

Software Verification and Analysis Randomized Model Finder

Eyer Leander, C'edric Jeanneret

Abstract—This document introduces a new method using randomness to find model for first order logic formulas. This new method has been tested on a simple example and results, even far from well-known model finder tool, are encouraging. This paper also present a solver for TPTP problems based on KodKod.

I. MOTIVATION

First order logic is present in many fields, like software verification, mathematics, philosophy, etc. Various applications requires to find an interpretation (the binding of the free variables to domain elements and the evaluations of functions or predicates) that satisfies a given first order logic formula. Among them, one may find¹:

- Showing the consistency of formal specifications
- Solving certain mathematical problems
- Finding counterexamples to false conjectures, thus avoiding infinite running of automatic theorem provers
- Providing semantic information to a resolution-based theorem prover, to guide its search for a proof

As they will be presented in next section, none of the existing tools for finding model is genuinely using randomness. However, in some SAT solvers, randomness is successfully used to improve the efficiency in finding assignment satisfying a set of propositional clauses. Why would randomness not be used to find model for first order logic ?

II. RELATED WORK

There are several ways to find a model for a first-order logic formula. One could for example search exhaustively for a model. Due to exponential explosion, such a method are limited to simple formulas or to small domains.

SEM-style tools performs backtracking searches. The search space is reduced using advanced techniques such like constraints propagation, symmetry detection, etc.

On the other hand, there is a complete family of tools based on the fact that model finding can be reduced to solving a problem in a simpler logic. For instance, MACE tool expresses the interpretation of a FOL formulas with some propositional clauses. These clauses would later be solved by a SAT-Solver, and it suffices to map the assignment back to an interpretation. If the reduction is sound, this interpretation is a model.

Paradox [1] is a improvement of MACE, introducing several techniques to reduce the size of the generated SAT problem.

In the same category, KodKod [2] accepts constraints from a language that combines first-order logic with relational algebra and transitive closure. Further more, it allows to specify partial solutions. KodKod also introduces an effective symmetry detection and provide an economical translation from relational logic to boolean logic.

III. MAIN IDEA

From previous section, it turns out that model finding is somehow a trade off between space (when translated to propositional clauses) and time (when search in initial logic is performed). Another observation one could do is that none of the existing tools exploit randomness to improve its efficiency (even if randomness can be used within the SAT-solver itself).

Randomness seems undesirable in the world of logics and proofs. However, in case where the model finder is used to find a solution to a

¹<http://lcs.ios.ac.cn/~zj/model.html>

problem (and not to show the non-existence of this solution), one may be interested in having a possibly approximate answer quickly. Based on these observations, we designed a new method to find model for first order logic.

Since the list of predicates and functions used in a set of FOL formulas is known, we propose to encode an interpretation as a list of values. This list would then be implicitly *attached* to a binary search tree. Consider a function f over a domain of size 2. Its interpretation can either be written explicitly (as in the table II) or implicitly (as the list of red elements of the figure 1).

A	A	A
A	A	B
A	B	A
A	B	B
B	A	A
B	A	B
B	B	A
B	B	B

TABLE I
INTERPRETATION OF A FUNCTION

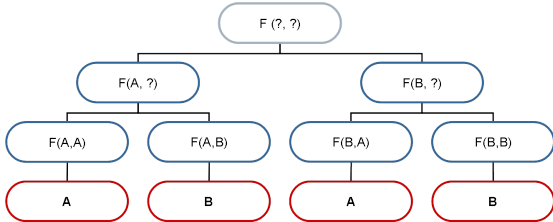


Fig. 1. Example of our interpretation's representation

Each elements in the list is either a domain element (for functions) or a boolean (for predicates). In any case, it can be encoded as a floating point number: let $0 \leq u < 1$. If the element is a boolean, its value is *true* if $u > 0.5$. On the other hand, if a domain element is expected, its value will be E_i , with $i = \lfloor u \cdot \text{domain's size} \rfloor$.

The interest of having a floating point representation is that it allow us to introduce the notion of *speed* at which the interpretation evolves. In other words, between two iterations, we have the following mathematical relation:

$$\vec{x}_{k+1} = \vec{x}_k + t \cdot \vec{v}$$

with \vec{x}_i being the float encoding of the interpretation at the iteration i , \vec{v} the speed, and a scalar t measuring the *time* elapsed during the iteration. Please not that components are normalized to the interval $[0, 1[$ at every iteration.

A search algorithm defines the initial interpretation \vec{x}_0 , the duration t_i and the speed \vec{v} of an iteration.

Since the domain is torroïdal, it is possible to perform exhaustive search with a constant time and speed. Details are omitted for sake of brevity, but the idea is to give the last component a speed such that its value changes at every iteration whereas the first components moves slowly (in a similar way than digits from an odometer in a car). Purely random search can also be modeled using this representation. It suffices to pick a new random speed at every iteration.

Actually, interpretation are either a model, or not. If we could give an estimation of how far from a model the interpretation is, model finding would be reduced to finding a minimum over a multi-dimensional bounded domain. Techniques exists to solve such problem, like gradient descent or particle swarm optimization. We propose to use the number of non satisfied axioms or conjectures as metric to estimate the *distance* to a model.

IV. IMPLEMENTATION

In the previous section we presented some new ideas in model finding techniques. This section will present the work we achieved towards these directions.

A. TPTP

We decided to use TPTP [3] as front-end language for our experiments. Firstly because it is accepted as a standard in automated reasoning community and secondly because an impressive library of problems is provided for testing and comparison purpose. In TPTP, problems are composed by a set of formulas or clauses. These are annotated with a name and a kind (hypothesis, axiom, theorem, conjecture, etc). Basically, a problem is solved if conjectures are proved using axioms. Using a model finder, this is done by

finding a model satisfying both the axioms and the negation of the conjecture.

Here is the abstract syntax used within our project²:

$$\begin{aligned}
 F & := A \mid F \wedge F \mid F \vee F \mid !F \\
 & \quad \mid \forall x. F \mid \exists x. F \\
 & \quad \mid F \Leftarrow F \mid F \Rightarrow F \mid F \Leftrightarrow F \\
 A & := \text{predName } T^* \\
 T & := \text{symbolName } T^*
 \end{aligned}$$

Roughly, formulas are composed by atoms using traditional first order logic connectors like conjunction, negation, implications, etc. Quantified formulas are also supported. Atoms are predicates, identified by a name and applied to a set of terms. Furthermore terms are functions, themselves identified by a name and applied to a set (possibly empty, in the case the function is called a constant) of terms.

To parse input files, we used an existing Java parser for TPTP ³ written by Andrey Chaltsev. The abstract syntax tree produced by this parser is then translated to an equivalent one in Scala. This translation is required because further tree manipulations will be implemented by using the Scala pattern matching mechanism.

B. KodKod

In order to get used with model finding problematics, we started to write a model finder based on KodKod, a relational model finder (see section II for more details). The tool is provided as a Java API, therefore, from a technical point of view, it is easy to interface it with our TPTP parser. Although KodKod supports all first order logics connectors, functions and predicates have to be expressed with relational operators. The following walk through explains how to translate a TPTP formula into a KodKod one.

- 1) Create a universe with n elements, n being the size of the searched model
- 2) Navigate through the abstract syntax tree. For every

- predicate of arity k :
 - a) create a relation of arity k
 - b) set its lower bound to empty set
 - c) set its upper bound to all possible k -tuples made of domain elements
 - d) store this relation in a symbol table
 - function of arity 0 (a constant):
 - a) create a relation of arity 1
 - b) set its lower bound to empty set
 - c) set its upper bound to the universe
 - d) store this relation in a symbol table
 - e) store an additional formula constraining the relation size to 1.
 - function of arity k :
 - a) create a relation of arity $k + 1$
 - b) set its lower bound to empty set
 - c) set its upper bound to all possible $k + 1$ -tuples made of domain elements
 - d) store this relation in a symbol table
- 3) Recursively build the KodKod abstract syntax tree:
- the conversion of logical connectors (conjunction, negation, etc) is straightforward
 - quantified formulas need special care:
 - a) create a new variable
 - b) store the variable on top of the symbol table
 - c) create a declaration claiming the freshly created variable is member of the universe
 - d) convert the inner formula
 - e) create the KodKod quantified formula using the converted inner formula and the declaration
 - f) remove the variable from the symbol table
 - equalities between two expressions are dealt by the `eq` construct
 - atoms (predicate applications) are transformed as testing whether the product of the arguments are member of the predicate relation
 - variable terms are replaced by variables found in the symbol table
 - constants are replaced by their relation
 - terms (function applications) are slightly more difficult to deal with: the function is curried and arguments are joined from the left to the relation of the function. For

²please note that this is not the TPTP concrete syntax

³http://www.freewebs.com/andrei_ch/

instance, $f(a, b)$ becomes $b.(a.f)$, $.$ being the joint operator.

- 4) build a formula as conjunction of declarations, axioms and the negation of the conjecture. This is the formula that has to be solved by KodKod.

C. RMF - Randomized Model Finder

1) Main Classes and Abstractions:

- **Iterable Interface:** the iterable interface lies at the heart of our implementation. It defines the following methods:
 - `hasNext()` - allows to see if this iterator has more elements available
 - `next()` - sets the iterator to the next element
 - `reset()` - reinitializes the iterator to it's first element
 - `rand()` - forces the iterator to enter a random state
- **VariableIterator:** the basic implementation of `Iterable` for a variable. It iterates over the range from 1 to the domain size given to it.
- **StackIterator:** used to connect two iterators (called *self* and *sub*) to work like a distance counter in a car. `next()` requests are forwarded to *sub* until *sub* reaches the end of it's domain, then *self* is increased and *sub* reset. The stack iterator is used to enumerate all possible interpretations in exhaustive search.
- **InterpretationElementRepr, FunctionRepr and PredicateRepr:** this class encodes an interpretation for a function or a predicate (the main difference is that the resultat of functions iterate over the whole domain while the resultat of predicates iterate only over two elements (true and false)). current The interpretation is represented by a stack iterator. The Function/Predicate representations themselves implement the `Iterable` interface as well and delegate requests to their interpretation objects. All function and predicate representations are connected in a stacked iterator which allows us to easily enumerate all possible combinations of function interpretations in the system.
- **Evaluator:** the evaluator evaluates a formula against an interpretation and decides if the

formula holds. This indicates that we have found a valid model.

2) *Abstraction of the Interpretation:* For each function and predicate symbol, the interpretation has to list the result of each combination of input values. We encode the interpretation as a list of numbers, such that the parameter values indicate the position in this list to load the result. The size of this list is $domain\ size^{arity}$. As an example, let's consider a predicate $p(a, b)$ of arity 2 in a domain of 3 and an interpretation for the equality function:

A	B	Res
1	1	1
1	2	2
1	3	2
2	1	2
2	2	1
2	3	2
3	1	2
3	2	2
3	3	1

The interpretation would thus be stored as the list (1, 2, 2, 2, 1, 2, 2, 2, 1).

3) *Searching interpretation:* Each of the numbers in our interpretation is stored as a variable iterator which are connected in a stack iterator. For exhaustive search, we simply enumerate all possible combinations of values in the list. For randomized search, we will randomize all positions at each iteration. Our experiments have shown that this technique is very inefficient and requires on average about as many iterations as there are possible iterations.

4) *Need for speed:* To reduce the number of iterations, we introduce the concept of speed. Each position in the interpretation list has a speed associated with it. The speed encodes the probability of that position changing in the next iteration. The idea is that 'correct' assignments in our interpretation remain stable while 'incorrect' ones change frequently until they reach a correct value. Changes in the speed are communicated back by the evaluator. The forall statement will test it's predicate against every element in the domain, and if it finds that an element is correct it propagates a decrease in speed for the assignment of values and their result. If the predicate is wrong an increase in speed is propagated.

For the equality predicate of the above example, consider the following interpretation:

1 1 2 2 2 2 2 2 1

The evaluator will check the value for (1,1) and find the value 1 which is correct, the speed for position one is thus decreased. The interpretation for (1,2) however is 1 which is wrong, the speed of change for the second position will thus be increased.

V. RESULTS

We analyzed three procedures for model finding:

- exhaustive search
- randomized search
- randomized search with speed

For testing we used a formula described the equality relation:

$$![H1, H2] : (q(H1, H2) \Leftrightarrow H1 = H2)$$

The formula has one correct model for any domain size.

Iterations	Domain 3	Domain 4
Exhaustive	238	31710
Randomized	493	60000
Speed	17	57

TABLE II

PERFORMANCE OF VARIOUS SEARCHES

The exhaustive search finds the model after analyzing about 50% of the possible interpretations since the model is located in the middle of our linear search space. Randomized search takes on average about twice as long which can be explained by the fact that interpretations might be visited several times (memory restrictions prevent us from storing which interpretations were analyzed already). The introduction of the speed heuristic allowed us to cut down the required number of iterations considerably. Also, while the number of analyzed interpretations grows linearly with the space of interpretations for exhaustive and randomized search, 'speed' based search is able to find the model in logarithmic space.

VI. LIMITATIONS

During our project, we had to face to some technical limitations. We started to write the search algorithms in a functional style, but high order functions (like `map` or `reduce`) were not usable because of stack overflows (due to the length of the interpretation lists, even for small domain or simple formulas). Even moving to imperative object-oriented code does not prevent us from heap overflow. According to this, Scala may not be the most appropriate language.

Our TPTP front end for KodKod is not able to take advantage of the available relational logic (like closure, etc). Handwritten translation of some TPTP problem (provided as examples in KodKod package) are by far more simple than the one automatically generated (because using relational constructs). So far, we were not able to extract partial instance from a TPTP problem. These facts may make KodKod solve our automated translation in a less efficient way than the translations provided and used as benchmarks in the KodKod publication.

Currently, our *implemented* metric only works with the universal quantifiers since violated predicates clearly indicate which elements of the interpretation needs to change. Existential quantifiers do not allow us that since it is not clear which combination of input values should change to fulfill the predicate at least once.

VII. FUTURE DIRECTION

Limitations presented in previous section let us propose some ideas for further work. First of all, the representation should be cut so that it contains only the information needed for an evaluation. In our project, interpretations contain the values for every functions or predicates, for every possible combination of arguments. Making the list shorter would reduce the time required for computation and the memory footprint. The search space could also be reduced by applying techniques presented in related works, like symmetric detection, constraints propagation, etc.

Due to the lack of time, we did not test different metrics, neither different optimization techniques like particle swarm optimization or gradient descent.

It would be valuable to measure the efficiency of reducing model finding to an optimization problem in a scientific way, by using, for example, the TPTP library.

VIII. CONCLUSION

During this project, we were confronted to the difficulties of finding model. Our intuition was that it was possible to move randomized SAT solver techniques upwards to first order logics. In some extent, we were successful in this *migration*. But the available time did not allow us to explore several path that remain open for further work.

ACKNOWLEDGEMENT

We would like to thank Doctor V. Kuncak for all what we learned during this semester and all the documentations he provided to us for this project

REFERENCES

- [1] K. Claessen and N. Sörensson, “New techniques that improve MACE-style model finding,” in *Proc. of Workshop on Model Computation (MODEL)*, 2003.
- [2] E. Torlak and D. Jackson, “Kodkod: A relational model finder,” in *Tools and Algorithms for Construction and Analysis of Systems (TACAS '07)*, 2007.
- [3] G. Sutcliffe and C. Suttner, “The TPTP Problem Library: CNF Release v1.2.1,” *Journal of Automated Reasoning*, vol. 21, no. 2, pp. 177–203, 1998.