hGen: A Random CNF Formula Generator for Hybrid Languages (System Description)

Carlos Areces

INRIA – Lorraine
areces@loria.fr

Juan Heguiabehere

LIT – Universiteit van Amsterdam
juanh@science.uva.nl

Abstract

hGen is a random CNF (conjunctive normal form) generator of formulas for sublanguages of $\mathcal{H}(@, \downarrow, \mathsf{A})$. It is an extension of the algorithm presented in [8].

1 Testing for Automated Reasoning

The development and comparison of automated reasoning tools requires the availability of suitable testbeds. The issue of testing for the satisfiability problem (and more generally the computational behaviour of the satisfiability problem) in classical languages like propositional and first-order logic has been widely investigated. The phase-transition phenomenon [3] for propositional logic has been verified through the use of *random* CNF generators like [11]. On the other hand, given the complexity of first-order languages, *hand-tailored* test sets are preferred in this case. The TPTP (Thousands of Problems for Theorem Provers) library [10] is the standard testbed for first-order provers.

The issue of testing for modal-like languages (including temporal, description and hybrid logics) has a much younger history. One of the problems in this field is, of course, the wide diversity of the possible target languages and their variations in computational complexity (ranging from NP-complete to EXPSPACE-complete and undecidable languages). One of the early testbeds for the basic modal language was the hand-tailored collection presented in [5] which was used in TABLEAUX'98. The fast development of modal provers that followed rendered this collection obsolete in a single year. In what fol-

lowed, efforts mainly focused in the design of appropriate random generators for the basic modal language [7,6,8].

The latest proposal – which allows for highly parameterized generation of modal formulas in CNF – is described in [8]. hGen extends this algorithm to allow the generation of formulas in any sublanguage of $\mathcal{H}(@, \downarrow, \mathsf{A})$, the basic modal language extended with nominals, state variables, satisfiability operators, the $\downarrow$-binder and the universal modality.

In the next section we briefly introduce random testing on CNF formulas, and then give details on $\mathrm{CNF}_{\Box_m}$, the general name used for generators of multi-modal formulas in conjunctive normal form. In Section 3 we introduce hGen, a CNF generator for hybrid languages, and in Section 4 we report on some preliminary testing.

2 Propositional and Modal CNF Generators

The satisfiability problem for propositional logic has been widely investigated, since it has many applications such as timetabling, code optimization, or cryptography. It is known that random CNF clauses of three or more literals capture the complexity of the satisfiability problem for the logic, and that random CNF clauses of more than 3 literals can be linearly encoded into CNF clauses of exactly 3 literals each. Therefore, even though there are many real-world problems and test sets available for propositional logic, one of the best known and most widely used test sets for propositional logic is Random 3SAT: a conjunction of L clauses of 3 random propositional literals each, chosen from a set of N different propositional variables. Since random 3SAT has become the *de facto* standard random test set for propositional satisfiability testing [2], developing a modal version of this test set has naturally received a lot of attention.

Random Modal CNF. A modal CNF formula (a $\mathrm{CNF}_{\Box_m}$ formula) is a conjunction of $\mathrm{CNF}_{\Box_m}$ clauses, where each clause is a disjunction of a certain number of propositional or modal literals. A literal is either an atom or its negation, and modal atoms are formulas of the form $\Box_i C$, where C is a $\mathrm{CNF}_{\Box_m}$ clause. A $3\mathrm{CNF}_{\Box_m}$ formula is a $\mathrm{CNF}_{\Box_m}$ formula where all clauses have exactly 3 literals.

A number of $\mathrm{CNF}_{\Box_m}$ formula generators have been proposed in the literature, see [7,6,8]. The latest version accepts five main parameters: the maximum modal depth D, the number of propositional variables N, the number of modalities m, the number of clauses L, and the probability p of an atom occurring at depths less than d being purely propositional. Although the usual number of literals per clause is 3, the generator gives a great degree of control over the clause size. In fact, both modal/propositional balance and clause size probability distributions can be specified either as constants or as a function of modal depth.

Given these parameters, a $\mathrm{CNF}_{\Box_m}$ formula of depth D is a set of L clauses, each made up of a number (chosen randomly according to the clause size probability distribution) of distinct modal CNF disjuncts, each consisting of either a proposition from the set $\{P_1, \ldots, P_N\}$ or, if $D > 0$, a formula $\Box_r C$, where $\Box_r \in \{\Box_1, \ldots, \Box_m\}$, and C is a $\mathrm{CNF}_{\Box_m}$ clause of depth $(D-1)$.

A standard test run using the $\mathrm{CNF}_{\Box_m}$ generator is as follows: all the parameters but L are fixed, and then a range for L is selected that covers the transition from 'only satisfiable formulas generated' to 'only unsatisfiable formulas generated'. A fixed number of formulas is generated of each of the possible configuration of the parameters, and given as input to the prover under test, generally with a time limit. Satisfiability rates, median/90^{th} percentile of CPU time elapsed, and other possible indicators are plotted against either L or L/N.

3 The Hybrid CNF Test Set

The hybrid CNF generator `hGen` extends the $\mathrm{CNF}_{\Box_m}$ generator described in [8]. `hGen` is implemented in Haskell and compiles with GHC 5.04 [4].

Parameters. The program accepts as parameters:

- The maximum nesting of operators D;
- The number of propositional variables, nominals, and state variables, N_p, N_n and N_x;
- The number of modalities, N_m;
- The number of clauses, L;
- The distribution of probabilities for clause size (a list $[f_1, \ldots, f_n]$, with f_i the relative frequency of clauses of size i);
- The probability for a disjunct of being non-atomic, p_{op};
- The relative frequencies of modalities, @-operators, $\downarrow$-operators, and the universal modality as main operator in non-atomic disjuncts, $p_{mod}, p_{down}, p_{at}, p_{univ}$;
- The relative frequencies of propositions, nominals and state variables in atomic disjuncts, $p_{prop}, p_{nom}, p_{svar}$;
- The probability for any literal of appearing negated, p_{neg};
- The number of instances to generate, *numinst*.

Given these parameters, a hybrid CNF formula of depth D is a set of L clauses, each made up of (a number chosen from $[f_1, \ldots, f_n]$) distinct hybrid CNF disjuncts, each consisting of either

- a proposition from the set $\{P_1, \ldots, P_{N_p}\}$, or
- a nominal from the set $\{n_1, \ldots, n_{N_n}\}$, or
- a state variable from the set $\{x_1, \ldots, x_{N_x}\}$, or
- if $D > 0$
  - a disjunct $\Box_r C$, where $\Box_r \in \{\Box_1, \ldots, \Box_{N_m}\}$ and C is a random hybrid CNF clause of depth $(D-1)$, or
  - a disjunct $@_n C$, where $n \in \{n_1, \ldots, n_{N_n}\}$ and C is a random hybrid CNF clause of depth $(D-1)$, or
  - a disjunct $\downarrow x_r \; op \; C$, where $x_r \in \{x_1, \ldots, x_{N_x}\}$, op is one of $\{@, \Box, \mathsf{A}\}$ which can still appear (i.e. its depth is non-zero) C is a random hybrid CNF clause with depth $(D-1)$, or
  - a disjunct $\mathsf{A}C$, where C is a random hybrid CNF clause of depth $(D-1)$.

The outline of the algorithm used to generate the formulas is given in Figure 1.

```
gen_clauses(params)
    for i := 1 to L do Cl_i := gen_cl(params);
    return (⋀_{i=1}^{L} Cl_i);

gen_cl(params)
    nd := rnd_length(params.C);
    nop := rnd_numops(nd, params);
    Atoms := rnd_atoms(params, nd − nop);
    Ops := rnd_opsd(nop, params);
    OC := {};
    foreach op_i in Ops
        OC := OC ∪ {op_i (gen_cl(params{depth := depth − 1))}
    return(⋁ OC ∨ ⋁ Atoms);

rnd_numops(nd, params)
    if (params.depth = 0) then 0
    else rnd_fc d(nd, params.pop);

rnd_atoms(params, nat)
    if (nat = 0) then {}
    else Atoms := rnd_atoms(params, nat − 1);
         atom := rnd_atom(Atoms, params);
         return(Atoms ∪ atom);

rnd_ops(n, params)
    if (n = 0) then {}
    else Ops := rnd_ops(params, n − 1);
         op := rnd_op(params);
         return(Ops ∪ op);
```

Fig. 1. Test generation structure

The **rnd_atom**(*Atoms*, *params*) function returns a random atom not in the set *Atoms*, respecting the relative frequencies of the different types of atom as given in *params*. **rnd_fc**(*nd*, *params.pop*) takes as arguments the number of disjuncts *nd* and the proportion of non-atomic disjuncts in a clause, *params.pop*. If $prop = nd \cdot params.pop$ is an integer, it returns *prop*, otherwise it returns $\lceil prop \rceil$ with probability $prop - \lfloor prop \rfloor$, or $\lfloor prop \rfloor$ otherwise (probability $\lceil prop \rceil - prop$). This prevents the accidental creation of clauses in which all disjuncts are atomic, which has been a source of triviality in modal CNF test sets [7,6,8]. **rnd_op** returns an operator according to the relative frequencies stated in *params*, optionally enforcing maximum nesting per operator. A special case is the $\downarrow$ operator, which always comes coupled with another operator; the reason is explained in Section 3.

The algorithm presented in [8] allows the specification of different clause size distributions for the different modal depths. In the presence of multi-modalities, the satisfiability operator and the universal modality the notion of modal-depth becomes rather involved. In hGen, we work instead with a global

notion of depth defined as operator nesting (this together with the probabilities for each operator, allows strict control over the generation of formulas for fragments of $\mathcal{H}(@, \downarrow, \mathsf{A})$ defined in terms of operator nesting). Clause size probability distribution is kept constant. We calculate the maximum nesting per operator from its probability of appearance and the total depth; whether the calculated depths should be enforced or not, is set from the command line. Since we're generating binders and variables, we ensure that every appearing variable is bound, and force bound variables to appear.

New triviality sources. Random generators should ensure that generated formulas do not contain trivial subformulas, whose truth value can be verified by simple syntactic checks (such formulas perturb probability distributions).

The extended expressivity of the target languages that hGen can handle introduces new tiviality sources. The following cases are handled by hGen.

For all ϕ, $\downarrow x_i.(x_i \vee \phi)$ is a tautology, and conversely for all ϕ, $\downarrow x_i.(\neg x_i \vee \phi)$ is equivalent to $\downarrow x_i.\phi$. Such formulas are never generated by hGen. Moreover, the $\downarrow$ operator does not change the state of evaluation allowing for formulas of operator depth > 0 that still require no model exploration. hGen introduces $\downarrow$ only in expressions of the form $\downarrow x_i(\neg)\Box_j\phi$, $\downarrow x_i(\neg)@_{n_j}\phi$, or $\downarrow x_i(\neg)\mathsf{A}\phi$. Otherwise the clause would be equivalent to one in which all the non-modal atoms are outside of the scope of the $\downarrow$ (since we're banning the bound variable from appearing at the same level it is bound in), effectively altering the clause size. This can only be done if we haven't yet reached the last level for all the other operators; in that case it is replaced by a propositional atom.

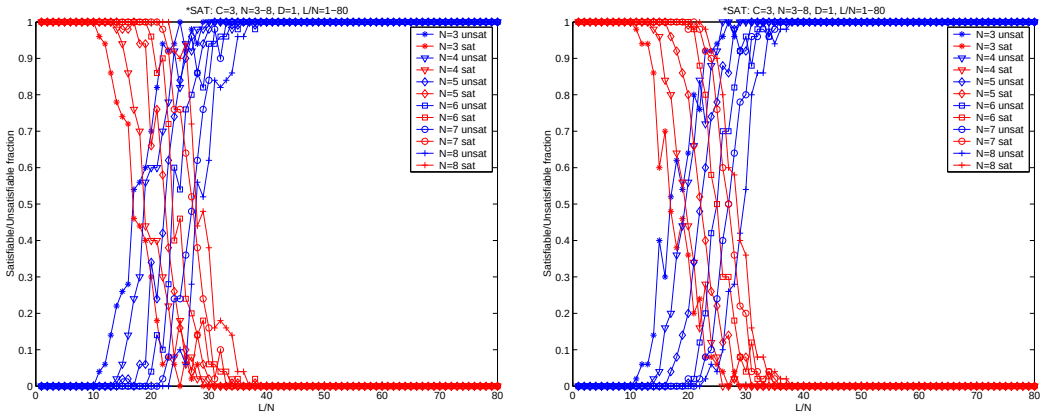
With respect to the @ operator, for any ϕ, $@_{n_1}(n_1 \vee \phi)$ is a tautology, and $@_{n_1}(\neg n_1 \vee \phi)$ is equivalent to $@_{n_1}\phi$. Again, such formulas are never generated.

4 Testing hGen

We present now some experiments we have done with hGen: (i) we test whether its only-modal behavior mimics that of the modal CNF generator, and (ii) we evaluate the behavior of HyLoRes (a resolution based prover for hybrid logics [1]) on hybrid input generated by hGen. Tests are performed using a 1.6 GHz Pentium 4 computer running Linux Red Hat 7.3.

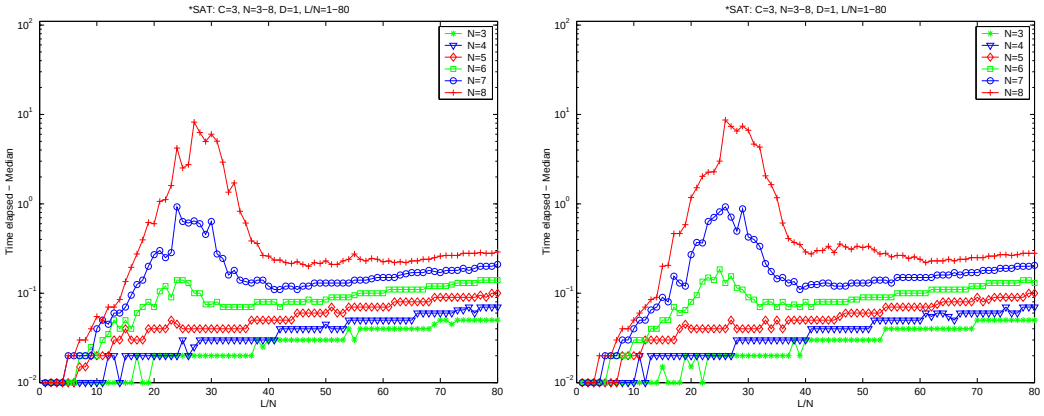
i) Using hGen as a modal test set. Since we are extending the language of the formulas generated with the modal CNF algorithm, it is important to verify that constraining the generator to modal formulas produced similar results to those obtained before. We decided to run a series of benchmarks and see if the results compared, in terms of mean difficulty, location of the easy-hard-easy pattern, and shape of the satisfiability fraction plot.

We fix all parameters but L, and ran the tests for L/N going from 1 to 80, with 50 instances per data point, for N going from 3 to 8. Modal depth was fixed at 1. We set the parameters of the generator to only produce modal formulas, and checked whether the runs showed any variations with respect



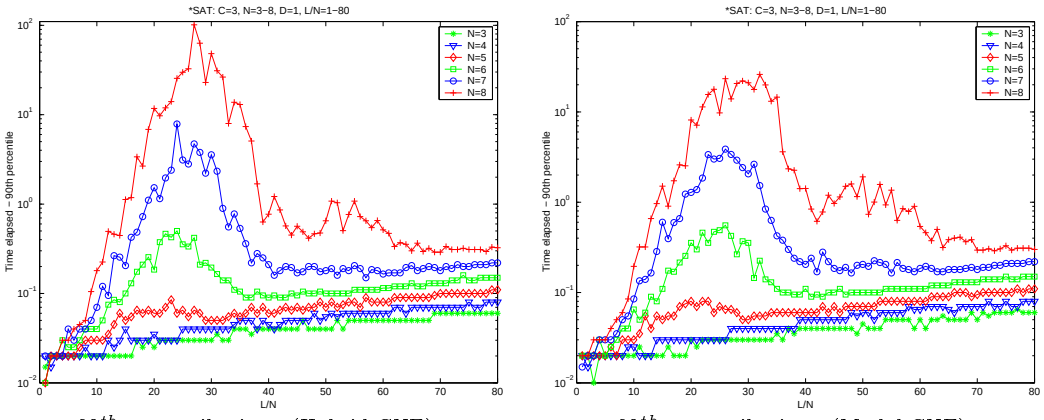
Satisfiable/unsatisfiable fraction (Hybrid CNF)

Satisfiable/unsatisfiable fraction (Modal CNF)



Median times (Hybrid CNF)

Median times (Modal CNF)



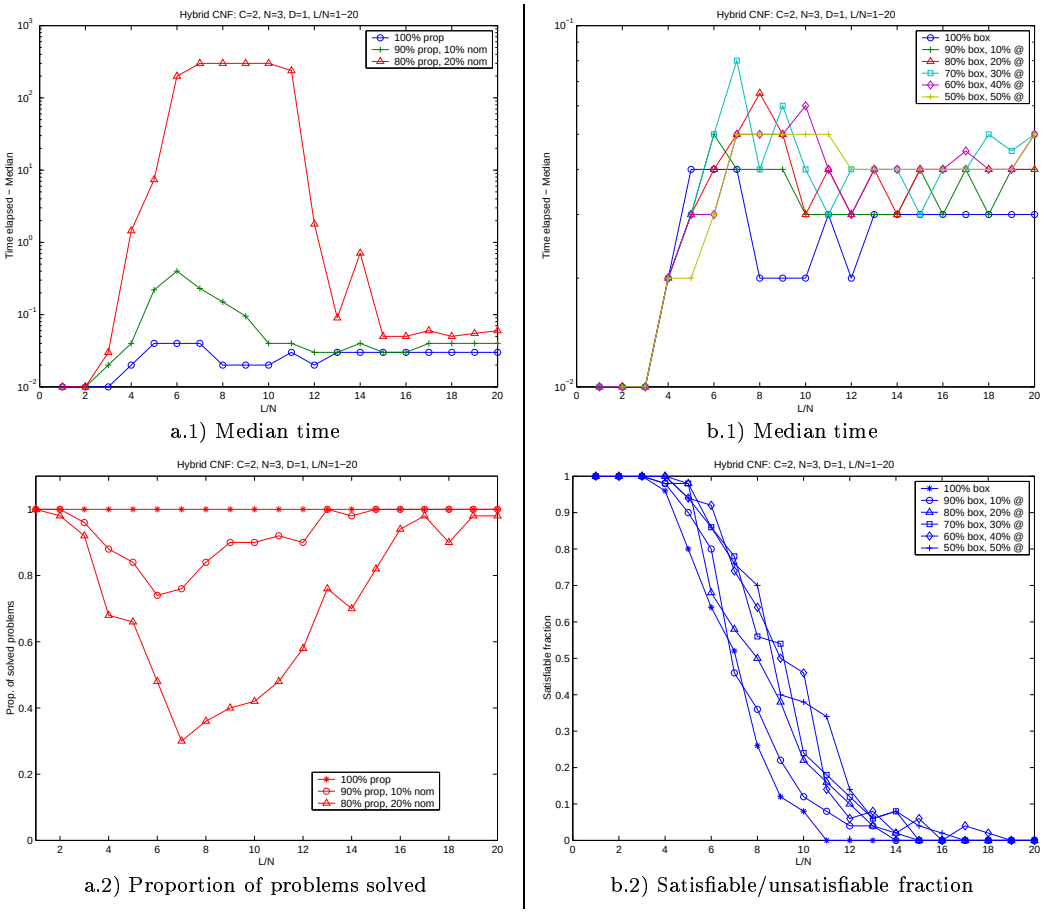
90^{th} percentile times (Hybrid CNF)

90^{th} percentile times (Modal CNF)

Fig. 2. Results of the comparison between Hybrid and Modal CNF

to runs of the Modal CNF test set for equivalent parameter sets. The prover we used for this benchmark was *SAT [9]; we ran the tests with a timeout of 300 seconds.

Results are displayed in Figure 2. The first two graphs display the satisfiable/unsatisfiable fractions; the second two show the median of the CPU time used for every data point, and the third ones show the 90^{th} percentile of the CPU times. The experiment confirmed that, for equivalent parameter



a.1) Median time

b.1) Median time

a.2) Proportion of problems solved

b.2) Satisfiable/unsatisfiable fraction

Fig. 3. Hybrid CNF tests

sets, the behavior of both test sets is very similar, in terms of location of the satisfiable/unsatisfiable transition and overall difficulty.

Of course, the Modal CNF test set allows for specification of clause size probability distribution and modal/propositional balance as a function of modal depth, while the hybrid CNF generator only accepts constant distributions, so the relationship between the test sets is more one of overlap than one of inclusion.

ii) hGen as a hybrid test set We present now some preliminary tests of the hybrid capabilities of HyLoRes, evaluated using hGen.

In Figure 3 a) we start with a purely modal base case, with $C = 2$, $N_p = 3$, $D = 1$, and gradually add nominals to the mix; that is, with $N_n = 5$ we keep $p_{svar} = 0$ and do one run with $p_{prop} = 1$, $p_{nom} = 0$, one with $p_{prop} = 9$, $p_{nom} = 1$, and one with $p_{prop} = 8$, $p_{nom} = 2$. The timeout was 300 seconds. Figure 3 a.1) shows the median time elapsed, while Figure 3 a.2) shows the proportion of problems solved. Here we see that even with slight increases of the quantity of nominals the difficulty rises sharply; HyLoRes is resolution based, and handling nominals is done by means of paramodulation with is very costly. The tests highlights the fact that optimizing paramodulation is

crucial for good performance. Figure 3 b) shows the effect of increasing the proportion of @-operators, starting from the same base case. We see that the difficulty changes very little (although the peak moves to the right), and the satisfiable/unsatisfiable transition moves to the right (see b.2)) as we increase the proportion of @-operators.

The sharp differences between the two experiments are to be expected, since the presence of nominals in a formula triggers paramodulation (potentially duplicating the size of the clause set and forcing a "global" inspection of the set of clauses), while the @-operator can be handled with a much more controlled mechanism (involving only "local" operations).

References

[1] Areces, C. and J. Heguiabehere, *Hylores: A hybrid logic prover based on direct resolution*, in: *Proc. of Advances in Modal Logic 2002*, Toulouse, France, 2002.

[2] Gent, I., H. van Maaren and T. Walsh, editors, "SAT 2000," IOS Press, 2000.

[3] Gent, I. and T. Walsh, *The SAT phase transition*, in: A. Cohn, editor, *Proc. of the 11th European Conference on Artificial Intelligence*, Amsterdam, The Netherlands, 1994, pp. 105–109.

[4] *The glasgow haskell compiler*, `www.haskell.org/ghc/`.

[5] Heuerding, A. and S. Schwendimann, *A benchmark method for the propositional modal logics K, KT, and S4*, Technical report IAM-96-015, University of Bern, Switzerland (1996).

[6] Horrocks, I., P. Patel-Schneider and R. Sebastiani, *An analysis of empirical testing for modal decision procedures*, Logic Journal of the IGPL **8** (2000), pp. 293–323.

[7] Hustadt, U. and R. Schmidt, *On evaluating decision procedures for modal logic*, in: M. Pollack, editor, *Proc. of the 15th International Joint Conference on Artificial Intelligence*, 1997, pp. 202–207.

[8] Patel-Schneider, P. and R. Sebastiani, *A new general method to generate random modal formulae for testing decision procedures*, Journal of Artificial Intelligence Research **18** (2003), pp. 351–389.

[9] *SAT Homepage. URL: `www.mrg.dist.unige.it/~tac/StarSAT.html`.

[10] Sutcliffe, G., C. Suttner and T. Yemenis, *The TPTP problem library*, in: A. Bundy, editor, *Proc. of CADE-12*, number 814 in LNAI, Nancy, France, 1994, pp. 252–266.

[11] van Gelder, A., *Problem generator* ***`mkcnf`*** (1993), contributed to the DIMACS 1993 Challenge archive.