Satisfiability Modulo Theories: state-of-the-art, contributions, project

Pascal Fontaine

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Satisfaisabilité Modulo Théories

Satisfiability Modulo Theories

state-of-the-art, contributions, project

Habilitation à Diriger des Recherches

Pascal Fontaine

14 août 2018

Membres du Jury:

Rapporteurs:

Sylvain Conchon	Université Paris-Sud 11, France
Renate Schmidt	University of Manchester, UK
Roberto Sebastiani	Università di Trento, Italy

Examinateurs:

Stephan Merz	Université de Lorraine, CNRS, Inria, Loria, France
Jeanine Souquières	Université de Lorraine, CNRS, Loria, France
Viorica Sofronie-Stokkermans	Universität Koblenz-Landau, Germany
Cesare Tinelli	The University of Iowa, USA

Loria
Laboratoire lorrain de recherche en informatique et ses applications

About this document

Unlike for PhD theses, where there is a firm deadline in the form of the end of available funding, the habilitation has no such constraint. The consequences of not going through the ritual of the habilitation remain for long imperceptible, except that more and more esteemed people around you start to tell you in a quite direct way, that you should submit your habilitation, now. I wish to thank those esteemed people, because their influence has been crucial in putting an end to procrastination and making me start thinking seriously about it. Among them, I wish to particularly thank Jeanine and Stephan for they put quite a lot of energy in pushing me.

Once it is clear that submitting a manuscript is unavoidable, the next question is the content of this manuscript. My time is extremely limited, between my teaching duties, my administrative tasks, the urge to continue doing research and supervising students, and taking care of a family. I must admit this had a direct influence on the form this document, which is simply a short and informal essay on the state-of-the-art in the field, followed by my contributions and project. I leave for later times the ambition to write a longer meaningful, complete, formal scientific memoir, and hope that readers of this short text will find it both easy and interesting to read.

For readers who want a more traditional entry point to Satisfiability Modulo Theories, I most strongly recommend the survey on SMT in the Handbook of Satisfiability [16] and this general audience article [15]. It may happen in this text that some common knowledge about SMT misses a reference about well known results; then the reader will have no problem to find the right reference in [16]. This document also introduces propositional satisfiability solving for which more information can be found in [116] from the same collection of articles [22] all related to various aspect of the satisfiability checking problem.

1 Introduction: on the importance of SMT

Propositional Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) have many industrial applications that should suffice to convince anyone of the interest to study this field. My involvement was mainly around verification; I contributed to the integration of SMT within verification platforms (Rodin, Atelier B) and interactive theorem provers with some success. However, if this serves well to justify my salary, the very reason for my interest for SMT does not rely on industrial applications. My previous life as a physics student gave me the very uncomfortable feeling that scientific reasoning is sometimes extremely imprecise, even when the right conclusions are drawn. I have found myself many times guilty of having completely false intuitions, and sometimes even writing very wrong proofs. I believe now that most hand written proofs are wrong or miss details, and a good hand written proof is not a correct one, but one that can easily be corrected. Rigor and precision are not easy to achieve. When the meanders of life brought me by pure chance to logic and to the right people to study it with, it was clear that I wanted to contribute to it. I was impressed by computer scientists who spend considerable time and energy to model, completely formally, large fragments of mathematics inside interactive theorem provers. In my very naive understanding of the world as a post teenager, I was even believing that some day, we would do the same with physics, chemistry, economy, law and politics. Part of this dream still lives on.

It was clear at that time that interactive theorem provers, although already awesome, were not yet perfect. In particular, automation was lacking. My bottom-up attitude to everything — after all, I studied physics to understand the world, and logic at first because I was unsatisfied with my sloppy understanding of the reasoning rules underlying physics — led me directly to propositional satisfiability checking. It was though also already clear to me that arithmetic, first-order logic, and quantifiers were to be added to the game for expressivity. In another environment or with a slightly different little influence, my core interest might have been superposition theorem proving [10, 99], but I discovered STeP (The Stanford Temporal Prover [23, 91]) and SVC (The Stanford Validity Checker [18], the ancestor of CVC4 [14]). Basically, my main research topic was settled on SMT. I can proudly assert that I never thought SMT would solve all problems easily. Indeed this has occupied me for nearly twenty years now.

In the first chapters, I will provide an informal and personal introduction

to SAT and SMT. The next chapters are dedicated to some of my research directions, where I will present my contributions and objectives. The last chapter is a digression with a plan for educating young computer scientists to the basics of computer science alongside my research interests.

2 A gentle introduction to SAT and SMT

It is not easy to understand the true reasons for the success of classical propositional logic, propositional logic for short in the following. It is arguably not the historical foundation of logic, Artistotle's syllogistic logic [6] being closer to first-order predicate logic than to propositional logic. Without a clear definition of *simplicity*, propositional logic can also not be said to be the simplest logic: intuitionistic propositional logic actually does not impose the law of the excluded middle, and in that sense, is simpler. But considering intuition, classical propositional logic is unchallenged: it is very easy to make children and students believe a proposition is either true or false, and quickly understand in this context the basic notions of logic, e.g., satisfiability, equivalence or what is a decision procedure. While the expressivity of propositional logic is quite limited, many relevant problems can be encoded within classical propositional logic. Last, but not least for our part, propositional logic is the basis of a series of logic with increasing expressivity. The methodology used in SMT solving values this, eventually reducing expressive languages into propositional logic.

The SAT problem, i.e., checking the satisfiability of propositional formulas, is an NP-complete problem. It is in NP, since it is possible to check in polynomial time if a formula is true under a given valuation of the propositions, assigning to each of them either the value true or false. It is NP-hard since any problem in NP can be encoded polynomially into SAT [37]. The SMT problem includes the SAT problem, that is, a SAT formula can be seen as an SMT formula, for the usual theories of interest. Thus SMT is NP-hard. Depending on the background theory and the restriction of the input language, the SMT problem may also be in NP. This is indeed the case whenever the satisfiability problem for conjunctive sets of literals in the SMT logic is itself in NP, e.g., for quantifier-free first-order logic with uninterpreted functions and predicates (see e.g., [98]), and linear arithmetic on rationals and integers (see e.g., [101]). In this context, solving the SMT problem with the help of a background SAT solver can just be seen as an on-the-fly encoding of an NP-complete problem into another. In the solving techniques, just like for the above complexity considerations, propositional SAT solving is central.

2.1 Solving the SAT problem

The paradigm at the core of most modern SAT solvers is Conflict-Driven Clause Learning (CDCL, see Chapter 4 of the Handbook of Satisfiability [22]). It is often claimed to be a refinement of the DPLL algorithm, by Davis, Puttnam, Logemann and Loveland [39, 38]. There is no doubt that the ideas in DPLL were of primal importance to the design of modern SAT solvers. But ignoring the history of the design of CDCL and understanding CDCL starting from truth tables might just be easier.

Checking if a propositional logic formula can be true or not can be as simple as checking the truth value for each possible assignment, i.e., building a truth table. Each line in the truth table states one assignment, and since there are only 2^n possible valuations assigning true or false to n propositional variables, a truth table for a formula with n variables contains 2^n lines. Creating the whole truth table is exponential in space, but keeping track of every assignment is of course not required. An algorithm using linear space would simply iterate over the assignments; when a satisfying assignment is found, the algorithm stops and declares the formula to be satisfiable. Let us however assume that the algorithm records the "failed" assignments, i.e., the ones that have been tried but do not satisfy the formula. The algorithm now picks successive assignments in no particular order, and uses exponential space to record them. Surprisingly, this makes the naive and inefficient algorithm of truth tables quite close to the state-of-the-art and efficient CDCL framework: iterating over assignments is also at the core of CDCL. Although CDCL does not record failing assignments, it learns conflict clauses, standing for all the assignments making this clause false, i.e., a set of failing assignments.

Consider now a formula $(b_1 \vee b_2) \wedge G(b_3, \dots b_n)$ over n Boolean variables $b_1, \dots b_n$. Obviously, if b_1 and b_2 are false, it is not necessary to complete the assignment with valuations for $b_3, \dots b_n$ and check every of the 2^{n-2} possible completions. Provided the algorithm can detect some cases of unsatisfiability with partial assignments, this can lead to a dramatic improvement over checking all 2^n full assignments. If the formula is a set of clauses, i.e., a conjunctive set of disjunctions of literals, the algorithm could for instance detect when a clause is false under the current partial assignment, that is, when all its literals are set to false by the current partial assignment. Assuming that the input formula is a conjunctive normal form (CNF, a conjunctive set of clauses) significantly simplifies automated reasoning techniques, and the price to pay is reasonable: every formula can be transformed into an equisatisfiable conjunctive normal form in linear time, by introducing new Boolean variables

for subformulas, a technique known as Tseitin's transformation [104, 121]. It can be argued that this transformation, even if linear and mostly reversible, potentially hurts the solving since the structure of the formula is somehow lost. In practice, it is easier to find mitigating techniques for this structure loss rather than designing algorithms for arbitrary formulas.

2.1.1 Boolean Constraint Propagation (BCP)

Although we just promised a dramatic improvement in the truth table algorithm by detecting unsatisfiable clauses early, we have not yet mentioned how to concretely detect when a clause becomes unsatisfiable, or *conflicting*, under a partial assignment. This task has to be done for every clause, and each time the partial assignment is extended: simply iterating over the whole set of clauses is thus impractical. If a partial assignment is extended with, let's say, a valuation to true for a propositional variable p, any clause containing p negatively might become conflicting and only those ones. This thus calls for an indexing of clauses by the negations of their literals. If a partial assignment is extended by setting a literal to true, only the clauses in the corresponding list might actually become conflicting.

Notice that, with such an indexing, every conflicting clause with n literals has been examined n times during the building of the partial assignment conflicting with the clause. And it is only useful the last time, i.e., when detecting the conflict. In fact, if a clause is not yet satisfied by the current partial assignment, it is only necessary that it appears in the list associated to the negation of one of its unassigned literals. To guarantee the detection of all conflicting clauses, it then suffices to enforce the invariant that every non-satisfied clause (that is, every clause without any literal set to true under the current assignment) is indexed by at least one of its unassigned literals, called the watched literal. This requires that, each time a literal is set to false, all the clauses indexed by this literal have to be indexed again by another of its literals that is unassigned or true under the current assignment. If it is not possible to find a replacement literal for a clause, the clause is in conflict with the current partial assignment, and no completion of the assignment can satisfy the CNF.

An obvious improvement is possible: rather than building partial assignments until a clause gets conflicting, it is much more profitable to extend the partial assignment as soon as a clause, called *unit*, imposes a choice on the assignment. In other words, if all the literals of a clause but one are false

under a partial assignment, then the last literal should be set to true. This is called *propagation*: partial assignment are extended with the aim of satisfying clauses rather than relying on guesses to make the clause true. Again, a possible but inefficient way to detect such propagating clauses would be to check all clauses every time a partial assignment is extended by a new literal. But it is also easy to improve the above technique based on watched literals to also accommodate propagation. To ensure detection of conflicts, it is necessary to have one (watched) unassigned literal per clause. To detect propagation, it suffices to inspect the clauses whose assigned literals are all false when the next but last unassigned literal becomes false. The same technique still works however, using two watched literals instead of one. The invariant becomes: every non-satisfied clause (that is, every clause without any literal set to true under the current assignment) is indexed by two of its unassigned (watched) literals. Every assignment, let's say, of a literal ℓ to false, might break the invariant. To repair it, the algorithm checks every clause with ℓ as watched literal, and, if it is not satisfied, ensures that it is indexed by two of its unassigned literals. Concretely, the clause is removed from the list associated to ℓ and added in the list associated to another of its unassigned literal while ensuring both watched literals are different. If this fails, it is either because there is only one unassigned literal, in which case the other literal has to be assigned positively immediately, or it is because there is no unassigned literals at all, and then the clause is conflicting. Conflicts can indeed still happen: a clause may have all its literals (including both watched literals) falsified by propagation from other clauses before it is itself considered for propagation. In the assignment stack — the stack of assigned literals by chronological order — some recently assigned literals (e.g., by propagation) may have not been yet fully taken into account for propagation.

Compared to the truth table algorithm with conflict detection on partial assignments, there is now less non-determinism in building the partial assignment. Some literals are now assigned by propagation. It can happen however that no literal can be propagated at some point. Then guessing is used to assign an additional literal, which may or may not trigger propagation. The process stops when a conflict occurs, or no more literal is unassigned, in which case the formula is satisfiable. In case of conflict, either no literal has been guessed and the formula is unsatisfiable, or one or several guessed literals have to be changed. Backtracking is the subject of Section 2.1.3.

Propagation is a central aspect of SAT solving, drastically reducing guessing and promoting deduction. In modern SAT solvers, the solving time is

generally dominated by Boolean constraint propagation, and most literal assignments are due to propagation. Watched literals, first introduced in the Chaff [95] SAT solver, were one essential factor of its success.

Using watched literals rather than a full indexing of clauses by literals trades the examination of each clause every time the partial assignment is extended, for a dynamic index instead of a static one. In modern SAT solving, it appears that this indexing is not as dynamic as one might fear. In a sense, thanks to the decision heuristics, the watched literals of a clause remain quite stable, because the order of assignment is also evolving slowly. The watchers stabilize themselves with respect to this order, i.e., they become the literals assigned last, and remain so for some time.

2.1.2 Conflict Analysis

Propagation prevents extending a partial assignment in an obviously wrong way, when a clause imposes the value of one Boolean variable because all but one of its literals are false. Conflicts can still occur, when a same variable can be propagated both positively and negatively by two different clauses. In such a situation, the partial assignment is not extendable to a model of the formula. The clause comprising the negation of all literals in this partial assignment is a logical consequence of the original formula, and therefore adding it to the formula preserves logical equivalence. It may however be possible to add a stronger clause, i.e., a clause containing only a subset of the negation of the literals in the assignment. This clause records the failing (partial) assignment, and the stronger it is, the more total assignments it discards. Furthermore, it would be nice if such a clause would prevent, through propagation, to reach the same state again. In this context also, a stronger clause is always better: it can be propagating earlier, and may cut out a larger part of the search space.

One admissible stronger clause would contain only the negations of decision literals, i.e., only the (negation of) literals in the assignment that have not been propagated but guessed. All the propagated literals are indeed a consequence by propagation of the decision literals. So if any of those propagated literal is false in a model of the formula, at least one of those decision literal should be false. The clause containing only the negations of decision literals in a conflicting partial assignment is a logical consequence of the original formula. Notice also that this clause is propagating in the expected way: it would prevent the decisions that led to the previous conflict to occur again. It would propagate the variable decided last, with the opposite polarity, cutting

off the search space that led to the previous conflict.

Among all the decision literals in the assignment stack in a conflict state, not all decision literals actually contributed to the conflict. To analyze this, it suffices to recall for each propagated literal the clauses that forced this propagation. One may then draw a directed implication graph, where each node is a literal, and its parents are the literals that implied the propagation. A node and all its parents thus represent a propagating clause, the parents corresponding to the negated literals in the clause. The directed graph is acyclic, since no propagated literal is indirectly responsible for its own value. All leafs (i.e., nodes without parents) in this graph are decision literals, since all propagated literals have parents. In a conflicting state, there exists a variable such that both polarities are present in the graph. For convenience, consider that there is a special conflict node whose unique parents are those two literals on the same variable. The interesting part of the implication graph is the conflict graph, i.e., the subgraph rooted by this conflict node. The full graph may have disconnected subgraphs, and many leafs without a directed path to the conflict node. Any decision literal that does not occur in the conflict graph did not take part in the chain of propagations that eventually led to the conflict, and can therefore be removed from the conflict clause.

It occurs that using a conflict clause containing only (the negation of) decision literals is often not the best choice. There indeed exists some freedom. Instead of a (decision) literal, one can for instance select all its consequences. Another legitimate conflict subgraph can be obtained by eliminating recursively from the conflict graph all parents of some nodes. Then, the leafs of this subgraph are a subset of the partial assignment that generate the conflict. It might then contain both decision and propagated literals. In practice, the first unique implication point (FUIP) [92] technique gives good results. Starting from the conflicting pair of literals, a growing conflict subgraph is built by repeatedly adding the parents of the literal propagated the latest, until there is only one leaf of the subgraph that is a descendant of the last decision. This leaf can be the decision itself, but it can also be a consequence of it that is sufficient (with all other decisions) to cause the conflict. It is called the first unique implication point. Using the FUIP technique is in practice better than collecting all relevant decision literals. Although there is no absolute theoretical reasons to prefer the FUIP, the rationale is that the unique implication point is in some sense the real reason of the conflict; it appears nearer to the conflict than the last decision literal, and continuing collecting reasons of the FUIP might augment the conflict clause with many more literals.

The above techniques keep only one literal at the last decision level, that is, the last decided literal or one literal propagated after this last decision. At previous decision levels however some literals in the conflict clause may imply some others. The conflict graph can also easily be used to remove such implied literals and thus refine the conflict clause to a stronger one [118]. In practice, this leads to significant speed-ups.

To refer again to truth tables, conflict analysis is a clever way to not try several times assignments that will fail to provide a model for the same reason, or in other words to cache reasons for failing assignments. The only requirement to the added clause is that it should be a consequence of the original set of clauses. In the case of FUIP conflict analysis, the learned clause is indeed a consequence, by resolution, of all the clauses at play in the conflict. As a side note, we can here notice that it is trivial for CDCL solvers to output resolution proofs of unsatisfiability.

An important question is whether CDCL solvers are able to find short proofs, if they exist. Truth tables cannot, of course, because an exponential number of interpretations with respect to the number of propositional variables have to be checked systematically, whatever the formula is. CDCL however, and mainly thanks to clause learning, is as powerful as general resolution [19]: with the right choices, it can find a short proof if there is a short resolution DAG-like proof.

2.1.3 Non-chronological backtracking

If a conflict is reached while building a partial assignment by deciding literals and using propagation, a conflict clause is learned. This clause can, as explained above, be computed using the FUIP technique, in which case it contains only one literal at the last decision level, i.e., either the (negation of the) last decided literal or one literal propagated after this last decision. Actually, having just one literal at the last decision level is a required feature of the conflict clause generation. Remember that, in the truth table algorithm, failed assignments are stored to avoid to check them several time; the conflict clause plays the same role, but with a bonus. There is no need to check if the conflict clause is invalidated. Thanks to propagation, this clause will prevent the same literals to be assigned again, since as soon as all but one literals involved in the previous conflict are assigned to true, the last becomes assigned to false by propagation. The core of the naive truth table algorithm is an iteration on assignments. In CDCL, conflict analysis and propagation together with

backtracking play the role of this iteration loop.

Every time a conflict happens, the conflict clause is learned, and the algorithm backtracks the assignment to the earliest decision level where the conflict clause is propagating. Another possibility would be to only backtrack the last and conflicting decision level and propagate from this point on. Besides experimental evidence, there are arguments however in favor of backtracking to the propagation level of the conflict clause. First, since decisions are to some extent arbitrary, it is preferred to assert every implied literal as soon as possible, soon meaning early in the assignment stack; the next conflict might then be more general, not influenced by the dispensable decision levels. Second, the code to support propagation without backtracking to the propagation level would be more complex, and might induce overhead to the algorithm: the watched literals technique is only applicable for the immediate propagation (when the penultimate literal of a clause becomes false), and is not suitable to remember the literals that have been propagated late.

As a final remark on backtracking, notice that, because the learned clause was computed using the FUIP and not the last decision literal, it may happen that the exact same decisions occur again, with the same order (but the stack would differ on the propagated literals). Some early solvers learned all the unique implication points clauses. It was later noticed that just learning the FUIP clause was experimentally better. It may be useful to reevaluate this in the recent contexts though.

2.1.4 Heuristics

Boolean constraint propagation, conflict analysis, learning and non-chronological backtracking are the core of the CDCL algorithm. Some freedom still exists in the implementation of the algorithm, and in particular, in the procedure that decides which Boolean variable to assign first and which polarity to pick. Remember that, for a satisfiable formula, a perfect decision heuristic would pick literals in such a way that the solver builds a full assignment without conflicting and backtracking, that is, in linear time. This is obviously a hard problem, and also due to the efficient data-structures and in particular the watched literal, the amount of knowledge on which to base the decision is limited. For instance, it could make sense to assign to true the literal occurring most often in non-satisfied clauses, but the solver data-structures do not provide this information at low cost. Modern SAT solvers rely on decision heuristics finely tuned to their inner working. A popular heuristic is

the Variable State Independent Decaying Sum (VSIDS) [95], which picks the most active variable first, a very active variable being one involved often in the recent conflicts. As a result, the VSIDS heuristic focuses sequentially on groups of tightly coupled variables.

When a variable is picked as a decision, it can be assigned either false or true. It happens that it is better to remember which polarity it has been last assigned, and picking the same polarity for the decision. This process, called phase caching [103] ensures the solver focuses on one part of the search rather than randomly exploring the search space. Assigning a variable by decision to a new polarity might indeed force the search in a completely different area of the search space. Again as it is the case for many techniques in SAT solving, this explanation comes from experimental observation rather than as a property of the algorithm.

If it is often good for a solver to focus on a part of the search space, and exhaust this part before examining another, this might also have drawbacks, and in particular, the solver might be stubborn at solving less relevant parts of the problem. Restarting (see e.g., [21, 80] for discussions on restart) the solver is beneficial in practice to get out of a corner, and is used quite a lot in modern tools. This is less fundamental for SMT instances than for hard SAT instances, since the SAT abstraction of SMT problems is most often very easy compared to the SAT problems usually given to solvers.

During the search, a solver generates many clauses, and many of them are used at most a couple of times. Learned clauses however slow down the solver, mostly since they have to be handled in the Boolean constraint propagation algorithm. It thus makes sense to periodically eliminate clauses, and modern solvers typically eliminate most of the learned clauses. Notice however that these might, depending on the strategy, break the completeness of the solver. Useful clauses might also be forgotten.

State-of-the-art solvers also use many techniques to simplify the input problem (preprocessing) [57], or even to simplify the problem within the search (inprocessing) [83]. Many of those techniques are specific to purely Boolean problems and do not apply to SMT. Others are simply not useful for Boolean abstractions of SMT problems. For instance, a lot of effort has been invested to exploit symmetries in Boolean problems [114]. Our symmetry preprocessing technique [50] (see selection of papers on page 65) tailored for SMT is however much easier to express directly at the SMT level rather than relying on this effort for Boolean symmetry breaking.

2.1.5 Drawbacks of modern SAT solving

If a variable appears only positively (or negatively) in a formula, a satisfying assignment can always be modified to assign this variable to true (resp. false). This simple observation leads to the pure literal rule: if an unassigned variable appears pure — i.e., always with the same polarity — in the unsatisfied clauses, it can be asserted with the polarity making all those clauses true. This is not a proper propagation, since it does not necessarily imply that there is no model with the variable having the opposite polarity, but setting the variable with this polarity cannot hurt. Modern SAT solvers do not use the pure literal rule since the optimized CDCL data-structure does not provide the facilities to quickly detect pure literals (under a given partial assignment). Anyway, the pure literal rule does not hold for Boolean abstractions of SMT formulas — a literal might appear pure although it would appear with opposite polarity in theory clauses — so in our context this is a lesser evil. The pure literal rule is however an example of a good technique that could not find its place in CDCL solvers.

The Boolean constraint propagation is triggered by watched literals. An alternative, less efficient, implementation could count, for each clause, the number of positive and negative literals, and could also take note for each unassigned literal, of how many clauses it could contribute to satisfy if assigned. It would trigger propagation when the number of negative literals in a clause with n literals is $n - 1$, and could implement a counter for satisfied clauses with a marginal cost. The CDCL data-structures in the algorithm do not allow to easily detect when all clauses are satisfied. There is furthermore no way to know if a decision actually helps satisfying clauses. As a consequence, some and even many literals in the stack of a satisfying assignment for a CNF might be unnecessary. Finding good partial satisfying assignments for formulas is important in the context of SMT solving in presence of hard theories or for quantifier handling; for instance, a decision procedure that is given a full assignment will have to deal with many literals irrelevant for satisfiability. We have contributed to an efficient algorithm to compute a minimal implicant (a "best" partial assignment) from a full assignment generated by a SAT solver [47].

2.2 Solving the SMT problem

In the context of Satisfiability Modulo Theories we consider formulas in more expressive languages than SAT. They may contain functions and predicates.

Variables are not only propositional but may take their value in a domain, either partially or fully specified by the theory, or left uninterpreted. As an example, the UFLIA language (Uninterpreted Functions and Linear Integer Arithmetic) of the SMT-LIB includes the usual Boolean connectives, equality, uninterpreted functions and predicates of any arity (including nullary: constants and propositions), the arithmetic comparison operators, and linear expressions on integers; the sort of each argument of each predicate and function symbol, and of the range of each function, can either be integer or one of finitely many uninterpreted sorts. Typical theories handled by SMT solvers include linear (or non-linear) arithmetic on reals and integers, uninterpreted symbols, arrays, bitvectors, abstract data-types and strings. Some SMT solvers can also reason on quantified formulas, but these quantifiers are currently only first-order.

Historically, research on practical SMT started after satisfiability checking for propositional logic, but quite some time before CDCL. For instance, one could arguably set the birth of modern age SMT to the design of the Stanford Pascal Verifier [96], the Stanford Temporal Prover STeP [23] used internally some SMT techniques, and the SVC solver [18] is an ancestor of the current CVC4 solver. The first SMT solvers using CDCL were adapting older SMT methods to suit CDCL, but they were not designed as extensions of CDCL for SMT. Only quite recently, with Model-Constructing Satisfiability Modulo Theories (MCSAT) [41], did the SMT community thoroughly try to extend the various phases (propagation, decision, learning) of CDCL to SMT. The success of CDCL is also starting to drive research on CDCL-inspired procedures for first-order logic [25].

In a propositional context with n variables, there are at any time at most $2n$ possible decisions and learning will produce a clause among 3^n possible ones since each variable may appear in a clause either positively, negatively, or not at all. If clauses are never forgotten, termination is a direct and trivial consequence of this finite space for the search tree. In an SMT context however, and for instance in presence of integer variables, there are an infinite number of possible decisions (e.g., a given variable may be given any integer value) or propagations (e.g., any bound on a variable might be propagated) and the learned facts can also belong to an infinite space since there are an infinite number of non-equivalent linear constraints that can be part of a learned clause. MCSAT [41] restores completeness with an explicit side condition that learning should produce clauses on a finite and fixed set of atomic formulas. Nonetheless, the freedom given by MCSAT is extremely large, and the success

of current MCSAT-based solvers (see e.g., [84, 85, 76, 125]) relies also on carefully designed heuristics. There is probably still a lot to understand how to optimally convert this freedom into a deterministic framework with a small search space. Classic SMT simply imposes the set of literals to be the obviously finite set of atoms occurring in the input formula. Only those literals can be propagated, and decisions are essentially similar to propositional ones.

Another orthogonal approach to SMT stems not from SAT but from first-order theorem proving (FOL). Driven by the success of SMT, there are several ongoing initiatives [3, 107] to combine theory reasoning within the superposition calculus. Again, for historical reasons, I have not worked on this trend, but the future will hopefully show convergence of both approaches; superposition is clearly the way to handle some types of quantifiers.

2.2.1 From SAT to SMT

Letting aside the urge to generalize CDCL to satisfiability modulo theories, a pragmatic approach is quite simple. Remember that Boolean propagation, early conflict detection, and precise conflict analysis are useful aspects for efficiency, but not mandatory for completeness. Applying them only leniently does not break completeness, but breaks down the behavior of the algorithm closer to the naive truth table approach. With this in mind, one can always view an SMT problem as a propositional problem with a hidden set of theory clauses specifying the relation between the atoms. The CDCL algorithm considers atoms as simple Boolean variables, and propagations only occur at the Boolean level, thanks to the clauses from the input problem, but without contribution from the hidden set of theory clauses. Soundness is recovered simply by discarding each set of literals computed as a model by the CDCL algorithm if it is not satisfiable according to the theory. The major advantage of this approach is that the underlying CDCL algorithm takes care of the Boolean structure of the formula, whereas the decision procedure for the theory only has to deal with conjunctive sets of literals. Refuting the tentative CDCL models is again done by adding new clauses to the original problem, these clauses containing only atoms from the input formula, thus guaranteeing termination. The original formula is abstracted down to a propositional one handled by the CDCL algorithm, and successively refined by the decision procedure that provide the missing clauses; the refinement loop somehow progressively reveals the theory clauses lacking from the original formula, and explaining the meaning of the atoms to the CDCL algorithm.

Unfortunately this approach shares several drawbacks that were already pointed out for truth tables. Indeed, the propositional abstraction is most of the time under-constrained: the CDCL algorithm would enumerate a large number of models, and each of them would have to be refuted one by one, pretty much like every interpretation is examined separately in a truth table. A mandatory improvement is for the theory decision procedure to provide, in case of conflict, a strong theory clause, i.e., a clause that does not contain the negation of all literals in the model of the propositional abstraction, but only of a small, as minimal as possible, subset of those literals. For each variable that is not mentioned in a stronger clause, the amount of models the clause can refute is doubled compared to a clause that would spuriously contain a literal on this variable; there is theoretically and experimentally a potential exponential gain.

Other improvements might be useful: the theory decision procedure might declare unsatisfiability of a partial model while it is built by the CDCL algorithm, without waiting for a full model of the propositional abstraction. Obviously, this is only practical if the decision procedure is efficient and will not significantly slow down the underlying CDCL algorithm. The procedure should furthermore be capable of incrementally checking the satisfiability of a set of literals: the amount of computation required to check the satisfiability of a set of literals, and then the same set augmented by a few new literals, is not exceedingly larger than for the last set alone. Also, the procedure should be backtrackable: once a set augmented by a few literals has been checked for satisfiability, only minimal work should be needed for the decision procedure to come back to the state after checking the original set of literals. These features are necessary since the literal stack of the CDCL algorithm will be checked many times for satisfiability by the theory decision procedure, with very minor differences between the successive checks.

Checking the consistency of the literal stack on-the-fly gives to the CDCL algorithm a more immediate knowledge of the hidden, virtual set of theory clauses. However, it is not yet as if those clauses were fully known to CDCL, since no propagation occurs. A decision procedure can feature theory propagation: while being aware of all the atoms in the input formula, it can assert their value in the CDCL stack as soon as it is implied by other literals in the stack. Even for simple decision procedures, this feature might come at a cost: while congruence closure is quasi-linear, congruence closure with full propagation is hard. The current trend in SMT is to do theory propagation with a best effort policy: decision procedures only do easy propagations.

Some theories however are better handled with a tighter coupling with propositional reasoning. Indeed, some data-structures are inherently Boolean. This includes bit-vectors (and machine integers) and floats. The difficulty then resides in designing the appropriate techniques to convert the problem into a Boolean one, e.g., using abstraction and lazy translation to SAT [124, 126].

2.3 Theories of interest

The SMT theory should answer a need for expressivity, have a decidable ground problem, be tractable and have a sufficiently efficient decision procedure for the purpose. The SMT framework ensures that the decision procedure only has to deal with conjunctions of literals.

One of the simplest and most useful theories to consider is the theory of equality and uninterpreted symbols (functions and predicates). It is handled in quasi-linear time by the congruence closure algorithm [98]. Despite its simplicity, it is useful in itself to model in an abstract way many aspects of computer systems. It is also at the core of decision procedures for more concrete data-structures, like arrays [119], or inductive data-types and co-datatypes [109].

Another obviously interesting set of theories are the arithmetic ones: there are indeed various theories differing by their expressivity and complexity, from difference logic [89] which has very efficient decision procedures but is quite inexpressive, to non-linear arithmetic on integers (a.k.a. Hilbert 10th problem) which is undecidable even for conjunctions of ground literals [93]. In between, a sweet spot is linear arithmetic on mixed integers and reals (or equivalently, rationals). The decision procedure for conjunctive sets of constraints is polynomial for reals and NP-complete for integers (see e.g., [24]), but current decision procedures are efficient for practical cases. Even in the real case, one relies on the simplex method, which is exponential [56, 24] in the worst case; in practice, simplex-based algorithms are more efficient than other algorithms that are optimal in the worst case [24].

Despite important applications — for instance, the B method [2] and the TLA+ [90] language use some set theory as their underlying logic — SMT solving on set theory was, at the time of starting writing these lines, surprisingly not very well supported in SMT besides some first results [11] and the existence of a preliminary standard [88]. Solvers for set theory based verification platforms (e.g., the prover for Atelier B) use mainly rewriting based heuristic techniques, that can provide good results even for quite evolved proof obli-

gations, but are even less predictable than the classical SMT approach and do not provide any guarantee of completeness. This is quite similar to the automation provided within some proof assistants, e.g., within Isabelle [102], but quite in opposition to the bottom-up SMT approach to add expressivity to SAT solving while maintaining decidability. A set theory in the SMT philosophy would be so restricted in expressivity that it would be of little help in itself. I believe the right approach is, just like for higher-order logic, to translate problems into SMT and progressively extend SMT to natively handle more and more useful constructions to deal with sets. In a sense, this is the line of the very promising approach in [11].

An essential aspect of SMT is that the theory reasoner can actually deal with a combination of theories. It is then possible to consider formulas containing, e.g., both uninterpreted symbols and arithmetic operators. Combination frameworks ensure that a decision procedure for the union of several theories can be built on the decision procedures for the component theories. On the one hand, these combination frameworks provide decidability results for combinations, and on the other, they also provide a modular way to actually build the decision procedures.

2.3.1 Uninterpreted symbols

The theory of equality, also known in the context of SMT as the theory of uninterpreted symbols, has one of the most mature decision procedures for SMT solving: it is efficient and has all the nice features for an optimal integration into the SMT framework.

In a nutshell, a union-find data-structure maintains a partition of terms into congruence classes according to asserted equalities. Possible inconsistencies with asserted disequalities are detected on the fly. A new asserted equality might result in a merge of two partition, if the terms asserted equal were not already in the same partition. The congruence closure algorithm also detects congruence, i.e., it ensures that two terms with the same top symbol and pairwise equal arguments are in the same partition. This is done efficiently by storing in a hash table the signature — the top symbol applied to congruence classes of the arguments — corresponding to each term. Notice that two terms with the same signature should belong to the same congruence class. This hash table can thus serve to detect congruence. Every time the congruence class of a term changes (because of some merge), the signature of its predecessors (the terms having that term as a direct argument) has to be updated. It may

happen that the updated signature conflicts with one that is already present in the hash table. If so, the two terms with the same signature should be put in the same class, if they aren't already. Conflict detection with the asserted disequalities is simply done by investigating, each time the class of a term is updated, the disequalities in which it takes part.

The algorithm directly inherits from union-find data-structures its $m + n \log n$ complexity for m operations on a partition of n objects assuming constant time operations on the hash table [54], thanks also to the fact that congruence and conflict detection can be implemented efficiently. Backtracking can also be done quite efficiently either using persistent data-structures that are designed to be backtrackable, or carefully engineering the algorithm to store in a stack-like manner the necessary information to undo each change.

To produce proofs and small conflicts, it is sufficient to maintain aside a congruence graph. Two classes are merged either because of an asserted equality or because of a congruence between two terms. For each merge, an edge between those two terms is added to the graph, and is labeled by the reason for the merge, that is, either by the asserted equality or by a placeholder standing for congruence. The congruence graph has one and only one path between two terms in the same class, and the equality of those terms is implied, by transitivity (and symmetry and reflexivity) of the equality, by the asserted equalities in the labels and by the equalities between directly connected congruent terms. The congruences themselves are a consequence (by congruence of the equality relation) of the fact that arguments are pairwise equal. From this congruence graph, it is not difficult to build a resolution proof involving only the asserted equalities and instances of the axioms (schemas) of symmetry, transitivity, reflexivity and congruence of equality. Collecting the involved asserted equalities, together with a conflicting disequality, provides a small conflict, although not the smallest one, and not even minimal. Providing the smallest conflict is NP-complete, and providing a minimal one is polynomial, but not linear [58, 59]. In practice, producing minimal conflicts rather than the (possibly non-minimal) small ones stemming from the congruence graph does not pay off the computing overhead with today's algorithms.

One last important aspect is theory propagation. Again, congruence closure can handle it very easily. Much like conflict detection, when a relevant equality becomes true because both members are put in a same class, the equality is propagated; it is also easy to propagate the disequalities congruent to some asserted disequality. However, we believe full disequality propagation

is a hard problem.[1]

2.3.2 Arithmetic

Although there are many results worth mentioning about arithmetic in SMT if one wants to be complete, it is not easy to present a mature description of an arithmetic decision procedure to embed in SMT. State-of-the-art SMT solvers use a simplex-based algorithm, often a variant of [55, 56], to handle linear arithmetic constraints occurring in formulas. It differs from text book simplex algorithms on these points: it is not an optimization algorithm but only finds a satisfiable assignment or concludes unsatisfiability, variable values do not necessarily need to be positive, the algorithm is essentially incremental and backtrackable. In a nutshell, an assignment for variables is maintained, and repaired to take into account every additional constraint incrementally. In contrast to usual linear programming software, it is generally implemented in exact rational arithmetic, and relies on a number of techniques to accommodate integer variables. Probably the best resource to understand the battery of methods useful to tackle linear arithmetic on mixed real and integer constraints as found in SMT is the description of the MathSAT procedure [75]. However, I believe there is still place for a lot of improvements. Although current algorithms are not really the bottleneck for practical use of SMT, mixed integer linear programming algorithms, e.g., Gurobi [100], are impressively more efficient than SMT solvers to solve conjunctions of linear arithmetic constraints. In fact, it greatly pays off to integrate a linear programming system into the linear arithmetic decision procedure [86]. One can argue that linear programming systems use machine arithmetic, and this accounts for the difference in efficiency. My intuition is that SMT algorithms are simply naive compared to the state-of-the-art in linear programming. The MCSAT approach for linear arithmetic [84] also exhibits interesting results; it would be of the highest interest to better understand its relation to the simplex algorithm as implemented in SMT.

Beyond linear arithmetic, and even less mature, are the procedures to handle non-linear arithmetic constraints and transcendental functions. We are actively contributing to this line of research, so we delay giving further information until the second part of this document.

[1] At the time of writing this lines, we believe the complexity of full disequality propagation is still an open problem.

2.3.3 Combining theories

Because of their applications, the input language of SMT solvers naturally often involves several kinds of symbols, related to different objects or theories, e.g., lists, arrays, or absolutely free data-structures, arithmetic constraints, and uninterpreted symbols. While it is possible to design a decision procedure for each of those theories separately, the only conceivable way to design a procedure to deal with sets of literals — remember that the SMT infrastructure allows the procedures to only reason on conjunctive sets of literals — mixing all theories is to proceed in a modular way. Now, given a set of literals mixing several disjoint theories, it is always possible to build an equisatisfiable set such that every literal is either totally or not at all relevant for each theory. Considering uninterpreted symbols and arithmetic, an occurrence of $p(x+1)$ can be replaced by the two literals $p(y)$ and $y = x+1$ while preserving satisfiability (p is an uninterpreted predicate, and y is a fresh symbol of appropriate type). This process is called purification. After purification, one can use the decision procedure for each theory on only the relevant literals. This simple procedure might suffice to detect an unsatisfiable set, but there are cases where it does not. It may indeed happen that there is a model for each theory of the combination, but there is no model for the combination of theories. This simply means it is not possible to build a model for the combination of the theories from the models for each theory. These models either disagree on the domains (they have different cardinalities) or on the interpretation for the shared symbols (and in particular equality).

In order to have a working combination of decision procedures, it is necessary to ensure that, if all decision procedures in the combination find a model, there exists a model also for the combination. This requires the decision procedures to agree both on the domains (i.e., their cardinalities) and on the interpretation of shared symbols. The original Nelson-Oppen combination scheme [97, 112, 120] considers disjoint theories: only equality and uninterpreted nullary symbols (called uninterpreted constants, variables, or parameters in the literature) are shared. It furthermore assumes that the theories are stably-infinite, which means that every satisfiable set of literals has a countable infinite model. In this context, it suffices that the decision procedures agree on an arrangement of the shared uninterpreted nullary symbols with respect to equality, e.g., by non-deterministically choosing a priori a partition of these symbols in equivalence classes. Thanks to the stably-infiniteness property, models found by the decision procedures can be made to agree on cardinalities, and on the interpretation of equality.

A priori agreeing on an arrangement, that is, a partition of the shared uninterpreted nullary symbols, is quite an inefficient way to ensure that decision procedures will have matching interpretations of equality. In practice, the decision procedure rather exchange information, and for instance, they propagate entailed equalities between shared terms to each other. For completeness though, and even in the case of simple combinations of disjoint stably-infinite theories, propagating only equalities is not sufficient. Consider for instance a combination including arithmetic and another theory, and five integer terms all between 0 and 3. Inevitably two of them are equal. Assume that the constraints in the other theory enforce those terms to all be distinct. No equality can be shared, but the combination is unsatisfiable. It is also necessary to propagate entailed disjunctions of equalities between shared terms; if such disjunctions are propagated, it is easy to prove that this propagation of information is equivalent to agreeing on a partition of the shared terms. So, for convex theories — theories such that, if a disjunction is entailed, one of the disjunct is entailed — propagating equalities is a complete way to agree on an arrangement. The empty theory (uninterpreted symbols with equality), and linear arithmetic on rationals can be considered as convex. For non-convex theories, it is very impractical and inefficient to detect and propagate entailed disjunctions of equalities. Part of the answer is provided by delayed theory combination [29], where disjunctions are handled by the underlying SAT solver. In the same philosophy, model-based combination [44, 45] exploits models provided by the individual decision procedures to enable the combination. These models are used to assert which shared terms are equal and which are not. Disagreement on these equalities leads to conflicts, but if an arrangement exists that would satisfy all theories, it will eventually be found after some iterations with the SAT solver.

If theories in a combination are not disjoint, because they share other symbols than equality and nullary uninterpreted symbols, the procedure also has to ensure that those shared symbols are interpreted by all theories in a compatible way. Local theory extensions [117] is a successful and practical framework for some kind of non-disjoint theories. Alternatively, disjoint theories can be combined by reduction to a decidable language [123]. Concretely however, non-disjoint combinations are not yet as commonly implemented as the classical Nelson-Oppen framework for stably-infinite theories. Some of those combinations have a high complexity, are somehow specific, or do not yet have the practical techniques and heuristics to make them work in practice. We will come back to this in the contribution, advances and perspectives part of this document.

2.3.4 Quantifiers

Solving the propositional satisfiability problem, checking the satisfiability of sets of literals with uninterpreted symbols, and combining disjoint theories are I believe mature aspects of SMT. Linear arithmetic decision procedures are satisfactory. Arithmetic beyond the linear case is still in its infancy. From our point of view, two major aspects that still require a lot of work and insight are non-disjoint combinations and, last but not least, handling quantifiers. SMT beyond first-order logic is mainly unexplored and central to our research project.

The Herbrand theorem (see [60] for a many-sorted version with equality) provides a tool to build a complete procedure for first-order logic formulas on the basis of a decision procedure for ground formulas. Indeed it suffices to instantiate an unsatisfiable formula sufficiently many times to get an unsatisfiable ground logical consequence, and a finite number of instances is always enough. The difficulty lies in the fact that, although the amount of necessary instances is finite, they have to be picked within an infinite though enumerable set. Since the set is enumerable, and assuming the ground problem is decidable — as it is for pure first-order logic with equality — a trivial complete procedure is to fairly enumerate instances and check the ground formulas so generated for unsatisfiability. This process will either eventually terminate on an unsatisfiable ground formula, it will terminate for lack of instances in the case of a finite Herbrand domain, or it will run for ever if the original formula is satisfiable. At the dawn of automated reasoning, this has been tried, quickly considered a dead end, and was abandoned in favor of the more promising approach of resolution and superposition theorem proving [10, 99]. Modern SAT and SMT solvers are now changing the landscape. The instantiation based SMT solver CVC4 is competitive in the annual competition of first-order provers, and even strong supporters of superposition are reconsidering instantiation [108].

Instantiation techniques in SMT often rely on the information provided by the satisfiability check of the ground abstraction of the formula, that is, the formula where all quantified subformulas are abstracted as propositional variables. This information can be a model, or simply a satisfying assignment for the ground atoms. Modern SMT solvers use a portfolio of instantiation techniques. Besides simply enumerating instances — which has its advantages, if done properly; we will come back to this in the perspective part of this document — the oldest technique is probably pattern-based E-matching [96, 51, 42]. Quantified formula are associated with patterns, that is, sets of terms

in the quantified formula containing all the quantified variables. Heuristics are designed to find out the effective patterns that will be efficient at finding the right instances of the formulas. Given a satisfying assignment for the ground formula, pattern-based E-matching simply tries to find out terms explicitly mentioned in this assignment which match the patterns, modulo the equalities in the satisfying assignment. This technique is thus perfectly suited to unfold axiomatized definitions.

Model-based quantifier instantiation [74] relies on a full model for the ground abstraction of the formula. The quantified formulas are evaluated in this full model, and all instances falsified in the model are added to the ground formula. It should be noted however that generating a full model also introduces quite a lot of arbitrary choices in this model, and thus also introduces randomness in the instantiation generation. As a consequence, many useless instances might actually be added to the ground formula. For instance, a partial model for a formula might leave a certain unary function totally unassigned. Any full model would though provide an assignment for this function, and this assignment would have to be iteratively repaired, by the instantiation generation process, until it agrees with all quantified formulas mentioning the function.

Conflicting instances is a technique that is not sufficient in itself, but that never generate useless instances [110, 13]. It tries to find, for each formula, an instance that will be sufficient to refute the current satisfying assignment for the ground atoms. Of course, this fails for most quantified formulas, but any instance generated this way is useful. In contrast to the previous two methods, this will not overwhelm the ground SMT solver with many instances.

3 Contributions, advances and perspectives

In the above, we gave a brief description of the state-of-the-art in SAT and SMT solving. My previous work as well as my current research are mainly focused on advancing this state-of-the-art. Many of my contributions are related to decision procedures and combinations, that is, procedures to decide the satisfiability of sets of literals with interpreted symbols, and ways to combine them into procedures to decide the satisfiability of sets of literals containing symbols from a mixture of decidable fragments. Section 3.1 gives an account of the previous results and planned work in this direction. Although the predilection domain of SMT is decidable fragments, SMT techniques are increasingly extended to also tackle much richer languages, beyond decidable

ones. In particular, I have been active in improving quantifier reasoning in SMT (Section 3.3), and a major part of my time in the near future will be devoted to bringing higher-order reasoning capabilities to SMT (Section 3.4). This includes studying the convergence of SMT with other kinds of approaches, notably superposition theorem provers. Computer algebra systems are yet another kind of automated reasoning engine; I have advocated in the past for more synergies between these systems and SMT, and my project includes pursuing this line of research, with the aim to boost arithmetic handling within SMT (Section 3.2). Improving the use of SMT solving not only requires augmenting its expressivity and efficiency, it is equally mandatory to facilitate embeddability of solvers in larger frameworks. Two issues can be pointed out. First, there should be languages and tools to help writing the input problems: this is one of the main goals of the SMT-LIB initiative, to which I contribute as one of the three current SMT-LIB managers. Second, solvers should provide a precise account of their results, i.e., by outputting either models or proofs. I have been very active already in promoting proofs for SMT solvers. The issue is however non trivial and will continue to require work in the future. Since this strongly relates also to the use of SMT in HOL platforms, both proofs and SMT for HOL are discussed together in a unique section (Section 3.4). Section 3.5 mentions the more applicative aspects of SMT to which I contributed.

3.1 Decision procedures and combination schemes

Uninterpreted symbols

One of the simplest decision procedure used in SMT decides the satisfiability of conjunctions of literals with only uninterpreted symbols and equality, a.k.a. the empty theory. In one of my very early works [63], we studied how to actually use this decision procedure in an SMT context — at that time, we were using BDDs as underlying procedure for propositional logic instead of CDCL. A key aspect for the success of using a decision procedure in an SMT framework is to be able to extract a small conflict set, i.e., a subset of an unsatisfiable input set of literals which is itself unsatisfiable. The smaller this conflict set is, the stronger the conflict clause is, and the overall convergence towards a model or unsatisfiability is quicker. Computing the smallest conflict set for sets of literals in the empty theory was long believed to be hard, although there was no proof of this. We recently proved it is indeed NP-complete [58, 59], by reduction of the SAT problem; this result is one of the selected papers (see

page 65). In practice, we use heuristics to compute small conflict sets. One might be tempted to hope for a better algorithm, but this result states that a perfect technique would anyway be computationally expensive.

Combining theories

Building decision procedures for expressive languages sometimes naturally leads to combining simpler language together. The classical combination framework for theories is due to Nelson and Oppen [97] and better understood later by Ringeissen [112] and Tinelli [120]. It assumes that the theories in the combination are decidable, disjoint — the only shared function or predicate symbols are the equality predicate and uninterpreted constants, i.e., nullary functions, often called variables in the context of theory combinations — and stably infinite — every satisfiable set of literals is satisfiable in an infinite model. Since my early works, I have contributed to better understand combination frameworks and lift the requirements for combination of theories to work. In an early work [65], I have considered non-stably infinite theories, expressed as a set of interpretations, i.e., not necessarily as a set of axioms. More recently [69], I studied how to build combination frameworks that preserve refutational completeness when combining disjoint but not necessarily decidable theories. I contributed also to a better understanding of Nelson-Oppen in practice, using (model-)equality propagation [45, 46]. Indeed, the Nelson-Oppen framework generally involves exchanging implied disjunctions of equalities, which is not practical. This model-equality propagation framework is quite similar in essence to the better known model-based theory combination approach [44].

Combining first-order fragments

The properties required from the theories in a combination are used to ensure completeness of the combination. The initial set of literals to consider in the union of theories is split in several sets, each considered in one theory. If every of these split sets is satisfiable in its theory, one has to guarantee that there is a model in the union of the theories. For disjoint theories, it is essentially necessary to ensure that cardinalities of the models for the splits sets are reconcilable; it is trivially ensured by the stably infinite property for instance. If one want to avoid this property, some other properties have to be found.

Among decidable theories, it is quite natural to consider the first-order decidable fragments. Most of them are not stably infinite. I have contributed

to show however that they have nice properties that compensate the lack of stable infiniteness. I have considered the Ackerman and guarded fragments in [4], the Löwenheim class (monadic first-order logic with equality, but without functions), the Bernays-Schönfinkel-Ramsey class, and the two-variable fragment of first-order logic in [61, 62].

It is much more difficult to drop the disjointness requirement for combining theories. However, I have showed that a few interesting cases, namely the Löwenheim and the Bernays-Schönfinkel-Ramsey classes, satisfy an appropriate property that enables non-disjoint combination, sharing unary predicates [32] (see selected paper on page 65). It is still an open problem to consider also other first-order decidable classes (e.g., the Ackerman class, the guarded fragment, and the two-variable fragment of first-order logic) in this non-disjoint framework.

I have contributed to other non-disjoint combinations of theories, but theses combination rather relate to data structures, discussed just below.

Combining data structures

From the verification point of view, data structures are, with arithmetic, extremely important to consider for combinations of decidable theories. I have contributed in early works to show that the empty theory [65], the theory of arrays [65], and theories for lists [71] can be combined with non-stably infinite theories. These theories behave sufficiently well not to impose strong conditions like stable infiniteness to the other theories in the combination.

In this context also, non-disjoint combinations are of great interest. It is indeed often the case that data structures involve functions connecting (bridging) them to some arithmetic theory. Consider for instance lists with length, or trees with depth or size. We have contributed to extend combination frameworks for these bridging functions. This is a kind of non-disjoint combination, since the bridging functions are relevant to both theories. Our approach [33, 35, 34] is based on rewriting and reduction to disjoint combinations.

Project

The ultimate goal in combining theories is, from my point of view, to get a framework to check the satisfiability of formulas in a modular way. This framework should benefit from the properties of some parts of the formulas,

for instance, if some part of the formula belongs to a decidable, stable infinite or convex fragment. If parts of the formulas are loosely connected by only a few shared symbols, non-disjoint combination would then enable to reason on those parts separately and in parallel, the framework taking care of exchanging the necessary information between the reasoners. Providing decision procedures to combine decidable fragments should not be the only goal, because in many cases, restricting languages to decidable ones is too much of a constraint for expressivity. Aiming at combination frameworks that maintain refutational completeness, notably when combining arbitrary first-order theories and decidable fragments, is also a valuable goal.

To further extend the expressivity and usefulness of SMT solvers, it seems natural to consider (and combine within the SMT solver) more decision procedures for data structures. In particular, distributed algorithms maintain specific data structures (e.g., distributed token trees for mutual exclusion) and employ variations of common data-structures (e.g., queues). Studying each of these possible data structures separately, and hard-coding each of them in an SMT solver is unfeasible. One solution to tackle this variety is to express those specific data structures using lower-level data structures that are directly supported by the reasoning infrastructure, such as arrays. This approach is unfortunately not very well suited for formal verification, especially when using automated tools: replacing the high-level operations by combinations of operations on low-level data structures makes the proof search prohibitively expensive. We propose a different approach based on plugging user-defined theories describing those specific data structures into SMT solvers, thus enabling the tools to reason on the high-level objects directly rather than on the complex implementation using lower-level constructs.

Enabling SMT solvers to accept plug-in theories poses serious practical, but also and foremost theoretical problems. As previously mentioned already, SMT solvers are based on combination frameworks that require that component theories satisfy precise and rather constraining properties. In order to accept plug-in theories, one must either (1) relax those constraining properties, specify criteria for the theories to be combinable in the framework, and define decision procedures that verify those criteria (like we have proposed in [4, 61]) or (2) find new ways to combine arbitrary decision procedures in a sound, if incomplete way. In practice the (semi-)decision procedures for plug-in theories would be implemented using generic automatic theorem provers. Those theories are indeed relatively small sets of quantified formulas, but proofs may be intricate. Saturation first-order theorem provers will explore the theories and

assumptions exhaustively in order to find a refutation. They thus provide efficient and refutationally complete reasoning engines. Furthermore, it has been shown that saturation theorem provers can be used to efficiently implement decision procedures for several decidable first-order fragments [7, 9, 73, 81]. For certain theories modeling data structures, it even appears [7, 115] that generic first-order provers compete with reasoning modules written specifically to handle those data structures. In that context, a specific decision procedure for a data structure will only need to be implemented if it is worthwhile, that is, if this data-structure occurs frequently in the usual verification tasks, if reasoning on this data structure represents a bottleneck for checking proof obligations, and if a specific decision procedure would bring a significant improvement in complexity or practical efficiency. In this line, we already proposed [69] and mentioned earlier a sound and complete way to combine a refutationally complete procedure and a decision procedure, to build a refutationally complete procedure for the union of a decidable theory and a disjoint arbitrary (not necessarily decidable) theory represented by a finite set of first-order axioms. We are working to further develop this framework, and to apply it in practice.

3.2 Automated reasoning vs. symbolic computation

Reasoning in various fragments of arithmetic is a fundamental and ubiquitous requirement in verification. By *arithmetic* we here mean any theory whose language includes arithmetic constants and functions (0, 1, $+$, $\times$ but possibly also mod, $\div$, exponentiation, trigonometric functions, *etc.*), interpreted over integer, rational, real or complex numbers (or a mix of those). There are well-known fundamental results about the decidability (and then, complexity) and undecidability of fragments of arithmetic that impose strict limits on the capabilities of automatic tools. For many practical applications, it is however enough to handle restricted subsets of arithmetic.

An important direction of research to extend the expressivity of SMT solvers is the quest for better arithmetic reasoning modules. If linear arithmetic on both integers and reals is quite well integrated in state-of-the-art SMT solvers, there are signs that SMT solvers handle those constraints in far less efficient ways than dedicated tools: the CVC4 solver managed to outperform all others [86] by finding an appropriate way to interact with a modestly efficient linear programming tool, despite inappropriate interface (that is, non-incremental, non-backtrackable, non-exact arithmetic) of the linear programming tool. The situation is even worse for non-linear arithmetic. Some inno-

vative techniques implemented in Z3 [85] are quite efficient for SMT problems in the SMT-LIB, but non-linear capabilities of SMT solvers currently hardly compare to dedicated tools on their own benchmarks. Non-linear arithmetic is a research subject in its own, with considerable knowledge accumulated over decades. We believe the right way to tackle this problem is to work together with the community of computer algebra, to increase their awareness of our problems, and to come up with solutions that will not reinvent the wheel. Working together might also have as a side effect to better advertise to the computer algebra some techniques that are successful within SMT, e.g. [84, 85].

I have been very active in bringing the computer algebra and SMT communities together, to share knowledge and techniques. Notably, the SMArT ANR-DFG project I have been coordinating (2014-2017) was along these lines, I was an organizer of the Dagstuhl seminar 15471 on Symbolic Computation and Satisfiability Checking in November 2015, and I am a principal investigator of the SC^2 initiative and the Coordination and Support Action H2020-FETOPEN-2015-CSA 712689 [1].

As a result of the SMArT ANR-DFG project, the veriT solver integrates the computer algebra system Reduce with Redlog as a means to decide the satisfiability of polynomial constraints on the reals. For this cooperation to work, it has been necessary to adapt real quantifier elimination methods for conflict set computation [82]: it is based on tracing the reasons of unsatisfiability and to compute a small conflict set using the collected information. The results of this cooperation using techniques from [82] are already good. In a work in progress unpublished at the time of writing these lines, this is further greatly improved by using as a preprocessor a thin layer implementing interval constraint propagation, i.e. a technique that computes over-approximations of the possible values for each variable in the constraints as intervals. These quick over-approximations are exploited fruitfully by the decision procedures that work downstream. Among my previous contributions, we also extended arithmetic reasoning capabilities of SMT solvers by adapting techniques developed in the context of computer algebra, namely subtropical satisfiability [70]. This heuristic boils down, when checking the satisfiability of a set of constraints, to investigate the constraints only at the extremities of the domain space of the variables: this works surprisingly often to actually find satisfiable points.

Project

My main objective for this line of work is to design decision procedures for rich arithmetic inside SMT solving.

For the linear case, more effort should be put in filling the gap between SMT decision procedures and optimization tools. Optimization tools are indeed typically much faster than SMT to handle conjunctions of constraints. The techniques used there should be carefully analyzed, to evaluate if they can be adapted to an SMT context. On a heuristic side, efficient simple, well known, heuristic techniques that work well for linear arithmetic should also be tried to tackle mixed real-integer polynomial constraints. Of course, the problem is undecidable [93], but heuristics might be good enough in practice. There might even be criteria to characterize problems that can be solved using these techniques. Among the heuristics to investigate are branch and bound, finite domain analysis, and adapting the method in [31, 30] to the non-linear case.

Interval constraint propagation (ICP, see e.g., [72]) is already in use in the veriT solver, bringing significant efficiency improvements for satisfiability modulo polynomial constraints. There are other places where ICP could also lead to efficiency improvements in procedures occurring in computer algebra systems. In particular, we believe this should be investigated in the context of virtual substitution (see e.g., [87]), not only as a preprocessing phase as it is currently in the veriT solver, but with a tighter integration at each step of the procedure. In a similar spirit, decision procedures for polynomial constraints used in the SMT framework might benefit from a tighter integration with the embedded SAT solver. SAT solvers are indeed recognized as being a very efficient way to handle disjunctions and case splits. Some techniques in computer algebra, notably for quantifier elimination, do intensively use case splittings. We believe there is an opportunity to build on SAT solvers, and reuse CDCL techniques for those case splittings.

Quantifier handling is another aspect for which computer algebra techniques and SMT could be combined, to tackle problems currently outside of the scope of each. While SMT deals with quantifiers using trigger or enumerative instantiation, quantifier elimination is at the heart of many computer algebra techniques. It might be possible to use quantifier elimination as found in computer algebra tools, as a mean to deal with some quantifiers found in formulas given to SMT. Currently, most SMT solvers are incomplete when given quantified formulas with quantifiers ranging over real or integer variables.

Just like not everything is linear when considering arithmetic in, e.g., verification tasks, polynomial arithmetic is sometimes not sufficient either. For transcendental functions, a very interesting approach consists in linearizing them incrementally [36]. The approach could be investigated further, e.g., by rather translating into polynomial constraints of low degree, and then use other usual techniques to handle those polynomial constraints.

When integrated within an SMT framework, it is useful that decision procedure produce proofs, and ideally proofs are small and easy to check. We are also currently investigating proofs for arithmetic decision procedures. In the linear case, Farkas' Lemma (see e.g., [24]) is the basis for providing proofs that are easily checkable; the simplex algorithm — currently at the core of most SMT reasoners for linear arithmetic — readily provide such proofs. Decision procedures for polynomial constraints, e.g., virtual substitution and cylindrical algebraic decomposition, do not readily provide such nice certificates. The Hilbert's Nullstellensatz and Stengle's Positivstellensatz could be used to represent unsatisfiability certificates for sets of polynomial constraints respectively on the complex and on the real numbers. It is currently not clear how to extract Stellensätze for decision procedures. It is however possible to obtain a certificate, when using Gröbner bases as a decision procedure for polynomial constraints over complex numbers. This has for instance been used for Coq [105].

3.3 Quantifier reasoning

Proof obligations often contain quantified formulas, for instance to express that a certain property holds for every pair of processes taking part in the distributed algorithm, and these formulas are usually quite large. In contrast to the quantifiers in axiom sets describing plug-in theories, quantifiers that arise in this context should be better handled by Skolemization and ground instantiation. Indeed, those quantifiers usually do not require deep reasoning, but finding the few right instances can be challenging. Guiding the instantiation of the quantified formulas in SMT is currently done by heuristic rules that can account for very efficient quantifier elimination. This approach often works very nicely but sometimes also fails, notably if quantified formulas require instantiation of many variables at the same time. The main state-of-the-art instantiation techniques are E-matching based on triggers [42, 52, 113], finding conflicting instances [110] and model-based quantifier instantiation [74, 111].

The theoretical foundation for instantiation as it is done in SMT solving is

simply the Herbrand theorem (see e.g., [60]): a formula in pure first-order logic with equality is satisfiable if and only if the set of all its ground instances is also satisfiable. A recent contribution, yet unpublished when writing these lines, refines the classical Herbrand theorem in the context of SMT to a stronger version that requires considering less instances. E-matching based on triggers and model-based quantifier instantiation were designed to avoid the numerous instances generated by simply using Herbrand instantiation. When done carefully and with the use of this stronger Herbrand theorem, our experiments show however that simple enumerative instantiation is actually quite powerful.

Among our recent contributions related to quantifier handling, we showed that the major instantiation techniques can be cast in a unifying framework for handling quantified formulas with equality and uninterpreted functions [13]. This framework reduces instantiation to the problem of E-ground (dis)unification, a variation of the classic rigid E-unification problem. We introduced a sound and complete calculus to solve this problem in practice: Congruence Closure with Free Variables (CCFV). In particular, we now use CCFV to implement finding conflicting instances. This greatly improved the performances of the veriT solver, and was also beneficial for the CVC4 solver.

Project

Many aspects of SMT solving have an engineering aspect, and this is particularly true for quantifier reasoning: among the many instances that are generated, how to sort out the good, useful ones, from those that would only distract the ground solver from its goal? To this aim, it is necessary to evaluate usefulness. Besides instances that are clearly not useful, and those that are very useful, the solver will anyway generate quite a lot of them that might be useful, but will turn out not to be much so. I believe it is of uttermost importance to clean the ground solver of these instance. Currently, neither veriT nor CVC4 implements this. Besides the obvious tedious engineering work necessary to fix this issue, there is also the scientific aspect of evaluating usefulness of instances. In SAT solving, finding out good criteria to eliminate learned clauses was the source of one of the most impressive improvement in SAT solving lately [8].

Since generating irrelevant instances is very harmful for quantifier reasoning in SMT, it is natural to invest efforts in improving the quality of the information provided by the ground solver. This was the motivation behind our work on computing prime implicants [47]: prime implicants clear the model

from the irrelevant literals, so that instantiation focuses only on relevant literals. In [42], a notion of relevancy is introduced with a similar objective. I believe more filtering can be valuable. A formula generally comprises both a problem and a set of sentences that describe the context (or theory) to study the problem. In some cases, the context is fixed by the application generating the formulas, and it is not useful to check it for consistency. The most valuable terms to take into account for instantiation are the one strongly related to the problem, and not the ones coming from the context and that are present in each of the generated formulas.

Pure quantified first-order logic is best handled with *resolution* and *superposition*-based theorem proving [9]. There has been some attempts to unify such techniques with SMT [43, 40, 26], but the main approach used in SMT remains instantiation, with quantified formulas reduced to ground ones and refuted with the help of decision procedures for ground formulas. Designing a framework unifying state-of-the-art SMT and superposition techniques is a long term interest of my research. The success of Avatar [122] confirms that automated reasoning has much to gain in mixing SMT and superposition approaches. Other step stones exist, and for instance, one direction I will follow in the near future is investigating the lifting up of resolutions occurring inside the SMT solver to first-order resolutions.

Some theories (like Presburger arithmetic or the theory of real closed fields) have dedicated quantifier elimination procedures. Unfortunately, those procedures currently do not work well in cooperation with either superposition, or with instantiation. Since adding uninterpreted symbols to arithmetic signature (see e.g., [77]) often breaks decidability of the language, combining theory-specific quantifier elimination procedures with first-order reasoning on uninterpreted symbols will of course not yield decision procedures. It would be nice however (and valuable for many applications) to preserve refutational completeness in practice on the considered first-order theory extended with uninterpreted symbols.

Finding conflicting instances, for which we now use the CCFV algorithm, is very efficient to generate useful instances. Given an assignment for the ground part of the formula, it finds out whether there exists one instance of the quantified formula that would alone contradict this assignment. The underlying CCFV algorithm can be implemented in a complete way to guarantee that if such a single instance exists, it will always be found. Finding conflicting instance however is not a refutationally complete procedure, since one single instance might not always suffice to refute the ground assignment.

We are currently working on extending conflict based instantiation to also be complete in the case when two instances are necessary, and eventually to be complete whatever the number of required instances is. Basically, we plan to use resolution on the quantified formulas and use conflicting instances as before. Completeness is not guaranteed in general, but it might be in certain fragments.

Finally, we believe quantifier handling in automated reasoning can be inspired by the efficient techniques developed in the context of modal logics, and conversely, that the SMT point of view can lead to more expressive modal logics which could be handled efficiently using SMT solvers [5].

3.4 SMT for HOL and proofs

Full automatic proving of formulas is not possible in all cases. In general, expert human guidance is necessary at the highest levels of a proof, or to state and prove intermediate lemmas that are used in subtle proof steps. Interactive theorem provers (ITPs) provide the necessary tools to handle those subtle tasks for which human guidance is mandatory. An ITP may also serve as a proof manager, allowing to handle proofs manually, but also delegating some of the proof steps to various automated theorem provers that act as back-ends. I believe this is the right way to provide a convenient verification platform benefiting form the advances in automated reasoning. In particular, the power of SMT solvers should be made available to ITP users.

To maintain high confidence in the verdicts obtained using the platform (i.e., the ITP with its back-ends), we rely on proof-producing back-ends. ITPs are generally based on a very small (a few hundred lines of code) and carefully engineered kernel. Any proof built in the ITP has to be certified through this kernel. If a back-end provides proofs, those proofs can also be certified through the kernel. Any deduced fact, either manually or automatically by a back-end, is thus accepted as a theorem with the same very high level of confidence. A buggy external automated prover, or a buggy conduit between the ITP and the external prover only has as a consequence in the worst case a certification failure, but will never lead to certify a false theorem. The two pioneer works in this approach for SMT are the conduits between HOL-Light and CVC Lite [94], and our work between haRVey (the predecessor of veriT) and the Isabelle ITP [66].

The integration of automated deduction tools as back-ends of ITPs includes many engineering tasks, but there are also several theoretical issues to

solve for an effective cooperation of tools. First, the size of the proofs output by the solvers should not explode: every proof step should be described succinctly by the output format, and the proof size should be at most linear with respect to reasoning time. Some methods used in highly optimized tools (especially for formula simplification, or Skolemization) are difficult to tackle without specialized rules, but those specialized rules (involving for instance deep inference) would basically be implemented very inefficiently within ITPs, leading in a very inefficient cooperation between the ITPs and the solvers. Second, there exists a trade-off between full (and long) proof traces and smaller "certificates" whose checking may involve a limited amount of proof search. Third, besides trivial simplification methods like pruning the generated proofs, and eliminating redundant deductions, there exist methods to compress proofs that may lead to a significant decrease in proof size, which in turn, could also lead to a significant decrease of the time needed to rebuild the proof inside the ITP. I have contributed to several techniques leading to improved proof size [67, 68]. Some of these techniques are now implemented in an independent tool, Skeptic [27]. Although we are not involved in its development, we are still working with the authors of Skeptic on finding new algorithms to further decrease proof sizes in the SMT context.

Another very important issue for the cooperation of ITPs and automated tools is the interface. The input language of SMT solvers is standardized in [106, 17], and this standard is quite widely accepted. This simplifies the construction of conduits to SMT solvers, since a conduit generating formulas in this standardized language generates formulas suitable for all SMT solvers accepting this standard. For the conduit to be fully functional however, proofs also have to be read from the SMT solver, and replayed back inside the ITP. There is currently no standardized SMT proof format. Together with Aaron Stump, we initiated in 2011 the PxTP series of workshop. One of the objectives of these workshops is specifically to promote the emergence of good proof formats for automated reasoning tools. In [20] we proposed a preliminary proof format for SMT. A lot of work remains however to obtain a proof format accepted by the whole community. The issues are that the format should be simple, versatile to express proofs from various decision procedures, and compact. I believe now that standardization cannot be enforced for proofs, but will come from de facto acceptation of the cleanest way of representing proofs in SMT. Contrary to the input format, it will always be necessary for the SMT solver to produce some work to output a proof.

More recently, I have contributed to a proof infrastructure for the pre-

processing of formulas [12]. The main components are a generic contextual recursion algorithm and an extensible set of inference rules. Clausification, skolemization, theory-specific simplifications, and expansion of 'let' expressions are instances of this framework. With suitable data structures, proof generation adds only a linear-time overhead, and proofs can be checked in linear time.

Currently there exist conduits to veriT for Coq, Isabelle, Why3, TLAPS and Rodin. The goal of my project is to improve veriT both on the input and on the output aspects to increase its usefulness in such conduits.

Project

A major part of my future research will be dedicated to Jasmin Blanchette's ERC starting grant Matryoshka[2], to which I am associated as senior collaborator. The Sledgehammer tool for Isabelle is essentially based on eager translation of higher-order proof obligations to first-order formulas that are discharged either by SMT solvers or superposition provers. This tool greatly contributes to improve efficiency of formal proof efforts, but it has limits: sometimes trivial proofs cannot be found because solvers are lost in translation. In the Matryoshka project, we want to augment SMT solvers with higher-order reasoning in a careful manner, to preserve their desirable properties. The approach is application-oriented, based on the available large repositories of interactive verification efforts.

We are currently investigating three aspects. First, we are extending congruence closure and instantiation to tackle higher-order constructions. Second, we are getting inspiration from the automated reasoning capabilities of interactive provers to add new kinds of reasoning methods inside SMT. Indeed, higher-order rewriting as implemented in proof assistants can sometimes be surprisingly good at finding proofs, whereas hammers fail on the same tasks. Last but not least, induction is often a crucial aspect for verification tasks, but this is currently not or insufficiently integrated into SMT solvers.

In the long run, the cooperation between automatic provers, proof assistants, and human users should be less fragmented: currently, users write theorems that are either proved by the assistant or the automatic provers via hammers. The future might well be that assistants and provers work together and rather ask users for input for hard parts of the proofs.

[2] `http://matryoshka.gforge.inria.fr/`

3.5 Tools and applications

The primal application of my research was verification of parameterized systems. In [64, 60], I showed that validation of inductive invariants for some kind of parameterized systems is a decidable problem, and SMT solvers are essential components of the decision procedure. Since then, I have been actively developing stand alone tools that can be used as back-ends for verification platform. With the veriT solver [28] (the system description paper is in the selection of papers on page 65), our goal is to provide a stable tool that implements our research algorithms and make them available in open-source.

On the front-end side, I have been contributing to the integration of SMT solvers in the verification platform Rodin [48, 49] (see selected paper on page 65). On a set of Event-B models stemming from industrial and academic problems, our techniques decrease by a factor of four the number of proof obligations whose verification requires human interaction. According to users, this is particularly helpful for verification conditions where the major challenge is the combinatorial aspect of the formula. Thanks to the SMT-LIB initiative and to the standard SMT-LIB language, building a plugin to use all SMT solvers is not more difficult than implementing a conduit for just one of them, at least for the common language. For a few years now I have been active as an SMT-LIB manager.

Project

Augmenting SMT with higher-order capabilities and improving proof generation will also improve the usefulness of SMT solvers for applications. I will continue to devote an increasing part of my time for the management of the SMT-LIB, and I am looking forward for the next SMT-LIB version, that should standardize many new features, including higher-order constructs.

Within the Matryoshka project, the objective is to improve SMT support for platforms. The priority targets are Isabelle and TLAPS. Sometimes however applications come from unexpected areas. For instance, SAT solvers are used to check the dependencies of software, or to look over the satisfiability of configurations for vehicles. The future applications for SMT might well be of a surprising nature; it is important to recognize them anyway.

4 From logic to education in computer science

As computer science reaches maturity, it is necessary to present it differently, packaged so that students can get the basics in a way that give them a deep understanding of the inner workings of modern computer systems and at the same time withhold the awfully specific details real systems are full of. I have been teaching at various levels and for various kinds of students, but I believe all need a more polished presentation of computer science than what is given to them now, which is too tightly tied to rising technologies. For instance, the religious wars between programming languages are particularly strong on the field of education. Maybe it is time to comprehend (again) programming without a specific modern programming language in mind. In the same vein, modern computers are only understandable with a simplified picture in mind. But this picture is not sufficiently well drawn.

I advocate that some basic computer science topics all go very well together: logic, computer structures, assembly language, algorithms and complexity, programming, compilers, and verification. There is some work to invest to present these in a connected way. In a logic course I have been giving to second year students for business informatics and cognitive sciences, I submitted a simple problem of comparing two pieces of code, differing only by the test in a conditional statement; the tests were logically equivalent but more than one student out of two asserted that the two pieces of code were behaving differently although no side effect could account for that. Logic is also strongly linked to computer structures. In a course I am teaching to the same second year students, we build, starting with logic gates, a full working RISC computer, simulated within a software, and that serves as a basis to program in assembly language. And programming this RISC machine in assembly language gives the students a full understanding of how loops, recursion, memory and pointers work. Logic comes again at hand when you show you can actually prove some code you have written, either in the low level, RISC assembly language, or in a higher level code. But again, understanding how the higher level code is translated in assembly language is instrumental to actually prove that the code is correct, since it gives it a clear semantics. This links verification, logic, computer structures and assembly language to some aspects of compilation. I have never taught algorithms and complexity, and I believe it is not easy to teach this to non-scientific students. But again, on a RISC machine, complexity has a precise definition: it is related to the number of basic instructions corresponding to an input of a given size. With a high level language, this notion is not clear at all: instructions deeply embedded

in the language have a non trivial (and often not documented) complexity. One school of thought is going away from the details in computer science. I believe we have to get back to details, but on a reconstructed abstraction of reality. Nobody can understand today's combustion engines in details, but understanding well conceived abstractions provides sufficient grounds not to be afraid when approaching the real thing. It is not only necessary to give knowledge to students so that they are able to perform in companies; providing them with a deep understanding of the principles underlying computer science will help them better. And this is not possible to do it anymore on the basis of state-of-the-art computers and programming languages.

5 Conclusion

It was once a time when logic and automated theorem proving was fully part of what people call *artificial intelligence*. Today, as I am writing these lines, there is again some diffuse fear that artificial intelligent entities will take over the world. Probably this is not more true than before, but it is true however that modern techniques of artificial intelligence have a deep impact on some scientific areas, and for instance completely modifies the state-of-the-art and techniques for speech recognition, or automatic treatment of natural language. It is not clear yet what impact it will have on automated reasoning. But I firmly believe it will not kill automated reasoning as we know it, just like it will not kill traditional programming. Probably, and this is a fact that is completely absent of my research program, machine learning will find its place in automated reasoning, and will greatly improve it. Perhaps automated reasoning might also find its place in artificial intelligence, as a complement module to rigorously check soundness of deduction, or to efficiently evaluate problems that can be translated to logic?

If we consider the publication of Hoare logic as a birth date [78], verification will soon turn fifty. Those years saw great hopes, and great disillusionment. Reading today Dijkstra's point of view [53] on the failure of programming in 1972 is funny in a sad way: many of his statements are still valid fifty years later. The verifying compiler, once expected as an outcome of the newborn science of verification, never really existed; Tony Hoare quite recently advocated to revive this challenge [79]. On the other hand, the situation is not that desperate. There exist verified compilers (see e.g. the CompCert project) and verified kernels, and large companies are surely all aware of and

use verification techniques.[3] It is impossible to know how the world would feel like without verification at all, but all major companies do find an economic value in maintaining a formal verification department. Many discoveries (think about quantum mechanics or DNA) are full of promises, and later reveal themselves to be more difficult to use than expected, but anyway lead to great practical advances. We are unable to prove every program easily just like we are unable to solve the Schrödinger equation for all systems. Anyway, the importance of verification and automated reasoning will continue to grow, while the pervasiveness of algorithms, computer systems and the cloud will increase. I do not believe it will one day become easy, but with the same effort, we will push verification much further. I do believe though that one day, automated deduction integrated within interactive theorem proving will discharge most non-creative thinking out of formally verified proofs.

[3] In the context of a project with RATP, I was myself quite surprised to realize that many people with a verification background work there.

References

[1] Erika Ábrahám, John Abbott, Bernd Becker, Anna Maria Bigatti, Martin Brain, Bruno Buchberger, Alessandro Cimatti, James H. Davenport, Matthew England, Pascal Fontaine, Stephen Forrest, Alberto Griggio, Daniel Kroening, Werner M. Seiler, and Thomas Sturm. SC^2: Satisfiability checking meets symbolic computation - (project paper). In Michael Kohlhase, Moa Johansson, Bruce R. Miller, Leonardo Mendonça de Moura, and Frank Wm. Tompa, editors, *Intelligent Computer Mathematics (CICM)*, volume 9791 of *Lecture Notes in Computer Science*, pages 28–43. Springer, 2016.

[2] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] Ernst Althaus, Evgeny Kruglov, and Christoph Weidenbach. Superposition modulo linear arithmetic SUP(LA). In Silvio Ghilardi and Roberto Sebastiani, editors, *Frontiers of Combining Systems (FroCoS)*, volume 5749 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2009.

[4] Carlos Areces and Pascal Fontaine. Combining theories: The ackerman and guarded fragments. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems (FroCoS)*, volume 6989 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2011.

[5] Carlos Areces, Pascal Fontaine, and Stephan Merz. Modal satisfiability via SMT solving. In Rocco De Nicola and Rolf Hennicker, editors, *Software, Services and Systems. Essays Dedicated to Martin Wirsing on the Occasion of His Emeritation*, volume 8950 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 2015.

[6] Aristotle. *Prior Analytics*. 350BCE. Translation into English by Arthur J.J. Jenkinson, 1928.

[7] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. New results on rewrite-based satisfiability procedures. *ACM Trans. Comput. Log.*, 10(1), 2009.

[8] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceed-*

ings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009, pages 399–404, 2009.

[9] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[10] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science B.V., 2001.

[11] Kshitij Bansal, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. A new decision procedure for finite sets and cardinality constraints in SMT. In Nicola Olivetti and Ashish Tiwari, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 9706 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2016.

[12] Haniel Barbosa, Jasmin Christian Blanchette, and Pascal Fontaine. Scalable fine-grained proofs for formula processing. In Leonardo de Moura, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 10395 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2017.

[13] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 10206 of *Lecture Notes in Computer Science*, pages 214–230. Springer, 2017.

[14] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[15] Clark Barrett, Daniel Kroening, and Thomas Melham. Problem solving for the 21st century: Efficient solvers for satisfiability modulo theories. Technical Report 3, London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, June 2014. Knowledge Transfer Report.

[16] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[17] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard : Version 2.0, March 2010. First official release of Version 2.0 of the SMT-LIB standard.

[18] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In Mandayam K. Srivas and Albert John Camilleri, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996.

[19] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[20] Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for SMT: a proposal. In *Workshop on Proof eXchange for Theorem Proving (PxTP)*, 2011.

[21] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008.

[22] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[23] Nikolaj Bjorner, Anca Browne, Eddie Chang, Michael Colon, Arjun Kapur, Zohar Manna, Sipma Sipma, and Tomas E. Uribe. STeP: The stanford temporal prover (educational release) user's manual. Technical Report CS-TR-95-1562, Stanford University, Department of Computer Science, November 1995.

[24] Alexander Bockmayr and V. Weispfenning. Solving numerical constraints. In John Alan Robinson and Andrei Voronkov, editors, *Handbook*

of Automated Reasoning, volume I, chapter 12, pages 751–842. Elsevier Science B.V., 2001.

[25] Maria Paola Bonacina, Ulrich Furbach, and Viorica Sofronie-Stokkermans. On first-order model-based reasoning. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, volume 9200 of *Lecture Notes in Computer Science*, pages 181–204. Springer, 2015.

[26] Maria Paola Bonacina, Christopher Lynch, and Leonardo Mendonça de Moura. On deciding satisfiability by theorem proving with speculative inferences. *Journal of Automated Reasoning*, 47(2):161–189, 2011.

[27] Joseph Boudou, Andreas Fellner, and Bruno Woltzenlogel Paleo. Skeptik: A proof compression system. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 8562 of *Lecture Notes in Computer Science*, pages 374–380. Springer, 2014.

[28] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In Renate Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156, Montreal, Canada, 2009. Springer.

[29] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 335–349. Springer, 2005.

[30] Martin Bromberger, Thomas Sturm, and Christoph Weidenbach. Linear integer arithmetic revisited. In Amy P. Felty and Aart Middeldorp, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 9195 of *Lecture Notes in Computer Science*, pages 623–637. Springer, 2015.

[31] Martin Bromberger and Christoph Weidenbach. Fast Cube Tests for LIA Constraint Solving. In Nicola Olivetti and Ashish Tiwari, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume

9706 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2016.

[32] Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. A gentle non-disjoint combination of satisfiability procedures. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 8562 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2014.

[33] Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. Satisfiability modulo non-disjoint combinations of theories connected via bridging functions. In Silvio Ghilardi, Ulrike Sattler, and Viorica Sofronie-Stokkermans, editors, *Automated Deduction: Decidability, Complexity, Tractability (ADDCT)*, 2014.

[34] Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. A polite non-disjoint combination method: Theories with bridging functions revisited. In Amy P. Felty and Aart Middeldorp, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 9195 of *Lecture Notes in Computer Science*, pages 419–433. Springer, 2015.

[35] Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. A rewriting approach to the combination of data structures with bridging theories. In Carsten Lutz and Silvio Ranise, editors, *Frontiers of Combining Systems (FroCoS)*, volume 9322 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2015.

[36] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Satisfiability modulo transcendental functions via incremental linearization. In Leonardo de Moura, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 10395 of *Lecture Notes in Computer Science*, pages 95–113. Springer, 2017.

[37] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[38] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[39] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[40] Leonardo Mendonça de Moura and Nikolaj Bjørner. Bugs, moles and skeletons: Symbolic reasoning for software development. In Jürgen Giesl and Reiner Hähnle, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 6173 of *Lecture Notes in Computer Science*, pages 400–411, Berlin, Heidelberg, 2010. Springer-Verlag.

[41] Leonardo Mendonça de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7737 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2013.

[42] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[43] Leonardo Mendonça de Moura and Nikolaj Bjørner. Engineering DPLL(T) + saturation. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 5195, pages 475–490, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] Leonardo Mendonça de Moura and Nikolaj Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, 2008.

[45] Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. Combining decision procedures by (model-)equality propagation. *Electronic Notes in Theoretical Computer Science*, 240:113–128, 2009. Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008), Salvador, Brazil, 26-29 August 2008.

[46] Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. Combining decision procedures by (model-)equality propagation. *Science of Computer Programming*, 77(4):518–532, 2012.

[47] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *Formal Methods In Computer-Aided Design (FMCAD)*, pages 46–52. IEEE, 2013.

[48] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. SMT solvers for Rodin. In John Derrick, John A. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *ABZ*, volume 7316 of *Lecture Notes in Computer Science*, pages 194–207. Springer, 2012.

[49] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. Integrating SMT solvers in Rodin. *Science of Computer Programming*, 94:130–143, 2014.

[50] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 6803 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2011.

[51] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, Hewlett Packard Laboratories, July 23 2003.

[52] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *J. ACM*, 52(3):365–473, 2005.

[53] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[54] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[55] Bruno Dutertre and Leonardo Mendonça de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 2006.

[56] Bruno Dutertre and Leonardo Mendonça de Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, 2006.

[57] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume

3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, June 2005.

[58] Andreas Fellner, Pascal Fontaine, Georg Hofferek, and Bruno Woltzenlogel Paleo. NP-completeness of small conflict set generation for congruence closure. In Vijay Ganesh and Dejan Jovanović, editors, *International Workshop on Satisfiability Modulo Theories (SMT)*, 2015.

[59] Andreas Fellner, Pascal Fontaine, and Bruno Woltzenlogel Paleo. NP-completeness of small conflict set generation for congruence closure. *Formal Methods in System Design*, 51(3):533–544, December 2017.

[60] Pascal Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, Institut Montefiore, Université de Liège, Belgium, September 2004.

[61] Pascal Fontaine. Combinations of theories for decidable fragments of first-order logic. In Silvio Ghilardi and Roberto Sebastiani, editors, *Frontiers of Combining Systems (FroCoS)*, volume 5749 of *Lecture Notes in Computer Science*, pages 263–278. Springer, 2009.

[62] Pascal Fontaine. Combinations of theories for decidable fragments of first-order logic, 2009. Available at `http://www.loria.fr/~fontaine/Fontaine12b.pdf`.

[63] Pascal Fontaine and E. Pascal Gribomont. Using BDDs with combinations of theories. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2514 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 2002.

[64] Pascal Fontaine and E. Pascal Gribomont. Decidability of invariant validation for parameterized systems. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 97–112. Springer-Verlag, 2003.

[65] Pascal Fontaine and E. Pascal Gribomont. Combining non-stably infinite, non-first order theories. In W. Ahrendt, P. Baumgartner, H. de Nivelle, S. Ranise, and C. Tinelli, editors, *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 2004)*, volume 125 of *Electronic Notes in Theoretical Computer Science*, pages 37–51, July 2005.

[66] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2006.

[67] Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploring and exploiting algebraic and graphical properties of resolution. In Aarti Gupta and Daniel Kroening, editors, *International Workshop on Satisfiability Modulo Theories (SMT)*, July 2010.

[68] Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Compression of propositional resolution proofs via partial regularization. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 6803 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2011.

[69] Pascal Fontaine, Stephan Merz, and Christoph Weidenbach. Combination of disjoint theories: Beyond decidability. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2012.

[70] Pascal Fontaine, Mizuhito Ogawa, Thomas Sturm, and Xuan-Tung Vu. Subtropical satisfiability. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems (FroCoS)*, volume 10483 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017.

[71] Pascal Fontaine, Silvio Ranise, and Calogero G. Zarba. Combining lists with non-stably infinite theories. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3452 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, 2005.

[72] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT*, 1(3-4):209–236, 2007.

[73] Harald Ganzinger and Hans De Nivelle. A superposition decision procedure for the guarded fragment with equality. In *Logic In Computer Science (LICS)*, pages 295–303. IEEE Computer Society Press, 1999.

[74] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[75] Alberto Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, January 2012.

[76] Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. Deciding floating-point logic with systematic abstraction. In Gianpiero Cabodi and Satnam Singh, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, pages 131–140. IEEE Computer Society, 2012.

[77] Joseph Y. Halpern. Presburger arithmetic with unary predicates is Π_1^1 complete. *The Journal of Symbolic Logic*, 56(2):637–642, June 1991.

[78] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[79] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[80] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In Manuela M. Veloso, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 2318–2323, 2007.

[81] Ullrich Hustadt, Renate A. Schmidt, and Lilia Georgieva. A survey of decidable first-order fragments and description logics. *Journal of Relational Methods in Computer Science*, 1:251–276, 2004.

[82] Maximilian Jaroschek, Pablo Federico Dobal, and Pascal Fontaine. Adapting real quantifier elimination methods for conflict set computation. In Carsten Lutz and Silvio Ranise, editors, *Frontiers of Combining Systems (FroCoS)*, volume 9322 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2015.

[83] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012.

[84] Dejan Jovanovic and Leonardo Mendonça de Moura. Cutting to the chase solving linear integer arithmetic. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 6803 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2011.

[85] Dejan Jovanović and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *Lecture Notes in Computer Science*, pages 339–354. Springer Berlin Heidelberg, 2012.

[86] Tim King, Clark Barrett, and Cesare Tinelli. Leveraging linear and mixed integer programming for smt. In Koen Claessen and Viktor Kuncak, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, pages 24:139–24:146, 2014.

[87] Marek Košta. New concepts for real quantifier elimination by virtual substitution, 2016. Doctoral Dissertation, Saarland University.

[88] Daniel Kröning, Philipp Rümmer, and Georg Weissenbacher. A proposal for a theory of finite sets, lists, and maps for the SMT-lib standard, 2009.

[89] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient Nelson-Oppen decision procedure for difference constraints over rationals. *Electronic Notes in Theoretical Computer Science*, 144(2):27–41, 2006.

[90] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, Mass., 2002.

[91] Zohar Manna, Anuchit Anuchitanukul, Nikolaj S. Bjørner, Anca Browne, Edward Chang, Michael Colon, Luca de Alfaro, Harish Devarajan, Henny B. Sipma, and Tomas Uribe. STeP: The stanford temporal prover. Technical Report CS-TR-94-1518, Stanford University, Computer Science Department, June 1994.

[92] João P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 220–227, San Jose, California, USA, November 10–14 1996. IEEE Computer Society.

[93] Yu. V. Matijasevič. Diophantine representation of recursively enumerable predicates. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 171–177, Amsterdam, 1971. North-Holland.

[94] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-light and CVC lite. *Electronic Notes in Theoretical Computer Science*, 144(2):43–51, 2006.

[95] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference (DAC)*, pages 530–535. ACM press, June 2001.

[96] C. G. Nelson. *Techniques for program verification.* Xerox, Palo Alto Research Center, 1980. CSL-81-10 // XEROX Palo Alto Research Center.

[97] Greg Nelson and Derek C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[98] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[99] Robert Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science B.V., 2001.

[100] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016.

[101] Christos H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, October 1981.

[102] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, USA, 1994.

[103] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[104] David A. Plaisted and Steven Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[105] Loic Pottier. Connecting gröbner bases programs with coq to do proofs in algebra, geometry and arithmetics. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *LPAR Workshops*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[106] Silvio Ranise and Cesare Tinelli. The SMT-LIB standard : Version 1.2, August 2006.

[107] Giles Reger, Nikolaj Bjørner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In Christoph Benzmüller, Geoff Sutcliffe, and Raúl Rojas, editors, *Global Conference on Artificial Intelligence (GCAI)*, volume 41 of *EPiC Series in Computing*, pages 39–52. EasyChair, 2016.

[108] Giles Reger, Martin Suda, and Andrei Voronkov. Instantiation and pretending to be an SMT solver with vampire. In Martin Brain and Liana Hadarean, editors, *International Workshop on Satisfiability Modulo Theories (SMT)*, 2017.

[109] Andrew Reynolds and Jasmin Christian Blanchette. A decision procedure for (co)datatypes in SMT solvers. *Journal of Automated Reasoning*, 58(3):341–362, 2017.

[110] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods In Computer-Aided Design (FMCAD)*, pages 195–202. IEEE, 2014.

[111] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In MariaPaola Bonacina, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2013.

[112] Christophe Ringeissen. *Combinaison de résolution de contraintes*. PhD thesis, Université de Nancy 1, INRIA-Lorraine, Nancy, France, December 1993.

[113] Philipp Rümmer. E-Matching with free variables. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 7180 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2012.

[114] Karem A. Sakallah. Symmetry and satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 289–338. IOS Press, 2009.

[115] Stephan Schulz and Maria Paola Bonacina. On Handling Distinct Objects in the Superposition Calculus. In B. Konev and S. Schulz, editors, *Proc. of the 5th International Workshop on the Implementation of Logics, Montevideo, Uruguay*, pages 66–77, 2005.

[116] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[117] Viorica Sofronie-Stokkermans. Hierarchical reasoning in local theory extensions and applications. In Franz Winkler, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen M. Watt, and Daniela Zaharie, editors, *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 34–41. IEEE Computer Society, 2014.

[118] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing (SAT)*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[119] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *Logic In Computer Science (LICS)*, pages 29–37. IEEE Computer Society, June 2001.

[120] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and Klaus U. Schulz, editors, *Frontiers of Combining Systems (FroCoS)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, March 1996.

[121] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[122] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer, 2014.

[123] Thomas Wies, Ruzica Piskac, and Viktor Kuncak. Combining theories with shared set operations. In Silvio Ghilardi and Roberto Sebastiani, editors, *Frontiers of Combining Systems (FroCoS)*, volume 5749 of *Lecture Notes in Computer Science*, pages 366–382. Springer, 2009.

[124] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. Approximations for model construction. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 8562 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 2014.

[125] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. Deciding bit-vector formulas with mcSAT. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *Lecture Notes in Computer Science*, pages 249–266. Springer, 2016.

[126] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. An approximation framework for solvers and decision procedures. *Journal of Automated Reasoning*, 58(1):127–147, 2017.

Glossary

BCP Boolean Constraint Propagation

CDCL Conflict-Driven Clause Learning

CNF Conjunctive Normal Form

DPLL Davis, Puttnam, Logemann and Loveland algorithm

FOL First-Order Logic

FUIP First Unique Implication Point

ITP Interactive Theorem Prover

MCSAT Model-Constructing Satisfiability Modulo Theories

RISC Reduced Instruction Set Computer

SAT Propositional Satisfiability

SMT Satisfiability Modulo Theories

VSIDS Variable State Independent Decaying Sum

A selection of five publications

- Andreas Fellner, Pascal Fontaine, and Bruno Woltzenlogel Paleo. NP-completeness of small conflict set generation for congruence closure. Formal Methods in System Design 51(3): 533-544 (2017)

  Proofs of NP-completeness are often fascinating: they are often not trivial to find although they can be obvious to follow. This is not an exception. The efficiency of satisfiability modulo theories (SMT) solvers is dependent on the capability of theory reasoners to provide small conflict sets, i.e. small unsatisfiable subsets from unsatisfiable sets of literals. We show here the NP-completeness of generating smallest conflict sets for one of the simplest decidable theory, i.e. sets of literals with only uninterpreted symbols and equalities. This is proved using a simple reduction from SAT.

- David Déharbe, Pascal Fontaine, Yoann Guyot and Laurent Voisin. Integrating SMT solvers in Rodin. Sci. Comput. Program. 94(2): 130–143 (2014)

  One of the most often used motivation for automated reasoning is verification. In this work, we investigated the use of SMT in addition to the traditional tools embedded in the Rodin platform for formal development in Event-B. Our contribution is the definition of a translation of Event-B proof obligations to the language of SMT solvers, its implementation in a Rodin plug-in, and an experimental evaluation on a large sample of industrial and academic projects. On this domain, adding SMT solvers to Atelier B provers reduces significantly the number of sequents that need to be proved interactively. SMT solvers are now regularly used within Rodin.

- Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. A Gentle Non-Disjoint Combination of Satisfiability Procedures. In Stéphane Demri, Deepak Kapur and Christoph Weidenbach, editors, In *Proc. International Joint Conference on Automated Reasoning (IJCAR)*, volume 8562 of *Lecture Notes in Computer Science (LNCS)*. pages 122–136, Springer, 2014.

  This result is at the same time hard (involving an extension of the Ramsey theorem) and quite simple to understand. A combination framework should ensure that the decision procedures in the combination share enough information to eventually cooperatively show that the set of formulas is unsatisfiable, or that the sharing of information will eventually stop and a model for the union of formulas can be built. This is similar in the case of disjoint or non-disjoint union of theories, only the shared information changes. For major classes of theories, namely the Löwenheim and Bernays-Schönfinkel-Ramsey classes, the non-disjoint combination of theories sharing only unary predicates (plus constants and the equality) is decidable. The shared information is the cardinality of the set of support of each shared unary predicate, and this is computable for those first-order decidable classes.

- David Déharbe, Pascal Fontaine, Stephan Merz and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, In *Proc. Conference on Automated Deduction (CADE)*, volume 6803 of *Lecture Notes in Computer Science (LNCS)*. pages 222–236, Springer-Verlag, 2011.

  In this work, we investigate using symmetries in formulas to improve the efficiency of satisfiability checking. This technique is based on the concept of (syntactic) invariance by permutation of constants. An algorithm for solving SMT by taking advantage of such symmetries is presented. The technique in itself is certainly interesting, but this paper is mainly an advocacy for better encoding of problems into SMT. For nearly ten years, many benchmarks in the QF_UF category of the SMT-LIB remained undefeated because they were encoded in a way that enforced solvers to study all equivalent permutations of the same easy problems.

- Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, In *Proc. Conference on Automated Deduction (CADE)*. volume 5663 of *Lecture Notes in Computer Science (LNCS)*, pages 151–156. Springer-Verlag, 2009.

  This is the published tool description of the veriT solver, which took a significant amount of my time to design and develop. The current version of the solver also owes Haniel Barbosa some impressive improvements, particularly on the quantifier handling aspects. The veriT solver is used as back-end in several platforms, e.g. Isabelle via Sledgehammer and Rodin via the SMT plugin.

NP-completeness of small conflict set generation for congruence closure

Andreas Fellner · Pascal Fontaine · Bruno Woltzenlogel Paleo

Abstract The efficiency of Satisfiability Modulo Theories (SMT) solvers is dependent on the capability of theory reasoners to provide small conflict sets, i.e. small unsatisfiable subsets from unsatisfiable sets of literals. Decision procedures for uninterpreted symbols (i.e. congruence closure algorithms) date back from the very early days of SMT. Nevertheless, to the best of our knowledge, the complexity of generating smallest conflict sets for sets of literals with uninterpreted symbols and equalities had not yet been determined, although the corresponding decision problem was believed to be NP-complete. We provide here an NP-completeness proof, using a simple reduction from SAT.

Keywords Satisfiability Modulo Theories · Decision procedures · Congruence closure · Complexity

1 Introduction

Satisfiability Modulo Theory solvers are nowadays based on a cooperation between a propositional satisfiability (SAT) solver and a theory reasoner for the combination of theories supported by the SMT solver. The propositional structure of the problem is handled by the SAT solver, whereas the theory reasoner only has to deal with conjunctions of literals. Very schematically (we

This work has been partially supported by the project ANR-13-IS02-0001 of the Agence Nationale de la Recherche, by the H2020-FETOPEN-2016-2017-CSA project SC^2 (712689), by an ÖAW APART Stipendium, by FWF W1255-N23, by ERC Start Grants Graph Games (279307) and Matryoshka (713999), and by FFG project number 845582 (TRUCONF).

A. Fellner
Austrian Institute of Technology and Vienna University of Technology (Austria)
E-mail: andreas.fellner@ait.ac.at

P. Fontaine
Inria, Loria, U. of Lorraine (France)

B. Woltzenlogel Paleo
Vienna University of Technology (Austria) and Australian National University (Australia)

refer to [1] for more details) the Boolean abstraction of the SMT problem is repeatedly refined by adding theory conflict clauses that eliminate spurious models of the abstraction, until either unsatisfiability is reached, or a model of the SMT formula is found. Refinements can be done by refuting models of the propositional abstraction one at a time. It is, however, much more productive to refute all propositional models that are spurious for the same reason at once. A model of the abstraction is spurious if the set of concrete literals corresponding to the abstracted literals satisfied by this model is unsatisfiable modulo the theory. Given such an unsatisfiable set of concrete literals, the disjunction of the negations of any unsatisfiable subset (a.k.a. *core*) is a suitable conflict clause. By backtracking and asserting the conflict clause, the SAT-solver is prevented from generating the spurious model again. The smaller the clause, the stronger it is and the more spurious models it prevents. Therefore, an optimal conflict clause, corresponding to a minimal unsatisfiable subset of literals (i.e. such that all its proper subsets are satisfiable) or even a minimum one (i.e. smallest among the minimals) is desirable. This feature of the theory reasoners to *generate small conflict sets* (a name adopted in [1]) from their input is also referred to as *proof production* [8,9] or *explanation generation* [10].

Decision procedures for the theory of uninterpreted symbols and equality can be based on congruence closure [7,3,10]. The decision problem is polynomial and even quasi-linear [3] with respect to the number of terms and literals in the input set. Producing minimal conflict sets also takes polynomial time. Indeed, testing if a set S remains unsatisfiable after removal of one of its literals is also polynomial. It suffices then to repeatedly test the $|S|$ literals of S to check if they can be removed. The set S pruned of its unnecessary literals is minimal. One could also profit from the incrementality of the decision procedure [6].

It has also been common knowledge that computing minimum conflict clauses for the theory of uninterpreted symbols and equality is a difficult problem. But, to our best knowledge, the complexity of finding the smallest conflict clause generation for sets of literals with uninterpreted symbols and equalities has never been established. The complexity of the corresponding decision problem (i.e. of whether there exists a conflict clause with size smaller than a given k) is mentioned to be NP-complete in [10] — with a reference to a private communication with Ashish Tiwari — but neither the authors of [10] nor Ashish Tiwari published a written proof of this fact[1].

Our interest in this problem arose from our work on Skeptik [2], a tool for the compression of proofs generated by SAT and SMT solvers. For the sake of moving beyond the purely propositional level, we have developed an algorithm for compressing congruence closure proofs, which consists of regenerating (possibly smaller) congruence closure conflict clauses while traversing the proof. Congruence closure conflict clauses are typically generated from paths in the *congruence graph* maintained by the congruence closure algorithm [5,10,9]. In order to obtain small conflict clauses, and thereby small congruence clo-

[1] We contacted both Ashish Tiwari and the authors of [10], who confirmed this.

sure proofs, we (dynamically) assigned weights to the congruence graph and searched for shortest paths in that graph. The weights of input equations would be 1, whereas the weight of a congruence edge would be the size of an explanation of its equation. This raised the question whether we could construct shortest conflict clauses as shortest paths in such weighted congruence graphs, by applying a polynomial time shortest path algorithm to a graph of polynomial size. We answered this question negatively by proving that the problem of deciding whether a shorter conflict clause exists is NP-hard. The goal of this article is to present this proof. The reason why the shortest path method is not able to find shortest conflict clauses is that the weights for congruence edges can not be accurately determined a priori. A preliminary version was presented at the SMT Workshop 2015 [4].

2 Preliminaries

We assume knowledge of propositional logic and quantifier-free first-order logic with equality and uninterpreted symbols, and only enumerate the notions and notations used in this article. A literal is either a propositional variable or the negation of a propositional variable. A clause is a disjunctive set of literals. A propositional variable x appears positively (negatively) in a clause C if $x \in C$ (resp. $\neg x \in C$). The notations $\{\ell_1, \dots \ell_n\}$ and $\ell_1 \vee \dots \vee \ell_n$ will be used interchangeably. A clause is tautological if and only if it contains a variable both positively and negatively. We shall tacitly assume that clauses are non-tautological, except when explicitly stated otherwise. Clauses being sets, they cannot contain multiple occurrences of the same literal. A formula in conjunctive normal form (CNF for short) is a conjunctive set of clauses. A total (partial) assignment $\mathcal{I}$ for a formula in propositional logic assigns a value in $\{\top, \bot\}$ to each (resp. some) propositional variable(s) in the formula. An assignment $\mathcal{I}$ for a formula F is a model of F, denoted $\mathcal{I} \models F$, if it makes the formula F true. A formula is satisfiable if it has a model, it is unsatisfiable otherwise. A total or partial assignment is perfectly defined by the set of literals it makes true. By default, an assignment is total unless explicitly said to be partial. A set of formulas E entails a (set of) formula(s) E', denoted $E \models E'$, if every model of E is a model of E'.

We now define the necessary notions for quantifier-free first-order logic.

Definition 1 (Terms and equations) A *signature* Σ is a finite set of function symbols $\mathcal{F}$ equipped with an *arity* function $\mathcal{F} \to \mathbb{N}$. A *constant* is a nullary function. A *unary* function has arity one. Given a signature Σ, the set of *terms* $\mathcal{T}^\Sigma$ is the smallest set containing all constants in $\mathcal{F}$ and all terms of the form $g(t_1, \dots, t_n)$, where g is a function symbol of arity n in $\mathcal{F}$ and $t_1, \dots, t_n$ are terms in $\mathcal{T}^\Sigma$. An *equation* between two terms s, t in $\mathcal{T}^\Sigma$ is denoted by $s = t$.

Signatures commonly include predicate symbols. Everything extends smoothly to signatures with predicates, but to simplify, a quantifier-free first-order logic

formula is here just a Boolean combination of equalities between terms; a literal is either an equation or the negation of an equation.

The terms $t_1, \ldots, t_n$ are *direct subterms* of $g(t_1, \ldots, t_n)$. The *subterm* relation is the reflexive and transitive closure of the direct subterm relation. Given a set of equations E, we denote by $\mathcal{T}(E)$ the set of terms and subterms occurring in the equations.

An assignment $\mathcal{I}$ on some signature maps each constant to an element in a universe $\mathcal{U}$, and each function symbol to a function of appropriate arity on $\mathcal{U}$. By extension, it assigns an element in $\mathcal{U}$ to every term, and a value to every equation $s = t$, namely $\top$ if $\mathcal{I}(s) = \mathcal{I}(t)$ and $\bot$ otherwise. Like in propositional logic, an assignment on some signature thus gives a truth value to every formula on this signature.

Definition 2 (Congruence relation) Given a set of terms $\mathcal{T}$ closed under the subterm relation, a relation $R \subseteq \mathcal{T} \times \mathcal{T}$ is a congruence if it is

- reflexive: $(t, t) \in R$ for each $t \in \mathcal{T}$;
- symmetric: $(s, t) \in R$ if $(t, s) \in R$;
- transitive: $(r, t) \in R$ if $(r, s) \in R$ and $(s, t) \in R$;
- compatible: $(g(t_1, \ldots, t_n), g(s_1, \ldots, s_n)) \in R$ if g is a n-ary function symbol and $(t_i, s_i) \in R$ for all $i = 1, \ldots, n$.

A congruence relation is also an equivalence relation, since it is reflexive, transitive and symmetric. Therefore a congruence relation partitions its underlying set of terms $\mathcal{T}$ into congruence classes, such that two terms (s, t) belong to the same class if and only if $(s, t) \in R$. The relations $\{(t, t) : t \in \mathcal{T}\}$ and $\mathcal{T} \times \mathcal{T}$ are trivial congruence relations. An assignment $\mathcal{I}$ on a signature Σ defines a congruence relation on any subset $\mathcal{T} \subseteq \mathcal{T}^{\Sigma}$, that is, $R = \{(s, t) \mid \mathcal{I}(s = t) = \top\}$.

An equation $s = t$ on terms in a set $\mathcal{T}$ can be seen as a singleton relation $\{(s, t)\} \subseteq \mathcal{T} \times \mathcal{T}$. By extension, a set of equations can also be seen as a relation, i.e., the union of the singleton relations.

Definition 3 (Congruence closure) The congruence closure E^* of a set of equations E on a set of terms $\mathcal{T}$ closed under the subterm relation is the smallest congruence relation on $\mathcal{T}$ containing E.

Since congruence relations are closed under intersection, the congruence closure of a set of equations always exists. Also notice that, if $(s, t) \in E^*$, then $E \models s = t$. We say that E is an *explanation* for $s = t$.

An algorithm computing the congruence closure of a relation is also a decision procedure for the problem of satisfiability of sets of equalities and disequalities in quantifier-free first-order logic with uninterpreted (predicates and) functions. It suffices indeed to compute the congruence closure of all equalities on the terms and subterms occurring in the literals. Then, the set of literals is satisfiable if and only if there is no disequality with both terms in the same class. A model can be built from the congruence closure, on a universe with cardinality equal to the number of classes in the congruence.

3 Congruence Closure in Practice

The algorithms we consider in the following take as input a set of literals E. Considering complexity, not only the cardinality of the set is important, but also the number of terms and subterms as well as the number of their occurrences. Congruence closure algorithms in modern SMT solvers typically represent terms with Directed Acyclic Graphs (DAGs) using maximal sharing, and not trees. The number of term and subterm occurrences does not matter, but only the number of distinct (sub)terms. The input is also typically not a set, but successive calls to an assertion function with a literal as argument: every repetition of the same literal then matters for complexity. Let us assume, however, that the input is a set E, terms are DAGs with maximal sharing (i.e. identity of atomic symbols and complex terms can be checked in constant time). Therefore, we characterize complexity results in terms of number of literals, terms and subterms of the input set, i.e. $|E|$ and $|\mathcal{T}(E)|$.

Since congruence relations are basically partitions of equivalent terms that additionally satisfy the compatibility property, it is unsurprising that practical congruence closure algorithms, or decision procedures for ground sets of first-order logic literals, are based on some kind of union-find data-structure. Terms (and subterms) are put into equivalence classes, according to the equalities in the input. The algorithms furthermore check, every time two classes of the partition are merged, whether any new equality induced by compatibility has to be taken into account. Also, it checks that the congruence is consistent with the set of disequalities. We refer the reader to [7,3,10] for more details. The complexity of those algorithms depend on the internal data-structures and on the representation of terms [3]. Algorithms typically implemented in SMT solvers have complexity $\mathcal{O}(|E| + (|\mathcal{T}(E)| \cdot \log |\mathcal{T}(E)|))$ assuming constant time operations on the hash table being used to detect new equalities induced by compatibility.

The generation of conflict sets or explanations is based on the congruence graph: its nodes are the terms and subterms considered by the algorithm. An edge in the graph is either a full edge, linking two nodes s and t and labeled by an input equation $s = t$, or a congruence edge (a dotted edge in the figures in this article), linking two terms with the same leading function symbol and labeled by the compatibility-deduced equality between both terms. The graph has a path between two terms if and only if they belong to the same congruence class. The equality between two terms in the same class is a logical consequence of the set of equations labeling the path. To get an explanation for the equality of two terms in the same class, that is, a set of input equations implying the equality of the two terms, it thus suffices to collect the set of equations labeling a path, and recursively replace any compatibility equation $g(t_1, \ldots, t_n) = g(s_1, \ldots, s_n)$ by the explanations of $t_1 = s_1, \ldots, t_n = s_n$.

Example 1 A congruence graph for two input equations $a = f(f(f(a)))$ and $a = f(f(f(f(f(a)))))$ is given on Figure 1. Labeling equations are omitted for simplicity. There is a path between a and $f(a)$, so both terms are equal if the

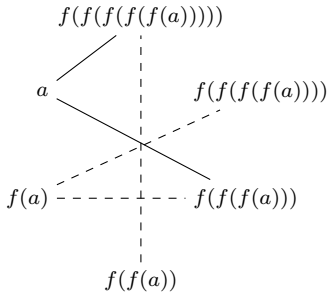


Fig. 1 An example congruence graph

input equations hold. To compute an explanation for $a = f(a)$, it suffices to collect the equalities on the path, that is, the input equation $a = f(f(f(a)))$ and the compatibility equation $f(a) = f(f(f(a)))$. This last equation should then be replaced by the equation between the arguments, i.e., $a = f(f(a))$ which is consequence, by transitivity, of another compatibility equation and of the other input equation $a = f(f(f(f(a))))$. Hence the explanation will contain both equations.

Practical congruence closure algorithms with explanation build a congruence graph while computing the congruence closure. Every time the decision procedure merges two classes, either because of an input equation or because an equality was deduced due to compatibility, a full- or congruence- edge is added to the graph. Since edges between nodes are only added when their respective congruence classes are merged, the path between two terms in the same class is unique. The explanation that two terms are equal is also unique, but there is no guarantee that this explanation is the smallest one. Indeed, it may happen that the algorithm considers, e.g. equations $a = b$ and $b = c$ before $a = c$, merging a, b and c before considering the last equation, and thus discarding $a = c$ as redundant: in that case, $a = c$ would have been the smallest proof that a and c are equal, but the congruence graph would only consider the two other equalities. There is not even a guarantee that the explanation is minimal. Again, the congruence closure algorithm can prove that $a = f(b)$ from the input equations $b = f(a)$, $f(a) = f(b)$ and $a = b$. The redundant equality $f(a) = f(b)$ would be recorded in the congruence graph, and thus be part of the explanation, if it is considered before $a = b$.

In practice, the congruence closure procedures implemented in SMT solvers produce explanations efficiently: the complexity of the explanation production is quasi-linear with respect to the explanation size, which is at most equal to the size of the input [10]. But the explanations are not optimal, i.e. they are not always the smallest. In fact, they are not even minimal. It is possible to compute minimal explanations in polynomial time; it suffices for instance to compute again the congruence closure iteratively removing every equation in the explanation, to see if it is redundant or not. One could (naively) hope to conceive a different congruence closure algorithm generating the smallest

explanation in polynomial time. For example, one might attempt to modify the iterative removal algorithm; or attempt to modify shortest path algorithms and apply them to congruence graphs enriched with redundant equations as labels. However, such attempts would be futile. As proven in the next section, the corresponding decision problem is NP-hard.

4 NP-Completeness of the Small Conflict Set Problem

The function problem of *generating* the smallest conflict set corresponds to the decision problem of *deciding* whether a conflict set with size smaller than a given k exists.

Definition 4 (Small conflict set problem) Given an unsatisfiable set E of literals in quantifier-free first-order logic with equality and $k \in \mathbb{N}$, the *small conflict set generation problem* is the problem of deciding whether there exists an unsatisfiable set $E' \subseteq E$ with $|E'| \leq k$.

If we had a polynomial-time algorithm α capable of generating the smallest conflict set for any unsatisfiable set E, then we could decide in polynomial time any instance of the small conflict set problem by applying α to E and checking whether $\alpha(E)$ has size smaller than k. However, as proven below, the small conflict set problem is NP-complete and, therefore, polynomial time generation of conflict sets with minimum size is not possible (unless P = NP). Our proof reduces the problem of deciding the satisfiability of a propositional logic formula in conjunctive normal form (SAT) to the small explanation problem.

Definition 5 (Small explanation problem) Given a set of equations $E = \{s_1 = t_1, \ldots s_n = t_n\}$, $k \in \mathbb{N}$ and a target equation $s = t$, the *small explanation problem* is the problem of answering whether there exists a set E' such that $E' \subseteq E$, $E' \models s = t$ and $|E'| \leq k$.

The small explanation problem and the small conflict set problem are closely related: there is a small explanation of size k of $s = t$ from E if and only if there is a small conflict of size $k + 1$ for $E \cup \{s \neq t\}$.

In the following we describe a polynomial translation from instances of the propositional satisfiability problem to instances of the small explanation problem. The translation consists of two parts: a translation of propositional formulas, here assumed, without loss of generality, to be in CNF (as shown in Definition 6), and a translation of assignments (as shown in Definition 7).

Definition 6 (CNF congruence translation) Let $\mathcal{C}$ be a set of propositional clauses $\{C_1, \ldots C_n\}$ using variables $x_1, \ldots, x_m$. The *congruence translation* $E_\mathcal{C}$ of $\mathcal{C}$ is defined as the set of equations

$$E_\mathcal{C} = \mathit{Connect} \cup \bigcup_{1 \leq i \leq n} \mathit{Clause}_i$$

with

$$\begin{aligned} \mathit{Connect} &= \{c'_i = c_{i+1} \mid 1 \leq i < n\} \\ \mathit{Clause}_i &= \{c_i = t_i(\hat{x}_j) \mid x_j \text{ appears in } C_i\} \\ &\cup \{t_i(\top_j) = c'_i \mid x_j \text{ appears positively in } C_i\} \\ &\cup \{t_i(\bot_j) = c'_i \mid x_j \text{ appears negatively in } C_i\} \end{aligned}$$

where $c_1, \ldots c_n, c'_1, \ldots c'_n, \hat{x}_1, \ldots \hat{x}_m, \top_1, \ldots \top_m, \bot_1, \ldots \bot_m$ are distinct constants, and $t_1, \ldots t_n$ are distinct unary functions.[2]

Remark 1 Note that the constants $\top_i$ and $\bot_i$ (for $1 \leq i \leq m$)) should not be confused with the Boolean values $\top$ and $\bot$. The intuitive relationship between these constants and the boolean values is established in Definition 7.

The translation of clauses is illustrated by the following example.

Example 2 Consider the set of clauses $\mathcal{C}$

$$\big\{C_1 = x_1 \vee x_2 \vee \neg x_3, C_2 = \neg x_2 \vee x_3, C_3 = \neg x_1 \vee \neg x_2\big\}.$$

Figure 2 represents the congruence translation of $\mathcal{C}$ graphically, an edge between two nodes meaning that the set contains an equation between the terms labeling the two nodes.

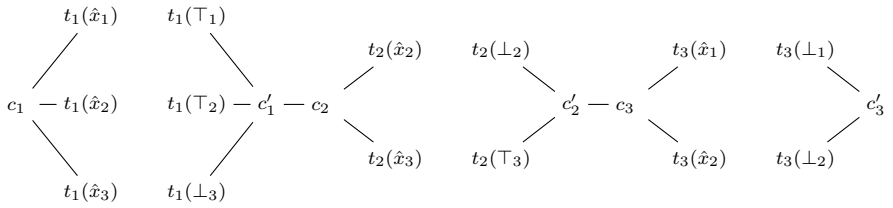


Fig. 2 The congruence translation $E_{\mathcal{C}} = \mathit{Connect} \cup \bigcup_{1 \leq i \leq n} \mathit{Clause}_i$ of $\mathcal{C}$.

Definition 7 (Assignment congruence translation) The *assignment congruence translation* $E_{\mathcal{I}}$ of an assignment $\mathcal{I}$ on propositional variables $x_1, \ldots, x_m$ is the set of equations

$$\begin{aligned} E_{\mathcal{I}} = \quad & \{\hat{x}_j = \top_j \mid 1 \leq j \leq m \text{ and } \mathcal{I} \models x_j\} \\ \cup\ & \{\hat{x}_j = \bot_j \mid 1 \leq j \leq m \text{ and } \mathcal{I} \models \neg x_j\} \end{aligned}$$

For convenience, we also define the set

$$\mathit{AssignmentEqs} = \{\hat{x}_j = \top_j, \hat{x}_j = \bot_j \mid 1 \leq j \leq m\}.$$

[2] It would be possible to define a translation without the c'_i constants, but they ease the presentation.

An assignment congruence translation is always a subset of *AssignmentEqs*. By extension, a subset of *AssignmentEqs* is said to be an assignment if it is the congruence translation of an assignment, that is, if it does not contain both $\hat{x}_j = \top_j$ and $\hat{x}_j = \bot_j$ for some j.

Example 3 (Example 2 continued) Consider the model $\mathcal{I} = \{x_1, \neg x_2, x_3\}$ of $\mathcal{C}$. Figure 3 gives a graphical representation of $E_\mathcal{I}$, whereas *AssignmentEqs* is represented in Figure 4. Notice that $E_\mathcal{C} \cup E_\mathcal{I} \models c_1 = c'_3$, and c_1 and c'_3 are connected in the congruence graph of $E_\mathcal{C} \cup E_\mathcal{I}$ (Figure 5), the path containing both full edges corresponding to equalities in $E_\mathcal{C} \cup E_\mathcal{I}$, and dotted edged corresponding to equalities due to the compatibility property of the congruence relation.

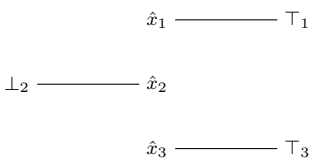


Fig. 3 Congruence translation of $\mathcal{I}$

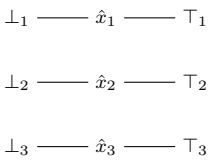


Fig. 4 *AssignmentEqs*

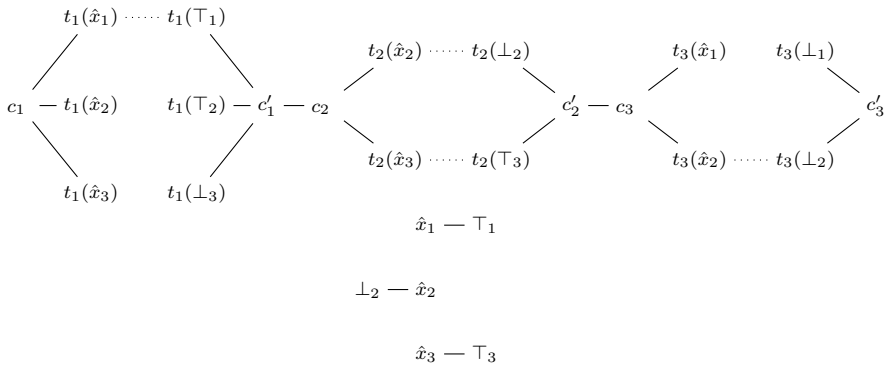


Fig. 5 The congruence graph for $E_\mathcal{C} \cup E_\mathcal{I}$

Lemma 1 *Consider a (partial or total) assignment $\mathcal{I}$ for non-tautological clauses $\mathcal{C} = \{C_1, \dots C_n\}$. Then $\mathcal{I} \models \mathcal{C}$ if and only if $E_\mathcal{I} \cup E_\mathcal{C} \models c_1 = c'_n$.*

Proof. Let the propositional variables in $\mathcal{C}$ be $x_1, \dots, x_m$.

($\Leftarrow$) Consider the congruence graph induced by $E_\mathcal{I} \cup E_\mathcal{C}$. Besides edges directly associated to equalities in the set, the only edges are congruence edges between terms $t_i(\hat{x}_j)$ and either $t_i(\top_j)$ or $t_i(\bot_j)$. So any path from c_1 to c'_n would go through such a congruence edge for each i. And such an edge exists for i if and only if the clause i is satisfied by $\mathcal{I}$.

($\Rightarrow$) If $\mathcal{I} \models \mathcal{C}$, then $\mathcal{I} \models C_i$ for each clause $C_i \in \mathcal{C}$. Assume $\mathcal{I}$ makes true a variable x_j, literal of C_i (the case of the negation of a variable is handled similarly). Then $E_\mathcal{I} \models t_i(\hat{x}_j) = t_i(\top_j)$, and $E_\mathcal{I} \cup \mathit{Clause}_i \models c_i = c'_i$. This is true for each i, and thanks to the equations in *Connect*, one can deduce using transitivity that $E_\mathcal{I} \cup E_\mathcal{C} \models c_1 = c'_n$. □

Lemma 2 *Consider a (partial or total) assignment $\mathcal{I}$ for non-tautological clauses $\mathcal{C} = \{C_1, \dots C_n\}$ on variables $x_1, \dots, x_m$. $|E_\mathcal{I} \cup E_\mathcal{C}|$ and $|\mathcal{T}(E_\mathcal{I} \cup E_\mathcal{C})|$ are polynomial in n and m.*

Proof. $E_\mathcal{I}$ contains at most m equations, since for no j both $\mathcal{I} \models x_j$ and $\mathcal{I} \models \neg x_j$. The set *Connect* contains exactly $n - 1$ equations. For every i, the set Clause_i contains at most $2m$ equations, resulting in $2mn$ equations for all clauses. In total, we thus have $|E_\mathcal{I} \cup E_\mathcal{C}| \leq n - 1 + m + 2mn$.

$E_\mathcal{I} \cup E_\mathcal{C}$ contains at most $2n + 3m + 3mn$ terms: $2n$ for c_i, c'_i, $3m$ for $\hat{x}_j, \top_j, \bot_j$ and $3mn$ for all possible combinations of $t_i(\hat{x}_j), t_i(\top_j), t_i(\bot_j)$. □

Considering again Example 3, and particularly Figure 5, any transitivity chain from c_1 to c'_3 will pass through c'_1, c_2, c'_2 and c_3. Any acyclic path from c_1 to c'_3 will contain 11 edges: 3 congruence edges, $3 * 2$ edges in Clause_i for $i = 1, 2, 3$ and 2 edges from *Connect*.

Since every interpretation $\mathcal{I}$ is such that $E_\mathcal{I} \subset \mathit{AssignmentEqs}$, one can try to relate the propositional satisfiability problem for a set of clauses $\mathcal{C} = \{C_1, \dots C_n\}$ to finding an explanation of $c_1 = c'_n$ in $\mathit{AssignmentEqs} \cup E_\mathcal{C}$. However, it is necessary that this explanation does not set $\hat{x}_j$ equal both to $\top_j$ and $\bot_j$, i.e. at most one of the two equations $\hat{x}_j = \top_j$ and $\hat{x}_j = \bot_j$ should be in the explanation. By restricting assignments to total ones, i.e. by enforcing that at least one of the two equations $\hat{x}_j = \top_j$ and $\hat{x}_j = \bot_j$ belongs to the explanation, it is also possible, with a single cardinality condition on the explanation size, to require that at most one of them belong to the explanation.

Lemma 3 *A set of non-tautological clauses $\mathcal{C} = \{C_1, \dots C_n\}$ using variables $x_1, \dots, x_m$ is satisfiable if and only if there is a set E' such that $E' \subseteq \mathit{AssignmentEqs} \cup E_{\mathcal{C}'}$, $E' \models c_1 = c'_{n+m}$ and $|E'| \leq 3n + 4m - 1$, where $\mathcal{C}'$ is $\mathcal{C}$ augmented with the tautological clauses $C_{n+i} = x_i \vee \neg x_i$ for $i = 1, \dots m$.*

Proof. ($\Rightarrow$) Consider a total model $\mathcal{I}$ for $\mathcal{C}$. We show that there is a set $E \subset E_{\mathcal{C}'}$, such that together with the congruence translation $E_\mathcal{I}$ of $\mathcal{I}$ it follows $E' = E \cup E_\mathcal{I} \models c_1 = c'_{n+m}$ and $|E'| \leq 3n + 4m - 1$.

The set $E_{\mathcal{I}}$ contains m equations, since it is the congruence translation of a total assignment.

For each clause C_i ($i = 1 \dots n+m$), there is a literal in C_i that is satisfied by the model $\mathcal{I}$. Let x_j be the variable of that literal.

Suppose $\mathcal{I} \models x_j$, then the set E contains equations $c_i = t_i(\hat{x}_j), t_i(\top_j) = c'_i$ of $Clause_i$. These equations are in $Clause_i$, because x_j is the satisfying literal of C_i, thus surely $x_j \in C_i$. From compatibility and the fact that $\hat{x}_j = \top_j \in E_{\mathcal{I}}$ it follows that $E \cup E_{\mathcal{I}} \models t_i(\hat{x}_j) = t_i(\top_j)$. Finally, from transitivity and the three equations $c_i = t_i(\hat{x}_j)$, $t_i(\hat{x}_j) = t_i(\top_j)$, $t_i(\top_j) = c'_i$ it follows that $E \cup E_{\mathcal{I}} \models c_i = c'_i$.

The case $\mathcal{I} \not\models x_j$ is symmetric, such that via equations $c_i = t_i(\hat{x}_j)$, $t_i(\hat{x}_j) = t_i(\bot_j)$, $t_i(\bot_j) = c'_i$, it follows $E \cup E_{\mathcal{I}} \models c_i = c'_i$.

In addition to 2 equations for each of the $(n+m)$ clauses, the set E contains all $n+m-1$ equations of *Connect*, that is $c'_i = c_{i+1}$ for $i = 1 \dots n+m-1$. From transitivity it follows that $E \cup E_{\mathcal{I}} \models c_1 = c'_{n+m}$.

In total, E contains $2(n+m)$ of the sets $Clause_i$, $n+m-1$ equations from *Connect* and m equations from $E_{\mathcal{I}}$, i.e. $|E| = 3n+4m-1$.

($\Leftarrow$) Suppose there is a set of equations $E' \subseteq \mathit{AssignmentEqs} \cup E_{\mathcal{C}'}$ such that $E' \models c_1 = c'_{n+m}$ and $|E'| \leq 3n+4m-1$. E' has to contain $2(n+m)$ equations from $Clause_i$ ($i = 1 \dots n+m$), that is one pair of equations $c_i = t_i(.)$ and $t_i(.) = c'_i$ for every clause, and $n+m-1$ equations from *Connect*, since by construction there is no other possibility to deduce $c_i = c'_i$. Furthermore, thanks to the tautological clauses, E' also has to contain at least $\hat{x}_j = \top_j$ or $\hat{x}_j = \bot_j$ for each $j \in \{1 \dots m\}$. Therefore, the cardinality condition $|E'| \leq 3n+4m-1$ and the fact that E' contains $3(n+m)-1$ equations from $Clause_i$ and *Connect*, requires that the E' contains at most one $\hat{x}_j = \top_j$ or $\hat{x}_j = \bot_j$ for each $j \in \{1 \dots m\}$. Therefore, we have that $E_{\mathcal{I}} = E' \cap \mathit{AssignmentEqs}$ is the congruence translation of an assignment and Lemma 1 guarantees the existence of a model for $\mathcal{C}'$, or equivalently for the original set of clauses $\mathcal{C}$. □

Example 4 In Lemma 3, the input formula is augmented with tautological clauses. We demonstrate here the necessity of these extra clauses on the unsatisfiable formula $\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$.

Figure 6 shows the congruence translation of φ together with a subset of *AssignmentEqs* that yields an explanation for $c_1 = c'_4$. This explanation picks, besides the necessary equations from the clause and connect parts, two equations from the *AssignmentEqs* part. However, this explanation maps x_1 to $\bot$ and $\top$ at the same time, and hence cannot correspond to a (consistent) assignment. With the addition of tautological clauses and because the number of equations in the explanation is upper bounded, spurious explanations of this kind are ruled out. This is illustrated in Figure 7, depicting the congruence translation of φ conjoined with the tautological clauses $(x_1 \vee \neg x_1)$ and $(x_2 \vee \neg x_2)$, together with the same subset of *AssignmentEqs* used in Figure 6. As desired, this subset is not an explanation of $c_1 = c'_6$, since the transitivity chain stops at $t_6(\hat{x}_2)$, x_2 being unassigned. In fact, in this congruence graph, there is no explanation of $c_1 = c'_6$ with less than 19 equations. This is

as expected, since φ is unsatisfiable and $3n+4m-1 = 19$ in our example with $n = 4$ clauses and $m = 2$ variables.

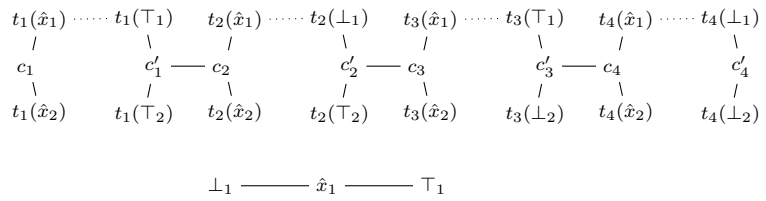


Fig. 6 The congruence translation of φ and a spurious short explanation.

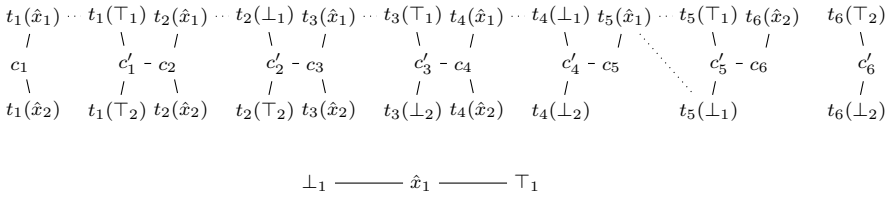


Fig. 7 The congruence translation of φ with tautological clauses.

Corollary 1 (NP-hardness) *The small explanation problem is NP-hard.*

Proof. Propositional satisfiability is NP-hard, and can be reduced in polynomial time to the small explanation problem. □

Lemma 4 (NP) *The small explanation problem is in NP.*

Proof. Let E be a set of equations and $s = t$ be a target equation. A solution to the explanation problem for some $k \in \mathbb{N}$ is a subset $E' \subseteq E$, such that $|E'| \leq k$. Let $n = |\mathcal{T}(E)| + |E|$ and $n' = |\mathcal{T}(E')| + |E'|$. We have $n' \leq n$, since $E' \subseteq E$ and every term in E' appears also in E. Checking whether E' is an explanation of $s = t$ can be done by computing its congruence closure, which is possible in polynomial time in n' [7] and thereby also in n. □

Theorem 1 (Small explanation NP-completeness) *The small explanation problem is NP-complete.*

Proof. By corollary 1 and lemma 4. □

Theorem 2 (Small conflict NP-completeness) *The small conflict set problem is NP-complete.*

Proof. The small conflict set problem is at least as hard as the small explanation problem since the small explanation problem has been showed to be reducible to the small conflict set problem. It is also in NP for exactly the same reason that the small explanation problem is. □

5 Conclusion

The conflict set generation feature of congruence algorithms is essential for practical SMT solving. Although one could argue that the important property of the generated conflicts is minimality (i.e. no useless literal is in the conflict), it is also interesting to consider producing the smallest conflict. We have shown that the problem of deciding whether a conflict of a given size exists is NP-complete. Therefore, it is generally intractable to obtain the smallest conflict.

In [6,8,9], methods to obtain small conflicts, but not necessarily the smallest, are discussed. In practice, it pays off to prioritize speed of the congruence closure algorithm and conflict generation over succinctness of conflicts. However, other applications sensitive to proof size may benefit from other methods prioritizing small conflict size, at a cost of less efficient solving. Thanks to the NP-completeness, one option could be to iteratively encode the small conflict problem into SAT, and use a SAT-solver to find successively smaller conflicts, until the smallest is found. Perhaps an encoding of the problem can be found that differentiates between hard constraints representing relevant instantiations of the axioms of equality as well as the target equation, and soft constraints representing the inclusion of input equations to an explanation. In that case, Max-SAT solvers could be used to find small explanations, in order to leverage efforts that combine decision procedures and optimization techniques.

Acknowledgment. We would like to thank Robert Nieuwenhuis and Ashish Tiwari for discussions and some preliminary ideas that led us to this proof. We are grateful to the anonymous reviewers of this paper and of [4] for their comments.

References

1. Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.
2. Joseph Boudou, Andreas Fellner, and Bruno Woltzenlogel Paleo. Skeptik: A proof compression system. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 8562 of *Lecture Notes in Computer Science*, pages 374–380. Springer, 2014.
3. Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, October 1980.
4. Andreas Fellner, Pascal Fontaine, Georg Hofferek, and Bruno Woltzenlogel Paleo. NP-completeness of small conflict set generation for congruence closure. In Vijay Ganesh and Dejan Jovanović, editors, *International Workshop on Satisfiability Modulo Theories (SMT)*, 2015.
5. Pascal Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, PhD thesis, Institut Montefiore, Université de Liege, Belgium, 2004.
6. Pascal Fontaine and E. Pascal Gribomont. Using BDDs with combinations of theories. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2514 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 2002.

7. Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.
8. Robert Nieuwenhuis and Albert Oliveras. Union-find and congruence closure algorithms that produce proofs. In Cesare Tinelli and Silvio Ranise, editors, *Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*, 2004.
9. Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Rewriting Techniques and Applications (RTA)*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.
10. Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.

Integrating SMT solvers in Rodin ☆

David Déharbe[a], Pascal Fontaine[b], Yoann Guyot[c], Laurent Voisin[d]

[a] *Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil*
[b] *University of Lorraine, Loria, Inria, France*
[c] *Cetic, Belgium*
[d] *Systerel, France*

Abstract

Formal development in Event-B generally requires the validation of a large number of proof obligations. Some tools automatically discharge a significant part of them, thus augmenting the efficiency of the formal development. We here investigate the use of SMT (Satisfiability Modulo Theories) solvers in addition to the traditional tools, and detail the techniques used for the cooperation between the Rodin platform and SMT solvers.

Our contribution is the definition of a translation of Event-B proof obligations to the language of SMT solvers, its implementation in a Rodin plug-in, and an experimental evaluation on a large sample of industrial and academic projects. On this domain, adding SMT solvers to Atelier B provers reduces significantly the number of sequents that need to be proved interactively.

Keywords: Formal methods, Event-B, SMT solving

1. Introduction

The Rodin platform [10] is an integrated design environment for the formal modeling notation Event-B [2]. Rodin is based on the Eclipse framework [25] and has an extensible architecture, where new features, or new versions of existing features, can be integrated by means of plug-ins. It supports the construction of formal models of systems as well as their refinement using the notation of Event-B, based on first-order logic, typed set theory and integer arithmetic. Event-B models should be consistent; for this purpose, Rodin generates proof obligations that need to be discharged (i.e., proved valid).

☆This work is partly supported by the project ANR-13-IS02-0001-01, the STIC Am-Sud project MISMT, CAPES grant BEX 2347/13-0, CNPq grants 308008/2012-0 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, `www.ines.org.br`), and EU funded project ADVANCE (FP7-ICT-287563).

Email addresses: `david@dimap.ufrn.br` (David Déharbe), `Pascal.Fontaine@inria.fr` (Pascal Fontaine), `yoann.guyot@cetic.be` (Yoann Guyot), `laurent.voisin@systerel.fr` (Laurent Voisin)

The proof obligations are represented internally as sequents, and a sequent calculus forms the basis of the verification machinery. Proof rules are applied to a sequent and produce zero, one or more new, usually simpler, sequents. A proof rule producing no sequent is called a discharging rule. The goal of the verification is to build a proof tree corresponding to the application of the proof rules, where all the leaves are discharging rules. In practice, the proof rules are generated by so-called *reasoners*. A reasoner is a plug-in that can either be standalone or use existing verification technologies through third-party tools. Most of the time, sequents are not amenable to finite domain encoding, and engines such as model checkers are not appropriate reasoners.

The usability of the Rodin platform, and of formal methods in general, greatly depends on several aspects of the verification activity:

Automation Ideally, the validity status of proof obligations is computed automatically by reasoners. If human interaction is required for discharging valid proof obligations (using an interactive theorem prover), productivity is negatively impacted.

Information Validation of proof obligations should not be sensitive to irrelevant modifications of the model. When modifying the model, large parts of the proof can be preserved if the precise facts used to validate each proof obligation are recorded. It is important that the reasoners are able to provide such sets of relevant facts, since they can then be used to automatically construct new proof rules to be stored and tried for after model changes. Also, other sequents (valid for the same reason) may be discharged by these rules without requiring another call to the reasoner.

In addition, when reasoners are able to generate counter-examples of failed proof obligations, this information can be very valuable to the user as hints to improve the model and the invariants.

Trust When a prover is used, either the tool itself or its results need to be certified; otherwise the confidence in the formal development is jeopardized.

In this paper, we address the application of a verification approach that may potentially fulfill these three requirements: *Satisfiability Modulo Theory* (SMT) solvers. SMT solvers can *automatically* handle large formulas of first-order logic with respect to some background theories, or a combination thereof, such as different fragments of arithmetic (linear and non-linear, integer and real), arrays, bit vectors, etc. They have been employed successfully to handle proof obligations with tens of thousands of symbols stemming from software and hardware verification. This paper extends the work presented in [15], and provides details of a translation of Event-B sequents to SMT input. The difficulty essentially lies in the way sets are translated. We here propose two approaches to tackle this challenge. Notice that these approaches could also be applied to other set-based formalisms such as the B method [1], TLA+ [20, 19], VDM [16] and Z [27].

The SMT-LIB initiative provides a standard for the input language of SMT solvers, and, in its last version [5], a command language defining a common

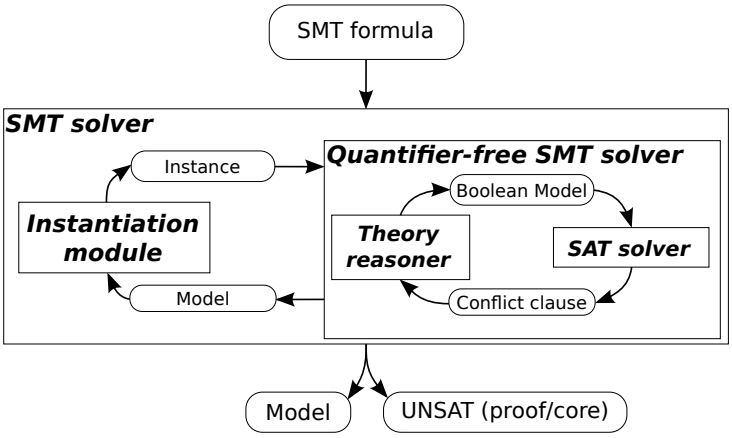


Figure 1: Schematic view of an SMT solver.

interface to interact with SMT solvers. We implemented a Rodin plug-in using this interface. The plug-in also extracts from the SMT solvers some additional *information* such as the relevant hypothesis. Some solvers (e.g. Z3 [12] and veriT [9]) are able to generate a comprehensive proof for validated formulas, which can be verified by a *trusted* proof checker [3]. In the longer term, besides automation, and information, trust may be obtained using a centralized proof manager. The plug-in is open-source, distributed under the same license as Rodin, and its source code is available in the main Rodin repository. The plug-in is easily installable by users through the update manager of the Rodin platform.

Overview. We start in Section 2 by giving some insights on the techniques employed in SMT solvers. Section 3 presents the translation of Rodin sequents to the SMT-LIB notation. Section 4 illustrates the approach through a simple example. Section 5 presents experimental results, based on the verification activities carried out for a variety of Event-B projects. We conclude by discussing future work.

Throughout the paper, formulas are expressed using the Event-B syntax [21], and sentences in SMT-LIB are typeset using a `typewriter` font.

2. Solving SMT formulas

In this section, we provide some insight about the internals of SMT solvers, in order to give to the reader an idea of the kind of formulas that can successfully be handled by SMT solvers. A very schematic view of an SMT solver is presented in Figure 1. Basically it is a decision procedure for quantifier-free formulas in a rich language coupled with an instantiation module that handles the quantifiers in the formulas by grounding the problem. For quantified logic, SMT solvers

are of course not decision procedures anymore, but they work well in practice if the necessary instances are easy to find and not too numerous.

We refer to [4] for more information about the techniques described in this section and SMT solving in general. There are several SMT solvers supporting quantifiers; the plug-in described in this paper makes use of Alt-Ergo [8], CVC3 [6], Z3 [12], and veriT [9]. This last solver is developed by two of the authors of this paper.

2.1. Quantifier-free formulas

Historically, the first goal of SMT solvers was to provide efficient decision procedures for expressive languages, beyond pure propositional logic. Those solvers have always been based on a cooperation of a Boolean engine, nowadays typically a SAT solver (see [7] for more information on SAT solver techniques and tools), and a theory reasoner to check the satisfiability of a set of literals in the considered language. The Boolean engine generates models for the Boolean abstraction of the input formula, whereas the theory reasoner refutes the sets of literals corresponding to these abstract models by conjunctively adding conflict clauses to the propositional abstraction. This exchange runs until either the Boolean abstraction is sufficiently refined for the Boolean reasoner to conclude that the formula is unsatisfiable, or the theory reasoner concludes that the abstract model indeed corresponds to a model of the formula.

The theory reasoners are themselves based on a combination of decision procedures for various fragments. In our context, the relevant decision procedures are congruence closure — to handle uninterpreted predicates and functions — decision procedure for arrays (typically reduced to some kind of congruence closure), and linear arithmetic. It is possible, using the Nelson-Oppen combination method [22, 26], to build a decision procedure for the union of the languages. The theory reasoner used in most SMT solvers is thus able to decide the satisfiability of literals on a language containing a mix of uninterpreted symbols, linear arithmetic symbols, and array operators.

For the theory reasoner and the SAT solver to cooperate successfully, some further techniques are necessary. Among these techniques, if a set of literals is found unsatisfiable, it is most valuable to generate small conflict clauses, in order to refine the Boolean abstraction as strongly as possible. Models of the propositional abstraction are checked for satisfiability while they are being built, so that unsatisfiability can be detected early. Finally, theory propagation, in which the theory reasoner provides hints for the SAT solver decisions, has proved to be very worthwhile in practice.

2.2. Instantiation techniques

Within SMT solvers, solving formulas with quantifiers is done by reduction to quantifier-free formulas, using instantiation. Indeed, formula $\forall \mathbf{x}\, \varphi(\mathbf{x})$ stands for a conjunction over all combinations of values for $\mathbf{x}$. Any formula of the form $\forall \mathbf{x}\, \varphi(\mathbf{x}) \Rightarrow \varphi(\mathbf{t})$, for any terms $\mathbf{t}$, can be added conjunctively to the input without changing its truth value. To show that a formula is unsatisfiable, it

is thus sufficient to find the right instances of quantified formulas to add to the input. In that context, even if the SMT solver abstracts $\forall \mathbf{x}\, \varphi(\mathbf{x})$ to a Boolean proposition, it is able to reason about the formulas with quantifiers. The quantifier instantiation module is responsible for producing lemmas of the form $\forall \mathbf{x}\, \varphi(\mathbf{x}) \Rightarrow \varphi(\mathbf{t})$. Automatically finding the right instances of quantified formulas is a key issue for the verification of sequents (as well as proof obligations produced in the context of a number of software verification tools). Generating too many instances may overload the solver with useless information and exhaust computing resources. Generating too few instances will result in an "unknown", and useless, verdict. Handling quantifiers within SMT solvers is still a very active research subject, and the methods to handle quantifiers vary greatly from one solver to another. We report here how veriT copes with quantified formulas. Several instantiation techniques are applied in turn: trigger-based, sort-based and superposition techniques.

The trigger-based and sort-based instantiation techniques are applied to top-most quantifiers, that is, to quantifiers that are not themselves in the scope of other quantifiers. Remember that, when checking the satisfiability of formulas, existential quantifiers with positive polarity and universal quantifiers with negative polarity can be eliminated by Skolemization. This satisfiability preserving transformation replaces suitable quantified variables by witnesses, introducing new uninterpreted symbols (constants or functions). In veriT, Skolemization automatically occurs for top-most quantifiers, whereas Skolemizable quantifiers in the scope of non-Skolemizable quantifiers are not eliminated. As a consequence, only Skolem constants are introduced, and no Skolem functions. Instantiation will remove top-most non-Skolemizable quantifiers, some quantifiers in the instance may then become top-most Skolemizable quantifiers in the process, and are in turn eliminated with the introduction of Skolem constants. This strategy is effective in practice.

In a quantified formula $Q\mathbf{x}\, \varphi(\mathbf{x})$, a trigger is a set of terms $T = \{t_1, \cdots t_n\}$ such that the free variables in T are the quantified variables $\mathbf{x}$ and each t_i is a sub-term of the matrix $\varphi(\mathbf{x})$ of the quantified formula. Trigger-based instantiation consists in finding, in the formula, sets of ground terms T' that match T, i.e., such that there is a substitution σ on $\mathbf{x}$, where the homomorphic extension of σ over T yields T'. Each such substitution defines an instantiation of the original quantified formula. Some verification systems allow the user to specify instantiation triggers. This is not the case in Rodin, and veriT applies heuristics to annotate quantified formulas with triggers.

If the trigger-based approach does not yield any new instance, veriT falls back to sort-based instantiation. All ground terms in the formula are collected, and each quantified formula is instantiated with every term that has the same type as the quantified variable.

Finally, veriT also features a module to communicate with a superposition-based first-order logic automated theorem prover, namely the E prover [24]. It is built upon automated deduction techniques such as rewriting, subsumption, and superposition and is capable of identifying the unsatisfiability of a set of quantified and non-quantified formulas. When such a set is found satisfiable,

lemmas are extracted from its output and communicated to the other reasoning modules of veriT. The E prover, like many saturation-based first-order provers, is complete for first-order logic with equality.

2.3. Unsatisfiable core extraction

Additionally to the satisfiability response, it is possible, in case of an unsatisfiable input, to ask for an *unsatisfiable core*. It may indeed be very valuable to know which hypotheses are necessary to prove a goal in a verification condition. For instance, the sequent (1) discussed in Section 4 and translated into the SMT input in Figure 6 is valid independently of the assertion labeled `grd1`; the SMT input associates labels to the hypotheses, guards, and goals, using the reserved SMT-LIB annotation operator `!`. A solver implementing the SMT-LIB unsatisfiable core feature could thus return the list of hypotheses used to validate the goals. In the case of the example in Figure 6, the guard is not necessary to prove unsatisfiability, and would therefore not belong to a good unsatisfiable core.

Recording unsatisfiable cores for comparison with new proof obligations is particularly useful in our context. Indeed, users of the Rodin platform will want to modify their models and their invariants, resulting in a need of validating again proof obligations mostly but not fully similar to already validated ones. If the changes do not impact the relevant hypotheses and goal of a proof obligation, comparison with the (previous) unsatisfiable core will discharge the proof obligation and the SMT solver will not need to be run again. Also a same unsatisfiable core is likely to discharge similar proof obligations, for instance generated for a similar transition, but differing for the guard.

The unsatisfiable core production for the veriT solver is related to the proof production feature. The solver is indeed able to produce a proof, and it has moreover a facility to prune the proof of unnecessary proof steps and hypotheses. It suffices thus to check the pruned proof and collect all hypotheses in that proof to obtain a super-set of the unsatisfiable core that is often minimal in practice. Other approaches for unsatisfiable core extraction for SMT are presented and discussed in [4].

3. Translating Event-B to SMT

Figure 2 gives a schematic view of the cooperation framework between Rodin and the SMT solver. Within the Rodin platform, each proof obligation is represented as a sequent, i.e. a set of hypotheses and a conclusion. These sequents are discharged using Event-B proof rules. Our strategy to prove an Event-B sequent is to build an SMT formula, call an SMT solver on this formula, and, on success, introduce a new suitable proof rule. This strategy is presented as a *tactic* in the Rodin user interface. Since SMT solvers answer the *satisfiability* question, it is necessary to take the negation of the sequent (to be validated) in order to build a formula to be refuted by the SMT solver. If the SMT solver does not implement unsatisfiable core generation, the proof rule will assert that the full Event-B sequent is valid (and will only be useful for that specific sequent). Otherwise an unsatisfiable core — i.e., the set of facts necessary to

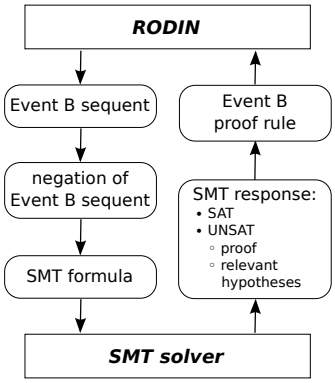


Figure 2: Schematic view of the interaction between Rodin and SMT solvers.

prove that the formula is unsatisfiable — is supplied to Rodin, which will extract a stronger Event-B proof rule containing only the necessary hypotheses. This stronger proof rule will hopefully be applicable to other Event-B sequents. If, however, the SMT solver is not successful, the application of the tactic has failed and the proof tree remains unchanged.

The SMT-LIB standard proposes several "logics" that specify the interpreted symbols that may be used in the formulas. Currently, however, none of these logics fits exactly the language of the proof obligations generated by Rodin. There exists a proposal for such a logic [18], but the existing SMT solvers do not yet implement corresponding reasoning procedures. Our pragmatic approach is thus to identify subsets of the Event-B logics that may be handled by the current tools, either directly or through some simple transformations. The translation takes as input the Event-B proof obligations. The representation of proof obligations is such that each identifier has been annotated with its type. In the type system, integers and Booleans are predefined, and the user may create new basic sets, or compose existing types with the powerset and Cartesian product constructors. Translating Boolean and arithmetic constructs is mostly straightforward, since a direct syntactic translation may be undertaken for some symbols: Boolean operators and constants, relational operators, and most of arithmetic (division and exponentiation operators are currently translated as uninterpreted symbols). As an example of transformation of an Event-B sequent to an SMT formula, consider the sequent with goal $0 < n + 1$ under the hypothesis $n \in \mathbb{N}$; the type environment is $\{n \oftype \mathbb{Z}\}$ and the generated SMT-LIB formula is:

```
(set-logic AUFLIA)
(declare-fun n () Int)
(assert (>= n 0))
(assert (not (< 0 (+ n 1))))
```

```
(check-sat)
```

The main issue in the translation of proof obligations to SMT-LIB is the representation of the set-theoretic constructs. We present successively two approaches. The simplest one, presented shortly for completeness, is based on the representation of sets as characteristic predicates [13]. Since SMT solvers handle first-order logic, this approach does not make it possible to reason about sets of sets. The second approach removes this restriction. It uses the *ppTrans* translator, already available in the Rodin platform; this translator removes most set-theoretic constructs from proof obligations by systematically expanding their definitions.

3.1. The λ-based approach

This approach implements and extends the principles proposed in [13] to handle simple sets. Essentially, a set is identified with its characteristic function. For instance the singleton $\{1\}$ is identified with $(\lambda x \oftype \mathbb{Z} \mid x = 1)$ and the empty set is identified with the polymorphic λ-expression $(\lambda x \oftype X \mid \text{FALSE})$, where X is a type variable. The union of (two) sets is a polymorphic higher-order function $(\lambda(S_1 \oftype X \to \text{BOOL}) \mapsto (S_2 \oftype X \to \text{BOOL}) \mid (\lambda x \oftype X \mid S_1(x) \lor S_2(x)))$, etc.

SMT-LIB does not provide a facility for λ-expressions, and has limited support for polymorphism. This approach requires several extensions to SMT-LIB: λ-expressions, a polymorphic sort system, and macro-definitions. Those extensions are actually implemented in the veriT parser. Consider the sequent $A \oftype \mathbb{P}(\mathbb{Z}) \vdash A \cup \{a\} = A$, the translation to this extended SMT-LIB language produces:

```
(declare-fun A (Int) Bool)
(declare-fun a () Int)
(define-fun (par (X) (union ((S1 (X Bool)) (S2 (X Bool))) (X Bool)
             (lambda ((x X)) (or (S1 x) (S2 x))))))
(define-fun enum ((x Int)) Bool (= x a))
(assert (not (= (union A enum) A)))
(check-sat)
```

where `X` denotes a sort variable. The function definitions `union` and `enum` are inserted by the translator. The former is part of a corpus of definitions for most of the set-theoretic constructs (see [13, 14] for details). The latter is created on-the-fly by the translator to denote the set $\{a\}$. Both definitions are composed of a list of sorted parameters, the sort of the result, and the body expressing the value of the result. The macro processor implemented in veriT transforms this goal to

```
(not (forall ((x Int)) (iff (or (A x) (= x a)) (A x))))
```

i.e., a first-order formula that may then be handled using usual SMT solving techniques. It is also possible to use veriT only as a pre-processor to produce plain SMT-LIB formulas that are amenable to verification using any SMT-LIB compliant solver.

$$
\begin{array}{rcl}
P & ::= & P \Rightarrow P \mid P \equiv P \mid P \land \cdots \land P \mid P \lor \cdots \lor P \mid \\
 & & \neg P \mid \forall L \cdot P \mid \exists L \cdot P \mid \\
 & & A = A \mid A < A \mid A \leq A \mid M \in S \mid B = B \mid I = I \\
L & ::= & I \cdots I \\
I & ::= & \mathit{Name} \\
A & ::= & A - A \mid A \text{ div } A \mid A \text{ mod } A \mid A \exp A \mid \\
 & & A + \cdots + A \mid A \times \cdots \times A \mid -A \mid I \mid \mathit{IntegerLiteral} \\
B & ::= & \mathsf{true} \mid I \\
M & ::= & M \mapsto M \mid I \mid \mathsf{integer} \mid \mathsf{bool} \\
S & ::= & I
\end{array}
$$

Figure 3: Grammar of the language produced by *ppTrans*. The non-terminals are P (predicates), L (list of identifiers), I (identifiers), A (arithmetic expressions), B (Boolean expressions), M (maplet expressions), S (set expressions).

As already mentioned, the main drawback of this approach is that sets of sets cannot be handled. It is thus restricted to simple sets and relations. Furthermore its reliance on extensions of the SMT-LIB format creates a dependence on veriT as a macro processor. The next approach lifts these restrictions.

3.2. The ppTrans approach

Our second approach uses the translator *ppTrans* provided by the *Predicate Prover* available in Rodin [17]. This tool translates an Event-B formula to an equivalent formula in a subset of the Event-B mathematical language. The grammar of this subset is shown in Figure 3. Note that the sole set-theoretic symbol is the membership predicate. In addition, the translator performs *decomposition* of binary relations and *purification*, namely it separates arithmetic, Boolean and set-theoretic terms. Finally *ppTrans* performs basic Boolean simplifications on formulas. In the following, we provide details on those transformations, using the notation $\varphi \rightsquigarrow \varphi'$ to express that the formula (or sub-term) φ is rewritten to φ'. Not only does this approach make the plug-in independent of veriT, but it is also more general with respect to the translation of relations and functions. However, in the class of formulas suitable for the λ-based approach, *ppTrans* would produce similar results.

Maplet-hiding variables. The rewriting system implemented in *ppTrans* cannot directly transform identifiers that are of type Cartesian product. In a preprocessing phase, such identifiers are thus decomposed, so that further rewriting rules may be applied. This decomposition introduces fresh identifiers of scalar type (members of some given set, integers or Booleans) that name the components of the Cartesian product. Technically, this pre-processing is as follows. We assume the existence of an attribute $\mathcal{T}$, such that $\mathcal{T}(e)$ is the type of expression e. Also, let $\mathsf{fv}(e)$ denote the free identifiers occurring in expression e. The decomposition of the Cartesian product identifiers is specified, assuming an unlimited supply of fresh identifiers (e.g. x_0, x_1,...), using the following two

definitions ∇ and ∇^T:

$$\begin{array}{rcl} \nabla(i) & = & \begin{cases} \nabla^T(\mathcal{T}(i)) & \text{if } i \text{ is a product identifier,} \\ i & \text{otherwise.} \end{cases} \\ \nabla^T(T) & = & \begin{cases} \nabla^T(T_1) \mapsto \nabla^T(T_2) & \text{if } \exists T_1, T_2 \cdot T = T_1 \times T_2, \\ \text{a fresh identifier } x_i & \text{otherwise.} \end{cases} \end{array}$$

For instance, assume $x \oftype \mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z})$; then $\nabla(x) = x_0 \mapsto (x_1 \mapsto x_2)$ and $\mathsf{fv}(\nabla(x)) = \{x_0, x_1, x_2\}$ are fresh identifiers.

The pre-processing behaves as follows:

- Quantified sub-formulas $\forall x \cdot \varphi(x)$, such that x is a product identifier, are rewritten to
$$\forall \mathsf{fv}(\nabla(x)) \cdot \varphi[\nabla(x)/x],$$
where $e[e'/x]$ denotes expression e where expression e' has been substituted for all free occurrences of x.

  Ex. $\forall a \cdot a = 1 \mapsto (2 \mapsto 3) \rightsquigarrow \forall a_0, a_1, a_2 \cdot a_0 \mapsto (a_1 \mapsto a_2) = 1 \mapsto (2 \mapsto 3)$.

- Let ψ denote the top-level formula and let $x_1 \dots x_n$ be the free Cartesian product identifiers of ψ. Then:
$$\begin{array}{rl} \psi \rightsquigarrow & \forall \mathsf{fv}(\nabla(x_1)) \cdots \mathsf{fv}(\nabla(x_n)) \cdot \\ & \quad (x_1 = \nabla(x_1) \land \cdots x_n = \nabla(x_n)) \Rightarrow \psi[\nabla(x_1)/x_1] \cdots [\nabla(x_n)/x_n]. \end{array}$$

  Ex. $\psi \equiv a = b \land a \in S$ with typing $\{a \oftype S, b \oftype S, S \oftype \mathbb{P}(\mathbb{Z} \times \mathbb{Z})\}$:

$$\begin{array}{rl} \psi \rightsquigarrow & \forall x_0, x_1, x_2, x_3 \cdot \\ & \quad (a = x_0 \mapsto x_1 \ \land \ b = x_2 \mapsto x_3) \Rightarrow \\ & \quad\quad (x_0 \mapsto x_1 = x_2 \mapsto x_3 \land x_0 \mapsto x_1 \in S) \end{array}$$

Purification. The goal of this phase is to obtain pure terms, i.e. terms that do not mix symbols of separate syntactic categories: arithmetic, predicate, set, Boolean, and maplet symbols. This is done by introducing new variables. In Event-B, heterogeneous terms result from the application of symbols with a signature with different sorts (e.g. symbol $\subseteq$ yields a predicate from two sets). This phase also eliminates some syntactic sugar. Figure 4 depicts the different syntactic categories, how the Event-B operators relate them, and the effect of desugarization. There is an arrow from category X to category Y if a term from X may have an argument in Y. For instance $. \in$ labels the arrow from P to A since the left argument of $\in$ may be an arithmetic term, e.g. in $x + y \in S$.

First, let us introduce informally the notation $Q^? P[e^*]$, where Q is $\forall$ or $\exists$, P a predicate, and e an expression in P such that the syntactic category of e is not the same as that of its parent (identifiers are considered to belong to all syntactic categories). This denotes the possible introduction of the quantifier Q on a fresh variable, so that heterogeneous sub-terms in e are purified, yielding e^*, as illustrated by the following examples:

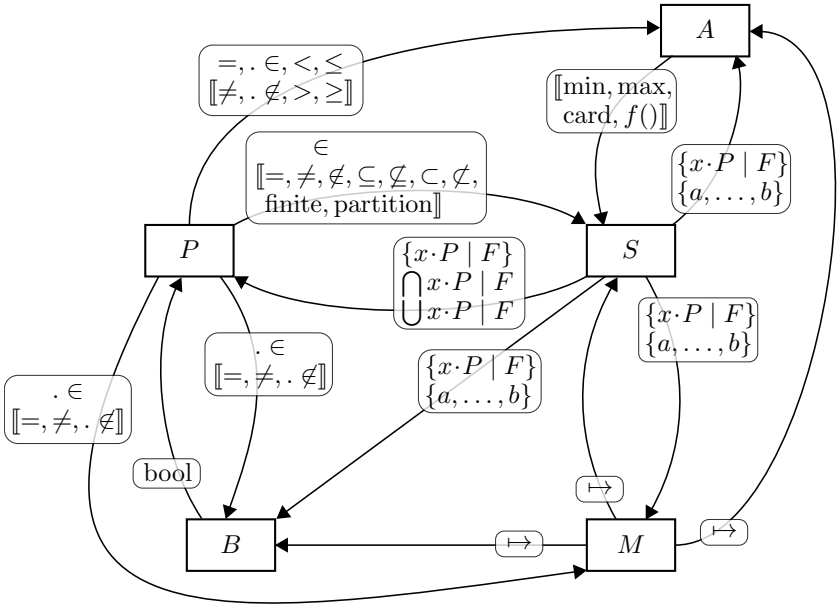


Figure 4: The different syntactic categories and the symbols relating them: A for arithmetic expressions, P for predicates, S for set expressions, B for Boolean expressions and M for maplet expressions. *ppTrans* removes all occurrences of the constructs delimited by double-brackets.

1. $\exists^?(a \mapsto (1 \mapsto 2))^* \in S$ represents $\exists x_0, x_1 \cdot x_0 = 1 \wedge x_1 = 2 \wedge a \mapsto (x_0 \mapsto x_1) \in S$ as 1 and 2 are not in the same syntactic category as the maplet.
2. $\forall^?(a \mapsto b)^* \in S$ does not introduce a quantification and denotes $a \mapsto b \in S$.

Appendix A presents the rewriting rules implemented in *ppTrans*. For readability, they are grouped thematically and the presentation order is not the same as the rewriting order. The symbols relating the syntactic categories P (predicates) and S (sets) are reduced to membership ($\in$) and equality ($=$) by application of the rules A.1–A.8 (see Appendix A.1). Moreover rules A.1 and A.2 are also applied when the arguments belong to other syntactic categories and are responsible for the elimination of all the occurrences of symbols $\neq$ and $\notin$. Applications of the equality symbol between syntactic categories S, M and B are removed by rules A.9–A.18 (Appendix A.2). Due to the symmetry property of equality, *ppTrans* also applies a symmetric version of each such rule. The symbols that embed arithmetic terms are taken care of with the rules in Appendix A.3. Rules A.19–A.21 first perform purification, and are followed by the application of rules A.22–A.29, responsible for the elimination of the symbols. Appendix A.4 contains rules to rewrite applications of the set membership symbol according to the rightmost argument. Rules A.30–A.39 expand the definitions for the different kinds of relation symbols. Rules A.40–A.61

handle miscellaneous other cases. Finally, Appendix A.5 presents the rules to rewrite applications of the set membership symbol according to both arguments, where the first argument is always a maplet. Again, through the application of rules A.62 to A.77, several symbols may be eliminated from the proof obligations.

All the rules in Appendix A are either sound purification rules, or the equivalence of the left and right side terms can easily be derived from the definitions (see [1]) of the eliminated symbols. Purification rules (Rules A.17, A.19 – A.21, A.44) eliminate heterogeneous terms and are only applied once. It is not difficult to order all other rules such that no eliminated symbol is introduced in subsequent rules. The rewriting system is thus indeed terminating.

Output to SMT-LIB format. Once *ppTrans* has completed rewriting, the resulting proof obligation is ready to be output in SMT-LIB format. The translation from *ppTrans*' output to SMT-LIB follows specific rules for the translation of the set membership operator. For instance assume the input has the following typing environment and formulas:

Typing environment	Formulas
$a \oftype S$	
$b \oftype T$	$a \in A$
$c \oftype U$	$a \mapsto b \in r$
$A \oftype \mathbb{P}(S)$	$a \mapsto b \mapsto c \in s$
$r \oftype \mathbb{P}(S \times T)$	
$s \oftype \mathbb{P}(S \times T \times U)$	

Firstly, for each basic set found in the proof obligation, the translation produces a sort declaration in SMT-LIB. However, as there is currently no logic in the SMT-LIB with powerset and Cartesian product sort constructors, *ppTrans* handles them by producing an additional sort declaration for each combination of basic sets (either through powerset or Cartesian product). Translating the typing environment thus produces a sort declaration for each basic set, and combination thereof found in the input. In SMT-LIB, sorts have a name and an arity, which is non-null for polymorphic sorts. The sorts produced have all arity 0, and for the above example, the following is produced:

$$
\begin{array}{rcl}
S & \rightsquigarrow & \texttt{(declare-sort S 0)} \\
T & \rightsquigarrow & \texttt{(declare-sort T 0)} \\
U & \rightsquigarrow & \texttt{(declare-sort U 0)} \\
\mathbb{P}(S) & \rightsquigarrow & \texttt{(declare-sort PS 0)} \\
\mathbb{P}(S \times T) & \rightsquigarrow & \texttt{(declare-sort PST 0)} \\
\mathbb{P}(S \times T \times U) & \rightsquigarrow & \texttt{(declare-sort PSTU 0)}
\end{array}
$$

Secondly, for each constant, the translation produces a function declaration of the appropriate sort:

$$
\begin{array}{rcl}
a \oftype S & \rightsquigarrow & \texttt{(declare-fun a () S)} \\
b \oftype T & \rightsquigarrow & \texttt{(declare-fun b () T)} \\
c \oftype U & \rightsquigarrow & \texttt{(declare-fun c () U)} \\
A \oftype \mathbb{P}(S) & \rightsquigarrow & \texttt{(declare-fun A () PS)} \\
r \oftype \mathbb{P}(S \times T) & \rightsquigarrow & \texttt{(declare-fun r () PST)} \\
s \oftype \mathbb{P}(S \times T \times U) & \rightsquigarrow & \texttt{(declare-fun s () PSTU)}
\end{array}
$$

Third, for each type occurring at the right-hand side of a membership predicate, the translation produces fresh SMT function symbols:

```
(declare-fun (MS0 (S PS) Bool))
(declare-fun (MS1 (S T PST) Bool))
(declare-fun (MS2 (S T U PSTU)) Bool)
```

The Event-B atoms can then be translated as follows:

$$
\begin{array}{rcl}
a \in A & \rightsquigarrow & \texttt{(MS0 a A)} \\
a \mapsto b \in r & \rightsquigarrow & \texttt{(MS1 a b r)} \\
a \mapsto b \mapsto c \in s & \rightsquigarrow & \texttt{(MS2 a b c s)}
\end{array}
$$

For instance, $A \cup \{a\} = A$ would be translated to $\forall x \cdot (x \in A \lor x = a) \Leftrightarrow x \in A$, that is, in SMT-LIB format:

```
(forall ((x S)) (= (or (MS0 x A) (= x a)) (MS0 x A)))
```

While the approach presented here covers the whole Event-B mathematical language and does not require polymorphic types or specific extensions to the SMT-LIB language, the semantics of some Event-B constructs is approximated because some operators become uninterpreted in SMT-LIB (chiefly membership but also some arithmetic operators such as division and exponentiation). However, we can recover their interpretation by adding axioms to the SMT-LIB benchmark, at the risk of decreasing the performance of the SMT solvers. Some experimentation is thus needed to find a good balance between efficiency and completeness.

Indeed, it appears experimentally that including some axioms of set theory to constrain the possible interpretations of the membership predicate greatly improves the number of proof obligations discharged. In particular, the axiom of elementary set (singleton part) is necessary for many Rodin proof obligations. The translator directly instantiates the axiom for all membership predicates. Assuming `MS` is the membership predicate associated with sorts `S` and `PS`, the translation introduces thus the following assertion:

```
(assert (forall ((x S))
          (exists ((X PS)) (and (MS x X)
                                (forall ((y S)) (=> (MS y X) (= y x)))))))
```

This particular assertion eliminates non-standard interpretations where some singleton sets do not exist. Without it, some formulas are satisfiable because of spurious models and the SMT solvers are unable to refute them.

4. A small Event-B example

As a concrete example of translation, this section presents the model of a simple job processing system consisting of a job queue and several active jobs. We define a given set *JOBS* to represent the jobs. The state of the model has two variables: *queue* (the jobs currently queued) and *active* (the jobs being processed). This state is constrained by the following invariants:

`inv1` : $active \subseteq JOBS$ (typing)

`inv2` : $queue \subseteq JOBS$ (typing)

`inv3` : $active \cap queue = \varnothing$ (a job can not be both active and queued)

One of the events of the system describes that a job leaves the queue and becomes active. It is specified as follows:

Event *SCHEDULE* $\widehat{=}$ (some queued job j becomes active)
any
j
where
`grd1` : $j \in queue$ (the job j is in the queue)
then
`act1` : $active := active \cup \{j\}$ (the job becomes active)
`act2` : $queue := queue \setminus \{j\}$ (the job is removed from the queue)
end

To verify that the invariant labeled `inv3` is preserved by the *SCHEDULE* event, the following sequent must be proved valid:

$$\texttt{inv1}, \texttt{inv2}, \texttt{inv3}, \texttt{grd1} \vdash \underbrace{(active \cup \{j\})}_{active'} \cap \underbrace{(queue \setminus \{j\})}_{queue'} = \varnothing \ . \tag{1}$$

The generated proof obligations thus aims to show that the following formula is unsatisfiable:

$$\begin{array}{l} active \subseteq JOBS \wedge \\ queue \subseteq JOBS \wedge \\ active \cap queue = \varnothing \wedge \\ j \in queue \wedge \\ \neg((active \cup \{j\}) \cap (queue \setminus \{j\}) = \varnothing) \ . \end{array}$$

This proof obligation does not contain sets of sets and both approaches apply. Figure 5 presents the SMT-LIB input obtained when the approach described in section 3.1 is applied. Line 2 contains the declaration of the sort corresponding to the given set *JOBS*. Lines 3–5 contain the declarations of the uninterpreted function symbols of the proof obligation, and are produced using the typing environment. Note that the sets *queue* and *active* are represented by unary predicate symbols. Next, the macros corresponding to the set operators $\varnothing$, $\in$,

```
(set-logic QF_AUFLIA)
(declare-sort JOBS 0)
(declare-fun active (JOBS) Bool)
(declare-fun queue (JOBS) Bool)
(declare-fun j () JOBS)
(define-fun (par (X) (emptyset ((x X)) Bool false)))
(define-fun (par (X) (in ((x X) (s (X Bool))) Bool (s x))))
(define-fun (par (X) (inter ((s1 (X Bool)) (s2 (X Bool))) (X Bool)
                        (lambda ((x X)) (and (s1 x) (s2 x))))))
(define-fun (par (X) (setminus ((s1 (X Bool)) (s2 (X Bool))) (X Bool)
                        (lambda ((x X)) (and (s1 x) (not (s2 x)))))))
(define-fun (par (X) (union ((s1 (X Bool)) (s2 (X Bool))) (X Bool)
                        (lambda ((x X)) (or (s1 x) (s2 x))))))
(define-fun enum ((elem JOBS)) Bool (= elem j))
(define-fun enum0 ((elem0 JOBS)) Bool (= elem0 j))
(assert (= (inter active queue) emptyset))
(assert (in j queue))
(assert (not (= (inter (union active enum) (setminus queue enum0))
                emptyset)))
(check-sat)
```

Figure 5: SMT-LIB input produced using the λ-based approach.

```
(set-logic AUFLIA)
(declare-sort JOBS 0)
(declare-sort PJ 0)
(declare-fun MS (JOBS PJ) Bool)
(declare-fun active () PJ)
(declare-fun j () JOBS)
(declare-fun queue () PJ)
(assert (! (forall ((x JOBS))
             (not (and (MS x active) (MS x queue)))) :named inv3))
(assert (! (MS j queue) :named grd1))
(assert (! (not (forall ((x0 JOBS))
                  (not (and (or (MS x0 active) (= x0 j))
                            (MS x0 queue)
                            (not (= x0 j)))))) :named goal))
(check-sat)
```

Figure 6: SMT-LIB input produced using the *ppTrans* approach.

$\cap$, $\setminus$, and $\cup$ are defined in lines 6–13. Lines 14–15 are the definitions of a macro that represents the singleton set $\{j\}$ (it occurs twice in the formula). Lines 16–19 are the result of the translation of the proof obligation itself.

Figure 6 presents the SMT-LIB input resulting from the translation approach described in section 3.2. Since the proof obligation includes sets of *JOBS*, a corresponding sort *PJ* and membership predicate *MS* are declared in lines 3–4. Then, the function symbols corresponding to the free identifiers of the sequent are declared at lines 5–7. Finally, the hypotheses and the goal of the sequent are translated to named assertions (lines 8–14).

The sequent described in this section is very simple and is easily verified by both Atelier-B provers and SMT solvers. It is noteworthy that the plug-in inspects sequents to decide which approach is applied. When the sequents contains no sets or only simple sets (i.e., no sets of sets), the λ-based approach is applied. Otherwise, the plug-in employs the *ppTrans* approach. The next section reports experiments with a large number of proof obligations and establishes a better basis to compare the effectiveness of these different verification techniques.

5. Experimental results

We evaluated experimentally the effectiveness of using SMT solvers as reasoners in the Rodin platform by means of the techniques presented in this paper. This evaluation complements the experiments presented in [15] and reinforces their conclusions. We established a library of 2,456 proof obligations stemming from Event-B developments collected by the European FP7 project Deploy and publicly available on the Deploy repository[1]. These developments originate from examples from Abrial's book [2], academic publications, tutorials, as well as industrial case studies.

One main objective of introducing new reasoners in the Rodin platform is to reduce the number of valid proof obligations that need to be discharged interactively by humans. Consequently, the effectiveness of a reasoner is measured by the number of proof obligations proved automatically by the reasoner.

Obviously, effectiveness should depend on the computing resources given to the reasoners. In practice, the amount of memory is seldom a bottleneck, and usually the solvers are limited by setting a timeout on their execution time. In the context of the Rodin platform, the reasoners are executed by synchronous calls, and the longer the time limit, the less responsive is the framework to the user. We have experimented different timeouts and our experiments have shown us that a timeout of one second seems a good trade-off: doubling the timeout to two seconds increases by fewer than 0.1% the number of verified proof obligations, while decreasing the responsiveness of the platform.

Table 1 compares different reasoners on our set of benchmarks. The second column corresponds to Rodin internal normalization and simplification proce-

[1] `http://deploy-eprints.ecs.soton.ac.uk`

dures. It shows that more than half of the generated proof obligations necessitate advanced theorem-proving capabilities to be discharged. The third column is a special-purpose reasoner, namely Atelier-B provers. They were originally developed for the B method and are also available in the Rodin platform. Although they are extremely effective, the Atelier-B provers now suffer from legacy issues. The last five columns are various SMT solvers applied to the proof obligations generated by the plug-in. The SMT solvers were used with a timeout of one second, on a computer equipped with an Intel Core i7-4770, cadenced at 3.40 GHz, with 24 GB of RAM, and running Ubuntu Linux 12.04. They show decent results, but they are not yet as effective reasoners as the Atelier-B theorem provers.

Number of proof obligations	Rodin	Atelier-B	alt-ergo-r217	cvc3-2011-11-21	veriT-dev-r2863	veriT & E-prover	z3-3.2
2456	1169	2260	2017	2218	2051	2160	2094

Table 1: Number of proof obligations discharged by the reasoners.

Although this comparison is interesting to evaluate and compare the different reasoners, it is not sufficient to evaluate the effectiveness of the approach presented in this paper. Indeed, nothing prevents users to use several other reasoners once one has failed to achieve its goal. In Table 2, we report how many proof obligations remain unproved after applying the traditional reasoners (Atelier-B theorem provers and the Rodin reasoner) in combination with each SMT solver, and with all SMT solvers.

Number of proof obligations left	alt-ergo-r217	cvc3-2011-11-21	veriT-dev-r2863	veriT & E-prover	z3-3.2	All SMT solvers
196	114	61	94	103	92	31

Table 2: Number of proof obligations *not* discharged by special-purpose reasoners and by each SMT solver.

Each of the SMT solvers seems a valuable complement to the special-purpose provers. However, we would also like to know whether the reasoning capacity of some of these solvers is somehow subsumed by another solver, or whether each SMT solver could provide a significant contribution towards reducing the

number of proof obligations that need to be discharged by humans. Table 3 synthesizes a pairwise comparison of the SMT solvers on our universe of proof obligations.

	alt-ergo	cvc3	veriT	veriT+E	z3
alt-ergo	2017	2001	1880	1967	1911
cvc3	2001	2218	1953	2088	2031
veriT	1880	1953	2051	1958	1878
veriT+E	1967	2088	1958	2160	2067
z3	1911	2031	1878	1972	2094

Table 3: Number of proof obligations verified by SMT solver A also discharged by solver B.

This comparison signals the results obtained when all available reasoners are applied: only 31 proof obligations are unproved, down from 196 resulting from the application of Atelier-B provers. It is also noteworthy that even though each SMT solver is individually less effective than Atelier-B provers, applied altogether, they prove all but 97 proof obligations. The important conclusion of our experiments is that there is strong evidence that SMT solvers complement in an effective and practical way the Atelier B provers, yielding significant improvements in the usability of the Rodin platform and its effectiveness to support the development of Event-B models.

6. Conclusion

SMT solving is a formal verification technique successfully applied to various domains including verification. SMT solvers do not have built-in support for set-theoretic constructs found in Rodin sequents. We presented here a translation approach to tackle this issue. We evaluated experimentally the efficiency of SMT solvers against proof obligations resulting from the translation of Rodin sequents. In our sample of industrial and academic projects, the use of SMT solvers on top of Atelier B provers reduces significantly the number of unverified sequents. This plug-in is available through the integrated software updater of Rodin (instructions at `http://wiki.event-b.org/index.php/SMT_Plug-in`).

The results are very encouraging and motivate us to progress further by implementing and evaluating new translation approaches, such as representing functions using arrays in the line of [11]. Elaborating strategies to apply different reasoners, based on some characteristics of the sequents is also a promising line of work. Another feature of some SMT solvers is that they can provide models when a formula is satisfiable. In consequence, it would be possible, with additional engineering effort, to use such models to report counter-examples in Rodin.

We believe that the approach presented in this paper could also be applied successfully for other set-based formalisms such as: the B method, TLA+, VDM and Z.

Cooperation of deduction tools is very error-prone, not only because it relies on the correctness of many large and complex tools, but also because of the translations. Certification of proofs in a centralized trusted proof manager would be the answer to this problem. Preliminary works in this direction exist [23].

Acknowledgments: This paper is a revised and extended version of [15]. We thank the anonymous reviewers of paper [15] and of this paper for their careful read and their remarks.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[3] M. Armand, G. Faure, B. Grégoire, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *First Int'l Conference on Certified Programs and Proofs, CPP 2011*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[4] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, Feb. 2009.

[5] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard Version 2.0, 2010.

[6] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[7] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[8] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, `http://alt-ergo.lri.fr`.

[9] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[10] J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. RODIN (Rigorous open Development Environment for Complex Systems). In *Fifth European Dependable Computing Conference: EDCC-5 supplementary volume*, pages 23–26, 2005.

[11] J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Scalable Automated Proving and Debugging of Set-Based Specifications. *Journal of the Brazilian Computer Society*, 9:17–36, 2003.

[12] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[13] D. Déharbe. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In M. Frappier, G. Uwe, K. Sarfraz, R. Laleau, and S. Reeves, editors, *Proceedings 2nd Int'l Conf. Abstract State Machines, Alloy, B and Z, ABZ 2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 2010.

[14] D. Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, 78(3):310 – 326, 2013.

[15] D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In J. Derrick, J. A. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Proc 3rd Int. Conf. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, volume 7316 of *Lecture Notes in Computer Science*, pages 194–207. Springer, 2012.

[16] C. B. Jones. *Systematic software development using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[17] M. Konrad and L. Voisin. Translation from Set-Theory to Predicate Calculus. Technical report, ETH Zurich, 2011.

[18] D. Kröning, P. Rümmer, and G. Weissenbacher. A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-LIB Standard. In *Informal proceedings, 7th Int'l Workshop on Satisfiability Modulo Theories (SMT) at CADE 22*, 2009.

[19] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, Mass., 2002.

[20] S. Merz. On the logic of TLA$^+$. *Computers and Informatics*, 22:351–379, 2003.

[21] C. Métayer and L. Voisin. The Event-B mathematical language, 2009. `http://deploy-eprints.ecs.soton.ac.uk/11/4/kernel_lang.pdf`.

[22] G. Nelson and D. C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.

[23] M. Schmalz. The logic of Event-B, 2011. Technical report 698, ETH Zürich, Information Security.

[24] S. Schulz. E - A Brainiac Theorem Prover. *AI Communications*, 15(2/3):111–126, 2002.

[25] The Eclipse Foundation. Eclipse SDK, 2009.

[26] C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems (FroCoS)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, Mar. 1996.

[27] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

Appendix A. Rewriting rules in *ppTrans*

Appendix A.1. Rules for sets and predicates

$$x \neq y \rightsquigarrow \neg(x = y) \tag{A.1}$$
$$x \notin s \rightsquigarrow \neg(x \in s) \tag{A.2}$$
$$s \subseteq t \rightsquigarrow s \in \mathbb{P}(t) \tag{A.3}$$
$$s \not\subseteq t \rightsquigarrow \neg(s \in \mathbb{P}(t)) \tag{A.4}$$
$$s \subset t \rightsquigarrow s \in \mathbb{P}(t) \wedge \neg(t \in \mathbb{P}(s)) \tag{A.5}$$
$$s \not\subset t \rightsquigarrow \neg(s \in \mathbb{P}(t)) \vee t \in \mathbb{P}(s) \tag{A.6}$$
$$\text{finite}(s) \rightsquigarrow \forall a \cdot \exists b, f \cdot f \in s \rightarrowtail a..b \tag{A.7}$$
$$\text{partition}(s, s_1, s_2, \ldots s_n) \rightsquigarrow \begin{array}{l} s = s_1 \cup s_2 \cup \ldots \cup s_n \wedge \\ s_1 \cap s_2 = \varnothing \wedge \cdots \wedge s_1 \cap s_n = \varnothing \wedge \\ \quad \vdots \\ s_{n-1} \cap s_n = \varnothing \end{array} \tag{A.8}$$

Appendix A.2. Elimination of equalities

The notation $\forall X_T \cdot P(X)$ stands for $\forall \mathsf{fv}(\nabla(e)) \cdot P(\nabla(e))$, with $\mathcal{T}(X) = T$.

$$s = t \rightsquigarrow \forall X_T \cdot X \in s \Leftrightarrow X \in t \tag{A.9}$$
$$x_1 \mapsto x_2 = y_1 \mapsto y_2 \rightsquigarrow x_1 = y_1 \wedge x_2 = y_2 \tag{A.10}$$
$$x = f(y) \rightsquigarrow y \mapsto x \in f \tag{A.11}$$
$$\text{bool}(P) = \text{bool}(Q) \rightsquigarrow P \Leftrightarrow Q \tag{A.12}$$
$$\text{bool}(P) = \text{TRUE} \rightsquigarrow P \tag{A.13}$$
$$\text{bool}(P) = \text{FALSE} \rightsquigarrow \neg P \tag{A.14}$$
$$x = \text{FALSE} \rightsquigarrow \neg(x = \text{TRUE}) \tag{A.15}$$
$$x = \text{bool}(P) \rightsquigarrow x = \text{TRUE} \Leftrightarrow P \tag{A.16}$$
$$e_l[\text{bool}(s)] = e_r \rightsquigarrow \forall^? e_l[\text{bool}(s)^*] = e_r \tag{A.17}$$
$$e = e \rightsquigarrow \top \tag{A.18}$$

Appendix A.3. Elimination of mixed arithmetic symbols

Purification

MOP stands for either min, max, card or a function application, $\prec$ for either $\leq$ or $<$ and $\succ$ for either $\geq$ or $>$.

$$e_l[\text{MOP}(s)] = e_r \rightsquigarrow \forall^? e_l[\text{MOP}(s)^*] = e_r \tag{A.19}$$
$$a[\text{MOP}(s)] \prec b \rightsquigarrow \forall^? a[\text{MOP}(s)^*] \prec b \tag{A.20}$$
$$a \prec b[\text{MOP}(s)] \rightsquigarrow \forall^? a \prec b[\text{MOP}(s)^*] \tag{A.21}$$

Elimination of mixed operators

$$n = \mathrm{card}(s) \rightsquigarrow \exists f \cdot f \in s \rightarrowtail\!\!\!\!\!\!\rightarrow 1..n \tag{A.22}$$
$$n = \min(s) \rightsquigarrow n \in s \wedge n \leq \min(s) \tag{A.23}$$
$$n = \max(s) \rightsquigarrow n \in s \wedge \max(s) \leq n \tag{A.24}$$
$$a \succ b \rightsquigarrow b \prec a \tag{A.25}$$
$$\max(s) \prec a \rightsquigarrow \forall x \cdot x \in s \Rightarrow x \prec a \tag{A.26}$$
$$\min(s) \prec a \rightsquigarrow \exists x \cdot x \in s \wedge x \prec a \tag{A.27}$$
$$a \prec \min(s) \rightsquigarrow \forall x \cdot x \in s \Rightarrow a \prec x \tag{A.28}$$
$$a \prec \max(s) \rightsquigarrow \exists x \cdot x \in s \wedge a \prec x \tag{A.29}$$

Appendix A.4. Rules based on the right argument of set membership

Elimination of membership in relations

The notation _func(f) specifies that f is a function, and abbreviates: $\forall A_U, B_V, C_V \cdot A \mapsto B \in f \wedge A \mapsto C \in f \Rightarrow B = C$, where $\mathcal{T}(f) = U \times V$.

$$e \in s \twoheadleftarrow\!\!\!\!\!\!\leftarrow t \rightsquigarrow e \in s \twoheadleftarrow t \wedge t \subseteq \mathrm{ran}(e) \tag{A.30}$$
$$e \in s \twoheadleftarrow t \rightsquigarrow e \in s \leftrightarrow t \wedge s \subseteq \mathrm{dom}(e) \tag{A.31}$$
$$e \in s \leftrightarrow\!\!\!\!\!\!\rightarrow t \rightsquigarrow e \in s \leftrightarrow t \wedge t \subseteq \mathrm{ran}(e) \tag{A.32}$$
$$e \in s \rightarrowtail\!\!\!\!\!\!\rightarrow t \rightsquigarrow e \in s \rightarrow t \wedge \_\mathrm{func}(e^{-1}) \tag{A.33}$$
$$e \in s \rightarrow t \rightsquigarrow e \in s \rightarrow t \wedge t \subseteq \mathrm{ran}(e) \tag{A.34}$$
$$e \in s \twoheadrightarrow t \rightsquigarrow e \in s \twoheadrightarrow t \wedge t \subseteq \mathrm{ran}(e) \tag{A.35}$$
$$e \in s \rightarrowtail t \rightsquigarrow e \in s \rightarrow t \wedge \_\mathrm{func}(e^{-1}) \tag{A.36}$$
$$e \in s \rightarrowtail\!\!\!\!\!\!\rightarrow t \rightsquigarrow e \in s \twoheadrightarrow t \wedge \_\mathrm{func}(e^{-1}) \tag{A.37}$$
$$e \in s \rightarrow t \rightsquigarrow e \in s \twoheadrightarrow t \wedge s \subseteq \mathrm{dom}(e) \tag{A.38}$$
$$e \in s \twoheadrightarrow t \rightsquigarrow e \in s \leftrightarrow t \wedge \_\mathrm{func}(e) \tag{A.39}$$

Other membership rewriting rules

The notation $\forall X_T \cdot P(X)$ stands for $\forall \mathsf{fv}(\nabla(e)) \cdot P(\nabla(e))$, with $\mathcal{T}(X) = T$.
Likewise, notation $\exists X_T \cdot P(X)$ stands for $\exists \mathsf{fv}(\nabla(e)) \cdot P(\nabla(e))$, with $\mathcal{T}(X) = T$.

$$e \in s \rightsquigarrow \top \qquad \text{if } \mathcal{T}(e) = s \tag{A.40}$$

$$e \in \varnothing \rightsquigarrow \bot \tag{A.41}$$

$$e \in \mathbb{P}(t) \rightsquigarrow \forall X_T \cdot X \in e \Rightarrow X \in t \qquad \text{if } \mathcal{T}(e) = \mathbb{P}(T) \tag{A.42}$$

$$e \in s \leftrightarrow t \rightsquigarrow \forall X_T \cdot X \in e \Rightarrow X \in s \times t \qquad \text{if } \mathcal{T}(e) = \mathbb{P}(T) \tag{A.43}$$

$$e \in f \rightsquigarrow \exists^? e^* \in f \qquad \text{if } f \text{ is an identifier} \tag{A.44}$$

$$e \in \mathbb{N} \rightsquigarrow 0 \leq e \tag{A.45}$$

$$e \in \mathbb{N}_1 \rightsquigarrow 0 < e \tag{A.46}$$

$$e \in \{x \cdot P \mid f\} \rightsquigarrow \exists x \cdot P \wedge e = f \tag{A.47}$$

$$e \in \left(\bigcap x \cdot P \mid f\right) \rightsquigarrow \forall x \cdot P \Rightarrow e \in f \tag{A.48}$$

$$e \in \left(\bigcup x \cdot P \mid f\right) \rightsquigarrow \exists x \cdot P \wedge e \in f \tag{A.49}$$

$$e \in \mathrm{union}(s) \rightsquigarrow \exists x \cdot x \in s \wedge e \in x \tag{A.50}$$

$$e \in \mathrm{inter}(s) \rightsquigarrow \forall x \cdot x \in s \Rightarrow e \in x \tag{A.51}$$

$$e \in r\,[s] \rightsquigarrow \exists X_T \cdot X \in s \wedge X \mapsto e \in r \qquad \text{if } \mathbb{P}(T) = \mathcal{T}(\mathrm{dom}(r)) \tag{A.52}$$

$$e \in f(s) \rightsquigarrow \exists X_T \cdot s \mapsto X \in f \wedge e \in X \qquad \text{if } T = \mathbb{P}(\mathcal{T}(e)) \tag{A.53}$$

$$e \in \mathrm{ran}(r) \rightsquigarrow \exists X_T \cdot X \mapsto e \in r \qquad \text{if } \mathbb{P}(T) = \mathcal{T}(\mathrm{dom}(r)) \tag{A.54}$$

$$e \in \mathrm{dom}(r) \rightsquigarrow \exists X_T \cdot e \mapsto X \in r \qquad \text{if } \mathbb{P}(T) = \mathcal{T}(\mathrm{ran}(r)) \tag{A.55}$$

$$e \in \{a_1, \ldots, a_n\} \rightsquigarrow e = a_1 \vee \cdots \vee e = a_n \tag{A.56}$$

$$e \in \mathbb{P}_1(s) \rightsquigarrow e \in \mathbb{P}(s) \wedge [\exists X_T \cdot X \in e] \qquad \text{if } \mathbb{P}(T) = \mathcal{T}(e) \tag{A.57}$$

$$e \in a..b \rightsquigarrow a \leq e \wedge e \leq b \tag{A.58}$$

$$e \in s \setminus t \rightsquigarrow e \in s \wedge \neg(e \in t) \tag{A.59}$$

$$e \in s_1 \cap \ \ldots \ \cap s_n \rightsquigarrow e \in s_1 \wedge \ \ldots \ \wedge e \in s_n \tag{A.60}$$

$$e \in s_1 \cup \ \ldots \ \cup s_n \rightsquigarrow e \in s_1 \vee \ \ldots \ \vee e \in s_n \tag{A.61}$$

Appendix A.5. Rules based on both arguments of set membership

$$e \mapsto f \in s \times t \rightsquigarrow e \in s \wedge f \in t \tag{A.62}$$

$$e \mapsto f \in r \rhd t \rightsquigarrow e \mapsto f \in r \wedge \neg(f \in t) \tag{A.63}$$

$$e \mapsto f \in s \lhd r \rightsquigarrow e \mapsto f \in r \wedge \neg(e \in s) \tag{A.64}$$

$$e \mapsto f \in r \rhd t \rightsquigarrow e \mapsto f \in r \wedge f \in t \tag{A.65}$$

$$e \mapsto f \in s \lhd r \rightsquigarrow e \mapsto f \in r \wedge e \in s \tag{A.66}$$

$$e \mapsto f \in \mathrm{id} \rightsquigarrow e = f \tag{A.67}$$

$$e \mapsto f \in r^{-1} \rightsquigarrow f \mapsto e \in r \tag{A.68}$$

$$e \mapsto f \in \mathrm{pred} \rightsquigarrow e = f + 1 \tag{A.69}$$

$$e \mapsto f \in \mathrm{succ} \rightsquigarrow f = e + 1 \tag{A.70}$$

$$\begin{aligned} e \mapsto f \in r_1 \lhd\!\!\!\!\lhd \cdots \lhd\!\!\!\!\lhd r_n \rightsquigarrow \quad & e \mapsto f \in r_n \vee \\ & e \mapsto f \in \mathrm{dom}(r_n) \lhd r_{n-1} \vee \\ & e \mapsto f \in \mathrm{dom}(r_n) \cup \mathrm{dom}(r_{n-1}) \lhd r_{n-2} \vee \\ & \vdots \\ & e \mapsto f \in \mathrm{dom}(r_n) \cup \cdots \mathrm{dom}(r_2) \lhd r_1 \end{aligned} \tag{A.71}$$

$$\begin{aligned} e \mapsto f \in r_1; \ldots; r_n \rightsquigarrow \quad & \exists X^1_{T_1}, \ldots X^{n-1}_{T_{n-1}} \cdot e \mapsto X^1 \in r_1 \wedge \ldots \wedge X^{n-1} \mapsto f \in r_n \\ & \text{if } \mathcal{T}(\mathrm{ran}(r_i)) = \mathbb{P}(T_i), 1 \leq i \leq n \end{aligned} \tag{A.72}$$

$$e \mapsto f \in r_1 \circ \ldots \circ r_n \rightsquigarrow e \mapsto f \in r_n; \ldots; r_1 \tag{A.73}$$

$$(e \mapsto f) \mapsto g \in \mathrm{prj}_1 \rightsquigarrow e = g \tag{A.74}$$

$$(e \mapsto f) \mapsto g \in \mathrm{prj}_2 \rightsquigarrow f = g \tag{A.75}$$

$$e \mapsto (f \mapsto g) \in p \otimes q \rightsquigarrow e \mapsto f \in p \wedge e \mapsto g \in q \tag{A.76}$$

$$(e \mapsto f) \mapsto (g \mapsto h) \in p \parallel q \rightsquigarrow e \mapsto g \in p \wedge f \mapsto h \in q \tag{A.77}$$

A Gentle Non-Disjoint Combination of Satisfiability Procedures

Paula Chocron[1,3], Pascal Fontaine[2], and Christophe Ringeissen[3⋆]

[1] Universidad de Buenos Aires, Argentina
[2] INRIA, Université de Lorraine & LORIA, Nancy, France
[3] INRIA & LORIA, Nancy, France

Abstract. A satisfiability problem is often expressed in a combination of theories, and a natural approach consists in solving the problem by combining the satisfiability procedures available for the component theories. This is the purpose of the combination method introduced by Nelson and Oppen. However, in its initial presentation, the Nelson-Oppen combination method requires the theories to be signature-disjoint and stably infinite (to guarantee the existence of an infinite model). The notion of gentle theory has been introduced in the last few years as one solution to go beyond the restriction of stable infiniteness, but in the case of disjoint theories. In this paper, we adapt the notion of gentle theory to the non-disjoint combination of theories sharing only unary predicates (plus constants and the equality). Like in the disjoint case, combining two theories, one of them being gentle, requires some minor assumptions on the other one. We show that major classes of theories, i.e. Löwenheim and Bernays-Schönfinkel-Ramsey, satisfy the appropriate notion of gentleness introduced for this particular non-disjoint combination framework.

1 Introduction

The design of satisfiability procedures has attracted a lot of interest in the last decade due to their ubiquity in SMT (Satisfiability Modulo Theories [4]) solvers and automated reasoners. A satisfiability problem is very often expressed in a combination of theories, and a very natural approach consists in solving the problem by combining the satisfiability procedures available for each of them. This is the purpose of the combination method introduced by Nelson and Oppen [15]. In its initial presentation, the Nelson-Oppen combination method requires the theories in the combination to be (1) signature-disjoint and (2) stably infinite (to guarantee the existence of an infinite model). These are strong limitations, and many recent advances aim to go beyond disjointness and stable infiniteness. Both corresponding research directions should not be opposed. In both cases, the problems are similar, i.e. building a model of $\mathcal{T}_1 \cup \mathcal{T}_2$ from a model of $\mathcal{T}_1$ and

⋆ This work has been partially supported by the project ANR-13-IS02-0001-01 of the Agence Nationale de la Recherche, by the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS), and by the STIC AmSud MISMT

a model of $\mathcal{T}_2$. This is possible if and only if there exists an isomorphism between the restrictions of the two models to the shared signature [24]. The issue is to define a framework to enforce the existence of this isomorphism. In the particular case of disjoint theories, the isomorphism can be obtained if the domains of the models have the same cardinality, for instance infinite; several classes of *kind* theories (shiny [25], polite [19], gentle [9]) have been introduced to enforce a (same) domain cardinality on both sides of the combination. For extensions of Nelson-Oppen to non-disjoint cases, e.g. in [24,27], cardinality constraints also arise. In this paper, we focus on non-disjoint combinations for which the isomorphism can be simply constructed by satisfying some cardinality constraints. More precisely, we extend the notion of gentle theory to the non-disjoint combination of theories sharing only unary predicates (plus constants and the equality). Some major classes of theories fit in our non-disjoint combination framework.

Contributions. The first contribution is to introduce a class of $\mathcal{P}$*-gentle* theories, to combine theories sharing a finite set of unary predicates symbols $\mathcal{P}$. The notion of $\mathcal{P}$-gentle theory extends the one introduced for the disjoint case [9]. Roughly speaking, a $\mathcal{P}$-gentle theory has nice cardinality properties not only for domains of models but also more locally for all Venn regions of shared unary predicates. We present a combination method for unions of $\mathcal{P}$-gentle theories sharing $\mathcal{P}$. The proposed method can also be used to combine a $\mathcal{P}$-gentle theory with another arbitrary theory for which we assume the decidability of satisfiability problems with cardinality constraints. This is a natural extension of previous works on combining non-stably infinite theories, in the straight line of combination methods à la Nelson-Oppen. Two major classes of theories are $\mathcal{P}$-gentle, namely the Löwenheim and Bernays-Schönfinkel-Ramsey (BSR) classes.

We characterize precisely the cardinality properties satisfied by Löwenheim theories. As a side contribution, bounds on cardinalities given in [8] have been improved, and we prove that our bounds are optimal. Our new result establishes that Löwenheim theories are $\mathcal{P}$-gentle.

We prove that BSR theories are also $\mathcal{P}$-gentle. This result relies on a non-trivial extension of Ramsey's Theorem on hypergraphs. This extension should be considered as another original contribution, since it may be helpful as a general technique to construct a model preserving the regions.

Related Work. Our combination framework is a way to combine theories with sets. The relation between (monadic) logic and sets is as old as logic itself, and this relation is particularly clear for instance considering Aristotle Syllogisms. It is however useful to again study monadic logic, and more particularly the Löwenheim class, and in view of the recent advances in combinations with non-disjoint and non-stably infinite theories.

In [26], the authors focus on the satisfiability problem of unions of theories sharing set operations. The basic idea is to reduce the combination problem into a satisfiability problem in a fragment of arithmetic called $BAPA$ (Boolean Algebra and Presburger Arithmetic). Löwenheim and BSR classes are also considered, but infinite cardinalities were somehow defined out of their reduction scheme,

whilst infinite cardinalities are smoothly taken into account in our combination framework. In [26], BSR was shown to be reducible to Presburger. We here give a detailed proof. We believe such a proof is useful since it is more complicated that it may appear. In particular, our proof is based on an original (up to our knowledge) extension of Ramsey's Theorem to accommodate a domain partitioned into (Venn) regions. Finally, the notion of $\mathcal{P}$-gentleness defined and used here is stronger than semi-linearity of Venn-cardinality, and allows non-disjoint combination with more theories, e.g. the guarded fragment.

In [21,22], a locality property is used to properly instantiate axioms connecting two disjoint theories. Hence, the locality is a way to reduce (via instantiation) a non-disjoint combination problem to a disjoint one. In that context, cardinality constraints occur when considering bridging functions over a data structure with some cardinality constraints on the underlying theory of elements [28,21,23].

In [12], Ghilardi proposed a very general model-theoretic combination framework to obtain a combination method à la Nelson-Oppen when $\mathcal{T}_1$ and $\mathcal{T}_2$ are two *compatible* extensions of the same shared theory (satisfying some properties). This framework relies on an application of the Robinson Joint Consistency Theorem (roughly speaking, the union of theories is consistent if the intersection is complete). Using this framework, several shared fragments of arithmetic have been successfully considered [12,16,17]. Due to its generality, Ghilardi's approach is free of cardinality constraints.

It is also possible to consider a general semi-decision procedure for the unsatisfiability problem modulo $\mathcal{T}_1 \cup \mathcal{T}_2$, e.g. a superposition calculus. With the rewrite-based approach initiated in [3], the problem reduces to proving the termination of this calculus. General criteria have been proposed to get modular termination results for superposition, when $\mathcal{T}_1$ and $\mathcal{T}_2$ are either disjoint [2] or non-disjoint [20]. Notice that the superposition calculus can also be used as a deductive engine to entail some cardinality constraints, as shown in [5].

Structure of the paper. Section 2 introduces some classical notations and definitions. In Section 3, we introduce the notion of $\mathcal{P}$-gentle theory and we present the related combination method for unions of theories sharing a (non-empty finite) set $\mathcal{P}$ of unary predicate symbols. All the theories in the Löwenheim class and in the BSR class are $\mathcal{P}$-gentle, as shown respectively in Section 4 and in Section 5. A simple example is given in Section 6. The conclusion (Section 7) discusses the current limitations of our approach and mentions some possible directions to investigate. Our extension of Ramsey's Theorem can be found in Appendix A.

2 Notation and Basic Definitions

A first-order language is a tuple $\mathcal{L} = \langle \mathcal{V}, \mathcal{F}, \mathcal{P} \rangle$ such that $\mathcal{V}$ is an enumerable set of variables, while $\mathcal{F}$ and $\mathcal{P}$ are sets of function and predicate symbols. Every function and predicate symbol is assigned an arity. Nullary predicate symbols are called proposition symbols, and nullary function symbols are called constant

symbols. A first-order language is called relational if it only contains function symbols of arity zero. A relational formula is a formula in a relational language. Terms, atomic formulas and first-order formulas over the language $\mathcal{L}$ are defined in the usual way. In particular an atomic formula is either an equality, or a predicate symbol applied to the right number of terms. Formulas are built from atomic formulas, Boolean connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\equiv$), and quantifiers ($\forall$, $\exists$). A literal is an atomic formula or the negation of an atomic formula. Free variables are defined in the usual way. A formula with no free variables is closed, and a formula without variables is ground. A universal formula is a closed formula $\forall x_1 \ldots \forall x_n.\varphi$ where φ is quantifier-free. A (finite) theory is a (finite) set of closed formulas. Two theories are disjoint if no predicate symbol in P or function symbol in F appears in both theories, except constants and equality.

An interpretation $\mathcal{I}$ for a first-order language $\mathcal{L}$ provides a non empty domain D, a total function $\mathcal{I}[f] : D^r \to D$ for every function symbol f of arity r, a predicate $\mathcal{I}[p] \subseteq D^r$ for every predicate symbol p of arity r, and an element $\mathcal{I}[x] \in D$ for every variable x. The cardinality of an interpretation is the cardinality of its domain. The notation $\mathcal{I}_{x_1/d_1,\ldots,x_n/d_n}$ for $x_1, \ldots, x_n$ different variables stands for the interpretation that agrees with $\mathcal{I}$, except that it associates $d_i \in D$ to the variable x_i, $1 \leq i \leq n$. By extension, an interpretation defines a value in D for every term, and a truth value for every formula. We may write $\mathcal{I} \models \varphi$ whenever $\mathcal{I}[\varphi] = \top$. Given an interpretation $\mathcal{I}$ on domain D, the *restriction* $\mathcal{I}'$ of $\mathcal{I}$ on $D' \subseteq D$ is the unique interpretation on D' such that $\mathcal{I}$ and $\mathcal{I}'$ interpret predicates, functions and variables the same way on D'. An *extension* $\mathcal{I}'$ of $\mathcal{I}$ is an interpretation on a domain D' including D such that $\mathcal{I}'$ restricted to D is $\mathcal{I}$.

A model of a formula (theory) is an interpretation that evaluates the formula (resp. all formulas in the theory) to true. A formula or theory is satisfiable if it has a model; it is unsatisfiable otherwise. A formula G is $\mathcal{T}$-satisfiable if it is satisfiable in the theory $\mathcal{T}$, that is, if $\mathcal{T} \cup \{G\}$ is satisfiable. A $\mathcal{T}$-model of G is a model of $\mathcal{T} \cup \{G\}$. A formula G is $\mathcal{T}$-unsatisfiable if it has no $\mathcal{T}$-models. In our context, a theory $\mathcal{T}$ is *decidable* if the $\mathcal{T}$-satisfiability problem for sets of (ground) literals is decidable in the language of $\mathcal{T}$ (extended with fresh constants).

Consider an interpretation $\mathcal{I}$ on a language with unary predicates $p_1, \ldots, p_n$ and some elements D in the domain of this interpretation. Every element $d \in D$ belongs to a *Venn region* $v(d) = v_1 \ldots v_n \in \{\top, \bot\}^n$ where $v_i = \mathcal{I}[p_i](d)$. We denote by $D_v \subseteq D$ the set of elements of D in the Venn region v. Notice also that, for a language with n unary predicates, there are 2^n Venn regions. Given an interpretation $\mathcal{I}$, D^c denotes the subset of elements in D associated to constants by $\mathcal{I}$. Naturally, D^c_v denotes the set of elements associated to constants that are in the Venn region v.

3 Gentle Theories Sharing Unary Predicates

From now on, we assume that $\mathcal{P}$ is a non-empty finite set of unary predicates. A $\mathcal{P}$-*union* of two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ is a union sharing only $\mathcal{P}$, a set of constants and the equality.

Definition 1. *An arrangement $\mathcal{A}$ for finite sets of constant symbols S and unary predicates $\mathcal{P}$ is a maximal satisfiable set of equalities and inequalities $a = b$ or $a \neq b$ and literals $p(a)$ or $\neg p(a)$, with $a, b \in S$, $p \in \mathcal{P}$.*

There are only a finite number of arrangements for given sets S and $\mathcal{P}$.

Given a theory $\mathcal{T}$ whose signature includes $\mathcal{P}$ and a model $\mathcal{M}$ of $\mathcal{T}$ on domain D, the *$\mathcal{P}$-cardinality* $\boldsymbol{\kappa}$ is the tuple of cardinalities of all Venn regions of $\mathcal{P}$ in $\mathcal{M}$ (κ_v will denote the cardinality of the Venn region v). The following theorem (specialization of general combination lemmas in e.g. [24,25]) states the completeness of the combination procedure for $\mathcal{P}$-unions of theories:

Theorem 1. *Consider a $\mathcal{P}$-union of theories $\mathcal{T}_1$ and $\mathcal{T}_2$ whose respective languages $\mathcal{L}_1$ and $\mathcal{L}_2$ share a finite set S of constants, and let L_1 and L_2 be sets of literals, respectively in $\mathcal{L}_1$ and $\mathcal{L}_2$. Then $L_1 \cup L_2$ is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable if and only if there exist an arrangement $\mathcal{A}$ for S and $\mathcal{P}$, and a $\mathcal{T}_i$-model $\mathcal{M}_i$ of $\mathcal{A} \cup L_i$ with the same $\mathcal{P}$-cardinality for $i = 1, 2$.*

The *spectrum* of a theory $\mathcal{T}$ is the set of $\mathcal{P}$-cardinalities of its models. The above theorem can thus be restated as:

Corollary 1. *The $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiability problem for sets of literals is decidable if, for any sets of literals $\mathcal{A} \cup L_1$ and $\mathcal{A} \cup L_2$ it is possible to decide if the intersection of the spectrums of $\mathcal{T}_1 \cup \mathcal{A} \cup L_1$ and of $\mathcal{T}_2 \cup \mathcal{A} \cup L_2$ is non-empty.*

To characterize the spectrum of the decidable classes considered in this paper, we introduce the notion of *cardinality constraint*. A *finite* cardinality constraint is simply a $\mathcal{P}$-cardinality with only finite cardinalities. An *infinite* cardinality constraint is given by a $\mathcal{P}$-cardinality $\boldsymbol{\kappa}$ with only finite cardinalities and a non-empty set of Venn regions V, and stands for all the $\mathcal{P}$-cardinalities $\boldsymbol{\kappa}'$ such that $\kappa'_v \geq \kappa_v$ if $v \in V$, and $\kappa'_v = \kappa_v$ otherwise. The *spectrum* of a finite set of cardinality constraints is the union of all $\mathcal{P}$-cardinalities represented by each cardinality constraint. It is now easy to define the class of theories we are interested in:

Definition 2. *A theory $\mathcal{T}$ is* $\mathcal{P}$-gentle *if, for every set L of literals in the language of $\mathcal{T}$, the spectrum of $\mathcal{T} \cup L$ is the spectrum of a computable finite set of cardinality constraints.*

Notice that a $\mathcal{P}$-gentle theory is (by definition) decidable. To relate the above notion with the gentleness in the disjoint case [9], observe that if p is a unary predicate symbol not occurring in the signature of the theory $\mathcal{T}$, then $\mathcal{T} \cup \{\forall x.p(x)\}$ is $\{p\}$-gentle if and only if $\mathcal{T}$ is gentle.

If a theory is $\mathcal{P}$-gentle, then it is $\mathcal{P}'$-gentle for any non-empty subset $\mathcal{P}'$ of $\mathcal{P}$. It is thus interesting to have $\mathcal{P}$-gentleness for the largest possible $\mathcal{P}$. Hence, when $\mathcal{P}$ is not explicitly given for a theory, we assume that $\mathcal{P}$ denotes the set of unary predicates symbols occurring in its signature. In the following sections we show that the Löwenheim theories and the BSR theories are $\mathcal{P}$-gentle.

The union of two $\mathcal{P}$-gentle theories is decidable, as a corollary of the following modularity result:

Theorem 2. *The class of $\mathcal{P}$-gentle theories is closed under $\mathcal{P}$-union.*

Proof. If we consider the $\mathcal{P}$-union of two $\mathcal{P}$-gentle theories with respective spectrums $\mathcal{S}_1$ and $\mathcal{S}_2$, then we can build some finite set of cardinality constraints whose spectrum is $\mathcal{S}_1 \cap \mathcal{S}_2$. □

Some very useful theories are not $\mathcal{P}$-gentle, but in practical cases they can be combined with $\mathcal{P}$-gentle theories. To define more precisely the class of theories $\mathcal{T}'$ that can be combined with a $\mathcal{P}$-gentle one, let us introduce the *$\mathcal{T}'$-satisfiability problem with cardinality constraints*: given a formula and a finite set of cardinality constraints, the problem amounts to check whether the formula is satisfiable in a model of $\mathcal{T}$ whose $\mathcal{P}$-cardinality is in the spectrum of the cardinality constraints. As a direct consequence of Corollary 1:

Theorem 3. *$\mathcal{T} \cup \mathcal{T}'$-satisfiability is decidable if the theory $\mathcal{T}$ is $\mathcal{P}$-gentle and $\mathcal{T}'$-satisfiability with cardinality constraints is decidable.*

Notice that $\mathcal{T}$-satisfiability with cardinality constraints is decidable for most common theories, e.g. the theories handled in SMT solvers. This gives the theoretical ground to add to the SMT solvers any number of $\mathcal{P}$-gentle theories sharing unary predicates.

From the results in the rest of the paper, it will also follow that the non-disjoint union (sharing unary predicates) of BSR and Löwenheim theories with one decidable theory accepting further constraints of the form $\forall x\,.\,((\neg)p_1(x) \wedge \ldots (\neg)p_n(x)) \Rightarrow (x = a_1 \vee \ldots x = a_m)$ is decidable. For instance, the guarded fragment with equality accepts such further constraints and the superposition calculus provides a decision procedure [11]. Thus any theory in the guarded fragment can be combined with Löwenheim and BSR theories sharing unary predicates.

In the disjoint case, any decidable theory expressed as a finite set of first-order axioms can be combined with a gentle theory [9]. Here this is not the case anymore. Indeed, consider the theory $\psi = \varphi \vee \exists x\, p(x)$ where p does not occur in φ; any set of literals is satisfiable in the theory ψ if and only if it is satisfiable in the theory of equality. If the satisfiability problem of literals in the theory φ is undecidable, the $\mathcal{P}$-union of ψ and the Löwenheim theory $\forall x\, \neg p(x)$ will also be undecidable.

4 The Löwenheim Class

We first review some classical results about this class and refer to [6] for more details. A Löwenheim theory is a finite set of closed formulas in a relational language containing only unary predicates (and no functions except constants). This class is also known as first-order relational monadic logic. Usually one distinguishes the Löwenheim class with and without equality. The Löwenheim class has the finite model property (and is thus decidable) even with equality. Full monadic logic *without equality*, i.e. the class of finite theories over a language containing symbols (predicates and functions) of arity at most 1, also has the finite model property. Considering monadic logic with equality, the class of

finite theories over a language containing only unary predicates and just two unary functions is already undecidable. With only one unary function, however, the class remains decidable [6], but does not have the finite model property anymore. Since the spectrum for this last class is significantly more complicated [13] than for the Löwenheim class we will here only focus on the Löwenheim class with equality (only classes with equality are relevant in our context), that is, without functions. More can be found about monadic first-order logic in [6,8]. In particular, a weaker version of Corollary 2 (given below) can be found in [8].

Previously [9,1], combining theories with non-stably infinite theories took advantage of "pumping" lemmas, allowing — for many decidable fragments — to build models of arbitrary large cardinalities. The following theorem is such a pumping lemma, but it considers the cardinalities of the Venn regions and not only the global cardinality.

Lemma 1. *Assume $\mathcal{T}$ is a Löwenheim theory with equality. Let q be the number of variables in $\mathcal{T}$. If there exists a model $\mathcal{M}$ on domain D with $|D_v \setminus D^c| \geq q$, then, for each cardinality $q' \geq q$, there is a model extension or restriction $\mathcal{M}'$ of $\mathcal{M}$ on domain D' such that $|D'_v \setminus D^c| = q'$ and $D'_{v'} = D_{v'}$ for all $v' \neq v$.*

Proof. Two interpretations $\mathcal{I}$ (on domain D) and $\mathcal{I}'$ (on domain D') for a formula ψ are *similar* if

- $|(D_v \cap D'_v) \setminus D^c| \geq q$;
- $D_{v'} = D'_{v'}$ for each Venn region v' distinct from v;
- $\mathcal{I}[a] = \mathcal{I}'[a]$ for each constant in ψ;
- $\mathcal{I}[x] = \mathcal{I}'[x]$ for each variable free in ψ.

Considering $\mathcal{M}$ as above, we can build a model $\mathcal{M}'$ as stated in the theorem, such that $\mathcal{M}$ and $\mathcal{M}'$ are similar. Indeed similarity perfectly defines a model with respect to another, given the cardinalities of the Venn regions.

We now prove that, given a Löwenheim formula ψ (or a set of formulas), two similar interpretations for ψ give the same truth value to ψ and to each sub-formula of ψ.

The proof is by induction on the structure of the (sub-)formula ψ. It is obvious if ψ is atomic, since similar interpretations assign the same value to variables and constants. If ψ is $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \Rightarrow \varphi_2$, the result holds if it also holds for φ_1 and φ_2.

Assume $\mathcal{I}$ makes true the formula $\psi = \exists x\, \varphi(x)$. Then there exists some $d \in D$ such that $\mathcal{I}_{x/d}$ is a model of $\varphi(x)$. If $d \in D'$, then $\mathcal{I}'_{x/d}$ is similar to $\mathcal{I}_{x/d}$ and, by the induction hypothesis, it is a model of $\varphi(x)$; $\mathcal{I}'$ is thus a model of ψ. If $d \notin D'$, then $d \in D_v$ and $|(D_v \cap D'_v) \setminus D^c| \geq q$. Furthermore, since the whole formula contains at most q variables, $\varphi(x)$ contains at most $q-1$ free variables besides x. Let $x_1, \ldots, x_m$ be those variables. There exists some $d' \in (D_v \cap D'_v) \setminus D^c$ such that $d' \neq \mathcal{I}[x_i]$ for all $i \in \{1, \ldots, m\}$. By structural induction, it is easy to show that $\mathcal{I}_{x/d}$ and $\mathcal{I}_{x/d'}$ give the same truth value to $\varphi(x)$. Furthermore $\mathcal{I}_{x/d'}$ and $\mathcal{I}'_{x/d'}$ are similar. $\mathcal{I}'$ is thus a model of ψ. To summarize, if $\mathcal{I}$ is a model of ψ, $\mathcal{I}'$ is also a model of ψ. By symmetry, if $\mathcal{I}'$ is a model of ψ, $\mathcal{I}$ is also a model of ψ. The proof for formulas of the form $\forall x\, \varphi(x)$ is dual. □

Lemma 1 has the following consequence on the acceptable cardinalities for the models of a Löwenheim theory:

Corollary 2. *Assume $\mathcal{T}$ is a Löwenheim theory with equality with n distinct unary predicates. Let r and q be respectively the number of constants and variables in $\mathcal{T}$. If $\mathcal{T}$ has a model of some cardinality κ strictly larger than $r + 2^n \max(0, q-1)$, then $\mathcal{T}$ has models of each cardinality equal or larger than $\min(\kappa, r + q\,2^n)$.*

Proof. If a model with such a cardinality exists, then there are Venn regions v such that $|D_v \setminus D^c| \geq q$. Then the number of elements in these Venn regions can be increased to any arbitrary larger cardinality, thanks to Lemma 1. If $\kappa > r + q\,2^n$, it means some Venn regions v are such that $|D_v \setminus D^c| > q$, and by eliminating elements in such Venn regions (using again Lemma 1), it is possible to obtain a model of cardinality $r + q\,2^n$. □

In [8], the limit is $q\,2^n$, q being the number of constants plus the maximum number of nested quantifiers. Now q is more precisely set to the number of variables, and the constants are counted separately. Moreover, $\max(0, q-1)$ replaces the factor q.

The case where q and r are both 0 corresponds to pure propositional logic (Löwenheim theories without variables and constants), where the size of the domain is not relevant. With $q = 1$ (one variable), there is no way to compare two elements (besides the ones associated to constants) and enforce them to be equal. It is still possible to constrain the domain to be of size at most r, using constraints like $\forall x\,.\,x = c_1 \vee \ldots \vee x = c_r$, but any model with one element not associated to a constant can be extended to a model of arbitrary cardinality (by somehow duplicating any number of time this element). Notice also that it is possible to set a lower bound on the size of the domain that can be $r + 2^n$. Consider for instance a set of sentences of the form $\exists x.(\neg)p_1(x) \vee \ldots (\neg)p_n(x)$; there are 2^n such formulas, each enforcing one Venn region to be non-empty.

Using several variables, a Löwenheim formula can enforce upper bounds larger than r on cardinalities. For $q = 2$, it is indeed easy to build a formula that has only models of cardinality at most $(q-1)\,2^n = 2^n$:

$$\forall x \forall y\,.\,\Big[\bigwedge_{0<i<j\leq n} p_i(x) = p_j(y)\Big] \Rightarrow x = y.$$

With a larger number of variables, the following formula ($q \geq 2$)

$$\forall x_1 \ldots \forall x_q\,.\,\Big[\bigwedge_{\substack{0<i<j\leq n \\ 0<i'<j'\leq q}} p_i(x_{i'}) = p_j(x_{j'})\Big] \Rightarrow \bigvee_{0<i'<j'\leq q} x_{i'} = x_{j'}$$

enforces the cardinality of the domain to be at most $(q-1)\,2^n$. To obtain a formula with constants that accepts only models of cardinality up to $r + 2^n \max(0, q-1)$, it suffices to add as a guard in the above formula the conjunctive sets of atoms expressing that the variables are disjoint from the r constants. So the above condition in Corollary 2 is the strongest one.

Besides the finite model property and the decidability of Löwenheim theories, Corollary 2 also directly entails the $\mathcal{P}$-gentleness:

Theorem 4. *Löwenheim theories on a language with unary predicates in $\mathcal{P}$ are $\mathcal{P}$-gentle.*

5 The Bernays-Schönfinkel-Ramsey Class

A Bernays-Schönfinkel-Ramsey (BSR for short) theory is a finite set of formulas of the form $\exists^*\forall^*\varphi$, where φ is a first-order formula which is function-free (but constants are allowed) and quantifier-free. Bernays and Schönfinkel first proved the decidability of this class without equality; Ramsey later proved that it remains decidable with equality. More can be found about BSR theories in [6]. Ramsey also gave some (less known) results about the spectrum of BSR theories [18]. We here give a proof that BSR theories are $\mathcal{P}$-gentle.

For simplicity, we will assume that existential quantifiers are Skolemized. In the following, a BSR theory is thus a finite set of universal function-free closed first-order formulas.

Lemma 2. *Let $\mathcal{T}$ be a BSR theory, and $\mathcal{M}$ be a model of $\mathcal{T}$ on domain D. Then any restriction $\mathcal{M}'$ of $\mathcal{M}$ on domain D' with $D^c \subseteq D' \subseteq D$ is a model of $\mathcal{T}$.*

Proof. Consider $\mathcal{M}$ and $\mathcal{M}'$ as above. Since $\mathcal{M}$ is a model of $\mathcal{T}$, for each closed formula $\forall x_1 \dots x_n \,.\, \varphi$ in $\mathcal{T}$ (where φ is function-free and quantifier-free), and for all $d_1, \dots, d_n \in D' \subseteq D$, $\mathcal{M}_{x_1/d_1,\dots,x_n/d_n}$ is a model of φ. This also means that, for all $d_1, \dots, d_n \in D'$, $\mathcal{M}'_{x_1/d_1,\dots,x_n/d_n}$ is a model of φ, and finally that $\mathcal{M}'$ is a model of $\forall x_1 \dots x_n \,.\, \varphi$. □

Intuitively, this states that the elements not assigned to ground terms (i.e. the constants) can be eliminated from a model of a BSR theory. It is known [18,9] that for any BSR theory $\mathcal{T}$ there is a computable finite number k such that if $\mathcal{T}$ has a model of cardinality greater or equal to k, then it has a model of any cardinality larger than k. Later in this section, we prove that the same occurs locally for each Venn region.

The notion of n-repetitive models, which we now define, is instrumental for this. Informally, a model is n-repetitive if it is symmetric for those elements of its domain that are not assigned to constants in the theory.

Definition 3. *An interpretation $\mathcal{I}$ on domain D for a BSR theory $\mathcal{T}$ is* n-repetitive for a set V of Venn regions *if, for each $v \in V$, $|D_v \setminus D^c| \geq n$ and there exists a total order $\prec$ on elements in $D_v \setminus D^c$ such that*

- *for every r-ary predicate symbol p in $\mathcal{T}$*
- *for all $d_1, \dots, d_r \in D$, and $d'_1, \dots, d'_r \in D$ with*
  - $|\{d_1, \dots, d_r\} \setminus D^c| \leq n$
  - $d'_i = d_i$ *if* d_i *or* $d'_i \in D^c \cup \bigcup_{v \notin V} D_v$
  - $v(d'_i) = v(d_i)$

- $d'_i \prec d'_j$ *iff* $d_i \prec d_j$, *if for some* $v' \in V$, $d_i, d_j \in D_{v'} \setminus D^c$

we have $\mathcal{I}[p](d_1, \ldots, d_r) = \mathcal{I}[p](d'_1, \ldots, d'_r)$.

Notice that a same interpretation can be n-repetitive for several Venn regions at the same time. Also, the above definition allows $D_v \setminus D^c$ to be empty for every $v \notin V$. Previously [9] (without distinguishing regions) we showed that one can decide if a BSR theory $\mathcal{T}$ is n-repetitive by building another BSR theory that is satisfiable if and only if $\mathcal{T}$ is n-repetitive. The same occurs to n-repetitiveness for Venn regions.

Theorem 5. *Consider a BSR theory* $\mathcal{T}$ *with* n *variables and a model* $\mathcal{M}$ *on domain* D. *If* $\mathcal{M}$ *is* n*-repetitive for the Venn regions* V *then, for any (finite or infinite) cardinalities* $\kappa_v \geq |D_v|$ $(v \in V)$, $\mathcal{T}$ *has a model* $\mathcal{M}'$ *extension of* $\mathcal{M}$ *on domain* D' *such that* $|D'_v| = \kappa_v$ *if* $v \in V$ *and* $D'_{v'} = D_{v'}$ *for all* $v' \notin V$.

Proof. Assume that $\prec$ are the total orders mentioned in Definition 3. We first build an extension $\mathcal{M}'$ of $\mathcal{M}$ as specified in the theorem, and later prove it is a model of $\mathcal{T}$.

Let E be the set of new elements $E = D' \setminus D$, and fix arbitrary total orders (again denoted by $\prec$) on $D'_v \setminus D^c$ for all $v \in V$ that extend the given orders on $D_v \setminus D^c$. Since $\mathcal{M}'$ is an extension of $\mathcal{M}$, the interpretation of the predicate symbols is already defined when all arguments belong to D. When some arguments belong to E, the truth value of an r-ary predicate p is defined as follows:

- $(d'_1, \ldots, d'_r) \notin \mathcal{M}'[p]$ for $|\{d'_1, \ldots, d'_r\} \setminus D^c| > n$: the interpretation of p over tuples with more than n elements outside D^c is fixed arbitrarily. Indeed, such tuples are irrelevant for the evaluation of the formulas of $\mathcal{T}$: terms occurring as arguments of a predicate are either variables or constants, and no more than n variables occur in any formula of $\mathcal{T}$.
- otherwise, to determine $\mathcal{M}'[p](d'_1, \ldots, d'_r)$, first choose $d_1, \ldots d_r \in D$ such that $d'_1, \ldots, d'_r$ and $d_1, \ldots d_r$ are related to each other just like in Definition 3. This is possible since, for every Venn region v for which the interpretation is repetitive, there are at least n elements in $D_v \setminus D^c$. Then $(d'_1, \ldots, d'_r) \in \mathcal{M}'[p]$ iff $(d_1, \ldots, d_r) \in \mathcal{M}[p]$. Observe that all possible choices of $d_1, \ldots, d_n$ lead to the same definition because $\mathcal{M}$ is n-repetitive.

The construction is such that $\mathcal{M}'$ is also n-repetitive for the same regions. It is also a model of $\mathcal{T}$: all formulas in $\mathcal{T}$ are of the form $\forall x_1 \ldots x_m \,.\, \varphi(x_1, \ldots, x_m)$, with $m \leq n$. For all $d'_1 \ldots, d'_m \in D'$, if $\{d'_1, \ldots, d'_m\} \subseteq D$ then

$$\mathcal{M}'_{x_1/d'_1, \ldots, x_m/d'_m}[\varphi(x_1, \ldots, x_m)] = \mathcal{M}_{x_1/d'_1, \ldots, x_m/d'_m}[\varphi(x_1, \ldots, x_m)]$$

since $\mathcal{M}'$ is an extension of $\mathcal{M}$. Otherwise, let $d_1, \ldots, d_m \in D$ be some elements related to $d'_1, \ldots, d'_m$ like in Definition 3. Since $\mathcal{M}'$ is n-repetitive,

$$\begin{aligned} \mathcal{M}'_{x_1/d'_1, \ldots, x_m/d'_m}[\varphi(x_1, \ldots, x_m)] &= \mathcal{M}'_{x_1/d_1, \ldots, x_m/d_m}[\varphi(x_1, \ldots, x_m)] \\ &= \mathcal{M}_{x_1/d_1, \ldots, x_m/d_m}[\varphi(x_1, \ldots, x_m)]. \end{aligned}$$

In both cases, $\mathcal{M}'_{x_1/d'_1, \ldots, x_m/d'_m}[\varphi(x_1, \ldots, x_m)]$ evaluates to true, and therefore $\mathcal{M}'$ is a model of $\forall x_1 \ldots x_n \,.\, \varphi(x_1, \ldots, x_m)$. □

Now it is possible to state that the full spectrum of a BSR theory only depends on (a finite set of) $\mathcal{P}$-cardinalities $\boldsymbol{\kappa}$ such that, for all Venn region v, $\kappa_v \leq k$ for some finite cardinality k only depending on the theory. The proof requires an extension of Ramsey's Theorem which can be found in the appendix A.

Theorem 6. *Given a BSR theory $\mathcal{T}$ with n variables, there exists a number k computable from the theory, such that, if $\mathcal{T}$ has a model $\mathcal{M}$ on domain D such that $|D_v \setminus D^c| \geq k$ for Venn regions $v \in V$, then it has a model which is n-repetitive for Venn regions V.*

Proof. Using Lemma 2, we can assume that $\mathcal{T}$ has a (sufficiently large) finite model $\mathcal{M}$ on domain D. We can assume without loss of generality that $\mathcal{M}$ is such that, for every predicate p of the language, $(d_1, \dots d_r) \notin \mathcal{M}[p]$ whenever there are more than n elements in $\{d_1, \dots d_r\} \setminus D^c$; indeed, these interpretations play no role in the truth value of a formula with n variables.

Let $\prec$ be an order on $D \setminus D^c$. Given two ordered (with respect to $\prec$) sequences $e_1, \dots, e_n$ and $e'_1, \dots, e'_n$ of elements in $D \setminus D^c$ such that $v(e_i) = v(e'_i)$ $(1 \leq i \leq n)$, we say that the configurations for $e_1, \dots, e_n$ and $e'_1, \dots, e'_n$ agree if for every r-ary predicate p, and for every $d_1, \dots, d_r \in D^c \cup \{e_1, \dots, e_n\}$, $(d_1, \dots, d_r) \in \mathcal{M}[p]$ iff $(d'_1, \dots, d'_r) \in \mathcal{M}[p]$, with $d'_i = e'_j$ if $d_i = e_j$ for some j, and $d'_i = d_i$ otherwise. Notice that there are only a finite number of disagreeing configurations for n elements in $D \setminus D^c$: more precisely a configuration is determined by at most $b = \sum_p (n + |D^c|)^{\text{arity}(p)}$ Boolean values, where the sum ranges over all predicates in the theory. Thus the number of disagreeing configurations is bounded by $C = 2^b$.

Interpreting configurations as colors, one can use the extension of Ramsey's Theorem given in Appendix A: according to Theorem 7, there is a computable function f such that, for any $N \in \mathbb{N}$, if $|D \setminus D^c|_V \geq f(n, N, C)$, then there exists a model on $D' \subseteq D$ with $|D' \setminus D^c|_V \geq N$ for which configurations agree if they have the same number of elements in each Venn region of V. Taking $N = n$, this is actually building a n-repetitive restriction of $\mathcal{M}$. □

The BSR class obviously has the finite model property, and is decidable. Lemma 2 and Theorems 5 and 6 above also prove that BSR theories are (gentle and) $\mathcal{P}$-gentle:

Corollary 3. *BSR theories on a language including unary predicates in $\mathcal{P}$ are $\mathcal{P}$-gentle.*

A simple constructive proof of this corollary would consider the finite number of all $\mathcal{P}$-cardinalities $\boldsymbol{\kappa}$ such that $\boldsymbol{\kappa}_v \leq k$ (where k comes from Theorem 6). All such $\mathcal{P}$-cardinalities can be understood as cardinality constraints, the extendable Venn regions being the ones for which $\boldsymbol{\kappa}_v > k$. Of course this construction is highly impractical, since it uses some kind of Ramsey numbers, known to be extremely large. In practice, we believe there are much better constructions: the important elements of the domain are basically only the ones associated to constants, and theoretical upper bounds are not met in non-artificial cases.

6 Example: Non-Disjoint Combination of Order and Sets

To illustrate the kind of theories that can be handled in our framework, consider a simple yet informative example with a BSR theory defining an ordering $<$ and augmented with clauses connecting the ordering $<$ and the sets p and q (we do not distinguish sets and their related predicates):

$$\mathcal{T}_1 = \begin{cases} \forall x.\ \neg(x < x) \\ \forall x, y, z.\ (x < y \wedge y < z) \Rightarrow x < z \\ \forall x, y.\ (p(x) \wedge \neg p(y)) \Rightarrow x < y \\ \forall x, y.\ (q(x) \wedge \neg q(y)) \Rightarrow x < y \end{cases}$$

and a Löwenheim theory

$$\mathcal{T}_2 = \begin{cases} \exists y \forall x.\ (p(x) \wedge q(x)) \equiv x = y \\ \forall x \exists y.\ p(x) \Rightarrow (x \neq y \wedge q(y)) \end{cases}$$

Notice that $\mathcal{T}_2$ is not a BSR theory due to the $\forall\exists$ quantification of its second axiom, but both theories $\mathcal{T}_1$ and $\mathcal{T}_2$ are actually $\mathcal{P}$-gentle. The theory $\mathcal{T}_1$ imposes either $p \cap \overline{q}$ or $\overline{p} \cap q$ to be empty (we will assume that the domain is non-empty and simplify the cardinality constraints accordingly). The theory $\mathcal{T}_2$ imposes the cardinality of $p \cap q$ to be exactly 1, and the cardinality of $\overline{p} \cap q$ to be at least 1. The following table collects the cardinality constraints:

	$\mathcal{T}_1$		$\mathcal{T}_2$
$\overline{p} \cap \overline{q}$	≥ 0	≥ 0	≥ 0
$\overline{p} \cap q$	0	≥ 0	≥ 1
$p \cap \overline{q}$	≥ 0	0	≥ 0
$p \cap q$	≥ 0	≥ 0	1

The theory $\mathcal{T}_1 \cup \mathcal{T}_2$ imposes $p \cap \overline{q}$ to be empty, in other words $p \subseteq q$. Moreover, the cardinality of $p \cap q$ is 1, and so it implies that the cardinality of p is 1. Hence, the set

$$\mathcal{T}_1 \cup \mathcal{T}_2 \cup \{p(a), p(b), a \neq b\}$$

is unsatisfiable. As a final comment, there could be theories using directly the Venn cardinalities as integer variables. For instance, imagine a constraint stating $|p| > 1$ in a theory including linear arithmetic on integers. This would of course be unsatisfiable with $\mathcal{T}_1 \cup \mathcal{T}_2$.

7 Conclusion

The notion of gentleness was initially presented as a tool to combine non-stably infinite disjoint theories. In this paper, we have introduced a notion of $\mathcal{P}$-gentleness which is well-suited for combining theories sharing (besides constants and the equality) only unary predicates in a set $\mathcal{P}$. The major contributions of this paper are that the Löwenheim theories and BSR theories are $\mathcal{P}$-gentle. A

corollary is that the non-disjoint union (sharing unary predicates) of Löwenheim theories, BSR theories, and decidable theories accepting further constraints of the form $\forall x\,.\,((\neg)p_1(x) \wedge \ldots (\neg)p_n(x)) \Rightarrow (x = a_1 \vee \ldots x = a_m)$ is decidable.

Our combination method is limited to shared unary predicates. Unfortunately, the theoretical limitations are strong for a framework sharing predicates with larger arities: for instance even the guarded fragment with two variables and transitivity constraints is undecidable [10], although the guarded fragment (or first-order logic with two variables) is decidable, and transitivity constraints can be expressed in BSR. The problem of combining theories with only a shared dense order has however been successfully solved [12,14]. In that specific case, there is again an implicit infiniteness argument that could be possibly expressed as a form of extended gentleness, to reduce the isomorphism construction problem into solving some appropriate extension of cardinality constraints. A clearly challenging problem is to identify an appropriate extended notion of gentleness for some particular binary predicates.

Also in future works, the reduction approach (Löwenheim and BSR theories can be simplified to a subset of Löwenheim) may be useful as a simplification procedure for sets of formulas that can be seen as non-disjoint (sharing unary predicates only) combinations of BSR, Löwenheim theories and an arbitrary first-order theory: this would of course not provide a decision procedure, but refutational completeness can be preserved. More generally we also plan to study how superposition-based satisfiability procedures could benefit from a non-disjoint (sharing unary predicates) combination point of view. In particular, superposition-based satisfiability procedures could be used as deductive engines with the capability to exchange constraints à la Nelson-Oppen.

The results here are certainly too combinatorially expensive to be directly applicable. However, this paper paves the theoretical grounds for mandatory further works that would make such combinations practical. There are important incentives since the BSR and Löwenheim fragments are quite expressive: for instance, it is possible to extend the language of SMT solvers with sets and cardinalities. Many formal methods are based on logic languages with sets. Expressive decision procedures (even if they are not efficient) including e.g. sets and cardinalities will help proving the often small but many verification conditions stemming from these applications.

References

1. Areces, C., Fontaine, P.: Combining theories: The Ackerman and Guarded fragments. In Tinelli, C., Sofronie-Stokkermans, V., eds.: Frontiers of Combining Systems (FroCoS). Volume 6989 of LNCS., Springer (2011) 40–54
2. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. **10**(1) (2009)
3. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Inf. Comput. **183**(2) (2003) 140–164
4. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T., eds.: Handbook of Satis-

fiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (February 2009) 825–885
5. Bonacina, M.P., Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decidability and undecidability results for Nelson-Oppen and rewrite-based decision procedures. In Furbach, U., Shankar, N., eds.: International Joint Conference on Automated Reasoning (IJCAR). Volume 4130 of LNCS., Springer (2006) 513–527
6. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Perspectives in Mathematical Logic. Springer-Verlag, Berlin (1997)
7. Chocron, P., Fontaine, P., Ringeissen, C.: A Gentle Non-Disjoint Combination of Satisfiability Procedures (Extended Version). Research Report 8529, Inria (2014) `http://hal.inria.fr/hal-00985135`.
8. Dreben, B., Goldfarb, W.D.: The Decision Problem: Solvable Classes of Quantificational Formulas. Addison-Wesley, Reading, Massachusetts (1979)
9. Fontaine, P.: Combinations of theories for decidable fragments of first-order logic. In Ghilardi, S., Sebastiani, R., eds.: Frontiers of Combining Systems (FroCoS). Volume 5749 of LNCS., Springer (2009) 263–278
10. Ganzinger, H., Meyer, C., Veanes, M.: The two-variable guarded fragment with transitive relations. In: Logic In Computer Science (LICS), IEEE Computer Society (1999) 24–34
11. Ganzinger, H., Nivelle, H.D.: A superposition decision procedure for the guarded fragment with equality. In: Logic In Computer Science (LICS), IEEE Computer Society Press (1999) 295–303
12. Ghilardi, S.: Model-theoretic methods in combined constraint satisfiability. Journal of Automated Reasoning **33**(3-4) (2004) 221–249
13. Gurevich, Y., Shelah, S.: Spectra of monadic second-order formulas with one unary function. In: Logic In Computer Science (LICS), Washington, DC, USA, IEEE Computer Society (2003) 291–300
14. Manna, Z., Zarba, C.G.: Combining decision procedures. In Aichernig, B.K., Maibaum, T.S.E., eds.: Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, Revised Papers. Volume 2757 of LNCS., Springer (2003) 381–422
15. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. on Programming Languages and Systems **1**(2) (October 1979) 245–257
16. Nicolini, E., Ringeissen, C., Rusinowitch, M.: Combinable extensions of Abelian groups. In Schmidt, R.A., ed.: Proc. Conference on Automated Deduction (CADE). Volume 5663 of LNCS., Springer (2009) 51–66
17. Nicolini, E., Ringeissen, C., Rusinowitch, M.: Combining satisfiability procedures for unions of theories with a shared counting operator. Fundam. Inform. **105**(1-2) (2010) 163–187
18. Ramsey, F.P.: On a Problem of Formal Logic. Proceedings of the London Mathematical Society **30** (1930) 264–286
19. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In Gramlich, B., ed.: Frontiers of Combining Systems (FroCoS). Volume 3717 of LNCS., Springer (2005) 48–64
20. Ringeissen, C., Senni, V.: Modular termination and combinability for superposition modulo counter arithmetic. In Tinelli, C., Sofronie-Stokkermans, V., eds.: Frontiers of Combining Systems (FroCoS). Volume 6989 of LNCS., Springer (2011) 211–226
21. Sofronie-Stokkermans, V.: Locality results for certain extensions of theories with bridging functions. In Schmidt, R.A., ed.: Proc. Conference on Automated Deduction (CADE). Volume 5663 of LNCS., Springer (2009) 67–83

22. Sofronie-Stokkermans, V.: On combinations of local theory extensions. In Voronkov, A., Weidenbach, C., eds.: Programming Logics - Essays in Memory of Harald Ganzinger. Volume 7797 of LNCS., Springer (2013) 392–413
23. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In Hermenegildo, M.V., Palsberg, J., eds.: Principles of Programming Languages (POPL), ACM (2010) 199–210
24. Tinelli, C., Ringeissen, C.: Unions of non-disjoint theories and combinations of satisfiability procedures. Theoretical Computer Science **290**(1) (2003) 291–353
25. Tinelli, C., Zarba, C.G.: Combining non-stably infinite theories. Journal of Automated Reasoning **34**(3) (April 2005) 209–238
26. Wies, T., Piskac, R., Kuncak, V.: Combining theories with shared set operations. In Ghilardi, S., Sebastiani, R., eds.: Frontiers of Combining Systems (FroCoS). Volume 5749 of LNCS., Springer (2009) 366–382
27. Zarba, C.G.: Combining sets with cardinals. J. Autom. Reasoning **34**(1) (2005) 1–29
28. Zhang, T., Sipma, H.B., Manna, Z.: Decision procedures for term algebras with integer constraints. Inf. Comput. **204**(10) (2006) 1526–1574

A An Extension of Ramsey's Theorem

We define an *n-subset* of S to be a subset of n elements of S. An *n-hypergraph* of S is a set of n-subsets of S. In particular, a 2-hypergraph is an (undirected) graph. The *complete* n-hypergraph of S is the set of all n-subsets of S, and its *size* is the cardinality of S. An n-hypergraph G is *colored* with c colors if there is a coloring function that assigns one color to every n-subset in G. In particular, a colored 2-hypergraph (that is, a colored graph), is a graph where all edges are assigned a color. Consider a set S of elements partitioned into disjoint regions $R = \{R_1, \dots R_m\}$. We say that a set $S' \subseteq S$ has *region size* larger than x and note $|S'|_R \geq x$ if $|S' \cap R_i| \geq x$ for all $i \in \{1, \dots, m\}$. We also say that an n-hypergraph is *region-monochromatic* if the color of each hyperedge only depends on the number of elements belonging to each region. Two hyperedges are said of the *same kind* if they have the same number of elements in each region; all hyperedges of the same kind of a region-monochromatic hypergraph thus have the same color. The following extension[4] of Ramsey's Theorem holds:

Theorem 7. *There exists a computable function f such that,*

- *for every number of colors c*
- *for every $n, N \in \mathbb{N}$*
- *for every complete n-hypergraph G on S colored with c colors*

if $|S|_R \geq f(n, N, c)$, then there exists a complete region-monochromatic n-subhypergraph of G on some $S' \subseteq S$ with $|S'|_R \geq N$.

Proof. The full proof can be found in [7]. □

[4] The classical Ramsey's Theorem is the case with only one region.

Exploiting symmetry in SMT problems

David Déharbe[1], Pascal Fontaine[2],
Stephan Merz[2], and Bruno Woltzenlogel Paleo[3⋆]

[1] Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil
david@dimap.ufrn.br
[2] University of Nancy and INRIA, Nancy, France
{Pascal.Fontaine,Stephan.Merz}@inria.fr
[3] Technische Universität Wien
bruno.wp@gmail.com

Abstract. Methods exploiting problem symmetries have been very successful in several areas including constraint programming and SAT solving. We here recast a technique to enhance the performance of SMT-solvers by detecting symmetries in the input formulas and use them to prune the search space of the SMT algorithm. This technique is based on the concept of (syntactic) invariance by permutation of constants. An algorithm for solving SMT by taking advantage of such symmetries is presented. The implementation of this algorithm in the SMT-solver veriT is used to illustrate the practical benefits of this approach. It results in a significant improvement of veriT's performances on the SMT-LIB benchmarks that places it ahead of the winners of the last editions of the SMT-COMP contest in the QF_UF category.

1 Introduction

While the benefit of symmetries has been recognized for the satisfiability problem of propositional logic [15], for constraint programming [9], and for finite model finding [4, 7, 11], SMT solvers (see [3] for a detailed account of techniques used in SMT solvers) do not yet fully exploit symmetries. Audemard et al. [1] use symmetries as a simplification technique for SMT-based model-checking, and the SMT solver HTP [14] uses some symmetry-based heuristics, but current state-of-the-art solvers do not exploit symmetries to decrease the size of the search space.

In the context of SMT solving, a frequent source of symmetries is when some terms take their value in a given finite set of totally symmetric elements. The idea here is very simple: given a formula G invariant by all permutations of some uninterpreted constants $c_0, \ldots, c_n$, for any model $\mathcal{M}$ of G, if term t does not contain these constants and $\mathcal{M}$ satisfies $t = c_i$ for some i, then there should be a model in which t equals c_0. While checking for unsatisfiability, it is thus sufficient to look for models assigning t and c_0 to the same value. This simple idea is very

⋆ This work was partly supported by the ANR DeCert project and the INRIA-CNPq project SMT-SAVeS.

effective, especially for formulas generated by finite instantiations of quantified problems. We have implemented our technique in a moderately efficient SMT solver (veriT [5]), and with this addition it outperforms the winners of recent editions of the SMT-COMP [2] contest in the QF_UF category. This indicates that detecting symmetries, automatically or based on hints in the input, can be important for provers to reduce the search space that they have to consider, just as some constraint solvers already take symmetry information into account.

Outline. We first introduce notations, then define symmetries and give the main theorem that allows us to reduce the search space. We recast an algorithm to exploit such symmetries in the context of SMT-solvers. Next, the classical pigeonhole problem is analyzed from the perspective of symmetries. Finally, some experimental results, based on the SMT-LIB, are provided and discussed.

2 Notations

A many-sorted first-order language is a tuple $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, d \rangle$ such that $\mathcal{S}$ is a countable non-empty set of disjoint sorts (or types), $\mathcal{V}$ is the (countable) union of disjoint countable sets $\mathcal{V}_\tau$ of variables of sort τ, $\mathcal{F}$ is a countably infinite set of function symbols, $\mathcal{P}$ is a countably infinite set of predicate symbols, and d assigns a sort in $\mathcal{S}^+$ to each function symbol $f \in \mathcal{F}$ and a sort in $\mathcal{S}^*$ to each predicate symbol $p \in \mathcal{P}$. Nullary predicates are propositions, and nullary functions are constants. The set of predicate symbols is assumed to contain a binary predicate $=_\tau$ for every sort $\tau \in \mathcal{S}$; since the sort of the equality can be deduced from the sort of the arguments, the symbol $=$ will be used for equality of all sorts. Terms and formulas over the language $\mathcal{L}$ are defined in the usual way.

An interpretation for a first-order language $\mathcal{L}$ is a pair $\mathcal{I} = \langle D, I \rangle$ where D assigns a non-empty domain D_τ to each sort $\tau \in \mathcal{S}$ and I assigns a meaning to each variable, function, and predicate symbol. As usual, the identity is assigned to the equality symbol. By extension, an interpretation $\mathcal{I}$ defines a value $\mathcal{I}[t]$ in D_τ for every term t of sort τ, and a truth value $\mathcal{I}[\varphi]$ in $\{\top, \bot\}$ for every formula φ. A model of a formula φ is an interpretation $\mathcal{I}$ such that $\mathcal{I}[\varphi] = \top$. The notation $\mathcal{I}_{s_1/r_1,\ldots,s_n/r_n}$ stands for the interpretation that agrees with $\mathcal{I}$, except that it associates the elements r_i of appropriate sort to the symbols s_i.

For convenience, we will consider that a theory is a set of interpretations for a given many-sorted language. The theory corresponding to a set of first-order axioms is thus naturally the set of models of the axioms. A theory may leave some predicates and functions uninterpreted: a predicate symbol p (or a function symbol f) is uninterpreted in a theory $\mathcal{T}$ if for every interpretation $\mathcal{I}$ in $\mathcal{T}$ and for every predicate q (resp., function g) of suitable sort, $\mathcal{I}_{p/q}$ belongs to $\mathcal{T}$ (resp., $\mathcal{I}_{f/g} \in \mathcal{T}$). It is assumed that variables are always uninterpreted in any theory, with a meaning similar to uninterpreted constants. Given a theory $\mathcal{T}$, a formula φ is $\mathcal{T}$-satisfiable if it has a model in $\mathcal{T}$. Two formulas are $\mathcal{T}$-equisatisfiable if one formula is $\mathcal{T}$-satisfiable if and only if the other is. A formula φ is a logical consequence of a theory $\mathcal{T}$ (noted $\mathcal{T} \models \varphi$) if every interpretation in $\mathcal{T}$ is a model

of φ. A formula φ is a $\mathcal{T}$-logical consequence of a formula ψ, if every model $\mathcal{M} \in \mathcal{T}$ of ψ is also a model of φ; this is noted $\psi \models_{\mathcal{T}} \varphi$. Two formulas ψ and φ are $\mathcal{T}$-logically equivalent if they have the same models in $\mathcal{T}$.

3 Defining symmetries

We now formally introduce the concept of formulas invariant w.r.t. permutations of uninterpreted symbols and study the $\mathcal{T}$-satisfiability problem of such formulas. Intuitively, the formula φ is invariant w.r.t. permutations of uninterpreted symbols if, modulo some syntactic normalization, it is left unchanged when the symbols are permuted. Formally, the notion of permutation operators depends on the theory $\mathcal{T}$ for which $\mathcal{T}$-satisfiability is considered, because only uninterpreted symbols may be permuted.

Definition 1. *A* permutation operator P *on a set* $\mathcal{R} \subseteq \mathcal{F} \cup \mathcal{P}$ *of uninterpreted symbols of a language* $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, d \rangle$ *is a sort-preserving bijective map from* $\mathcal{R}$ *to* $\mathcal{R}$*, that is, for each symbol* $s \in \mathcal{R}$*, the sorts of* s *and* $P[s]$ *are equal. A permutation operator homomorphically extends to an operator on terms and formulas on the language* $\mathcal{L}$*.*

As an example, a permutation operator on a language containing the three constants c_0, c_1, c_2 of identical sort, may map c_0 to c_1, c_1 to c_2 and c_2 to c_0.

To formally define that a formula is invariant by a permutation operator *modulo some rewriting*, the concept of $\mathcal{T}$-preserving rewriting operator is introduced.

Definition 2. *A* $\mathcal{T}$-preserving rewriting operator R *is any transformation operator on terms and formulas such that* $\mathcal{T} \models t = R[t]$ *for any term, and* $\mathcal{T} \models \varphi \Leftrightarrow R[\varphi]$ *for any formula* φ*. Moreover, for any permutation operator* P*, for any term and any formula,* $R \circ P \circ R$ *and* $R \circ P$ *should yield identical results.*

The last condition of Def. 2 will be useful in Lemma 6. Notice that R must be idempotent, since $R \circ P \circ R$ and $P \circ R$ should be equal for all permutation operators, including the identity permutation operator.

To better motivate the notion of a $\mathcal{T}$-preserving rewriting operator, consider a formula containing a clause $t = c_0 \vee t = c_1$. Obviously this clause is symmetric if t does not contain the constants c_0 and c_1. However, a permutation operator on the constants c_0 and c_1 would rewrite the formula into $t = c_1 \vee t = c_0$, which is not syntactically equal to the original one. Assuming the existence of some ordering on terms and formulas, a typical $\mathcal{T}$-preserving rewriting operator would reorder arguments of all commutative symbols according to this ordering. With appropriate data structures to represent terms and formulas, it is possible to build an implementation of this $\mathcal{T}$-preserving rewriting operator that runs in linear time with respect to the size of the DAG or tree that represents the formula.

Definition 3. *Given a $\mathcal{T}$-preserving rewriting operator R, a permutation operator P on a language $\mathcal{L}$ is a* symmetry operator *of a formula φ (a term t) on the language $\mathcal{L}$ w.r.t. R if $R[P[\varphi]]$ and $R[\varphi]$ (resp., $R[P[t]]$ and $R[t]$) are identical.*

Notice that, given a permutation operator P and a linear time $\mathcal{T}$-preserving rewriting operator R satisfying the condition of Def. 3, it is again possible to check in linear time if P is a symmetry operator of a formula w.r.t. R. In the following, we will assume a fixed rewriting operator R and say that P is a symmetry operator if it is a symmetry operator w.r.t. R.

Symmetries could alternatively be defined semantically, stating that a permutation operator P is a symmetry operator if $P[\varphi]$ is $\mathcal{T}$-logically equivalent to φ. The above syntactical symmetry implies of course the semantical symmetry. But the problem of checking if a permutation operator is a semantical symmetry operator has the same complexity as the problem of unsatisfiability checking. Indeed, consider the permutation P such that $P[c_0] = c_1$ and $P[c_1] = c_0$, and a formula ψ defined as $c = c_0 \wedge c \neq c_1 \wedge \psi'$ (where c, c_0 and c_1 do not occur in ψ'). To check if the permutation operator P is a semantical symmetry operator of ψ, it is necessary to check if formulas ψ and $P[\psi]$ are logically equivalent, which is only the case if ψ' is unsatisfiable.

Definition 4. *A term t (a formula φ) is* invariant w.r.t. permutations of uninterpreted constants $c_0, \ldots, c_n$ *if any permutation operator P on $c_0, \ldots, c_n$ is a symmetry operator of t (resp. φ).*

The main theorem follows: it allows one to introduce a symmetry breaking assumption in a formula that is invariant w.r.t. permutations of constants. This assumption will decrease the size of the search space.

Theorem 5. *Consider a theory $\mathcal{T}$, uninterpreted constants $c_0, \ldots, c_n$, a formula φ that is invariant w.r.t. permutations of $c_i, \ldots, c_n$, and a term t that is invariant w.r.t. permutations of $c_i, \ldots, c_n$. If $\varphi \models_{\mathcal{T}} t = c_0 \vee \ldots \vee t = c_n$, then φ is $\mathcal{T}$-satisfiable if and only if*

$$\varphi' =_{\text{def}} \varphi \wedge (t = c_0 \vee \ldots \vee t = c_i)$$

is also $\mathcal{T}$-satisfiable. Clearly, φ' is invariant w.r.t. permutations of $c_{i+1}, \ldots, c_n$.

Proof: Let us first prove the theorem for $i = 0$.

Assume that $\varphi \wedge t = c_0$ is $\mathcal{T}$-satisfiable, and that $\mathcal{M} \in \mathcal{T}$ is a model of $\varphi \wedge t = c_0$; $\mathcal{M}$ is also a model of φ, and thus φ is $\mathcal{T}$-satisfiable.

Assume now that φ is $\mathcal{T}$-satisfiable, and that $\mathcal{M} \in \mathcal{T}$ is a model of φ. By assumption there exists some $j \in \{0, \ldots, n\}$ such that $\mathcal{M} \models t = c_j$, hence $\mathcal{M} \models \varphi \wedge t = c_j$. In the case where $j = 0$, $\mathcal{M}$ is also a model of $\varphi \wedge t = c_0$. If $j \neq 0$, consider the permutation operator P that swaps c_0 and c_j. Notice (this can be proved by structural induction on formulas) that, for any formula ψ, $\mathcal{M} \models \psi$ if and only if $\mathcal{M}_{c_0/d_j, c_j/d_0} \models P[\psi]$, where d_0 and d_j are respectively $\mathcal{M}[c_0]$ and $\mathcal{M}[c_j]$. Choosing $\psi =_{\text{def}} \varphi \wedge t = c_j$, it follows that $\mathcal{M}_{c_0/d_j, c_j/d_0} \models P[\varphi \wedge t = c_j]$, and thus $\mathcal{M}_{c_0/d_j, c_j/d_0} \models P[\varphi] \wedge t = c_0$ since t is invariant w.r.t.

permutations of $c_0, \ldots, c_n$. Furthermore, since φ is invariant w.r.t. permutations of $c_0, \ldots, c_n$, $R[P[\varphi]]$ is φ for the fixed $\mathcal{T}$-preserving rewriting operator. Since R is $\mathcal{T}$-preserving, $\mathcal{M}_{c_0/d_j, c_j/d_0} \models P[\varphi]$ if and only if $\mathcal{M}_{c_0/d_j, c_j/d_0} \models R[P[\varphi]]$, that is, if and only if $\mathcal{M}_{c_0/d_j, c_j/d_0} \models \varphi$. Finally $\mathcal{M}_{c_0/d_j, c_j/d_0} \models \varphi \wedge t = c_0$, and $\mathcal{M}_{c_0/d_j, c_j/d_0}$ belongs to $\mathcal{T}$ since c_0 and c_j are uninterpreted. The formula $\varphi \wedge t = c_0$ is thus $\mathcal{T}$-satisfiable.

For the general case, notice that $\varphi'' =_{\text{def}} \varphi \wedge \neg(t = c_0 \vee \ldots \vee t = c_{i-1})$ is invariant w.r.t. permutations of $c_i, \ldots, c_n$, and $\varphi'' \models_{\mathcal{T}} t = c_i \vee \ldots \vee t = c_n$. By the previous case (applied to the set of constants $c_i, \ldots, c_n$ instead of $c_0, \ldots, c_n$), φ'' is $\mathcal{T}$-equisatisfiable to $\varphi \wedge \neg(t = c_0 \vee \ldots \vee t = c_{i-1}) \wedge t = c_i$. Formulas φ and

$$\big(\varphi \wedge \neg(t = c_0 \vee \ldots \vee t = c_{i-1})\big) \vee \big(\varphi \wedge (t = c_0 \vee \ldots \vee t = c_{i-1})\big)$$

are $\mathcal{T}$-logically equivalent. Since $A \vee B$ and $A' \vee B$ are $\mathcal{T}$-equisatisfiable whenever A and A' are $\mathcal{T}$-equisatisfiable, φ is $\mathcal{T}$-equisatisfiable to

$$\big(\varphi \wedge \neg(t = c_0 \vee \ldots \vee t = c_{i-1}) \wedge t = c_i\big) \vee \big(\varphi \wedge (t = c_0 \vee \ldots \vee t = c_{i-1})\big).$$

This last formula is $\mathcal{T}$-logically equivalent to

$$\varphi \wedge (t = c_0 \vee \ldots \vee t = c_{i-1} \vee t = c_i)$$

and thus the theorem holds. □

Checking if a permutation is syntactically equal to the original term or formula can be done in linear time. And checking if a formula is invariant w.r.t. permutations of given constants is also linear: only two permutations have to be considered instead of the $n!$ possible permutations.

Lemma 6. *A formula φ is invariant w.r.t. permutations of constants $c_0, \ldots, c_n$ if both permutation operators*

- *P_{circ} such that $P_{\text{circ}}[c_i] = c_{i-1}$ for $i \in \{1, \ldots, n\}$ and $P_{\text{circ}}[c_0] = c_n$,*
- *P_{swap} such that $P_{\text{swap}}[c_0] = c_1$ and $P_{\text{swap}}[c_1] = c_0$*

are symmetry operators for φ.

Proof: First notice that any permutation operator on $c_0, \ldots, c_n$ can be written as a product of P_{circ} and P_{swap}, because the group of permutations of $c_0, \ldots, c_n$ is generated by the circular permutation and the swapping of c_0 and c_1. Any permutation P of $c_0, \ldots, c_n$ can then be rewritten as a product $P_1 \circ \cdots \circ P_m$, where $P_i \in \{P_{\text{circ}}, P_{\text{swap}}\}$ for $i \in \{1, \ldots, m\}$. It remains to prove that any permutation operator $P_1 \circ \cdots \circ P_m$ is indeed a symmetry operator. This is done inductively. For $m = 1$ this is trivially true. For the inductive case, assume $P_1 \circ \cdots \circ P_{m-1}$ is a symmetry operator of φ, then

$$\begin{aligned} R[(P_1 \circ \ldots \circ P_m)[\varphi]] &\equiv R[P_m[(P_1 \circ \cdots \circ P_{m-1})[\varphi]]] \\ &\equiv R[P_m[R[(P_1 \circ \cdots \circ P_{m-1})[\varphi]]]] \\ &\equiv R[P_m[\varphi]] \\ &\equiv R[\varphi] \end{aligned}$$

```
1  P := guess_permutations(φ);
2  foreach {c_0, ..., c_n} ∈ P do
3  |  if invariant_by_permutations(φ, {c_0, ..., c_n}) then
4  |  |  T := select_terms(φ, {c_0, ..., c_n}) ;
5  |  |  cts := ∅ ;
6  |  |  while T ≠ ∅ ∧ |cts| ≤ n do
7  |  |  |  t := select_most_promising_term(T, φ) ;
8  |  |  |  T := T \ {t} ;
9  |  |  |  cts := cts ∪ used_in(t, {c_0, ..., c_n}) ;
10 |  |  |  let c ∈ {c_0, ..., c_n} \ cts;
11 |  |  |  cts := cts ∪ {c};
12 |  |  |  if cts ≠ {c_0, ..., c_n} then
13 |  |  |  |  φ := φ ∧ ( ⋁_{c_i ∈ cts} t = c_i );
14 |  |  |  end
15 |  |  end
16 |  end
17 end
18 return φ;
```

Algorithm 1: A symmetry breaking preprocessor.

where $\equiv$ stands for syntactical equality. The first equality simply expands the definition of the composition operator $\circ$, the second comes from the definition of the $\mathcal{T}$-preserving rewriting operator R, the third uses the inductive hypothesis, and the last uses the fact that P_m is either P_{circ} or P_{swap}, that is, also a symmetry operator of φ. □

4 SMT with symmetries: an algorithm

Algorithm 1 applies Theorem 5 in order to exhaustively add symmetry breaking assumptions on formulas. First, a set of sets of constants is guessed (line 1) from the formula φ by the function *guess_permutations*; each such set of constants $\{c_0, \ldots, c_n\}$ will be successively considered (line 2), and invariance of φ w.r.t. permutations of $\{c_0, \ldots, c_n\}$ will be checked (line 3). Notice that function *guess_permutations*(φ) gives an approximate solution to the problem of partitioning constants of φ into classes $\{c_0, \ldots, c_n\}$ of constants such that φ is invariant by permutations. If the $\mathcal{T}$-preserving rewriting operator R is given, then this is a decidable problem. However we have a feeling that, while the problem is still polynomial (it suffices to check all permutations with pairs of constants), only providing an approximate solution is tractable. Function *guess_permutations* should be such that a small number of tentative sets are returned. Every tentative set will be checked in function *invariant_by_permutations* (line 3); with appropriate data structures the test is linear with respect to the size of φ (as a corollary of Lemma 6).

As a concrete implementation of function *guess_permutations*(φ), partitioning the constants in classes that all give the same values to some functions $f(\varphi, c)$ works well in practice, where the functions f compute syntactic information that is unaffected by permutations, i.e. $f(\varphi, c)$ and $f(P[\varphi], P[c])$ should yield the same results. Obvious examples of such functions are the number of appearances of c in φ, or the maximal depth of c within an atom of φ, etc. The classes of constants could also take into account the fact that, if φ is a large conjunction, with $c_0 \neq c_1$ as a conjunct (c_0 and c_1 in the same class), then it should have $c_i \neq c_j$ or $c_j \neq c_i$ as a conjunct for every pair of different constants c_i, c_j contained in the class of c_0 and c_1. In veriT we use a straightforward detection of clusters $c_0, \ldots, c_n$ of constants such that there exists an inequality $c_i \neq c_j$ for every $i \neq j$ as a conjunct in the original formula φ.

Line 3 checks the invariance of formula φ by permutation of $c_0, \ldots, c_n$. In veriT, function *invariant_by_permutations*($\varphi, \{c_0, \ldots, c_n\}$) simply builds, in linear time, the result of applying a circular permutation of $c_0, \ldots, c_n$ to φ, and the result of applying a permutation swapping two constants (for instance c_0 and c_1). Both obtained formulas, as well as the original one, are normalized by a rewriting operator sorting arguments of conjunctions, disjunctions, and equality according to an arbitrary term ordering. The three formulas should be syntactically equal (this is tested in constant time thanks to the maximal sharing of terms in veriT) for *invariant_by_permutations*($\varphi, \{c_0, \ldots, c_n\}$) to return true.

Lines 4 to 15 concentrate on breaking the symmetry of $\{c_0, \ldots, c_n\}$. First a set of terms

$$T \subseteq \{t \mid \varphi \models t = c_0 \vee \ldots \vee t = c_n\}$$

is computed. Again, function *select_terms*($\varphi, \{c_0, \ldots, c_n\}$) returns an approximate solution to the problem of getting all terms t such that $t = c_0 \vee \ldots \vee t = c_n$; an omission in $\mathcal{T}$ would simply restrict the choices for a good candidate on line 7, but would not jeopardize soundness. Again, this is implemented in a straightforward way in veriT.

The loop on lines 6 to 15 introduces a symmetry breaking assumption on every iteration (except perhaps on the last iteration, where a subsumed assumption would be omitted). A candidate symmetry-breaking term $t \in T$ is chosen by the call *select_most_promising_term*(T, φ). The efficiency of the SMT solver is very sensitive to this selection function. If the term t is not important for unsatisfiability, then the assumption would simply be useless. In veriT, the selected term is the most frequent constant-free term (i.e. the one with the highest number of clauses in which it appears), or, if no constant-free terms remains, the one with the largest ratio of the number of clauses in which the term appears over the number of constants that will be required to add to *cts* on line 11; so actually, *select_most_promising_term* also depends on the set *cts*.

Function *used_in*($t, \{c_0, \ldots, c_n\}$) returns the set of constants in term t. If the term contains constants in $\{c_0, \ldots, c_n\} \setminus cts$, then only the remaining constants can be used. On line 10, one of the remaining constants c is chosen nondeterministically: in principle, any of these constants is suitable, but the choice

may take into account accidental features that influence the decision heuristics of the SMT solver, such as term orderings.

Finally, if the symmetry breaking assumption $\bigvee_{c_i \in cts} t = c_i$ is not subsumed (i.e. if $cts \neq \{c_0, \ldots, c_n\}$), then it is conjoined to the original formula.

Theorem 7. *The formula φ obtained after running Algorithm 1 is $\mathcal{T}$-satisfiable if and only if the original formula φ_0 is $\mathcal{T}$-satisfiable.*

Proof: If the obtained φ is $\mathcal{T}$-satisfiable then φ_0 is $\mathcal{T}$-satisfiable since φ is a conjunction of φ_0 and other formulas (the symmetry breaking assumptions).

Assume that φ_0 is $\mathcal{T}$-satisfiable, then φ is $\mathcal{T}$-satisfiable, as a direct consequence of Theorem 5. In more details, in lines 6 to 15, φ is always invariant by permutation of constants $\{c_0, \ldots, c_n\} \setminus cts$, and more strongly, on line 13, φ is invariant by permutations of constants in cts as defined in line 9. In lines 4 to 15 any term $t \in T$ is such that $\varphi \models_{\mathcal{T}} t = c_0 \vee \ldots \vee t = c_n$. On lines 10 to 14, t is invariant with respect to permutations of constants in cts as defined in line 9. The symmetry breaking assumption conjoined to φ in line 13 is, up to the renaming of constants, the symmetry breaking assumption of Theorem 5 and all conditions of applicability of this theorem are fulfilled. □

5 SMT with symmetries: an example

A classical problem with symmetries is the pigeonhole problem. Most SMT or SAT solvers require exponential time to solve this problem; these solvers are strongly linked with the resolution calculus, and an exponential lower bound for the length of resolution proofs of the pigeon-hole principle was proved in [10]. Polynomial-length proofs are possible in stronger proof systems, as shown by Buss [6] for Frege proof systems. An extensive survey on the proof complexity of pigeonhole principles can be found in [13]. Polynomial-length proofs are also possible if the resolution calculus is extended with symmetry rules (as in [12] and in [17]).

We here recast the pigeonhole problem in the SMT language and show that the preprocessing introduced previously transforms the series of problems solved in exponential time with standard SMT solvers into a series of problems solved in polynomial time. This toy problem states that it is impossible to place $n+1$ pigeons in n holes. We introduce n uninterpreted constants $h_1, \ldots, h_n$ for the n holes, and $n+1$ uninterpreted constants $p_1, \ldots, p_{n+1}$ for the $n+1$ pigeons. Each pigeon is required to occupy one hole:

$$p_i = h_1 \vee \ldots \vee p_i = h_n$$

It is also required that distinct pigeons occupy different holes, and this is expressed by the clauses $p_i \neq p_j$ for $1 \leq i < j \leq n+1$. One can also assume that the holes are distinct, i.e., $h_i \neq h_j$ for $1 \leq i < j \leq n$, although this is not needed for the problem to be unsatisfiable.

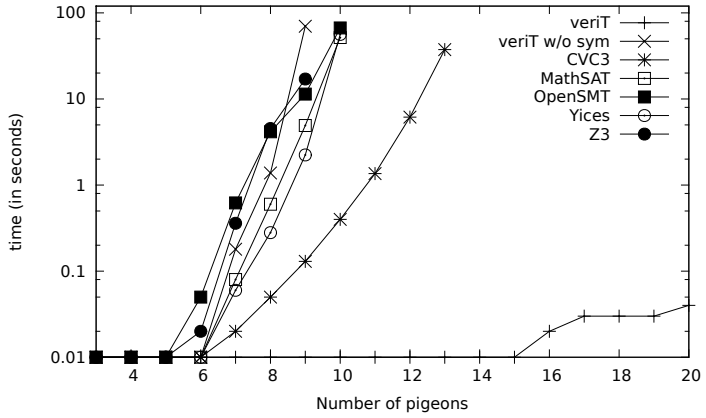


Fig. 1. Some SMT solvers and the pigeonhole problem

The generated set of formulas is invariant by permutations of the constants $p_1, \ldots, p_{n+1}$, and also by permutations of constants $h_1, \ldots, h_n$; very basic heuristics would easily guess this invariance. However, it is not obvious from the presentation of the problem that $h_i = p_1 \vee \ldots \vee h_i = p_{n+1}$ for $i \in [1..n]$, so any standard function *select_terms* in the previous algorithm will fail to return any selectable term to break the symmetry; this symmetry of $p_1, \ldots, p_{n+1}$ is not directly usable. It is however most direct to notice that $p_i = h_1 \vee \ldots \vee p_i = h_n$; *select_terms* in the previous algorithm would return the set of $\{p_1, \ldots, p_{n+1}\}$. The set of symmetry breaking clauses could be

$$
\begin{aligned}
& p_1 = h_1 \\
& p_2 = h_1 \vee p_2 = h_2 \\
& p_3 = h_1 \vee p_3 = h_2 \vee p_3 = h_3 \\
& \quad \vdots \\
& p_{n-1} = h_1 \vee \ldots \vee p_{n-1} = h_{n-1}
\end{aligned}
$$

or any similar set of clauses obtained from these with by applying a permutation operator on $p_1, \ldots, p_{n+1}$ and a permutation operator on $h_1, \ldots, h_n$. Without need for any advanced theory propagation techniques[4], $(n+1) \times n/2$ conflict clauses of the form $p_i \neq h_i \vee p_j \neq h_i \vee p_j \neq p_i$ with $i < j$ suffice to transform the problem into a purely propositional problem. With the symmetry breaking clauses, the underlying SAT solver then concludes (in polynomial time) the unsatisfiability of the problem using only Boolean Constraint Propagation.

Without the symmetry breaking clauses, the SAT solver will have to investigate all $n!$ assignments of n pigeons in n holes, and conclude for each of those assignments that the pigeon $n+1$ cannot find any unoccupied hole.

[4] Theory propagation in veriT is quite basic: only equalities deduced from congruence closure are propagated. $p_i \neq h_i$ would never be propagated from $p_j = h_i$ and $p_i \neq p_j$.

The experimental results, shown in Figure 1, support this analysis: all solvers (including veriT without symmetry heuristics) time out[5] on problems of relatively small size, although CVC3 performs significantly better than the other solvers. Using the symmetry heuristics allows veriT to solve much larger problems in insignificant times. In fact, the modified version of veriT solves every instance of the problem with as many as 30 pigeons in less than 0.15 seconds.

6 Experimental results

In the previous section we showed that detecting and breaking symmetries can sometimes decrease the solving time from exponential to polynomial. We now investigate its use on more realistic problems by evaluating its impact on SMT-LIB benchmarks.

Consider a problem on a finite domain of a given cardinality n, with a set of arbitrarily quantified formulas specifying the properties for the elements of this domain. A trivial way to encode this problem into quantifier-free first-order logic, is to introduce n constants $\{c_1, \ldots, c_n\}$, add constraints $c_i \neq c_j$ for $1 \leq i < j \leq n$, Skolemize the axioms and recursively replace in the Skolemized formulas the remaining quantifiers $Qx.\varphi(x)$ by conjunctions (if Q is $\forall$) or disjunctions (if Q is $\exists$) of all formulas $\varphi(c_i)$ (with $1 \leq i \leq n$). All terms should also be such that $t = c_1 \vee \ldots \vee t = c_n$. The set of formulas obtained in this way is naturally invariant w.r.t. permutations of $c_1, \ldots, c_n$. So the problem in its most natural encoding contains symmetries that should be exploited in order to decrease the size of the search space. The QF_UF category of the SMT library of benchmarks actually contains many problems of this kind.

Figure 2 presents a scatter plot of the running time of veriT on each formula in the QF_UF category. The x axis gives the running times of veriT without the symmetry breaking technique presented in this paper, whereas the times reported on the y axis are the running times of full veriT. It clearly shows a global improvement; this improvement is even more striking when one restricts the comparison to unsatisfiable instances (see Figure 3); no significant trend is observable on satisfiable instances only. We understand this behavior as follows: for some (not all) satisfiable instances, adding the symmetry breaking clauses "randomly" influences the decision heuristics of the SAT solver in such a way that it sometimes takes more time to reach a satisfiable assignment; in any way, if there is a satisfiable assignment, then all permutations of the uninterpreted constants (i.e. the ones for which the formula is invariant) are also satisfiable assignments, and there is no advantage in trying one rather than an other. For unsatisfiable instances, if terms breaking the invariance play a role in the unsatisfiability of the problem, then adding the symmetry breaking clauses always reduces the number of cases to consider, potentially by a factor of $n^n/n!$ (where n is the number of constants), and have a negligible impact if the symmetry breaking terms play no role in the unsatisfiability.

[5] The timeout was set to 120 seconds, using Linux 64 bits on Intel(R) Xeon(R) CPU E5520 at 2.27GHz, with 24 GBytes of memory.

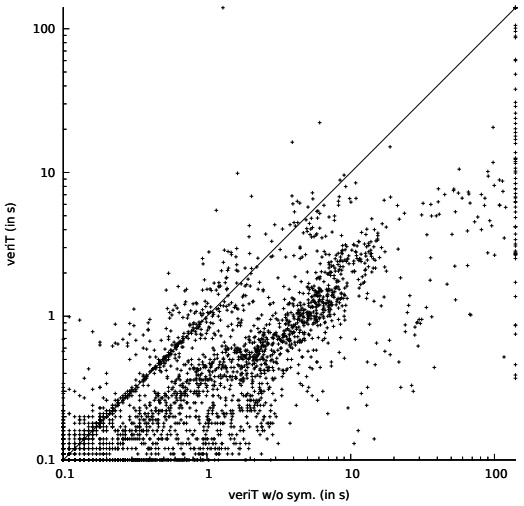


Fig. 2. Efficiency in solving individual instances: veriT vs. veriT without symmetries on all formulas in the QF_UF category. Each point represents a benchmark, and its horizontal and vertical coordinates represent the time necessary to solve it (in seconds). Points on the rightmost and topmost edges represent a timeout.

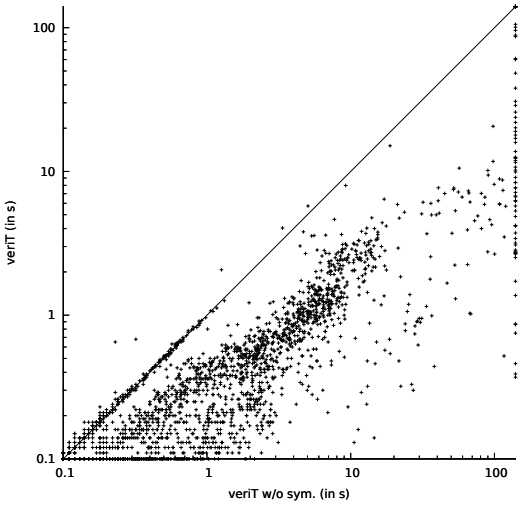


Fig. 3. Efficiency in solving individual instances: veriT vs. veriT without symmetries on the unsatisfiable instances of the QF_UF category.

	Nb. of instances		Instances within time range (in s)						Total time	
	success	timeout	0-20	20-40	40-60	60-80	80-100	100-120	T	T'
veriT	6633	14	6616	9	2	1	3	2	3447	5127
veriT w/o sym.	6570	77	6493	33	14	9	12	9	10148	19388
CVC3	6385	262	6337	20	12	7	5	4	8118	29598
MathSAT	6547	100	6476	49	12	6	3	1	5131	7531
openSMT	6624	23	6559	43	13	6	1	2	5345	8105
Yices	6629	18	6565	32	23	5	1	3	4059	6219
Z3	6621	26	6542	33	23	15	4	4	6847	9967

Table 1. Some SMT solvers on the QF_UF category

To compare with the state-of-the-art solvers, we selected all competing solvers in SMT-COMP 2010, adding also Z3 (for which we took the most recent version running on Linux we could find, namely version 2.8), and Yices (which was competing as the 2009 winner). The results are presented in Table 1. Columns T and T' are the total time, in seconds, on the QF_UF library, excluding and including timeouts, respectively. It is important to notice that these results include the whole QF_UF library of benchmarks, that is, with the diamond benchmarks. These benchmarks require some preprocessing heuristic [16] which does not seem to be implemented in CVC3 and MathSAT. This accounts for 83 timeouts in CVC3 and 80 in MathSAT. According to this table, with a 120 seconds timeout, the best solvers on QF_UF without the diamond benchmarks are (in decreasing order) veriT with symmetries, Yices, MathSAT, openSMT, CVC3. Exploiting symmetries allowed veriT to jump from the second last to the first place of this ranking. Within 20 seconds, it now solves over 50 benchmarks more than the next-best solver.

Figure 4 presents another view of the same experiment; it clearly shows that veriT is always better (in the number of solved instances within a given timeout) than any other solver except Yices, but it even starts to be more successful that Yices when the timeout is larger than 3 seconds. Scatter plots of veriT against the solvers mentioned above give another comparative view; they are available in Appendix A. Again the benefits on the zone with a time smaller that 3 seconds on both axes is not always clear. Also, bear in mind that the satisfiable instances do not benefit from the technique and still exhibit on the scatter plot the somewhat poor efficiency of veriT without symmetries. But the zone between 3 and 120 seconds on the x axis is clearly more populated than the zone between 3 and 120 seconds on the y axis.

Table 2 presents a summary of the symmetries found in the QF_UF benchmark category. Among 6647 problems, 3310 contain symmetries tackled by our method. For 2698 problems, the symmetry involves 5 constants; for most of them, 3 symmetry breaking clauses were added.

The technique presented in this paper is a preprocessing technique, and, as such, it is applicable to the other solvers mentioned here. We conducted an experiment on the QF_UF benchmarks augmented with the symmetry breaking

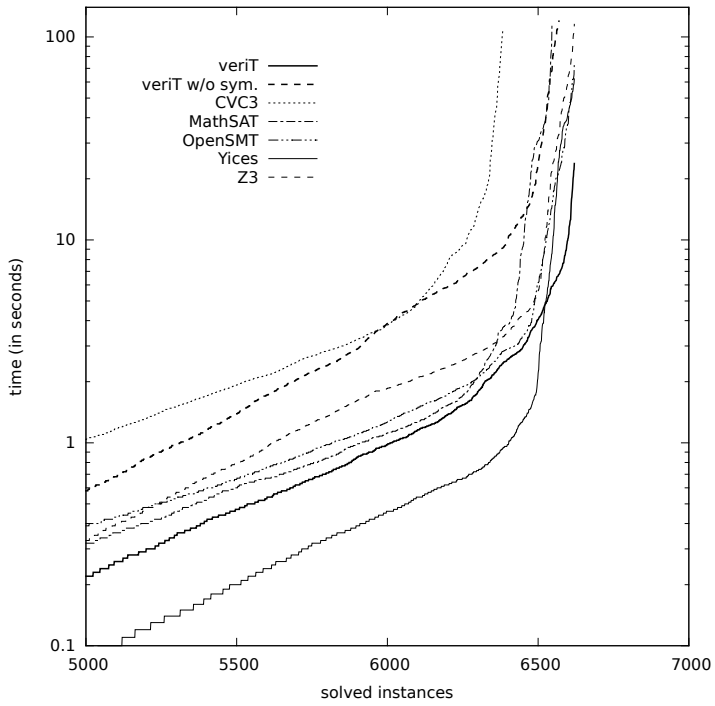


Fig. 4. Number of solved instances of QF_UF within a time limit, for some SMT solvers.

n_c \ n_{sym}	2	3	4	5	6	7	8	9	10	11
1	2									
2		12		8						
3			24	2668						
4				22	92	3				
5					122	166				
6						156				
7							17			
8								11		
9									5	
10										2
Total	2	12	24	2698	214	325	17	11	5	2

Table 2. Symmetries detected for the QF_UF category: n_{sym} indicates the number of constants involved in the symmetry, n_c the number of symmetry breaking clauses.

clauses. We observed the same kind of impressive improvement for all solvers. The most efficient solvers solve all but very few instances (diamond benchmarks excluded): within a time limit of 120s and on the whole library, Yices only fails for one formula, CVC for 36, and the others fails for 3 or 4 formulas. We also observe a significant decrease in cumulative times, the most impressive being Yices solving the full QF_UF library but one formula in around 10 minutes. Scatter plots exhibiting the improvements are available in Appendix B.

7 Conclusion

Symmetry breaking techniques have been used very successfully in the areas of constraint programming and SAT solving. We here present a study of symmetry breaking in SMT. It has been shown that the technique can account for an exponential decrease of running times on some series of crafted benchmarks, and that it significantly improves performances in practice, on the QF_UF category of the SMT library, a category for which the same solver performed fastest in 2009 and 2010. It may be argued that the heuristic has only be shown to be effective on the pigeonhole problem and competition benchmarks in the QF_UF category. However, we believe that in their most natural encoding many concrete problems contain symmetries; provers in general and SMT solvers in particular should be aware of those symmetries to avoid unnecessary exponential blowup. We are particularly interested in proof obligations stemming from verification of distributed systems; in this context many processes may be symmetric, and this should translate to symmetries in the corresponding proof obligations.

Although the technique is applicable in the presence of quantifiers and interpreted symbols, it appears that symmetries in the other SMT categories are somewhat less trivial, and so, require more clever heuristics for guessing invariance, as well as more sophisticated symmetry breaking tools. This is left for future work. Also, our technique is inherently non-incremental, that is, symmetry breaking assumptions should be retrieved, and checked against new assertions when the SMT solver interacts in an incremental manner with the user. This is not a major issue, but it certainly requires a finer treatment within the SMT solver than simple preprocessing.

The veriT solver is open sourced under the BSD license and is available on `http://www.veriT-solver.org`.

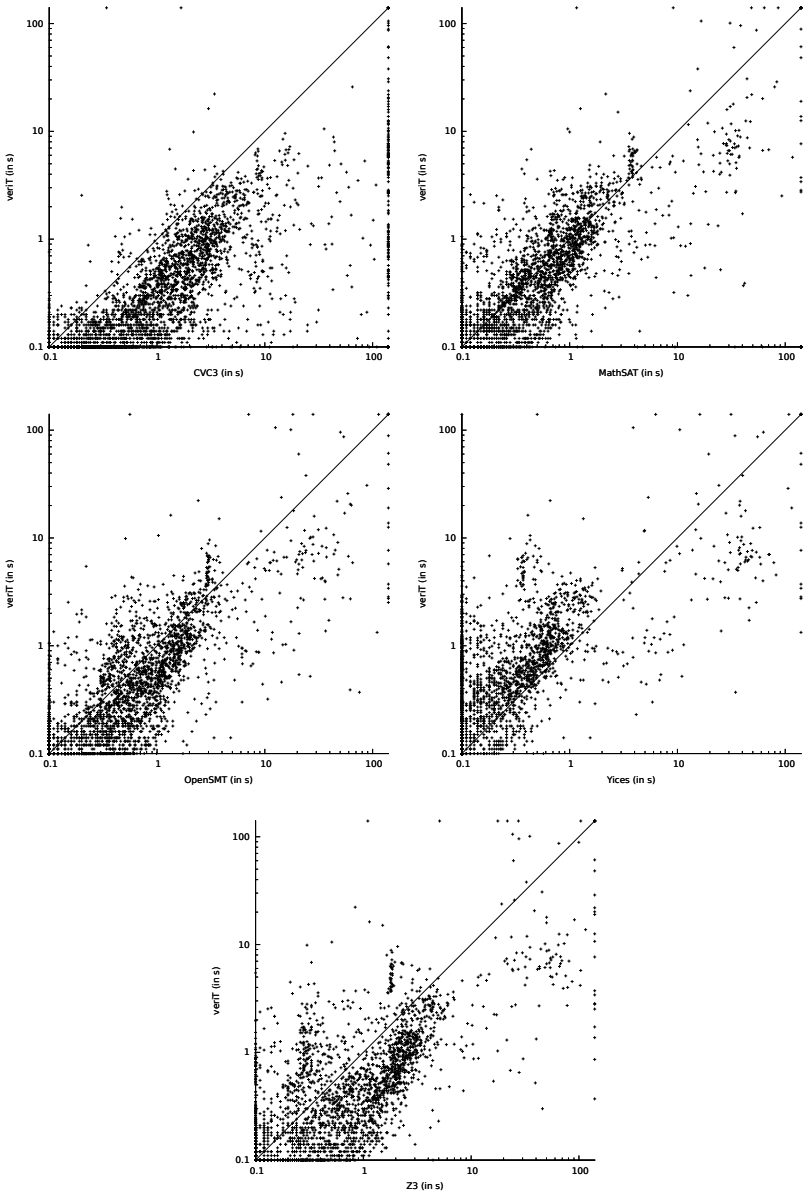
Acknowledgements. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several universities as well as other funding bodies (see `https://www.grid5000.fr`). We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Bounded model checking for timed systems. In D. Peled and M. Y. Vardi, editors, *In IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 2529 of *LNCS*, pages 243–259. Springer, 2002.
2. C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 20–23. Springer, 2005.
3. C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, Feb. 2009.
4. P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *J. Applied Logic*, 7(1):58–74, 2009.
5. T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.
6. S. R. Buss. Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, 52:916–927, 1987.
7. K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.
8. D. Déharbe, P. Fontaine, S. Merz, and B. W. Paleo. Exploiting symmetry in SMT problems, 2011. Available at `http://www.loria.fr/~fontaine/Deharbe6b.pdf`.
9. I. P. Gent, K. E. Petrie, and J.-F. Puget. *The Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, chapter Symmetry in Constraint Programming, pages 329–376. Elsevier, 2006. Edited by Francesca Rossi, Peter van Beek and Toby Walsh.
10. A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297 – 308, 1985.
11. X. Jia and J. Zhang. A powerful technique to eliminate isomorphism in finite model search. In U. Furbach and N. Shankar, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*, pages 318–331. Springer Berlin / Heidelberg, 2006.
12. B. Krishnamurthy. Short proofs for tricky formulas. *Acta Inf.*, 22:253–275, August 1985.
13. A. A. Razborov. Proof complexity of pigeonhole principles. In *Conference on Developments in Language Theory (DLT)*, pages 100–116. Springer-Verlag, 2002.
14. K. Roe. The heuristic theorem prover: Yet another smt-modulo theorem prover. In T. Ball and R. B. Jones, editors, *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 467–470. Springer, 2006.
15. K. A. Sakallah. Symmetry and satisfiability. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 289–338. IOS Press, Feb. 2009.
16. O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 265–279. Springer, July 2002.
17. S. Szeider. The complexity of resolution with generalized symmetry rules. *Theory Comput. Syst.*, 38(2):171–188, 2005.

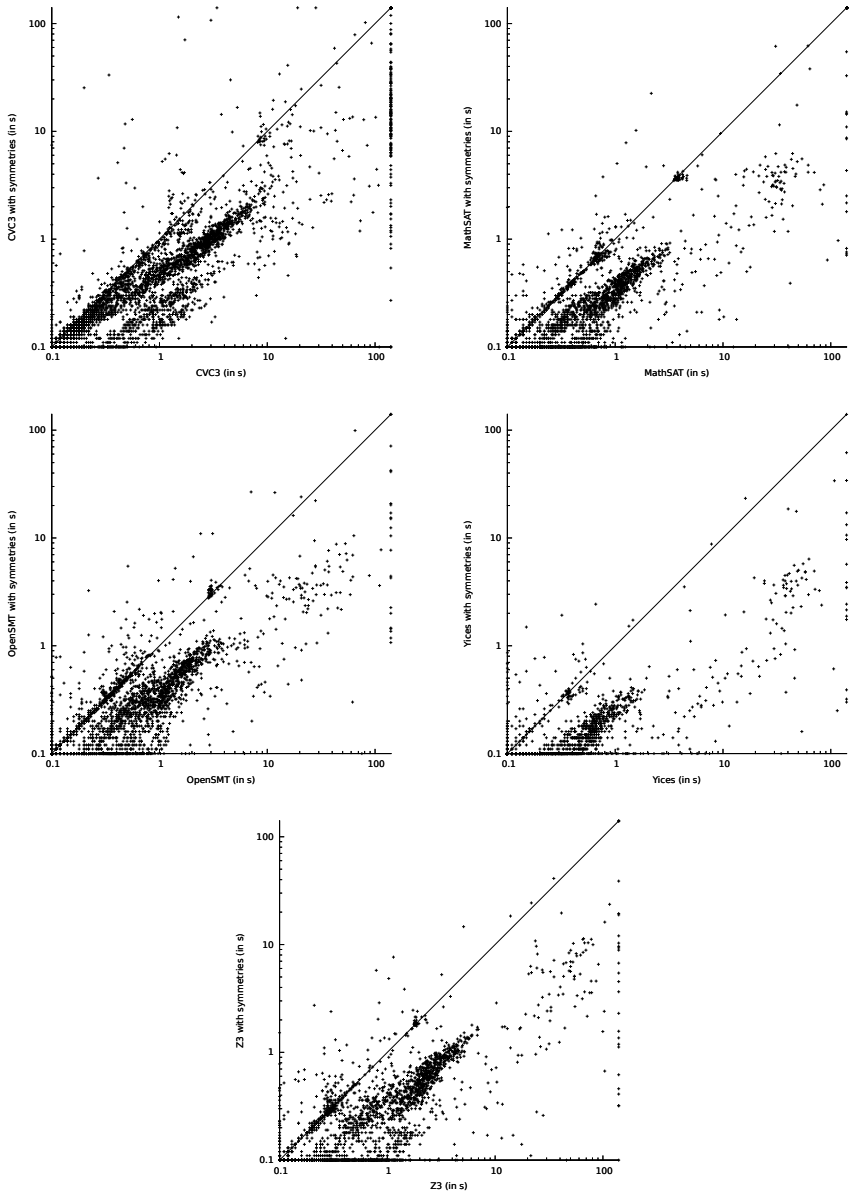
A veriT and other solvers

Here are scatter plots exhibiting the performances of veriT in solving individual instances of QF_UF veriT against some other SMT solvers.



B Other solvers on formulas with symmetry breaking clauses

These scatter plots exhibit the gain of efficiency from symmetries for some state-of-the-art solvers. They compare running times on QF_UF formulas with symmetry breaking clauses and on original formulas.



The following table presents the results of some SMT solvers on QF_UF formulas without and with symmetry breaking clauses. 83 timeouts for CVC and 80 for MathSAT are due to the diamond benchmarks.

	Nb. of instances		Instances within time range (in s)						Total time	
	success	timeout	0-20	20-40	40-60	60-80	80-100	100-120	T	T'
veriT	6633	14	6616	9	2	1	3	2	3447	5127
veriT w/o sym.	6570	77	6493	33	14	9	12	9	10148	19388
CVC	6385	262	6337	20	12	7	5	4	8118	29598
MathSAT	6547	100	6476	49	12	6	3	1	5131	7531
openSMT	6624	23	6559	43	13	6	1	2	5345	8105
Yices	6629	18	6565	32	23	5	1	3	4059	6219
Z3	6621	26	6542	33	23	15	4	4	6847	9967
Hereunder are results on formulas with symmetry breaking clauses										
CVC	6528	119	6463	42	9	7	2	5	6495	10815
MathSAT	6563	84	6556	4	1	2	0	0	1665	2145
openSMT	6644	3	6634	6	2	1	1	0	1982	2342
Yices	6646	1	6642	3	0	1	0	0	710	830
Z3	6644	3	6640	3	1	0	0	0	1612	1972

veriT: an open, trustable and efficient SMT-solver

Thomas Bouton[2], Diego Caminha B. de Oliveira[2],
David Déharbe[1], and Pascal Fontaine[2]

[1] Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil
david@dimap.ufrn.br
[2] LORIA–INRIA, Nancy, France
{Thomas.Bouton,Diego.Caminha,Pascal.Fontaine}@loria.fr

Abstract. This article describes the first public version of the satisfiability modulo theory (SMT) solver veriT. It is open-source, proof-producing, and complete for quantifier-free formulas with uninterpreted functions and difference logic on real numbers and integers.

1 Introduction

We present the satisfiability modulo theory (SMT) solver veriT, a joint work of University of Nancy, INRIA (Nancy, France) and Federal University of Rio Grande do Norte (Natal, Brazil). veriT provides an open, trustable and reasonably efficient decision procedure for the logic of unquantified formulas over uninterpreted symbols, difference logic over integer and real numbers, and the combination thereof. This corresponds to the logics identified as QF_IDL, QF_RDL, QF_UF and QF_UFIDL in the SMT-LIB benchmarks [15, 3]. veriT also includes quantifier reasoning capabilities through the integration of a first-order prover and quantifier instantiation heuristics. Finally, veriT has proof-production capabilities; it outputs proofs that may be used or checked by external tools.

veriT is incremental, i.e. after each satisfiability check, new formulas can be added conjunctively to the already checked set of formulas. The input format is the SMT-LIB language [15], but veriT can also be used as a library with an API following the guidelines of [12]. The tool is open-source and distributed under the BSD licence at `http://www.verit-solver.org`. Internally, the solver is organized to be easily extended by plugging new decision procedures in a Nelson-Oppen like combination schema. Although not (yet) as fast as the solvers performing best in the SMT competition [3], veriT has a decent efficiency. We thus claim that it can already be useful in verification platforms where an open-source license, extensibility, and proof certification are important.

Selected features of the veriT solver and an experimental evaluation of its efficiency are presented in Section 2 and 3, respectively. Future developments are described in Section 4.

2 System description

The reasoning core of veriT uses a SAT solver [9] to produce models of the Boolean abstraction of the input formula. Such propositional assignments are

given to a so-called *theory reasoner*, responsible for verifying if they are models in the background theory. This theory reasoner is a fully incremental combination of decision procedures *à la* Nelson and Oppen, where non-convexity of theories is handled using the model-equality propagation technique [7] which integrates model-based guessing [5] in a classical Nelson-Oppen equality exchange. Equality propagation is controlled by the congruence closure algorithm.

The remainder of this section describes some special features of veriT: integration of a third-party first-order prover, extension of the input language with macro definitions, and production of proofs certifying the produced results.

2.1 Integrating a first-order prover

As a particular feature inherited from its predecessor haRVey [8] and, to complement very simple instantiation heuristics, the veriT solver includes a first-order logic (FOL) superposition prover. However, veriT greatly improves the integration of the FOL prover with the other decision procedures, notably with congruence closure. Indeed, the first-order prover is seen within the combination *à la* Nelson-Oppen as a "decision procedure" that takes an arbitrary FOL theory as a parameter. However, due to the cost of running the FOL prover and to its non-incremental nature (when used as a black box), this procedure is called in last resort. A FOL theory is computed from the quantified sub-formulas in the assignment, abstracting ground sub-terms in order to minimize the number of relevant symbols in the theory. In addition, information from congruence closure is used to abstract all subterms in the assignment that do not contain such relevant symbols.

The prover may deduce that the given set of formulas is unsatisfiable. In that case, the deduction tree is parsed to obtain the relevant unsatisfiable subset of the input. A conflict clause is then built using this set and, again, information from the congruence closure data structures. Since the prover is given an upper limit of resources, it always terminates. If the prover terminates without proving the unsatisfaibility of the given set of formulas, ground equalities and deduced ground clauses are identified and propagated back to veriT.

In many cases where the superposition calculus is a decision procedure [2] for the theory represented by the quantified formulas, our technique simulates a Nelson-Oppen combination with on-the-fly purification. It has been shown that first-order generic provers may perform quite well even compared to dedicated decision procedures (see for instance [1]). Currently, the E-prover [16] is used as the first-order prover, and we plan to include Spass [18], which provides better sort handling. Fine-tuning the interplay between the instantiation heuristics and the e-prover is essential for efficiency and remains to be done.

2.2 Macros

The input format for veriT is the SMT-LIB language extended with macro definitions. This syntactic sugar is particularly useful for instance to write formulas containing simple sets constructions (see Figure 1). After β-reduction, and after

rewriting equalities between predicates and functions (for instance, if p and q are unary predicates, $p = q$ is rewritten as $\forall x \,.\, p(x) \equiv q(x)$), the obtained formula is first-order. Such formulas may contain quantifiers but, if no function is used, they belong to the Bernays-Schönfinkel-Ramsey fragment (the Bernays-Schönfinkel class with equality) which is decidable. veriT can use the embedded FOL prover as a decision procedure for this fragment. In many more intricate cases [10], the resulting formula still belongs to a decidable fragment, but the decision procedure may become very expensive. To be practical, such cases require heuristics that have not yet been implemented in veriT.

This macro feature is indeed used in tools (e.g. CRefine [14]) that generate verification conditions for formal developments in set-based modelling languages, such as Circus [4], and that integrate veriT as a verification engine to discharge these proof obligations.

```
(benchmark SET008_3p
  :logic UNKNOWN
  :extrasorts (ELMT)
  :extrapreds ((B ELMT) (C ELMT))
  :extramacros
   ((emptyset (lambda (?v ELMT) . false))
    (intersection (lambda (?p (ELMT boolean)) (?q (ELMT boolean)) .
                     (lambda (?x ELMT) . (and (?p ?x) (?q ?x)))))
    (difference (lambda (?p (ELMT boolean)) (?q (ELMT boolean)) .
                   (lambda (?x ELMT) . (and (?p ?x) (not (?q ?x)))))))
  :formula (not (= (intersection (difference B C) C) emptyset)))
```

Fig. 1. A simple example with the macro capability.

2.3 Proofs

Proof production has two goals. First, this feature increases the confidence in the tool, the proofs being checked by an independent module inside veriT. Second, skeptical proof assistants can use such traces to reconstruct proofs of formulas discharged by veriT (see [11]).

In Figure 2, we give an example of a very simple formula, and the proof output by veriT. Each line states a fact that can be assumed to hold. It is identified by a number, followed by a list starting with a label identifying the rule used to deduce the fact, followed by a clause, and optionally ended by numerical parameters. In our context, a clause is a disjunctive list of formulas (not literals), maybe containing a sole formula. The numerical parameters depend on the rule, and may be either identifiers of previous clauses (e.g. in the resolution rule), or other place information. As an example, the *and* rule (for instance the second line in Figure 2: `(and ((= a c)) 1 0)`) takes two numerical parameters. The first

```
(benchmark example
  :logic QF_UF
  :extrafuns ((a U) (b U) (c U) (f U U))
  :extrapreds ((p U))
  :formula (and (= a c) (= b c)
                (or (not (= (f a) (f b)))
                    (and (p a) (not (p b))))))

1:(input ((and (= a c) (= b c)
               (or (not (= (f a) (f b))) (and (p a) (not (p b)))))))
2:(and ((= a c)) 1 0)
3:(and ((= b c)) 1 1)
4:(and ((or (not (= (f a) (f b))) (and (p a) (not (p b))))) 1 2)
5:(and_pos ((not (and (p a) (not (p b)))) (p a)) 0)
6:(and_pos ((not (and (p a) (not (p b)))) (not (p b))) 1)
7:(or ((not (= (f a) (f b))) (and (p a) (not (p b)))) 4)
8:(eq_congruent ((not (= b a)) (= (f a) (f b))))
9:(eq_transitive ((not (= b c)) (not (= a c)) (= b a)))
10:(resolution ((= (f a) (f b)) (not (= b c)) (not (= a c))) 8 9)
11:(resolution ((= (f a) (f b))) 10 2 3)
12:(resolution ((and (p a) (not (p b)))) 7 11)
13:(resolution ((p a)) 5 12)
14:(resolution ((not (p b))) 6 12)
15:(eq_congruent_pred ((not (= b a)) (p b) (not (p a))))
16:(eq_transitive ((not (= b c)) (not (= a c)) (= b a)))
17:(resolution ((p b) (not (p a)) (not (= b c)) (not (= a c))) 15 16)
18:(resolution () 17 2 3 13 14)
```

Fig. 2. A simple example with its proof.

numerical parameter refers to the clause C in a previous numbered rule (i.e. `1` refers to the clause in the `input` rule, at line 1). This clause C is unit and is hence represented as a list of one formula (the whole input formula in our example), and this formula is a conjunction $a_0 \wedge \cdots \wedge a_n$. Obviously, each subformula $a_0, \ldots a_n$ is a consequence of C, and the second parameter just gives the identifier of the formula in the new clause, i.e. the second numerical parameter in rule at line 2 indicates the formula at position 0 in the input.

veriT already provides proof production for formulas with arbitrary Boolean structure and uninterpreted functions, and is being extended to linear arithmetics. The first line is the input. Every other fact is either a consequence of previous ones, or is a tautology. The input formula being unsatisfiable, the last deduced fact is the empty clause. In the example, lines 2 to 7 account for the conjunctive normal form transformation. Lines 8, 9, 15, and 16 are tautologies related to the theory of equality. The remaining facts are deduced by resolution from the other ones.

Since every proof-related information is handled through a unique module inside veriT, any proof format for SMT (for instance [17, 6]) can be adopted as

soon as it becomes a standard. Although our previous experiments [11] showed that the proof size was not the bottleneck for the cooperation with skeptical proof assistants, the implementation of techniques to greatly reduce the size of our proof traces is planned.

3 Experimental evaluation

We evaluated veriT, CVC3 and Z3 (both using the latest available version in February 2009) against the SMT-LIB benchmarks for QF_IDL, QF_RDL, QF_UF and QF_UFIDL (June 2008 version) using an Intel(R) Pentium(R) 4 CPU at 3.00 GHz with 1 GiB of RAM and a timeout of 120 seconds. The following table presents, for each solver, the number of completed benchmarks.

Solver	QF_UF (6656)	QF_UFIDL (432)	QF_IDL (1673)	QF_RDL (204)	all (8965)
veriT	6323	332	918	100	7673
CVC3	6378	278	802	45	7503
Z3	6608	419	1511	158	8696

This clearly shows that, although veriT is not yet as efficient as competition winning tools [3], its efficiency is decent. The proof production capability of veriT does not come at a cost on efficiency.

4 Future work

veriT is a new SMT-solver that provides an open framework to generate certifiable proofs without sacrificing too much efficiency. The future developments will notably include features related to efficiency and expressiveness. Considering efficiency aspects, the tool does not yet implement theory propagation [13], a technique that is known to greatly improve the efficiency of SMT solvers. Concerning expressiveness, the linear arithmetic decision procedure currently only handles difference logic; we are now developing a reasoning engine for linear arithmetic based on the Simplex method, which will extend completeness to full linear arithmetic. Quantifier reasoning will be improved by including new instantiation heuristics, as well as adding support for patterns guiding quantifier instantiations. We also plan to integrate the proof production capabilities of the embedded FOL prover with that of veriT.

Finally, we are working on the application of veriT to formal development efforts, mainly of concurrent systems. In that context, the ability to handle sets is very helpful; a major objective is to improve the support for such constructions.

Acknowledgments: We would like to thank Stephan Merz for his comments and guidance. The ancestor of the veriT solver, haRVey, was initiated by the third author and Silvio Ranise, to whom we are indebted of several ideas used in veriT. We are also grateful to the anonymous reviewers for their remarks.

References

1. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Trans. Comput. Log.*, 10(1), 2009.
2. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.
3. C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In *Computer Aided Verification (CAV)*, pages 20–23, 2005.
4. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
5. L. de Moura and N. Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, 2008.
6. L. M. de Moura and N. Björner. Proofs and refutations, and Z3. In *LPAR Workshops*, volume 418 of *CEUR Workshop Proceedings*, 2008.
7. D. Déharbe, D. de Oliveira, and P. Fontaine. Combining decision procedures by (model-)equality propagation. In *Brazil. Symp. Formal Methods*, pages 51–66, 2008.
8. D. Déharbe and S. Ranise. Bdd-driven first-order satisfiability procedures (extended version). Research report 4630, LORIA, 2002.
9. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 333–336. Springer, 2003.
10. P. Fontaine. Combinations of theories and the Bernays-Schönfinkel-Ramsey class. In *4th Int'l Verification Workshop (VERIFY)*, 2007.
11. P. Fontaine, J.-Y. Marion, S. Merz, L. P. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *Tools and Algorithms for Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 167–181. Springer, 2006.
12. J. Grundy, T. Melham, S. Krstić, and S. McLaughlin. Tool building requirements for an API to first-order solvers. *Electronic Notes in Theoretical Computer Science*, 144:15–26, 2005.
13. R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 321–334. Springer, 2005.
14. M. Oliveira, C. Gurgel, and A. de Castro. Crefine: Support for the Circus refinement calculus. In *IEEE Intl. Conf. Software Engineering and Formal Methods*, pages 281–290. IEEE Comp. Soc. Press, 2008.
15. S. Ranise and C. Tinelli. The SMT-LIB standard : Version 1.2, Aug. 2006.
16. S. Schulz. System Description: E 0.81. In *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.
17. A. Stump. Proof Checking Technology for Satisfiability Modulo Theories. In *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.
18. C. Weidenbach, R. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System description: Spass version 3.0. In *Conference on Automated Deduction (CADE)*, volume 4603 of *LNCS*, pages 514–520. Springer, 2007.