Debug KLEE with GDB Part 4

Weikun Han

Understand KLEE constraints:

When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the path condition which must hold on execution of that path [1].

The various Expr classes mostly model the llvm instruction set. ref<Expr> is used to maintain the reference count but also embeds any constant expressions. In fact in the current code base ConstantExprs should almost never be created. Most of the Expr's are straightforward. Some of the most important ones are Concat?Expr, which join some number of bytes into a larger type, ExtractExpr which extracts smaller types from larger ones, and ReadExpr which is a symbolic array access [2].

1. Use grep command in Linux to search function include name "constraints."

```
$ grep -rn "constraints" ../
```

```
../lib/Expr/Constraints.cpp:110:        it = constraints.begin(), ie = constraints.end(
); it != ie; ++it) {
../lib/Expr/Constraints.cpp:148:      // traversing the constraints looking for equaliti
es is hardly the
../lib/Expr/Constraints.cpp:158:    constraints.push_back(e);
../lib/Expr/Constraints.cpp:163:    constraints.push_back(e);
../unittests/Solver/SolverTest.cpp:82:    ConstraintManager constraints;
../unittests/Solver/SolverTest.cpp:83:    constraints.addConstraint(expr);
../unittests/Solver/SolverTest.cpp:85:    bool success = solver.mustBeTrue(Query(constra
ints, queryExpr), res);
../docs/overview:13:/// solution to the symbolic constraints can be found.
../docs/overview:14:/// KLEE makes queries to STP to solve symbolic constraints.
```

By looking up search result and base on last 3 parts research, I can see the function which deal with "constraints" is locate in Constraints.cpp. However, I still need keep explore exactly which function deal with "constraints."

Next, I go inside Constraints.cpp and look around line 168.

```
167
168 void ConstraintManager::addConstraint(ref<Expr> e) {
169   e = simplifyExpr(e);
170   addConstraintInternal(e);
171 }
```

And I find above function is second lowest level function in Constraints.cpp. Therefore, I can use GDB setup the break point here in order to figure out more information in "constraints".

2. Use GDB to setup break point at Constraints.cpp:168.

When your program has stopped, the first thing I need to know is where it stopped and how it got there. When your program stops, the GDB commands for examining the stack allow you to see all of this information [3].

```
(gdb) b Constraints.cpp:168
(gdb) info b
```

```
(gdb) info b
Num     Type           Disp Enb Address            What
5       breakpoint     keep y   0x0000000000c1bda5 in klee::ConstraintManager::a
ddConstraint(klee::ref<klee::Expr>) at Constraints.cpp:168
        breakpoint already hit 8 times
```

Next, I use GDB run program, and it can stop Constrains.cpp:168. Therefore, I want to know where it stops and how it gets there.

Now, I can use winheight or list command in GDB to check currently executes code. And use backtrace command in GDB to print a backtrace of the entire stack

```
(gdb) winheight
(gdb) bt
```

```
Constraints.cpp
115           } else {
116             equalities.insert(std::make_pair(*it,
117                                              ConstantExpr::alloc(1, Expr::Bool)));
118           }
119         } else {
120           equalities.insert(std::make_pair(*it,
121                                            ConstantExpr::alloc(1, Expr::Bool)));
122         }
123       }
124
125       return ExprReplaceVisitor2(equalities).visit(e);
126     }
127
128     void ConstraintManager::addConstraintInternal(ref<Expr> e) {
129       // rewrite any known equalities and split Ands into different conjuncts
130
131       switch (e->getKind()) {
132       case Expr::Constant:
133         assert(cast<ConstantExpr>(e)->isTrue() &&
134                "attempt to add invalid (false) constraint");
135         break;
136
137         // split to enable finer grained independence and other optimizations
138       case Expr::And: {
139         BinaryExpr *be = cast<BinaryExpr>(e);
140         addConstraintInternal(be->left);
141         addConstraintInternal(be->right);
142         break;
ulti-thre Thread 0x7ffff7fd67 In: klee::ConstraintManager::addConstra* L128  PC: 0xc1bada
#0  klee::ConstraintManager::addConstraintInternal (this=0x26b88f0, e=...)
    at Constraints.cpp:128
#1  0x0000000000c1be2f in klee::ConstraintManager::addConstraint (this=0x26b88f0, e=...)
    at Constraints.cpp:170
#2  0x0000000000b3b286 in klee::ExecutionState::addConstraint (this=0x26b8880, e=...)
    at /home/weikun/Projects/klee/include/klee/ExecutionState.h:168
#3  0x0000000000b23ba0 in klee::Executor::addConstraint (this=0x2649970, state=...,
    condition=...) at Executor.cpp:1038
#4  0x0000000000b233f0 in klee::Executor::fork (this=0x2649970, current=...,
    condition=..., isInternal=false) at Executor.cpp:996
#5  0x0000000000b2726e in klee::Executor::executeInstruction (this=0x2649970, state=...,
    ki=0x26d5010) at Executor.cpp:1632
#6  0x0000000000b2fbc9 in klee::Executor::run (this=0x2649970, initialState=...)
    at Executor.cpp:2798
---Type <return> to continue, or q <return> to quit---
```

From result, I can clearly understand exactly what is going on.

Also, I can check with information in the doxygen.

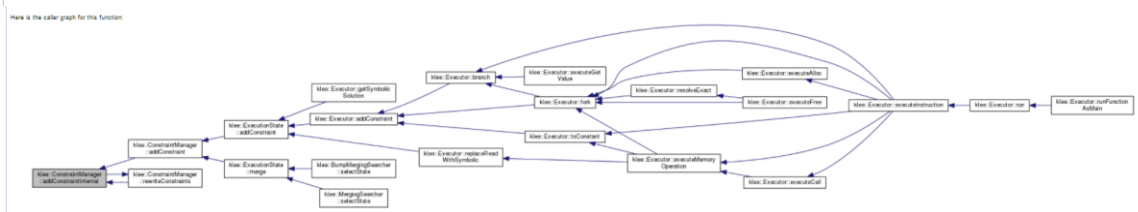
void ConstraintManager::addConstraintInternal (ref< Expr > e)

Definition at line 120 of file Constraints.cpp.

References klee::Expr::And, klee::Expr::Constant, constraints, klee::Expr::Eq, klee::Expr::getKind(), klee::BinaryExpr::left, rewriteConstraints(), and klee::BinaryExpr::right.

Referenced by addConstraint(), and rewriteConstraints().

Here is the call graph for this function:



From the doxygen, I can understand more information than backtrace command in GDB.

3. Use GDB to print variable includes "constraints" in Constraints.cpp:128.

Now, I can check back source code in Constraints.cpp.

```
128 void ConstraintManager::addConstraintInternal(ref<Expr> e) {
129   // rewrite any known equalities and split Ands into different conjuncts
130
131   switch (e->getKind()) {
132   case Expr::Constant:
133     assert(cast<ConstantExpr>(e)->isTrue() &&
134            "attempt to add invalid (false) constraint");
135     break;
136
137     // split to enable finer grained independence and other optimizations
138   case Expr::And: {
139     BinaryExpr *be = cast<BinaryExpr>(e);
140     addConstraintInternal(be->left);
141     addConstraintInternal(be->right);
142     break;
143   }
144
145   case Expr::Eq: {
146     if (RewriteEqualities) {
147       // XXX: should profile the effects of this and the overhead.
```

Here, I can see variable "e" is "constraints" base on the source code and last 3 parts research.

Next, let me print variable "e".

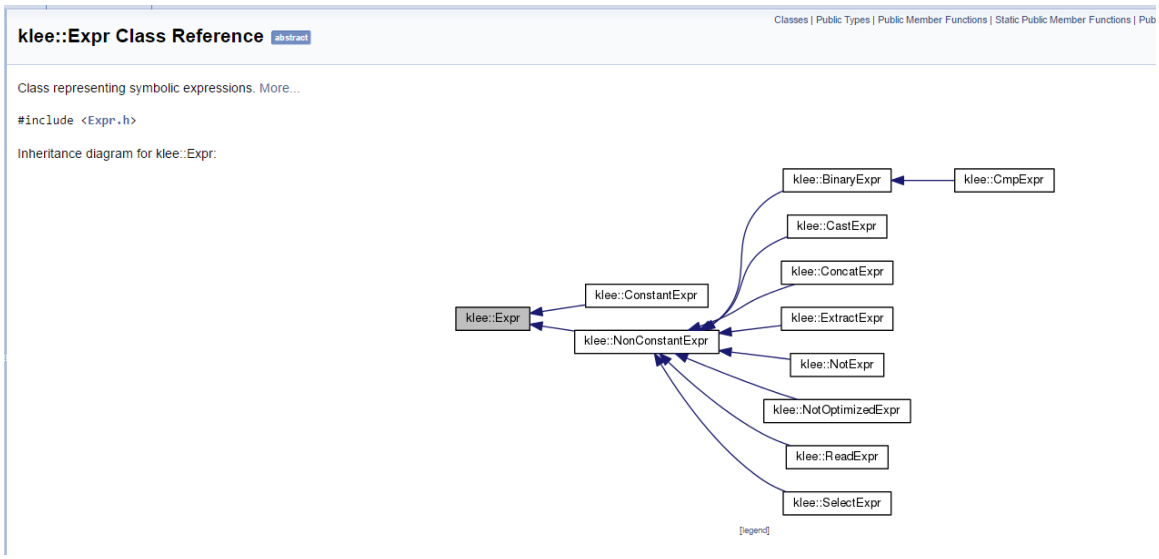
```
(gdb) p *e
```

```
(gdb) p *e
$2 = (klee::Expr &) @0x26943c0: {
  _vptr.Expr = 0x2561c18 <vtable for klee::EqExpr+16>, static count = 74,
  static MAGIC_HASH_CONSTANT = 39, static InvalidWidth = 0, static Bool = 1,
  static Int8 = 8, static Int16 = 16, static Int32 = 32, static Int64 = 64,
  static Fl80 = 80, refCount = 10, hashValue = 3262965214}
(gdb)
```

Next, from result, I cannot see exactly “constraints” looks like. However, I can know the “constraints” is processed in **class** (klee::Expr).

4. Check class (klee::Expr) in order check “constraints”.

First, I go to doxygen or check back source code. Here, I check doxygen easier to understand all information in class.



From the doxygen, I can understand *e result in part 3.

Let us take look at attributes in **class** (klee::Expr) in order to understand *e result in part 3.

Public Attributes

unsigned	**refCount**

Static Public Attributes

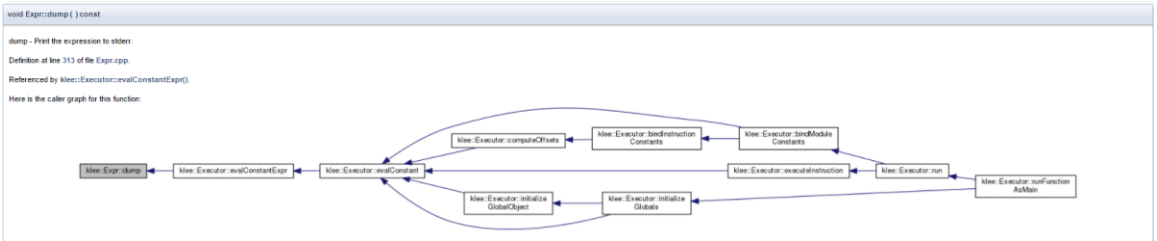
static unsigned	**count** = 0
static const unsigned	**MAGIC_HASH_CONSTANT** = 39
static const **Width**	**InvalidWidth** = 0
static const **Width**	**Bool** = 1
static const **Width**	**Int8** = 8
static const **Width**	**Int16** = 16
static const **Width**	**Int32** = 32
static const **Width**	**Int64** = 64
static const **Width**	**Fl80** = 80

Protected Attributes

unsigned	**hashValue**

Now, I now all attributes come from and it means. However, it can let me know “constraints” looks like.

Keeping search in doxygen, I find one **function** (klee::Expr::dump) can help me to cheak “constraints.”



This function can help me to print the expression to stderr.

5. Use function (klee::Expr::dump) to print "constraints".

```
(gdb) p E->dump()
```

```
klee::ConstantExpr::classof (E=0x26943a0)
    at /home/weikun/Projects/klee/include/klee/Expr.h:1090
(gdb) p E->dump()
(Eq 0
     (ReadLSB w32 0 a))
$18 = void
```

```
klee::ConstantExpr::classof (E=0x2695400)
    at /home/weikun/Projects/klee/include/klee/Expr.h:1090
(gdb) p E->dump()
(Eq false
     (Eq 0
         (ReadLSB w32 0 a)))
$19 = void
```

```
klee::ConstantExpr::classof (E=0x2694950)
    at /home/weikun/Projects/klee/include/klee/Expr.h:1090
(gdb) p E->dump()
(Slt (ReadLSB w32 0 a)
      0)
$20 = void
```

```
klee::ConstantExpr::classof (E=0x2692430)
    at /home/weikun/Projects/klee/include/klee/Expr.h:1090
(gdb) p E->dump()
(Eq false
     (Slt (ReadLSB w32 0 a)
          0))
$21 = void
```

Here, I print all "constraints." First I will explain why result has 4 constraints. Next, I will explain each constraint means.

To understand that, I need compare with original C++ file.

```
#include <klee/klee.h>

int get_sign(int x) {
  if (x == 0)
    return 0;

  if (x < 0)
    return -1;

  else
    return 1;
}
```

In original C++ file, it has only one function, and this function has 3 conditions. First, the 4 constrains are: KLEE make 3 conditions into 2 parts, and each part split into TRUE and FALSE. Next, the constraint (Eq 0 (ReadLSB w32 0 a)) is correspond to condition (if (x == 0)), the constraint (Eq false (Eq 0 (ReadLSB w32 0 a))) is correspond to condition (else), the constraint ((Slt 0 (ReadLSB w32 0 a) 0)) is correspond to condition (if (x < 0)), and the constraint (Eq false (Slt 0 (ReadLSB w32 0 a) 0)) is correspond to condition (else).

n. References:

[1] Cristian Cadar, Daniel Dunbar, Dawson Engler, *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,* 8th USENIX Symposium on Operating Systems Design and Implementation, 2009

[2] KLEE, https://klee.github.io/, March 2017.

[3] Richard Stallman, Roland Pesch, Stan Shebs, *Debugging with GDB: the GNU source-level debugger,* Ninth Edition, Free Software Foundation, ISBN 1-882114-77-9.