Solver description: `orsat`

Olivier Roussel
CRIL - CNRS FRE 2499 – rue de l'Université – SP 16 – 62307 Lens Cedex, France

1 Introduction

The submitted solver `orsat` version 0.1 alpha is an early and unachieved release of a experimental solver with temporary codename `orsat::qbf1`. This solver has been developed in a few weeks to experiment with a new SAT library as well as with a new QBF algorithm. By lack of time, many features are missing in the submitted version or are poorly implemented. No good performance can be expected from this version of the solver but, at this point, this doesn't allow us to draw any conclusion on the library or the new algorithm.

2 The `orsat` Library

The `orsat` library is yet another SAT library which is currently in the early phase of its development. It aims at facilitating the experimentation of algorithms for problems around SAT. Therefore, its primary goal is to be very flexible in order to suit any algorithm requirements. At the same time, this flexibility shouldn't impact performances too much (but we know we have to pay a price for this flexibility). This implies that, when we write an algorithm, we want to to be able to pick up from a list the sole features we need to develop that algorithm. One solution would be to develop an implementation with all features controlled by `#ifdef` structures. Unfortunately, this is quickly unmanageable.

The solution adopted in the `orsat` library is to use objects, inheritance and templates. Each object implements one feature and templates let us compose these features with a maximum of liberty. The library is implemented in C++ because this language give us more control on the program. Java for example looks unadapted to efficient generic programming. It is well-known that C++ doesn't check anything because this is considered a waste of time. This is a serious concern but not a major drawback because a number of tools help us detect errors.

The basic idea of the library is to build objects gradually from small pieces. For example, the class `BasicLiteral` contains nothing but the identifier of the literal and the opposite of the literal. When there's a need to give values to the literals and to use watched literals, we'd like to write something like `WatchedLiterals< SemanticLiteral<BasicLiteral> >` which must generate an inheritance tree rooted at `BasicLiteral`. C++ templates let us write this but to avoid cumbersome casts we have to use one more trick : the Curiously Recurring Template Pattern (CRTP) where the base class is passed as a template parameter the most derived class (leaf type). Basically, to define the exact kind of literal to use, one just writes something like

```
class Literal : public
  WatchedLiterals<
    SemanticLiteral<
      BasicLiteral<Literal> > >
{};
```

The `orsat` library also defines a generic search algorithm which mainly handles backtracking (intelligent or chronological). This generic search will give the ability to

- use non standard algorithms with non binary choice points
- save and restore the search (checkpointing)
- distribute the search over several hosts
- get a graphical representation of the search tree

For example, all that was needed to write the submitted QBF algorithm was to define the classes `QBFAlgorithm` and `Branch` and write the three lines below.

```
QBFAlgorithm<Formula> algo(formula);
GenericSearch<QBFAlgorithm<Formula>,
  BranchList<Branch> > search(algo);
search.explore();
```

More information on the `orsat` library can be found in [1]. The first stable public release is expected during the summer 2004.

3 The `orsat::qbf1` Algorithm

The principle of the algorithm used in the submitted version of the solver is to consider a QBF formula f as an extended SAT formula. In fact each QBF clause can be seen as a SAT clause containing the existential literals augmented with the remaining universal literals. In a sense, this algorithm changes the quantifier order of the formula since it first consider existential variables (for practical reasons, they are considered in the order defined by the prefix) and only after universal variables (in any order).

The solver tries to satisfy each clause by first assigning values to the existential literals. If this succeeds, the QBF formula is trivially true. When an interpretation I_k of existential literals can no more be extended with some other existential literal, this means that f_{I_k} (the simplification of f by interpretation I_k) contains only universal literals. Basically, the algorithm just tries to prove that for any assignation of the universal variables, there exist an assignation of existential variables satisfying the matrix (policy). The implicants of f_{I_k} define for which assignations of universal variables the policy I_k can be selected to satisfy the matrix. When there is only one universal quantifier, this simply means checking that $\bigvee_k f_{I_k}$ is a tautology.

Things are just a little bit more complicated when there's an alternation of universal and existential quantifiers. Let's take formula $\forall a_1 \exists e_1 \forall a_2 \exists e_2 \Phi$ as an example. When e_1 changes of value (let's say it becomes true), we must check that for this new value and for all valuation of a_2, there exist a value of e_2 which satisfy the matrix Φ. The values of (a_2, e_2) which satisfied the matrix and that we had collected when e_1 was false are no more of interest. In other words, during the search, when an existential literal e changes of value, implicants of an f_{I_k} containing a universal literal a which come after e in the prefix (noted $e \succ a$) must be forgotten. At the same time, we must combine implicants from different f_{I_k} to generate implied implicants by eliminating the least universal literals according to order $\succ$. This simply means that when we have policies for each value of a_2 (for a given a_1 and e_1), we can affirm that we have a policy for a_1.

To sum up, we must combine implicants of the different f_{I_k} to eliminate as much as we can the least universal literals and check if we can obtain a tautology. Each time an existential literal e changes of value, we must forget the implicants containing a universal literal a s.t. $e \succ a$. To detect a tautology, we actually work on the negation of the formula and try to generate inconsistency.

To each group of universally quantified literal, we associate a bucket which contains the negation of the implicants of f_{I_k} (this is a set of clauses). Each time a clause is added to a bucket, we generate implicates by eliminating variables of this group of universally quantified literal. If this succeeds in generating a clause with no literal of this group, we send it to the upper bucket. Whenever an existential literal changes of value, we clear all buckets below this existential literal (this is why we assign value to existential literal in the order of the prefix). The QBF formula is true if and only if the upper bucket produces the empty clause.

The submitted solver name comes from the fact that basically this algorithm is an extension of a SAT algorithm and with little programming effort it is able to solve both SAT and QBF formulas.

The current implementation is unfortunately missing a number of important features and cannot be actually competitive with other solvers.

4 Conclusion

The solver `orsat` experiments with new ideas, both at the implementation level by providing a new SAT library, and at the algorithm level by considering the QBF formula as an extended SAT formula. The `orsat` library is still in the early phase of its development but already has some interesting characteristics that may prove quite useful. The `orsat::qbf1` algorithm must be further refined and evaluated beyond this first and very basic implementation.

References

[1] *The orsat web page* http://www.cril.univ-artois.fr/∼roussel/orsat