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 conjunctions of theory reasoning problems. We can reduce 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 unsatisfiable 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 an SMT formula that seem non-propositional have to be processed by the dedicated decision procedure. Bitvectors (i.e., integers of a fixed maximum size), for example, can be bit-blasted by turning every bit into a propositional constant. Formulas containing only bitvector arithmetic and propositional variables do 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 (a clause with only a single literal) 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.

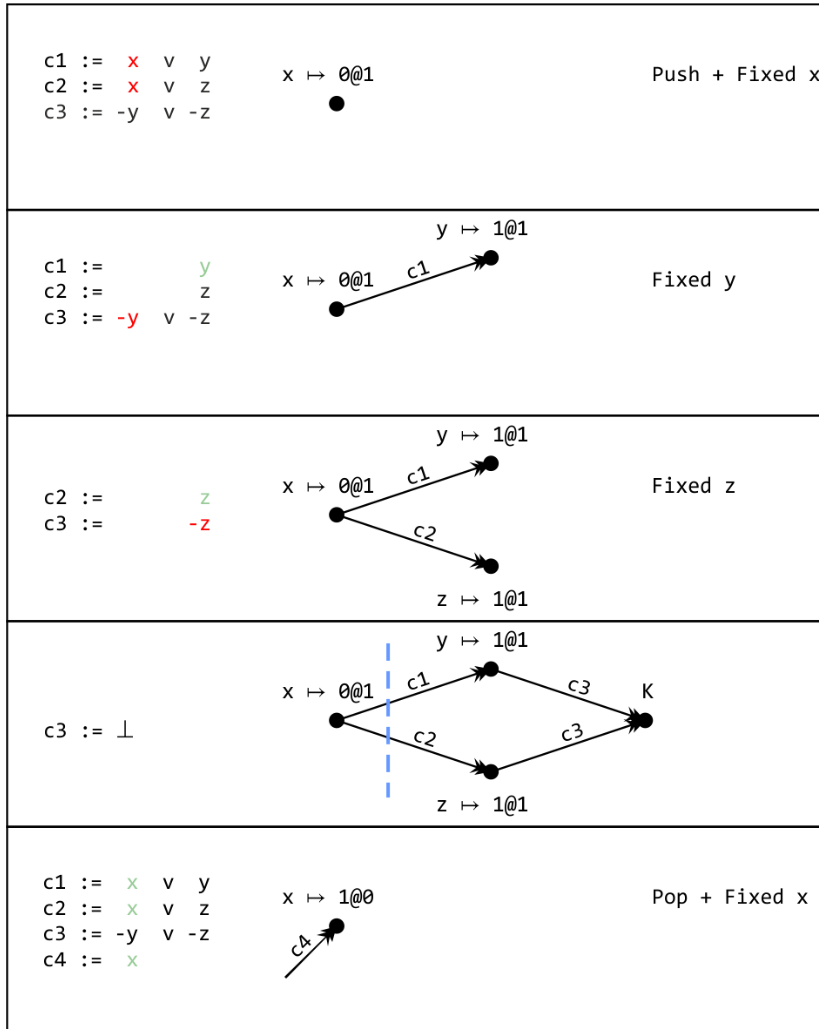

Figure 1: CDCL for $(x \vee y) \wedge (x \vee z) \wedge (\neg y \vee \neg z)$

Initially, as there is no unit clause, we need to start guessing and assign x to 0. (This is a very bad decision, but 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 will definitely not have 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 revert 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 can then 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.

2.1 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, block it by adding an appropriate clause, and

incrementing some counter. We repeat this step until the solver cannot find any further solutions.

To demonstrate the propagator in combination with bitvectors we use the following encoding of the n-queens problem:

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 the line i.

```
z3::expr_vector queens(ctx);
for (unsigned i = 0; i < n; i++) {
  queens.push_back(
    ctx.bv_const(("q" + to_string(i)).c_str(), bits)
  );
}
```

We, furthermore, have the following constraints:

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

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

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

```
z3::expr_vector distinct(ctx);
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]));
    }
}
```

(Note that we are actually dealing with a SAT problem as all the queen constants q_i are bitvectors.)

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.

2.2 First Appearance of the Propagator

Although this process works well, it turns out that we can speed it up quite a lot by using a user-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 is connected to all variables. This forces the solver to come up with another model. For the solver it looks like it could 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. This is very similar to adding blocking clauses explicitly.

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(ctx));
```

We then mark all those subterms in our formula that should be tracked by the propagator. We mark a term by calling "add":

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

Z3 will now report whenever the value of the registered expression is (temporally) fixed. We now have to register two functions. One that is called whenever one of the registered 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 tell the solver to call the virtual *fixed* and *final* of the class if something interesting happens. Apart from these two functions, there are other optional callback function like *created* (see later).

Furthermore, we have to override three functions: *push*, *pop*, and *fresh*. (They are registered automatically as soon as the class is initialized.) The function *fresh* is not really interesting for our purpose right now (see later as well), 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 the solver 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 (unsigned i = 0; i < num_scopes; i++) {
    unsigned lastCnt = fixedCnt.top();
    fixedCnt.pop();
    // Remove fixed values from candidate model
    unsigned j = fixedValues.size();
    while (j > lastCnt) {
      currentModel.erase(fixedValues[j - 1]);
      j--;
    }
    fixedValues.resize(lastCnt);
  }
}
user_propagator_base* fresh(z3::context&) override {
  return this; // Won't be called in our example
}
```

The general pattern is to maintain a (partial) candidate model ($currentModel$), a stack ($fixedCnt$) remembering how many values are fixed up to each decision level (remember: a new decision level starts with each call of *push*), and a list of all values that currently fixed ($fixedValues$).

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.

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 learned clauses from time to time. The solver does not interpret our manually inserted contradictions as real assertions, but rather as consequences of the original problem (although they are not, but the solver does not know this). Therefore, we have to assume that the solver might come up with a model we have already received before. If we would directly add blocking clauses to the solver, we would not

have this problem.

```
void final() override {
  this->conflict(fixedValues);
  if (modelSet.find(currentModel) ==
        modelSet.end()) {
    // Model hasn't been found so far
    solutionNr++;
    modelSet.insert(currentModel);
  }
}
void fixed(z3::expr const &ast,
           z3::expr const &value) override {

  fixedValues.push_back(ast);
  currentModel[ast] = (unsigned)value.get_numeral_int;
}
```

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 have this problem as Z3 has only to process a single query. Later we will see if it really paid off.

2.3 A Custom n-Queens Theory

Reducing user-propagators to a tool for efficiently adding blocking clauses is of course far too restrictive. We can even build "custom theories". We will now define an "n-queens theory". As we have seen before, a decision procedure receives a conjunction of literals over a given theory and the procedure is asked if this is feasible. We do something very similar, but we do not have to wait until the solver has values for all the literals, but also when only some of them are assigned. 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 allowed with respect to the previous placements. 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(z3::expr const &ast,
           z3::expr const &value) override {

  unsigned queenId = queenToY[ast];
  unsigned queenPos = bvToInt(value);

  if (queenPos >= board) {
    z3::expr_vector conflicting(ast.ctx());
    conflicting.push_back(ast);
    this->conflict(conflicting);
    return;
  }

  for (const z3::expr& fixed: fixedValues) {
    unsigned otherId = queenToY[fixed];
    unsigned otherPos = currentModel[fixed];

    if (queenPos == otherPos) {
      z3::expr_vector conflicting(ast.ctx());
      conflicting.push_back(ast);
      conflicting.push_back(fixed);
      this->conflict(conflicting);
      continue;
    }
    int diffY = abs((int)queenId - (int)otherId);
    int diffX = abs((int)queenPos - (int)otherPos);
    if (diffX == diffY) {
      z3::expr_vector conflicting(ast.ctx());
      conflicting.push_back(ast);
      conflicting.push_back(fixed);
      this->conflict(conflicting);
    }
  }

  fixedValues.push_back(ast);
  currentModel[ast] = queenPos;
}
```

Done. There is nothing more to do. We do not even have to formally assert anything. We just call the solver with the introduced n queen constants $q_1, \ldots, q_n$ and the new propagator. For the solver it looks like it just 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.

2.4 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 are given in milliseconds and the values are the means 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 the conflicts internally. We enumerate the models within Z3. The last strategy does not use any logical encoding, but adds conflict nodes whenever we found a correct model (model enumeration) or receive a (partial) assignment that does not satisfy our n-queen constraints (which we did not specify anywhere in the formula).

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 can really speed up the whole process. However, we have to deal with the internals of the underlying SAT solver and finding the right candidates for the added conflict can become difficult in more complex problems.

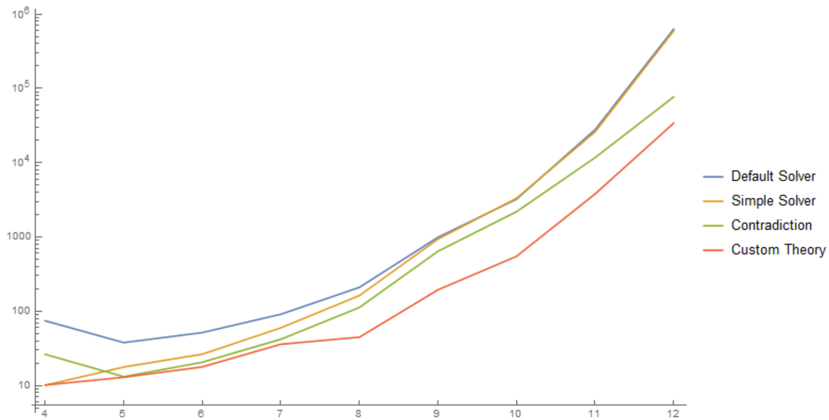

Figure 3: Plot of the runtimes (Logarithmically scaled)

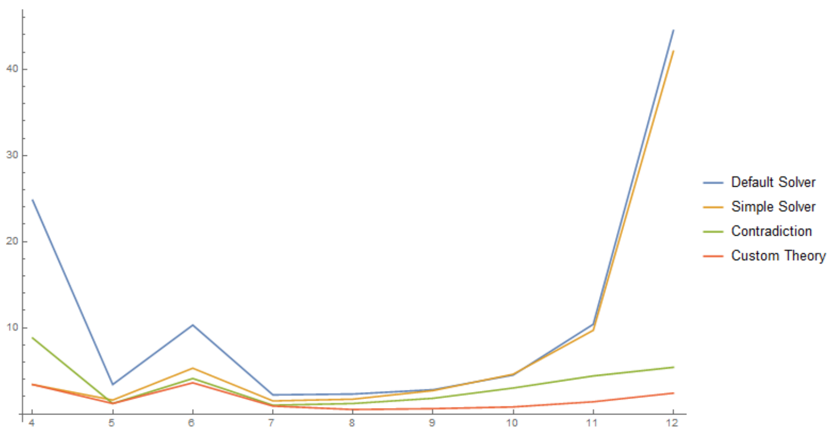

Figure 4: Plot of the runtimes per model

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, other encodings 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.

3 Quantifiers

One of the ways Z3 can deal with quantifiers is MBQI (model based quantifier instantiation). The idea is to ignore quantifiers in the first place and build a candidate model for the remaining formula. As soon as such a candidate model is found, the solver checks if this candidate model also satisfies the quantified subformulas.

Assume, for example, we have a universal quantifier with positive polarity in the formula to check for satisfiability. In this case, we cannot simply find appropriate values for the bound variables (as it should be true for all possible values). We, therefore, ignore the quantifiers in the first place (i.e., assume the whole quantified subformula is just a propositional auxiliary constant) and build a candidate model for the relaxed formula.

In case the auxiliary variable has to be true according to the candidate model, we need to check if the corresponding quantifier (we assume a universal quantifier) is also true with respect to the current candidate model. To find out, we negate the subformulas, skolemise it, and check for satisfiability. ($\forall x : P(x)$ is true if $\neg P(k)$ is false where k is a fresh Skolem constant.) If this negation is unsatisfiable we know that the found candidate model really behaves as expected. In case the negated subformula is satisfiable the universal quantifier is not satisfied with respect to the found candidate model. We can then take the found counterexample and instantiate the quantified subformula according to it (or some generalization) and add it to the original formula. This prevents the top-level search from coming up with the same candidate model.

For example: If we want to solve

$$x \geq 0 \wedge P(0) \wedge \forall y(P(y) \Rightarrow x \neq y)$$

via MBQI the formula is considered as $x \geq 0 \wedge P(0) \wedge A$ where A is a fresh propositional constant. The solver then might come up with the model $x \mapsto 0, P(.) \mapsto \top$ and $A \mapsto \top$. As the original formula contains at least one quantifier that could not be removed beforehand, we must check if the model satisfies the subformula $\exists y \neg(\top \Rightarrow 0 \neq y)$. ($x$ and P were replaced because of the candidate model.) By skolemisation we check if $\neg(\top \Rightarrow 0 \neq k_1))$ is satisfiable (where k_1 is a fresh Skolem constant). We get a model for this subformula that assigns $k_1 = 0$. Hence, we know that the previous candidate model does not work. We instantiate the quantified subformula by $y \mapsto 0$ and add it to the original problem: $x \geq 0 \wedge P(0) \wedge A \wedge (P(0) \Rightarrow x \neq 0)$. We might now get the model $x \mapsto 1, P(.) \mapsto \top$ and $A \mapsto \top$. We again do the subquery for $\neg(\top \Rightarrow 1 \neq k_2))$ and we again get a counterexample, so we instantiate again by $y \mapsto 1$. The new formula is $x \geq 0 \wedge P(0) \wedge A \wedge (P(0) \Rightarrow x \neq 0) \wedge (P(1) \Rightarrow x \neq 1)$. Now we can come up with the candidate $x \mapsto 1, P(x) \mapsto x \neq 1$ and $A = \top$. Now $\neg(k_3 \neq 1 \Rightarrow 1 \neq k_3))$ is indeed unsatisfiable which shows that the candidate model is a model for the overall formula.

If the quantified subformula contains again other quantifiers, we do the same trick in a recursive manner.

4 Using quantifiers with the propagator

A question that may arise is if we can use the propagator together with the MBQI process. Obviously we cannot register the bound variables of a quantifier but only free occurrences of variables. (We would have to register the Skolem constants introduced in the subquery, but we do not have them beforehand.)

In principle, we can do the subquery done by MBQI on our own and use the propagator in the subquery to register the (now free) variables. However, might not be desired as Z3 does some additional magic.

The way Z3 can be used to deal directly with quantification is by introducing and observing auxiliary function symbols through the propagator. In the following, we will consider a (frankly rather arbitrary constructed) optimization problem of the previous N-Queens counting problem.

4.1 N-Queens Optimisation

As before, we consider the n-queens problem. However, this time we want to maximize the horizontal distribution of queens in adjoint rows. Let $V(q_1, \ldots, q_n)$ be an abbreviation of the formula ensuring that the queens $q_1, \ldots, q_n$ are positioned in an allowed manner as used in the previous encoding. Formally, we want to solve the problem:

$$V(q_1, \ldots, q_n) \wedge \forall q'_1, \ldots, q'_n (V(q'_1, \ldots, q'_n) \Rightarrow |q_1 - q_2| + \ldots + |q_{n-1} - q_n| \geq |q'_1 - q'_2| + \ldots + |q'_{n-1} - q'_n|)$$

Of course, we can encode this in Z3 (we have to replace the absolute value by if-then-else, but otherwise the translation is direct). We then can use the propagator to implicitly define $V(q_1, \ldots, q_n)$ but we cannot do so for $V(q'_1, \ldots, q'_n)$ because we do not get candidate values for the q'_i as they are universally quantified. The easiest solution to use the propagator is to do the final maximality check" (i.e., the universally quantified part) manually in the *final* callback of

the formula. This is very similar to what Z3 does internally:

```
void final() override {
  int max1 = 0;
  for (unsigned i = 1; i < board; i++) {
      max1 += abs(
        (signed)currentModel[i] -
        (signed)currentModel[i - 1]);
  }
  z3::solver subquery(ctx(),   z3::solver::simple());

  subquery.add(assertion);
  subquery.add(z3::ugt(manhattanSum, max1));
  if (subquery.check() == z3::unsat)
    return; // model is already maximal

  z3::model counterExample =   subquery.get_model();

  int prev, curr = -1;
  int max2 = 0;
  for (int i = 0; i < queens.size(); i++) {
    prev = curr;
    curr = counterExample.eval(queens[i])
           .get_numeral_int();
    if (i == 0) continue;
    max2 += abs(curr - prev);
  }
  z3::expr_vector vec(ctx());
  this->propagate(vec, z3::uge(manhattanSum, max2));
}
```

We create a new solver instance and ask it if there is a solution that is strictly better than the one found so far. (As we use the user-propagator we know the current value.) If there is none, we are done. If there is one, we get a counterexample and add through the *propagate*-function a new formula to the problem that states that we are looking for a formula that is at least as good as the counterexample. As the formula we add through *propagate* obviously contradicts the current candidate-model Z3 will backtrack.

In this example, we could as well directly propagate in the *final* that we want a better solution until the solver returns that the formula is unsatisfiable.

In these cases we decide if an assignment is fine by encoding the part checking if the quantifier is satisfied directly in the user-propagator. (This works just fine in our example, but might be more difficult in other situations.) Therefore, we can to some extent also interact with Z3's MBQI.

The way we can do so is by using user-functions. We can declare them by using the *user_propagate_function* function. This call has a very similar outcome as

the ordinary API function *function* that can be used to declare an uninterpreted function symbol. However, the UFs generated by *user_propagate_function* are dealt with in a special way by the solver. (Note that using a user function may be slower than an ordinary UF. In case it is not required to use a user-function, an ordinary function is a better choice. Furthermore, user functions will not show up in any model.)

If we now also register a *created* callback

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

we will be notified through the *created* function whenever any instance of the declared user-functions has been encountered the first time. (Not only within the scope of a quantifier. The *created* + user-functions are in principle independent of MBQI!)

We declare a user-function $valid(x_1, \ldots, x_n)$ that should be true iff the queens $x_1, \ldots, x_n$ are positioned in an allowed way. (Similar to the abbreviation V before, but now it is really a function symbol.)

```
z3::sort_vector domain(context);
for (int i = 0; i < queens.size(); i++) {
    domain.push_back(queens[i].get_sort());
}
z3::func_decl validFunc =
  context.user_propagate_function(
    context.str_symbol("valid"),
    domain,
    context.bool_sort());
```

The considered formula is now

$$
\begin{aligned}
&valid(q_1, \ldots, q_n) \land \\
&\forall q'_1, \ldots, q'_n (valid(q'_1, \ldots, q'_n) \Rightarrow \\
&\qquad\qquad |q_1 - q_2| + \ldots + |q_{n-1} - q_n| \geq |q'_1 - q'_2| + \ldots + |q'_{n-1} - q'_n|)
\end{aligned}
$$

Whenever the solver encounters an instance of *valid* that it has not seen so far, this expression is automatically registered as if we had called *add* on the expression and the *created* callback is called. To force the user-function to behave as desired, we can register all the function's arguments as well (this has

to be done manually):

```
void created(const z3::expr &func) override {
  z3::expr_vector args = func.args();
  for (unsigned i = 0; i < args.size(); i++) {
    z3::expr arg = args[i];
    if (!arg.is_numeral()) {
      this->add(arg);
    }
  }
}
```

As the solver "ignores" the quantifier before the MBQI subquery is done, we get the *created* call only when the bound variables of the quantifier have been replaced already by Skolem constants. As these are not quantified but free in the sub-query, we can register them (in the subquery).

The question now is which callback-functions should be called by the MBQI sub-solver. The *created*-callback of our propagator-object will be called for $valid(q_1, \ldots, q_n)$ because this is encountered by the outer solver. However, the callbacks for the subqueries may go somewhere else. We can control where they should go to by the *fresh* callback that we previously implemented by `return this;`. This callback is called every time the solver created a subcontext/subsolver. We may return a propagator-object that should be associated with this new context/solver. We may return the same propagator-object (in this case, this propagator will get callbacks from both solvers) or a new one. As the callbacks give no hint from which solver they come, it is necessary to return a different object if we need to distinguish where the callback came from:

```
 user_propagator_created* childPropagator = nullptr;

 user_propagator_base* fresh(z3::context &ctx)
                                              override {
  childPropagator = new user_propagator_created(ctx);
  return childPropagator;
}

~user_propagator_created() {
  delete childPropagator;
}
```

(Note, that all further MBQI checks of the solver will be done with the same sub-solver, so there will be only a single *fresh*-callback in our example. Furthermore, we are responsible for deallocating the new object.)

In the C++ API we have to register all the previously registered callbacks

in the new propagator again:

```
user_propagator_created(z3::context &c) :
        z3::user_propagator_base(c) {

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

This new ("inner") propagator-object will be notified whenever something interesting happens in the forall part of the formula. The original ("outer") propagator-object will be notified whenever something happens in the non-universally quantified part of the formula.

The "inner" propagator-object will now get callbacks that might look like, $valid(k!1, \dots, k!n)$ where these $k!i$ represent the constants generated through skolemisation. We will now register these constants within the *created* callback to get notified whenever Z3 assigned some value to them. We can now again use *conflict* and *propagate* to communicate with the calling solver.

Important: We have to deal with both: The function should evaluate to true and evaluate to false. In case the solver decides that the function should evaluate to true, we have to add conflicts whenever we notice that at least two of its arguments (queen positions) conflict. In case the function should evaluate to false, and we detect that all queens are positioned just fine, we have to add a conflict. In case the user-function is not implemented completely or consistently through the propagator the solver may not terminate! The code for the *fixed*-callback is omitted in favour of readability.

For example, let us briefly analyse the behaviour with $n = 5$:

```
Created (0): (valid q0 q1 q2 q3 q4)
Registered q0
Registered q1
Registered q2
Registered q3
Registered q4
```

First, the outer propagator-object (0) gets the *created* callback as the user-function *valid* has been encountered the first time with the arguments $q_1, \dots, q_n$ (in this order). We register all its arguments. From this point on, we get *fixed*-callbacks for the function and all its arguments:

```
Fixed (0) (valid q0 q1 q2 q3 q4) to true
Fixed (0) q4 to #b00000
Fixed (0) q3 to #b00000
...
```

We have to make sure that $(valid q0 q1 q2 q3 q4)$ is fixed to *true* if and only if its arguments are positioned fine. As soon as Z3 found a feasible placement, we

get the *final* callback. This tells us that Z3 solved the $valid(q_1, \ldots, q_n) \wedge A$ part of the formula (where A is the auxiliary constant replacing the universal part). Now, Z3 checks if the model is also a model for the universal part. Z3 notifies the outer propagator that it created a new context via *fresh* and we return a new propagator-object (1) that is from now on responsible for the universal subqueries:

```
Final (0)
Fresh context
```

This inner propagator-object will be notified that ($valid$ $k!5$ $k!4$ $k!3$ $k!2$ $k!1$) has been encountered. The $k!i$ ($1 \leq i \leq 5$) are the Skolem constants introduced. We do the same trick again as before, but this time in the inner propagator-object. We register all its arguments and make sure that *valid* behaves as expected:

```
Created (1): (valid k!5 k!4 k!3 k!2 k!1)
Registered k!5
Registered k!4
Registered k!3
Registered k!2
Registered k!1
Fixed (1) (valid k!5 k!4 k!3 k!2 k!1) to true
Fixed (1) k!3 to #b10001
Fixed (1) k!2 to #b01100
...
Final (1)
```

Z3 now found a counterexample to $(valid(q'_1, \ldots, q'_n) \Rightarrow |q_1 - q_2| + \ldots + |q_{n-1} - q_n| \geq |q'_1 - q'_2| + \ldots + |q'_{n-1} - q'_n|)$ where $q_1, \ldots, q_n$ were replaced by the candidate model from the outer run. Z3 learned that the candidate model was not correct and that it has to instantiate the quantifier. Using the values of the found model($k!1, \ldots, k!5$) is possible, but for performance reasons it tries to find a more general instantiation. For example, $k!1$ (the Skolem constant used as the last argument of *valid*) seems to be equal to the candidate value of $q3$. Therefore, Z3 instantiates $q4'$ (the bound variable at the last position of *valid*) by $q3$ and so on:

```
Created (0): (valid q2 q1 q0 q4 q3)
```

This is done until the inner-solver returns unsatisfiable. (Or the outer solver returns unsatisfiable, but this should not happen in our example, as the problem is of course satisfiable.)

4.2 Results

In contrast to the original n-queens encoding, the most user-propagator intensive solution (i.e., the one using user-functions) is not necessarily the best. It turned

out that the direct encoding without any interference of the user-propagator is far more performant than the encoding that uses the user-propagator + user-function. One of the reasons for this is that Z3 also analyses the structure of the problem to guess suitable instantiations. If we hide internals of the problem in a user-function, the performance gain of using a custom-theory may not exceed the disadvantage of loosing helpful information.

5 Some Remarks

If we want to put `z3::expr` or `z3::expr_vector` into hashtables like into STL's `std::unordered_map` we need to define equality and hash functions for them. Z3 provides them already, but we need to link them with the C++ STL containers for example via instantiating `std::hash` and `std::equal_to`.

All the things discussed here work only if we consider bitvectors or booleans. It would not work if we assume that we position the queens on integer rather than on bitvector positions, as we would not get candidate values for them. Of course, we can use user-propagation in formulas as well in which more complex theories like integer, strings, or array occur, but we have to make sure that we only register expressions that are either a boolean or bitvector. For example, we can register neither x nor y nor $x + y$ if x and y are integers in the formula $x + y = 0$. However, there is no problem with registering the whole subformula $x + y = 0$ as this has a boolean type.

Last, we may not only add conflicts between whole bitvectors. If we, for example, want to make sure that a (potentially large) bitvector x is odd, it is totally fine to register the expression `(_ extract 0 0) x` even if this expression does not occur in the original formula. Registering the expression will make sure that we will get an appropriate callback in case x is assigned. We can then add a contradiction in the corresponding *fixed* callback if the assigned value is 0. In this way, we can also add contradictions between parts of bitvectors.

6 Source Code

The complete (slightly adopted and optimized) source code for the considered n-queen problems discussed here can be found in Z3's GitHub repository (*examples* folder).