6) b := c
ASSIGN (6,7) a := b
ASSUME (7,8) (EQ h i)
ASSUME (8,9) (NOT (EQ g i))
ASSUME (8,10) (NOT (EQ f i))
ASSUME (8,11) (NOT (EQ e i))
ASSUME (8,12) (NOT (EQ d i))
ASSUME (8,13) (NOT (EQ c i))
```

```
ASSUME (8,14) (NOT (EQ b i))
ASSUME (8,15) (NOT (EQ a i))

ERROR 9
ERROR 10
ERROR 11
ERROR 12
ERROR 13
ERROR 14
ERROR 15

ENTRY 0
```

exp4.c

```
#include <assert.h>

int main()
{
    int x, y, z, a, b;

    a = x;
    if ( b == 0 )
    {
        assert(!(a < y && x == y));
    }
    else if ( b == 1 )
    {
        assert(!(a > y && x == y));
    }
    else if ( b == 2 )
    {
        assert(!(x > y && a == y));
    }
    else if ( b == 3 )
    {
        assert(!(x < y && a == y));
    }

    return 0;
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) a := x
ASSUME (1,2) TRUE
ASSUME (1,3) TRUE
```

```
ASSUME (1,4) TRUE
ASSUME (1,5) TRUE
ASSUME (2,6) (AND (< a y) (EQ x y))
ASSUME (3,6) (AND (> a y) (EQ x y))
ASSUME (4,6) (AND (> x y) (EQ a y))
ASSUME (5,6) (AND (< x y) (EQ a y))

ERROR 6

ENTRY 0
```

tut2.c

```
int STATUS_SUCCESS = 0;
int STATUS_UNSUCCESSFUL = -1;
struct Irp {
    int Status;
    int Information;
};

struct Requests {
    int Status;
    struct Irp *irp;
    struct Requests *Next;
};

struct Device {
    struct Requests *WriteListHeadVa;
    int writeListLock;
};

void FSMInit() {
    // code to initialize the global lock to unlocked state
}

void FSMLock() {
    // code for acquiring the lock
}

void FSMUnLock() {
    // code for releasing the lock
}

void SmartDevFreeBlock(struct Requests *r) {
    // code omitted for simplicity
}

void IoCompleteRequest(struct Irp *irp, int status) {
```

```
    // code omitted for simplicity
}

struct Device devE;

void main () {
    int IO_NO_INCREMENT = 3;
    int nPacketsOld, nPackets;
    struct Requests *request;
    struct Irp *irp;
    struct Device *devExt;

    FSMInit();
    devExt = &devE;


    /* driver code */
    do {
        FSMLock();
        nPacketsOld = nPackets;

        request = devExt->WriteListHeadVa;

        if(request!=0 && request->Status!=0){
            devExt->WriteListHeadVa = request->Next;

            FSMUnLock();
            irp = request->irp;

            if((*request).Status >0) {
                (*irp).Status = STATUS_SUCCESS;
                (*irp).Information = (*request).Status;
            } else {
                (*irp).Status = STATUS_UNSUCCESSFUL;
                (*irp).Information = (*request).Status;
            }
            SmartDevFreeBlock(request);
            IO_NO_INCREMENT = 3;
            IoCompleteRequest(irp, IO_NO_INCREMENT);
            nPackets = nPackets + 1;
        }
    } while (nPackets != nPacketsOld);
    FSMUnLock();
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) IO_NO_INCREMENT := 3
```

```
ASSIGN (1,2) lock := 0
ASSIGN (2,3) devExt := devE
ASSUME (3,26) (NOT (EQ lock 0))
ASSIGN (3,4) (EQ lock 0)
ASSIGN (4,5) lock := 1
ASSIGN (5,6) nPacketsOld := nPackets
ASSIGN (6,7) request := devExt->WriteListHeadVa
ASSUME (7,22) (NOT (AND (NEQ request 0) (NEQ request->status 0)))
ASSUME (7,8) (AND (NEQ request 0) (NEQ request->status 0))
ASSIGN (8,9) devExt->WriteListHeadVa := request->Next
ASSUME (9,26) (EQ lock 0)
ASSUME (9,10) (NOT (EQ lock 0))
ASSIGN (10,11) lock := 0
ASSIGN (11,12) irp := request->irp
ASSUME (12,16) (<= request->Status 0)
ASSUME (12,13) (NOT (<= request->Status 0))
ASSIGN (13,14) irp->Status := 0
ASSIGN (14,15) irp->Information := request->Status
ASSUME (15,18) TRUE
ASSIGN (16,17) irp->Status := -1
ASSIGN (17,18) irp->Information := request->Status
ASSIGN (18,19) request := request
ASSIGN (19,20) IO_NO_INCREMENT := 3
ASSIGN (20,21) irp := irp
ASSIGN (21,22) nPackets := ( + nPackets 1 )
ASSUME (22,7) (NOT (EQ nPackets nPacketsOld))
ASSUME (22,23) (EQ nPackets nPacketsOld)
ASSUME (23,26) (EQ lock 0)
ASSIGN (23,24) (NOT (EQ lock 0))
ASSIGN (24,25) lock := 0

ERROR 26

ENTRY 0
```

inf1.c

```
#include <assert.h>

int main()
{
    int x = 0, y = 0;

    y = y + 1;
    while ( x >= 0 )
    {
        x = x + y;
```

```
        y = y - 1;
    }
    assert( x >= 0 );
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) x := 0
ASSIGN (1,2) y := 0
ASSIGN (2,3) y := (+ y 1)
ASSUME (3,4) (>= x 0)
ASSUME (3,7) (NOT (>= x 0))
ASSIGN (4,5) x := (+ x y)
ASSIGN (5,6) y := (- y 1)
ASSUME (6,3) TRUE

ERROR 7

ENTRY 0
```

inf2.c

```
#include <assert.h>

int main()
{
    int x = 0, y = 0;

    y = y + 1;
    while ( x >= 0 )
    {
        x = x + y;
        y = y + 1;
    }
    assert( x >= 0 );
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) x := 0
ASSIGN (1,2) y := 0
ASSIGN (2,3) y := (+ y 1)
ASSUME (3,4) (>= x 0)
ASSUME (3,7) (NOT (>= x 0))
ASSIGN (4,5) x := (+ x y)
ASSIGN (5,6) y := (+ y 1)
```

```
ASSUME (6,3) TRUE

ERROR 7

ENTRY 0
```

inf3.c

```
#include <assert.h>

int main()
{
    int x = 0, y = 0;

    y = y + 1;
    while ( x >= 0 )
    {
        x = x + y;
    }
    assert( x >= 0 );
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) x := 0
ASSIGN (1,2) y := 0
ASSIGN (2,3) y := (+ y 1)
ASSUME (3,4) (>= x 0)
ASSUME (3,7) (NOT (>= x 0))
ASSIGN (4,5) x := (+ x y)
ASSUME (5,3) TRUE

ERROR 7

ENTRY 0
```

inf4.c

```
#include <assert.h>

int main()
{
    int x = 0, y = 0;

    while ( x >= 0 )
    {
        x = x + y;
```

```
        y = y - 1;
    }
    assert( x >= 0 );
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) x := 0
ASSIGN (1,2) y := 0
ASSUME (2,3) (>= x 0)
ASSUME (2,6) (NOT (>= x 0))
ASSIGN (3,4) x := (+ x y)
ASSIGN (4,5) y := (- y 1)
ASSUME (5,2) TRUE

ERROR 6

ENTRY 0
```

inf5.c

```
#include <assert.h>

int main()
{
    int x = 0, y = 0;

    while ( x >= 0 )
    {
        x = x + y;
        y = y + 1;
    }
    assert( x >= 0 );
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) x := 0
ASSIGN (1,2) y := 0
ASSUME (2,3) (>= x 0)
ASSUME (2,6) (NOT (>= x 0))
ASSIGN (3,4) x := (+ x y)
ASSIGN (4,5) y := (+ y 1)
ASSUME (5,2) TRUE

ERROR 6
```

```
ENTRY 0
```

inf6.c

```
#include <assert.h>

int main()
{
    int x = 0, y = 0;

    while ( x >= 0 )
    {
        x = x + y;
    }
    assert( x >= 0 );
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) x := 0
ASSIGN (1,2) y := 0
ASSUME (2,3) (>= x 0)
ASSUME (2,6) (NOT (>= x 0))
ASSIGN (3,4) x := (+ x y)
ASSUME (4,2) TRUE

ERROR 6

ENTRY 0
```

inf7.c

```
#include <assert.h>

int main()
{
    int x;

    for ( x = 0; x <= 50; x += 2 );
    assert( x != 51 );
    return 0;
}
```

The corresponding control flow automata:

```
ASSIGN (0,1) x := 0
ASSUME (1,2) (<= x 60)
ASSIGN (2,3) x := (+ x 2)
ASSUME (3,1) TRUE
ASSUME (1,4) (NOT (<= x 60))
ASSUME (4,5) (EQ x 61)

ERROR 5

ENTRY 0
```

Last update: Mar. 21, 2005 << Back Top