Z3: User Propagator

Clemens Eisenhofer

Experience report, in joint work with Nikolaj Bjørner

In the following we will have a look at Z3's user propagator feature. Defining custom propagators allow us to register callbacks that are called during solving. As we need to know to some extend how SMT solvers work internally we will start with a brief discussion how solving is done.

1 Background

SMT solvers mostly decompose the input problem into a propositional SAT problem and a "structureless" theory reasoning problem. We can convert any problem into a propositional formula by replacing complex first-order literals by propositional variables. For example, $\neg((x + 1 = y \land y = 2 * x) \Rightarrow (x = 1))$ could be translated into $\neg((a \land b) \Rightarrow c)$. This problem will be handed over to an ordinary SAT solver that either gives us a model or tells us that this formula is unsatisfiable. If the result is unsatisfible we are done as this also means that the original SMT formula is unsatisfiable. However, the inverse direction is not true. Although the example from before is in fact unsatisfiable (w.r.t. integer arithmetic) we can get a model for the propositional skeleton. In this example we can get $a \mapsto 1, b \mapsto 1, c \mapsto 0$. With this model we construct a pure conjunctive SMT formula that asks explicitly if this assignment is viable: $x + 1 = y \land y = 2 * x \land x \neq 1$. Now the SMT solver can ask a decision procedure if it is possible to and what concrete values can be assigned to the constants. If we get a positive answer we are done and can return a model to the user. Otherwise we found out that at least this assignment is not possible and we have to find another one. We can now simply add to our propositional skeleton a clause that asserts that we want to exclude this assignment. In our toy example this would mean our new SAT formula is: $\neg((a \land b) \Rightarrow c) \land (\neg a \lor \neg n \lor c)$ which is now unsatisfiable.

But not all parts of a SMT formula that seem non-propositional have to be processed by the dedicated decision procedure. Bitvectors (i.e., integers of a fixed size), for example, can be bitblasted by turning every bit into a propositional constant. Formulas containing only bitvector arithmetic and propositional variables would not even require a separate decision procedure and could be solved solely by the SAT solver.

Let us now briefly also discuss the way modern SAT solvers work. Most of them use conflict driven clause learning (CDCL) which is a derivative of the DPLL algorithm. In a nutshell: The solver converts the formula into a conjunctive normal form (CNF) and then tries to propagate unit clauses as good as possible. If there is nothing more to propagate it guesses a truth-assignment for a variable and proceeds propagating. We consider a very simple example: We want to check $(x \vee y) \wedge (x \vee z) \wedge (\neg y \vee \neg z)$ for satisfiability. We will go through the first steps of solving the formula with CDCL.

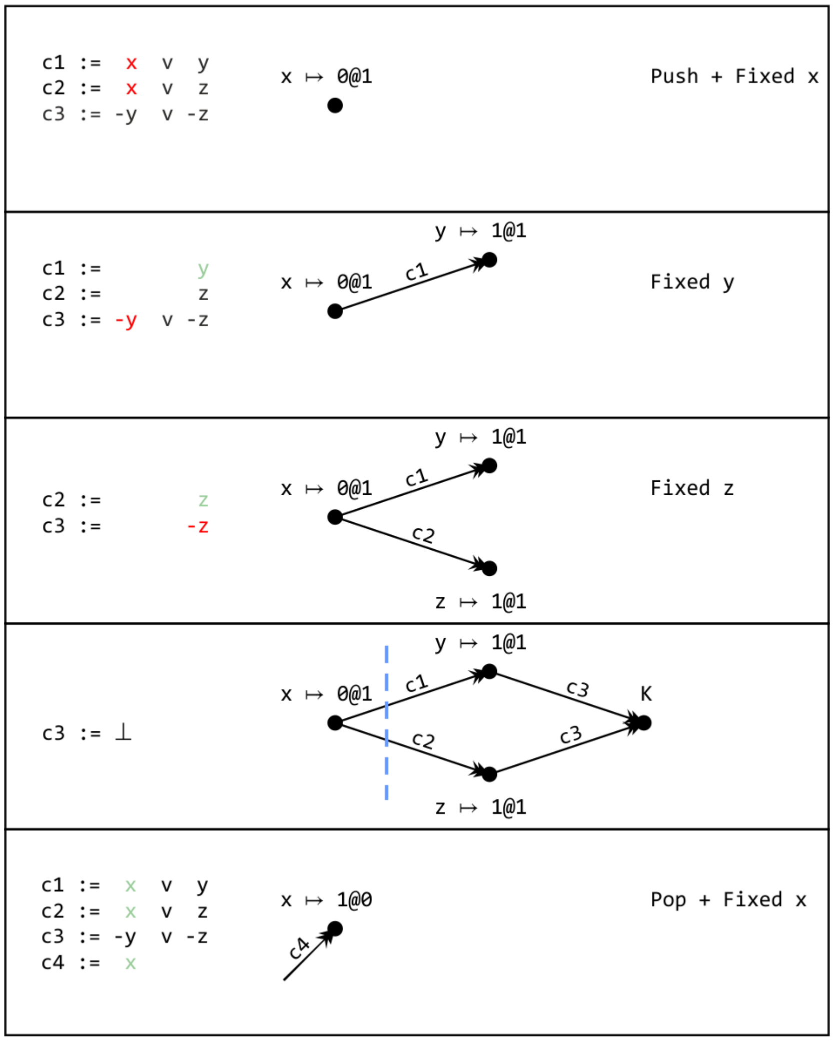

Figure 1: CDCL for $(x \vee y) \wedge (x \vee z) \wedge (\neg y \vee \neg z)$

Initially, as there is no unit clause we need to start guessing and assign x to 0. (This is a very bad decision but do not forget that this is an example.) As this is an arbitrary guess we potentially have to revert this step later via backtracking. We *push* the current state and *fix* the value of x to 0 (Image 1). As we need to satisfy clause $c1$ somehow and we already know that $x \mapsto 0$ does not satisfy it we need to set y to 1 (Image 2). In other words: We need to make this decision and therefore we do not push the current state (as we won't need to backtrack to this particular state). We do the same for variable z and set it to 1 as well (Image 3). Again, this is a completely deterministic step. This, however, contradicts clause $c3$ (Image 4). We now have to backtrack by undoing the previous decisions. (i.e., we *pop* the last state. In CDCL we can sometimes even pop multiple decisions at once.) As the name conflict driven clause learning indicates we now analyse our contradiction. We want to separate the most recent decision from the contradictory node K. By propositional resolution along the path between the node $x \mapsto 0@1$ and the node K we construct the formula $x \vee x$ which is equivalent to x. Adding this clause eliminates the problematic path to the contradiction globally in our problem. So we add this clause to our clause set, clear the problematic part of the graph, and proceed.

2 Done with the background. Let's start with the foreground!

But let us come back to Z3's user propagator: We can register functions that are called if some of the previously mentioned events occur. We get notified if the solver makes nondeterministic decisions (*push*) or reverts them (*pop*). We can also get notified as soon as a value is fixed during the CDCL process. However, this does not necessarily mean that every decision the solver makes corresponds to fixing a variable. If we consider bitvectors the *fixed*-event is only called as soon as all bits of the corresponding bitvector are set. We then can also read off the current numerical candidate value of the bitvector.

You might now ask "Fine, we can observe what the solver does. What's the point?"

We can not only observe what is done but also actively interfere and guide the solver. For example: We can tell the solver during its model search that there is a contradiction between some variables. i.e., we can manually introduce contradictory nodes or tell the solver to learn new formulas. In terms of our CDCL example before this means connecting variable assignment nodes like $y \mapsto 1@1$ to a contradictory node which forces the CDCL solver to backtrack.

3 N-Queens

To make this less abstract we will consider a concrete C++ example. Our task: We want to count the number of solutions for the n-queens problem (`https://en.wikipedia.org/wiki/Eight_queens_puzzle`) on a $n \times n$ chess board. The way enumerating/counting solutions in Z3 is mostly implemented is quite simple: We find a model and then block it by adding a blocking clause. Then we request a new model until the solver cannot find more.

To demonstrate the propagator in combination with bitvectors we use the following encoding:

Every line has to contain a single queen somewhere. We, therefore, define for every line i ($0 \leq i < n$) a variable q_i that represents the position of the queen in line i.

```
std::vector<z3::expr> queens;
int bits = log2i(n) + 1;
for (int i = 0; i < n; i++) {
    queens.push_back(context.bv_const(
        ("q"s + to_string(i)).c_str(), bits));
}
```

We, furthermore, have the following assertions:

- For every queen q_i: $0 \leq q_i < n$ (Queens have to be on the board.):

```
for (int i = 0; i < n; i++) {
    solver.add(z3::uge(queens[i], 0));
    solver.add(z3::ule(queens[i], n - 1));
}
```

- $\mathit{distinct}(\{q_1, \ldots q_n\})$ (Queens cannot attack vertically.):

```
z3::expr_vector distinct(context);
for (const z3::expr & queen : queens) {
    distinct.push_back(queen);
}
solver.add(z3::distinct(distinct));
```

- For every position $1 \leq i < j \leq n : j - i \neq q_j - q_i \land j - i \neq q_i - q_j$ (Queens cannot attack diagonal.):

```
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        solver.add((j - i) !=
            (queens[j] - queens[i]));
        solver.add((j - i) !=
            (queens[i] - queens[j]));
    }
}
```

It is crucial, that we are actually dealing with a SAT problem.

Say we get a model: $\{q_1 \mapsto v_1, \ldots, q_n \mapsto v_n\}$. We can simply assert additionally $q_1 \neq v_1 \lor \ldots \lor q_n \neq v_n$ to get a different model. We repeat this until we get unsatisfiable as a result.

Although this process works well, it turns out that we can speed up quite a lot by using a custom propagator. We define a class `user_propagator`. The idea is very simple. We wait until the solver finds a complete truth-assignment to all variables and then add a contradictory node that forces the solver to come up with another model. Officially the solver will not find a single model as we always claim that the solver came up with an invalid one because the assignment of the variables contradict each other.

So, how does it work?

As not all of Z3's solvers support user propagators (in particular the default solver does not) we have to use a simple one. Precisely: Z3's "Simple Solver".

```
z3::solver solver(ctx, Z3_mk_simple_solver(context));
```

We then mark all those subterms in our formula that should be tracked by the propagator. We mark a term by calling "add" which assigns a unique id to the term. From this point on we can talk about the value of the subterm via this identifier.

```
for (int i = 0; i < queens.size(); i++) {
    unsigned id = propagator->add(queens[i]);
    idMapping[id] = i;
}
```

We now have to register two functions. One that is called as soon as a constant's value is fixed (*fixed*) and one that is called if the solver thinks that it successfully assigned a value to all constants (*final*).

```
this->register_fixed();
this->register_final();
```

These two calls register the virtual *fixed* and *final* of the class. (Alternatively, we could also pass a function to these register-functions.) Apart from these two functions there is a third optional callback function *eq* that is called whenever the solver decides that two registered (bitvector) terms are equal. However, we do not need this function in our example.

Furthermore, we have to override three functions: *push*, *pop*, and *fresh*. (They are registered automatically as soon as the class is initialised.) The function *fresh* is not really interesting for our purpose but we nonetheless have to implement it. *push* is called every time the solver makes a decision as discussed previously. *pop* is called if the solver backtracks. As it might undo multiple decisions at once via smart backtracking the function's argument tells us how many decisions were discarded at once.

```
void push() override {
    fixedCnt.push(fixedValues.size());
}
```

```
void pop(unsigned num_scopes) override {
    for (int i = 0; i < num_scopes; i++) {
        fixedValues.resize(fixedCnt.top());
        fixedCnt.pop();
    }
}
user_propagator_base* fresh(Z3_context ctx) override {
    return this; // Won't be called in our example
}
```

We use these two functions to keep track of how many constants have been fixed so far. If the solver tells us that it assigned some value to a bitvector via calling our *fixed*-function we need to track this information as we cannot get it from somewhere else. In the *final*-method we simply add a contradiction (*conflict*) between all the constants assigned and increment a variable counting the number of models. (For completeness: We can not only add conflicts between assignments but also introduce new (arbitrary) formulas via the method *propagate*.)

However, we also need to remember the actual model although we are not necessarily interested in it. The problem is that the solver might drop some of the learned clauses from time to time. The solver does not interpret our manually inserted contradictions as real assertions but rather as hints. Therefore, we have to assume that the solver might come up with a model we have received before. (If we would directly add blocking clauses to the solvers assertions we would not have this problem.)

```
void final() override {
    this->conflict(fixedValues.size(),
        fixedValues.data());
    if (modelSet.find(*currentModel) ==
            modelSet.end()) {
        // Model wasn't found so far
        solutionId++;
        modelSet.insert(*currentModel);
        currentModel = currentModel->clone();
    }
}
unsigned bvToInt(z3::expr e) {
    return (unsigned)e.get_numeral_int();
}
void fixed(unsigned id, z3::expr const &e) override {
    fixedValues.push_back(id);
    currentModel->set(id_mapping[id], bvToInt(e));
}
```

So, why do we mess around with it if the introduced contradictions are not even hard? Switching around between Z3 and our program code that adds blocking clauses is an additional overhead that should not be underestimated. Z3 has to start up and shut down every time at least to some extend. If we add custom contradictions we do not suffer this have this problem as Z3 has only to process a single query. At the end of this article we will see if it really payed off.

Reducing user propagators to a tool for efficiently adding blocking clauses is of course far too restrictive. We can even build custom theories to some extend. We will now define a "n-queens theory". As we have seen before, a decision procedure receives a conjunction of literals over a given theory. Unfortunately, our custom decision procedures are restricted to types like boolean constants and bitvectors (as the SAT solver can provide us candidate intermediate models for these sorts). As before we keep track of the solver's assignments. As soon as we get a value for a queen's position we decide if the assignment is feasible. We check in our C++ program whether a queen can be positioned at the proposed location by checking against all other queen positions assigned so far. If the position is illegal we can add a contradiction between the two involved queens.

Precisely we change our *fixed*-function to:

```
 void fixed(unsigned id, z3::expr const &e) override {
    unsigned queenPos = bvToInt(e);
    if (queenPos >= board) {
         this->conflict(1, &id);
         return;
    }
    for (unsigned fixed : fixedValues) {
         int otherId = id_mapping[fixed];
         int otherPos = (*currentModel)[fixed];
         if (queenPos == otherPos) {
              const unsigned conflicting[] =
                   { id, fixed };
              this->conflict(2, conflicting);
              continue;
         }
         unsigned queenId = id_mapping[id];
         int diffY = abs((int)queenId - otherId);
         int diffX = abs((int)queenPos - otherPos);
         if (diffX == diffY) {
              const unsigned conflicting[] =
                   { id, fixed };
              this->conflict(2, conflicting);
         }
    }
    fixedValues.push_back(id);
    currentModel->set(id_mapping[id], queenPos);
}
```

Interestingly, we do not have to formally assert anything to do this. Officially the solver has to assign values to the bitvector constants $q_1, \ldots, q_n$ such that (*assert true*) is satisfied. Sounds easy, however, due to our propagator we eliminate all candidate models that do not obey our constraints now expressed in C++ code. The advantage of this way is that we do not have to deal with the (expensive) bitvector arithmetic on the SAT level but instead on the C++ code level. This is far more efficient.

4 Results

A lot of words but no time measurements so far. Let's change that:

	n								
	4	5	6	7	8	9	10	11	12
Def.S.	74.4	37.8	51.6	91.1	210.0	992.6	3233.5	27884.3	631980
Simpl.S.	10.1	17.7	26.4	59.8	162.6	937.9	3326.9	26086.9	597348
Contr.	26.3	13.1	20.6	42.0	112.5	642.2	2192.7	11731.3	77122
Cust.Th.	10.1	12.9	17.8	35.9	44.7	194.3	548.7	3801.6	34096

Figure 2: Runtime for different values of n.
(All measurements in milliseconds and the data is the mean over 5 repetitions.)

We consider 4 different strategies: The first one uses Z3's default solver and the bitvector constraints we defined before. The second strategy differs only in the aspect that we use the simple solver that theoretically supports user propagators. Both strategies enumerate the models by adding blocking clauses. The third strategy also uses the constraints but adds conflict nodes internally. We enumerate the models within Z3. The last strategy does not use any asserts but adds conflict nodes whenever we found a complete feasible model (model enumeration) or receive a (partial) assignment that does not satisfy our n-queen constraints (which we did not specify anywhere).

We can observe two things: Firstly, that finding the number of solutions for the n-queens problem quickly becomes quite computationally expensive and secondly that using a user propagator really speeds up the whole process although we might reconsider the same model.

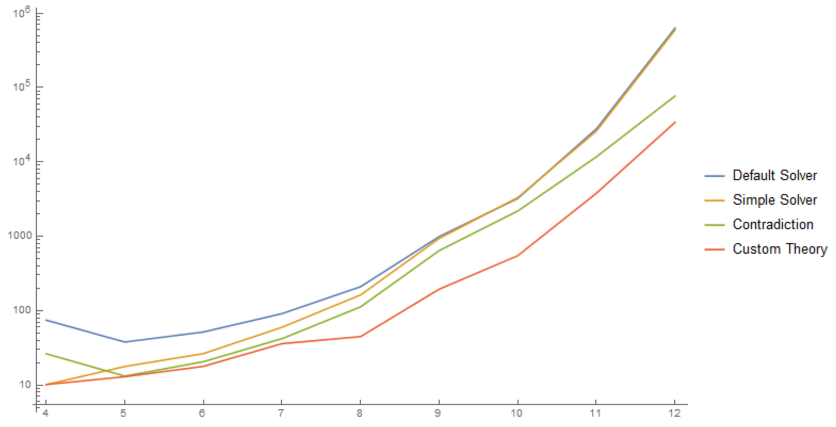

Figure 3: Plot of the runtimes (Logarithmically scaled)

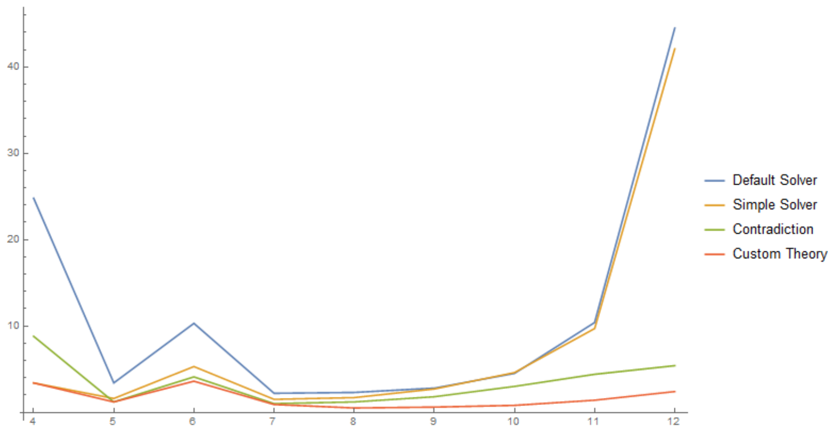

Figure 4: Plot of the runtimes per model

Note that other encodings of the n-queens problem are possible as well. For example via pseudo-boolean functions, explicit plain propositional logic, and many more. However, note that other encoding might be more difficult to deal with for the solver or require a far more complex problem encoding. For example, a pure propositional encoding is many times larger than our bitvector encoding/C++ constraints.

5 Source Code

The complete (slightly adopted) source code for the n-queens problem discussed here can be found online: `https://github.com/Z3Prover/z3/tree/master/examples/c%2B%2B/userPropagator.cpp`