

SPASS+T

Virgile Prevosto^{1*} and Uwe Waldmann²

¹CEA – LIST, Centre de Saclay
Software Reliability Laboratory (LSL)
91191 Gif-sur-Yvette Cedex, France
`virgile.prevosto@m4x.org`

²Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
`uwe@mpi-inf.mpg.de`

Abstract

SPASS+T is an extension of the superposition-based theorem prover SPASS that allows us to enlarge the reasoning capabilities of SPASS using an arbitrary SMT procedure for arithmetic and free function symbols as a black-box. We discuss the architecture of SPASS+T and the capabilities, limitations, and applications of such a combination.

1 Introduction

SPASS (Weidenbach et al. [10]) is a saturation-based theorem prover for first-order logic based on the superposition calculus. Such provers are notoriously bad at dealing with integer or real arithmetic – encoding numbers in binary or unary is not really a viable solution in most application contexts. We have extended SPASS in such a way that we can link it to a rather arbitrary SMT (*Satisfiability Modulo Theories*) procedure for arithmetic and free function symbols, for instance our own MODUPROVE system (based on the DPLL(T) framework of Ganzinger et al. [5]) or the CVC Lite prover by Barrett and Berezin [2]. Briefly, the resulting system, called SPASS+T, works as follows: SPASS uses its deduction rules to generate formulas as usual. In addition, it passes to the SMT procedure all the formulas that can be handled by the procedure, i. e., all ground formulas. As soon as one of the two systems encounters a contradiction, the problem is solved.

The scenario described so far is a special case of hierarchic theorem proving: a combination of a *base prover* that deals with some subclass of formulas, say, formulas in linear arithmetic, and an *extension prover* that deals with formulas over the complete signature

*This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors. Project website: <http://www.verisoft.de>.

and passes all formulas to the base prover that the latter can handle. The correctness of such a hierarchic combination of deductive systems is obvious. In theory, conditions for completeness have been given by Bachmair, Ganzinger, Sofronie-Stokkermans, and Waldmann [1, 6]. For instance, one can get a complete hierarchic combination of provers if base and non-base vocabulary are separated using abstraction, the term ordering is chosen in such a way that base terms/atoms are smaller than non-base terms/atoms, the base prover can deal with arbitrary formulas over the base vocabulary, the extension is sufficiently complete,¹ and the base theory is compact [1].

In practice, however, sufficient completeness need not hold (and cannot be checked automatically), abstraction enlarges the search space, SMT procedures for some useful theories can only deal with ground formulas (this is for instance the case for Nelson-Oppen combinations of arithmetic and free function symbols), and even compactness may be an issue. As an example, consider the two clauses $\forall x, y. (f(x) \neq y + y)$ and $\forall x, y. (f(x) \neq y + y + 1)$. If the base theory is Presburger arithmetic (linear integer arithmetic), then the conjunction of these two clauses (expressing the fact that $f(x)$ is neither even nor odd) is inconsistent. Still any finite set of ground instances of these two formulas is consistent. Even unification modulo the base theory would not help.

An integration of first-order theorem provers and SMT procedures can therefore only be a pragmatic one: Completeness can be achieved for special classes of inputs, but not in general. If we work with non-ground problem description but ground goal formulas, then there is some hope that we can ultimately produce sufficiently many ground formulas so that the SMT procedure can find the contradiction. If we want to solve non-ground problems, in particular if we want to find solutions for variables, then the purely hierarchic approach must be supplemented by some knowledge about arithmetic that is built-in directly into SPASS.

2 SPASS

SPASS (Weidenbach et al. [10]) is known to be one of the most advanced implementations of the superposition calculus. The superposition calculus is a refutationally complete procedure for arbitrary first-order clauses with equality, that is, it provides a semi-decision procedure for the unsatisfiability of sets of clauses. Theorem proving methods such as resolution or superposition aim at deducing a contradiction from a set of formulas by recursively inferring new formulas from given ones. New formulas are derived according to a set of inference rules. In the case of superposition, these rules are restricted versions of paramodulation, resolution, and factoring, parameterized by a reduction ordering \succ that is total on ground expressions (that is, ground atoms and ground terms) and by a *selection function* S , which assigns to each clause a (possibly empty) multiset of (occurrences of) *negative* literals. The literals in $S(C)$ are called *selected*. Selected literals, besides being negative, can be arbitrarily chosen. Ordering and selection function impose various restrictions on the possible inferences, which are crucial for the efficiency of theorem provers like SPASS. Let us consider one of the inference rules of the superposition calculus as an example:

¹Intuitively, sufficient completeness means that every ground term of a base sort is provably equal to a ground term consisting only of base symbols.

$$\text{Negative superposition} \quad \frac{D' \vee t = t' \quad C' \vee \neg s[u] = s'}{(D' \vee C' \vee \neg s[t'] = s')\sigma}$$

if (i) the literal $t = t'$ is strictly maximal in the first premise, (ii) no literal is selected in the first premise, (iii) either the literal $\neg s[u] = s'$ is selected in the second premise or it is maximal and no literal is selected, (iv) $t \not\leq t'$, (v) $s[u] \not\leq s'$, (vi) u is not a variable, and (vii) σ is a most general unifier of t and u .

This inference rule combines the unification of t and the subterm u of s with subsequent replacement of $u\sigma$ by $t'\sigma$. When we speak of a *superposition inference* we mean an arbitrary rule of the calculus.

Note that a clause with selected literals cannot serve as the left premise of a superposition inference, and that maximal terms are superposed either on maximal or selected literals, and a smaller term is never replaced by a larger one. Moreover, only the maximal sides of equations are replaced. The pattern of interplay between ordering restrictions and the selection function is the same for all inference rules of the calculus.

The local restrictions of the superposition inference rules are supplemented by a global redundancy criterion that allows us to discard formulas that are provably unnecessary for deriving a contradiction.

3 System Architecture

SPASS+T consists of three modules: the theorem prover SPASS, a (rather arbitrary) SMT procedure for integer or real arithmetic and free function symbols, and a module SMT-Control connecting both. The proof systems are only loosely coupled; they wait for each other only if they have completed their own task.

SPASS sends three kinds of messages to SMT-Control:

- all the ground formulas present in the input or derived during the saturation;
- *proof_found* (signaling that SPASS has derived the empty clause);
- *end_of_file* (signaling that SPASS has saturated its input).

SMT-Control collects the formulas sent by SPASS and repeatedly sends portions of the set of formulas to the SMT procedure,² until one of the following events occurs:

- SPASS sends *proof_found*;
- the SMT procedure detects unsatisfiability;
- the SMT procedure detects satisfiability after SPASS has sent *end_of_file* and the SMT procedure has seen all the output of SPASS.

The result is “proof found” in the first two cases, “no proof found” in the third one. Lacking completeness, “no proof found” does in general not imply satisfiability.

²Under the assumption that the SMT procedure works incrementally. Alternatively the SMT procedure can be restarted with increasing subsets of the set of formulas.

4 The Simple Case

There are applications where we can guarantee that every base theory formula generated by SPASS is ground. In this case, the very simple setup described above is already sufficient. An example is pointer data structure verification à la McPeak and Necula [7]. In this scenario, we consider recursive data structures involving pointers to records or to *nil*. Record fields can either be scalar values or pointers. In the axioms used to describe the behaviour of a data structure, all variables range over pointer values (i. e., there is no quantification over scalar values). Record fields are encoded as (partial) functions, so $x.data$, i. e., the *data* field of the record that x points to, is written as $data(x)$. McPeak and Necula require that every occurrence of such a function must be guarded by a condition that ensures that the argument is non-*nil*. For instance, the formula stating that the *prev* pointer is the inverse of the *next* pointer in a doubly linked list looks like

$$\forall x. (x \neq nil \wedge next(x) \neq nil \rightarrow prev(next(x)) = x),$$

and the formula stating that the *data* field of any record is positive looks like

$$\forall x. (x \neq nil \rightarrow data(x) > 0).$$

To avoid positive disjunctions, we replace the guard formula $x \neq nil$ by $isrecord(x)$, obtaining

$$\begin{aligned} \forall x. (isrecord(x) \wedge isrecord(next(x)) \rightarrow prev(next(x)) = x), \\ \forall x. (isrecord(x) \rightarrow data(x) > 0). \end{aligned}$$

Since the SMT procedure accepts not only the symbols of the arithmetical theory, such as $+$, $-$, \cdot , $<$, or \leq , but also free function or predicate symbols, the distinction between base and non-base symbols is somewhat arbitrary – if it is useful, we may treat both theory symbols and a subset of the free symbols as base symbols. Let us therefore consider the guard predicate $isrecord$ as the only non-base symbol in the signature. If we *select* the occurrences of $isrecord(t)$ in the antecedent, we ensure that the only inferences that are possible with such clauses are (repeated) resolution steps with positive occurrences of $isrecord(s)$. Since $isrecord(s)$ occurs positively only in goal formulas (or formulas recursively derived from goal formulas), and since the term s is always ground in these formulas, the base formulas resulting from resolution are ground and can be passed to the SMT procedure. In effect, SPASS+T mimics the derivation steps of McPeak and Necula, and the completeness result of McPeak and Necula carries over to SPASS+T.

5 Theory Instantiation

If non-ground theory literals are not guarded, then it can easily happen that the usual inference rules of SPASS do not produce those ground instances that the SMT procedure would need to find a contradiction. Consider the following example from Boyer and Moore [3]: Let min and max be non-theory function symbols denoting the minimum and the maximum element of a list of numbers. Suppose that we want to refute $\neg (l < max(a) + k)$ using the assumptions $l \leq min(a)$ and $0 < k$ and the universally quantified

lemma $\forall x. (\min(x) \leq \max(x))$. Analogously to Boyer and Moore, we need an additional inference rule that computes the required ground instance of this lemma. For efficiency reasons, we restrict ourselves to instantiations where a term headed by a non-theory symbol occurs in at least one other ground clause – if a non-theory term occurs in only one clause, this clause is very unlikely to be useful to the SMT procedure, at least in linear arithmetic. We obtain the following inference rule:

$$\textit{Theory instantiation} \quad \frac{C[s] \quad L[t] \vee D}{(L[t] \vee D)\sigma}$$

if $L[t] \vee D$ is not ground, t is headed by a non-theory function symbol and occurs in a maximal literal $L[t]$ immediately below a theory symbol, all function or predicate symbols occurring above t in $L[t]$ are theory symbols or equality, $C[s]$ is ground, σ is an mgu of s and t , and $(L[t] \vee D)\sigma$ is ground.

In the example above, the *theory instantiation* inference

$$\frac{l \leq \min(a) \quad \min(x) \leq \max(x)}{\min(a) \leq \max(a)}$$

yields the ground clause that the SMT procedure needs to derive a contradiction. There is one problem with this rule, though: The generated formula $(L[t] \vee D)\sigma$ is subsumed by the second premise $L[t] \vee D$; therefore SPASS will delete it again. We export it to the SMT procedure before the redundancy check, but we must also ensure that $(L[t] \vee D)\sigma$ can again be used as a first premise in a *theory instantiation* inference. To this end, we introduce a special new predicate symbol *ground* and split the inference rule into two parts:

$$\textit{Theory instantiation I} \quad \frac{C[s]}{\textit{ground}(s)}$$

if $C[s]$ is ground and s is headed by a non-theory function symbol.

$$\textit{Theory instantiation II} \quad \frac{\textit{ground}(s) \quad L[t] \vee D}{\textit{ground}(u_1) \dots \textit{ground}(u_n) \quad \mathbf{export}(D')}$$

if $L[t] \vee D$ is not ground, t is headed by a non-theory function symbol and occurs in a maximal literal $L[t]$ immediately below a theory symbol, all function or predicate symbols occurring above t in $L[t]$ are theory symbols or equality, σ is an mgu of s and t , $D' = (L[t] \vee D)\sigma$ is ground, and u_1, \dots, u_n are the ground terms occurring in D' that are different from s and headed by a non-theory function symbol.

In the example above, the *theory instantiation I* inference

$$\frac{l \leq \min(a)}{\textit{ground}(\min(a))}$$

and the *theory instantiation II* inference

$$\frac{\textit{ground}(\min(a)) \quad \min(x) \leq \max(x)}{\textit{ground}(\max(a)) \quad \mathbf{export}(\min(a) \leq \max(a))}$$

cause $\min(a) \leq \max(a)$ to be exported to the SMT procedure and ensure that the clause $\text{ground}(\max(a))$ is available for further *theory instantiation II* inferences within SPASS.

If we select guard literals whenever possible, and if we choose the atom ordering so that non-theory predicates are larger than theory predicates, then the *theory instantiation* rule is used only as a last resort: It is applied only if there are no non-theory guard literals that might produce ground instances in a more restricted manner.

6 Arithmetic Simplification

There are several reasons to supplement the external SMT procedure in SPASS+T by simplification techniques that are built-in directly into SPASS+T. First of all, such simplification techniques offer the chance to find solutions for variables, say, to replace a clause $\forall x. (\neg x + 2 = 5 \vee p(x))$ by $\forall x. (\neg x = 3 \vee p(x))$ and then by $p(3)$. Second, they allow us to reduce different numeric subexpressions to the same number, so that the search space of SPASS+T is not cluttered by different, but numerically equivalent clauses, such as $p(2 + 1)$, $p(1 + 2)$, $p(1 + (1 + 1))$, etc. Third, by applying arithmetic simplification in advance, an equation like $\forall x. (x + 0 = x)$ can be used to simplify a formula such as $p(((a + 2) - 1) - 1)$ to $p(a)$.

In SPASS+T, we use a combination of additional input axioms encoding some fragment of arithmetic and specialized simplification rules dealing with the numeric part. The latter include rules for evaluation of constant numeric subexpressions, such as

$$\frac{C[c_1 + c_2]}{C[c_0]}$$

and

$$\frac{C[(t + c_1) + c_2]}{C[t + c_0]}$$

where c_1 and c_2 are numeric constants and $c_0 = c_1 + c_2$, and rules for elementary (in-)equation solving, for instance

$$\frac{C \vee [\neg] t + c_1 \sim c_2}{C \vee [\neg] t \sim c_0}$$

where $c_0 = c_2 - c_1$ and \sim is equality or an ordering relation.

All the arithmetic simplification rules we have implemented have the property that the resulting formula is smaller than the original one in any Knuth-Bendix ordering, provided that all constants have the same weight. Still it should be clear that there is some risk of losing proofs by applying arithmetic simplification. As a trivial example consider the two clauses $\forall x. (p(3 \cdot x + 4))$ and $\neg p(3 \cdot 5 + 4)$. These clauses are obviously contradictory, but lacking theory unification, there is no way to derive a contradiction as soon as the second clause has been simplified to $\neg p(19)$.

In contrast to formulas that describe the relationships between concrete numerical constants, equations such as $\forall x. (x + 0 = x)$, $\forall x. (x - x = 0)$, or $\forall x, y. ((x - y) + y = x)$ are preferably added to the input as ordinary axioms. In this way they are not only available for simplification, but also for superposition or resolution inferences. In our experiments, this capability turned out to be extremely useful for refuting non-ground queries.

The *integer ordering expansion* rule, which is essential for some kinds of inductive proofs, is in some way a hybrid between the two cases above.

$$\text{Integer ordering expansion} \quad \frac{C \vee s \leq t}{C \vee s = t \vee s \leq t - 1 \quad C \vee s = t \vee s + 1 \leq t}$$

It is an inference rule, rather than a simplification rule, and it corresponds to resolution inferences with the axioms $\forall x, y. (\neg x \leq y \vee x = y \vee x \leq y - 1)$ and $\forall x, y. (\neg x \leq y \vee x = y \vee x + 1 \leq y)$. These are *unordered* resolution inferences, however, so SPASS would not perform them. The *integer ordering expansion* rule fills this gap, but it has to be restricted, for instance by limiting applications to clauses with only one positive literal; otherwise it turns out to be too productive.

7 Experiments

We have tested SPASS+T both on a list of sample conjectures [8] (theorems and non-theorems) from the TPTP library [9] and on our own list of benchmark problems, mostly combining arithmetic and data structures. We used SPASS+T with CVC Lite 2.5 as external decision procedure for the union of the theory of uninterpreted functions and integer arithmetic. SPASS+T options were chosen identically for all benchmark problems; in particular, arithmetic simplification, theory instantiation, and (a very restricted form of) integer ordering expansion were switched on, and precedences were chosen uniformly in such a way that non-theory predicates (notably the containment relation for collections) became larger than theory predicates.³ Moreover, the following set of auxiliary axioms was supplied:

- INT01 $\forall X, Y. ((X - Y) + Y = X).$
- INT02 $\forall X. (X + 0 = X).$
- INT03 $\forall X. (0 + X = X).$
- INT04 $\forall X. (X < X + 1).$
- INT05 $\forall X, Y. (X < Y \rightarrow X \leq Y).$
- INT06 $\forall X. (X \leq X).$
- INT07 $\forall X, Y. (X < Y \leftrightarrow Y > X).$
- INT08 $\forall X, Y. (X \leq Y \leftrightarrow Y \geq X).$
- INT09 $\forall X, Y. (\neg (X < Y \wedge X \geq Y)).$
- INT10 $\forall X, Y. (\neg (X \leq Y \wedge X > Y)).$
- INT11 $\forall X, Y. (X \leq Y \wedge Y \leq X \rightarrow X = Y).$
- INT12 $\forall X, Y. (X - Y = X + (-Y)).$
- INT13 $\forall X. (-(-X) = X).$
- INT14 $\forall X. (X \cdot 0 = 0).$
- INT15 $\forall X. (0 \cdot X = 0).$
- INT16 $\forall X. (X \cdot 1 = X).$
- INT17 $\forall X. (1 \cdot X = X).$
- INT18 $\forall X. (X/1 = X).$
- INT19 $\forall X. (X \bmod 1 = 0).$

³Using the `set_DomPred` option of SPASS.

The experiments were carried out on an office PC with a 2.40 GHz Intel Pentium 4 CPU and 512 MB RAM running Debian Linux. All times reported are wall-clock time in seconds.

7.1 TPTP Integer Arithmetic Problems

The list of integer arithmetic problems in the TPTP library contains 147 theorems and 36 non-theorems [8]. Most of the problems are rather easy. The results are summarized in the following table:

Category	Theorems	Non-theorems
Number of problems:	147	36
Proof found:	140	0
Termination in less than 1 second:	136	0
Termination in 1 to 8 seconds:	4	0
No proof found:	7	36
Termination in less than 1 second:	6	31
Termination in 1 to 8 seconds:	1	3
Non-termination:	0	2

7.2 Further Problems

Integer arithmetic. In addition to the TPTP list, we have developed a collection of 75 theorems mostly combining arithmetic and data structures to test SPASS+T. In the following list, the second column gives the runtime of SPASS+T (wall clock time in seconds) and the result, where “+” means “Proof found”, “-” means “No proof found”, and “ ∞ ” means that the time limit of 600 seconds was exceeded.

- (1) 1.82 - $\forall X, Z. (\exists Y. (X + Y = Z))$
- (2) 0.16 + $\forall Y, Z. (\exists X. (X + Y = Z))$
- (3) 0.68 - $\forall X, Z. (\exists Y. (X - Y = Z))$
- (4) 0.16 + $\forall Y, Z. (\exists X. (X - Y = Z))$
- (5) 0.33 - $\exists X. (X + X < -10)$
- (6) 0.19 + $\exists X. (X \cdot 3 + (-5) < -12)$
- (7) 0.23 + $\exists X, Y. (3 \cdot X + 5 \cdot Y = 24)$
- (8) 0.22 + $\exists X, Y. (3 \cdot X + 5 \cdot Y = 23)$
- (9) 0.38 - $\exists X, Y. (3 \cdot X + 5 \cdot Y = 22)$
- (10) 0.69 + $\forall X, Y. (\neg 4 \cdot X + 6 \cdot Y = 21)$
- (11) 1.01 + $\forall X, Y, Z. (2 \cdot X + Y + Z = 10 \wedge X + 2 \cdot Y + Z = 10 \rightarrow \neg Z = 0)$

$$(12) \quad 1.65 + \quad \forall X, Y, Z. (X < 5 \wedge Y < 3 \wedge X + 2 \cdot Y > 7 \rightarrow Y = 2)$$

$$(13) \quad 4.77 + \quad \forall X. (\exists Y. (Y < X \wedge \neg \exists Z. (Y < Z \wedge Z < X)))$$

$$(14) \quad 0.21 + \quad \neg \exists X. (0 < X \wedge \forall Y. (Y < X \rightarrow Y + 1 < X))$$

Integer arithmetic plus free function or predicate symbols.

$$(15) \quad 0.18 + \quad \forall X, Y. (X + Y = f(X) \wedge Y - f(X) = 0 \rightarrow Y = f(0))$$

$$(16) \quad 0.91 + \quad \forall X. (f(X) > X) \rightarrow f(f(f(6))) \geq 9$$

$$(17) \quad 3.75 + \quad \forall X. (f(X) > X) \rightarrow \forall Y, Z. (f(f(Y) + Z) \geq Y + Z + 2)$$

$$(18) \quad 1.23 + \quad \forall X, Y. (X < Y \rightarrow f(X) < f(Y)) \rightarrow \forall Y. (f(f(Y) + 2) > f(f(Y)))$$

$$(19) \quad 1.40 - \quad \forall X, Y. (X < Y \rightarrow f(X) < f(Y)) \rightarrow \forall Y. (f(f(Y) + 2) > f(f(Y)) + 1)$$

$$(20) \quad 0.18 + \quad \forall X. (f(X) > 1) \rightarrow 7 - 2 \cdot f(3) < 4$$

$$(21) \quad 1.01 + \quad \forall X. (f(X) > X) \rightarrow \forall X. (X - f(X) < 0)$$

$$(22) \quad 1.17 - \quad \forall X, Y. (g(X, Y) = g(X, Y + 2)) \wedge g(3, 3) = g(3, 4) \rightarrow g(3, 2) = g(3, 5)$$

$$(23) \quad 0.20 + \quad p(14 \cdot 3 + 8) \rightarrow p(50)$$

$$(24) \quad 0.24 + \quad \forall X. (p(X + 3)) \rightarrow p(5)$$

$$(25) \quad 0.46 - \quad \forall X. (p(2 \cdot X)) \rightarrow p(10)$$

$$(26) \quad 0.59 - \quad p(0) \wedge \forall X. (p(X) \rightarrow p(X + 1)) \wedge \forall X. (p(X) \rightarrow p(X - 1)) \rightarrow \forall X. (p(X))$$

Integer arithmetic and arrays. The formulas

$$\text{ARR01} \quad \forall A, I, X. (\text{read}(\text{write}(A, I, X), I) = X).$$

$$\text{ARR02} \quad \forall A, I, J, X. (I = J \vee \text{read}(\text{write}(A, I, X), J) = \text{read}(A, J)).$$

were used to axiomatize arrays for the following problems:

$$(27) \quad 0.29 + \quad \forall A, A'. (A = \text{write}(\text{write}(\text{write}(\text{write}(A', 3, 33), 4, 444), 5, 55), 4, 44) \\ \rightarrow \text{read}(A, 3) = 33)$$

$$(28) \quad 0.61 + \quad \forall A, A', I. (A = \text{write}(\text{write}(\text{write}(\text{write}(A', 3, 33), 4, 44), 5, 55), I, 66) \\ \rightarrow \text{read}(A, 4) = 44 \vee \text{read}(A, 4) = 66)$$

$$(29) \quad 1.43 + \quad \forall A, A', I. (A = \text{write}(\text{write}(\text{write}(\text{write}(A', 3, 33), 4, 444), 5, 55), 4, 44) \\ \wedge 3 \leq I \wedge I \leq 4 \\ \rightarrow 33 \leq \text{read}(A, I) \wedge \text{read}(A, I) \leq 44)$$

$$(30) \quad 0.25 + \quad \forall A, A'. (A = \text{write}(\text{write}(\text{write}(\text{write}(A', 3, 33), 4, 444), 5, 55), 4, 44) \\ \rightarrow \exists I. (\text{read}(A, I) = 33))$$

- (31) 0.15 + $\forall A, A'. (A = \text{write}(\text{write}(\text{write}(\text{write}(A', 3, 33), 4, 444), 5, 55), 4, 44) \rightarrow \exists I. (\text{read}(A, I) < 40 \wedge 30 < \text{read}(A, I)))$
- (32) 0.39 + $\forall A, A'. (\forall I. (\text{read}(A', I) > I) \wedge A = \text{write}(\text{write}(A', 3, 5), 7, 9) \rightarrow \forall I. (\text{read}(A, I) > I))$
- (33) 3.78 + $\forall A, A', J. (\forall I. (\text{read}(A', I) > I) \wedge A = \text{write}(A', J, J + 3) \rightarrow \forall I. (\text{read}(A, I) > I))$
- (34) 0.37 + $\forall A, A'. (A = \text{write}(\text{write}(\text{write}(\text{write}(A', 3, 33), 4, 444), 5, 55), 4, 44) \wedge \forall I. (\text{read}(A', I) < 100) \rightarrow \forall I. (\text{read}(A, I) < 100))$
- (35) 0.30 + $\forall A, A', J, K. (\forall I. (J \leq I \wedge I \leq K \rightarrow \text{read}(A', I) > 0) \wedge A = \text{write}(A', K + 1, 3) \rightarrow \forall I. (J \leq I \wedge I \leq K + 1 \rightarrow \text{read}(A, I) > 0))$
- (36) 5.65 + $\forall A, A', J, K. (\forall I. (J \leq I \wedge I \leq K \rightarrow \text{read}(A', I) > 0) \wedge A = \text{write}(A', J + 2, \text{read}(A', J + 1) + 1) \rightarrow \forall I. (J \leq I \wedge I \leq K \rightarrow \text{read}(A, I) > 0))$
- (37) 3.52 - $\forall A, J, K. (\forall I. (J \leq I \wedge I \leq K \rightarrow \text{read}(A, I) > 0) \rightarrow \forall I. (J + 3 \leq I \wedge I \leq K \rightarrow \text{read}(A, I) > 0))$
- (38) 31.75 + $\forall A. (\forall I. (1 \leq I \wedge I \leq 10 \rightarrow \text{read}(A, I) > I) \wedge \forall I. (11 \leq I \wedge I \leq 20 \rightarrow \text{read}(A, I) > 20 - I) \rightarrow \forall I. (6 \leq I \wedge I \leq 15 \rightarrow \text{read}(A, I) > 5))$
- (39) 0.23 + $\forall A. (\forall I. (20 \leq I \wedge I \leq 30 \rightarrow \text{read}(A, I) = I + 25) \rightarrow \exists I. (\text{read}(A, I) = 50))$
- (40) 0.56 - $\forall A. (\forall I. (20 \leq I \wedge I \leq 30 \rightarrow \text{read}(A, I) = 2 \cdot I + 3) \rightarrow \exists I. (\text{read}(A, I) = 53))$
- (41) 0.27 + $\forall A, A', I, J, K. (\neg \text{read}(A', I) = \text{read}(A', J) \wedge A = \text{write}(\text{write}(\text{write}(A', J, 0), K, \text{read}(A', K) + 1), I, 0) \rightarrow \exists L. (\neg \text{read}(A, L) = \text{read}(A', L)))$
- (42) 9.90 + $\forall A, A', I, J, K. (A = \text{write}(\text{write}(\text{write}(A', I, 3), I + 2, 2), I + 4, 1) \wedge I \leq J \wedge J \leq I + 3 \rightarrow \exists K. (J \leq K \wedge K \leq J + 3 \wedge \text{read}(A, K) \leq 3))$

Integer arithmetic and collections. The formulas

$$\text{COL01 } \forall I. (\neg I \in \emptyset).$$

$$\text{COL02 } \forall I, S. (I \in \text{add}(I, S)).$$

$$\text{COL03 } \forall I, S. (\neg I \in \text{remove}(I, S)).$$

$$\text{COL04 } \forall I, S, J. (I \in S \vee I = J \leftrightarrow I \in \text{add}(J, S)).$$

$$\text{COL05 } \forall I, S, J. (I \in S \wedge \neg I = J \leftrightarrow I \in \text{remove}(J, S)).$$

were used to axiomatize collections of integers for the following problems:

$$(43) \quad 0.31 + \quad \neg 4 \in \text{add}(1, \text{add}(3, \text{add}(5, \emptyset)))$$

$$(44) \quad 0.56 + \quad \forall S, I, J. (S = \text{add}(5, \text{add}(3, \text{add}(1, \emptyset))) \\ \wedge I \in S \wedge J \in S \wedge \neg I = J \\ \rightarrow I + J < 9)$$

$$(45) \quad 0.23 + \quad \forall S, S'. (\forall I. (I \in S' \rightarrow I > 0) \\ \wedge S = \text{add}(2, \text{remove}(7, S')) \\ \rightarrow \forall I. (I \in S \rightarrow I > 0))$$

$$(46) \quad 0.27 + \quad \forall S, S'. (\forall I. (I \in S' \rightarrow I > 0) \\ \wedge S = \text{remove}(4, \text{remove}(1, \text{remove}(2, S'))) \\ \rightarrow \forall I. (I \in S \rightarrow I > 2))$$

$$(47) \quad 0.64 + \quad \forall S. (\forall I. (I \in S \rightarrow I \geq 0) \\ \wedge \neg 0 \in S \wedge \neg 1 \in S \\ \rightarrow \forall I. (I \in S \rightarrow I \geq 2))$$

$$(48) \quad 0.23 + \quad \forall S. (\forall I. (I \in S \rightarrow I \geq 0) \\ \wedge \forall I. (I \in S \leftrightarrow I \in \text{remove}(0, \text{remove}(1, S)))) \\ \rightarrow \forall I. (I \in S \rightarrow I \geq 2))$$

$$(49) \quad 1.92 + \quad \forall S, S', J. (\forall I. (I \in S' \rightarrow I > 0) \\ \wedge J \in S' \wedge S = \text{add}(J + 2, \text{remove}(J, S')) \\ \rightarrow \forall I. (I \in S \rightarrow I > 0))$$

$$(50) \quad 2.29 + \quad \forall S, S', J, K. (\forall I. (I \in S' \rightarrow I > 0) \\ \wedge J \in S' \wedge K \geq 0 \wedge S = \text{add}(J + K, \text{remove}(J, S')) \\ \rightarrow \forall I. (I \in S \rightarrow I > 0))$$

$$(51) \quad 1.37 + \quad \forall S, S'. (\forall I. (I \in S' \rightarrow I > 0) \\ \wedge \forall I. (I \in S \rightarrow \exists J. (J \in S' \wedge I > J)) \\ \rightarrow \forall I. (I \in S \rightarrow I > 1))$$

$$(52) \quad 1.95 + \quad \forall S, S'. (\forall I. (I \in S' \rightarrow I > 0) \\ \wedge \forall I. (I \in S \rightarrow \exists J. (J \in S' \wedge 2 \cdot I - 5 \cdot J > 0)) \\ \rightarrow \forall I. (I \in S \rightarrow I > 2))$$

$$(53) \quad 1.90 + \quad \forall S, S'. (\forall I. (I \in S' \rightarrow I > 0) \\ \wedge \forall I. (I \in S \rightarrow \exists J, K. (J \in S' \wedge K \in S' \wedge I = J + K)) \\ \rightarrow \forall I. (I \in S \rightarrow I > 1))$$

$$(54) \quad 0.36 + \quad \forall S. (S = \text{add}(10, \text{add}(30, \text{add}(50, \emptyset))) \\ \rightarrow \exists I. (20 \leq I \wedge I \leq 40 \wedge I \in S))$$

Integer arithmetic and collections with counting. The formulas

- CCO01 $\forall I. (\neg I \in \emptyset).$
 CCO02 $\forall I, S. (I \in \text{add}(I, S)).$
 CCO03 $\forall I, S. (\neg I \in \text{remove}(I, S)).$
 CCO04 $\forall I, S, J. (I \in S \vee I = J \leftrightarrow I \in \text{add}(J, S)).$
 CCO05 $\forall I, S, J. (I \in S \wedge \neg I = J \leftrightarrow I \in \text{remove}(J, S)).$
 CCO06 $\forall S. (\text{count}(S) \geq 0).$
 CCO07 $\forall S. (S = \emptyset \leftrightarrow \text{count}(S) = 0).$
 CCO08 $\forall I, S. (\neg I \in S \leftrightarrow \text{count}(\text{add}(I, S)) = \text{count}(S) + 1).$
 CCO09 $\forall I, S. (I \in S \leftrightarrow \text{count}(\text{add}(I, S)) = \text{count}(S)).$
 CCO10 $\forall I, S. (I \in S \leftrightarrow \text{count}(\text{remove}(I, S)) = \text{count}(S) - 1).$
 CCO11 $\forall I, S. (\neg I \in S \leftrightarrow \text{count}(\text{remove}(I, S)) = \text{count}(S)).$
 CCO12 $\forall I, S. (I \in S \rightarrow S = \text{add}(I, \text{remove}(I, S))).$

were used to axiomatize collections of integers with a counting operation for the following problems:

- (55) $22.09 + \quad \forall S. (\text{count}(\text{remove}(5, S)) \geq 7 \\ \rightarrow \text{count}(\text{remove}(4, S)) \geq 6)$
- (56) $3.71 + \quad \forall S. (\text{count}(\text{add}(5, S)) = \text{count}(\text{add}(3, S)) \\ \rightarrow \text{count}(\text{remove}(5, S)) = \text{count}(\text{remove}(3, S)))$
- (57) $2.20 + \quad \forall S, I. (\text{count}(S) + 1 \geq \text{count}(\text{add}(I, S)))$
- (58) $11.95 + \quad \forall S, I, K. (K > 0 \rightarrow \text{count}(S) + K \geq \text{count}(\text{add}(I, S)))$
- (59) $19.49 + \quad \forall S, I, K. (K > 0 \rightarrow \text{count}(S) + K \geq \text{count}(\text{add}(I, \text{add}(I, S))))$
- (60) $0.42 + \quad \forall S, I. (\text{count}(\text{remove}(2, S)) = 0 \\ \wedge \text{count}(\text{remove}(3, S)) = 0 \\ \rightarrow \text{count}(\text{remove}(I, S)) = 0)$
- (61) $0.32 + \quad \forall S. (2 \in S \wedge \text{count}(S) = 1 \rightarrow \neg 5 \in S)$
- (62) $23.22 + \quad \forall S. (2 \in S \wedge 3 \in S \wedge \text{count}(S) = 2 \rightarrow \neg 5 \in S)$
- (63) $28.16 + \quad \forall S, I. (2 \in S \wedge 3 \in S \wedge \text{count}(S) = 2 \wedge I > 3 \rightarrow \neg I \in S)$
- (64) $118.41 + \quad \forall S, I, J. (I < 3 \wedge 6 < J \\ \wedge I \in S \wedge J \in S \\ \wedge \text{count}(S) = 2 \\ \rightarrow \neg 5 \in S)$
- (65) $0.34 + \quad \exists S, I. (\text{count}(S) + 1 > \text{count}(\text{add}(I, S)))$

- (66) 309.91 + $\exists S. (\text{count}(S) = 3)$
- (67) 269.01 + $\forall S, I, J, K. (I > J \wedge J > K$
 $\wedge I \in S \wedge J \in S \wedge K \in S$
 $\rightarrow \text{count}(S) > 2)$
- (68) 0.53 + $\forall S, J. (\forall I. (I \in S \rightarrow J > I)$
 $\rightarrow \text{count}(\text{add}(J, S)) > \text{count}(S))$
- (69) 4.04 + $\text{count}(\text{add}(1, \text{add}(3, \emptyset))) = 2$
- (70) 1.02 + $\text{count}(\text{add}(5, \text{remove}(3, \text{add}(3, \emptyset)))) = 1$
- (71) $\infty - \text{count}(\text{add}(1, \text{add}(5, \text{remove}(3, \text{add}(2, \emptyset)))))) = 3$
- (72) 3.49 + $\forall S, I. (\text{count}(\text{remove}(I, \text{add}(I, S))) = \text{count}(\text{remove}(I, S)))$
- (73) $\infty - \forall S, I. (\text{count}(\text{add}(0, \text{remove}(I, \text{add}(I, S))))$
 $= \text{count}(\text{add}(0, \text{remove}(I, S))))$

Integer arithmetic and pointer data types. The formulas

- PNT01 $\forall X. (\neg \text{isrecord}(X) \rightarrow \text{length}(X) = 0)$
- PNT02 $\forall X. (\text{isrecord}(X) \rightarrow \text{length}(X) \geq 1)$
- PNT03 $\forall X. (\text{isrecord}(X) \rightarrow \text{length}(X) = \text{length}(\text{next}(X)) + 1)$
- PNT04 $\forall X. (\neg \text{isrecord}(X) \rightarrow \neg \text{isrecord}(\text{split1}(X)))$
- PNT05 $\forall X. (\text{isrecord}(X) \rightarrow \text{isrecord}(\text{split1}(X)))$
- PNT06 $\forall X. (\text{isrecord}(X) \rightarrow \text{data}(\text{split1}(X)) = \text{data}(X))$
- PNT07 $\forall X. (\text{isrecord}(X) \wedge \neg \text{isrecord}(\text{next}(X))$
 $\rightarrow \neg \text{isrecord}(\text{next}(\text{split1}(X))))$
- PNT08 $\forall X. (\text{isrecord}(X) \wedge \text{isrecord}(\text{next}(X))$
 $\rightarrow \text{next}(\text{split1}(X)) = \text{split1}(\text{next}(\text{next}(X))))$
- PNT09 $\forall X. (\neg \text{isrecord}(X) \rightarrow \neg \text{isrecord}(\text{split2}(X)))$
- PNT10 $\forall X. (\neg \text{isrecord}(\text{next}(X)) \rightarrow \neg \text{isrecord}(\text{split2}(X)))$
- PNT11 $\forall X. (\text{isrecord}(X) \wedge \text{isrecord}(\text{next}(X)) \rightarrow \text{isrecord}(\text{split2}(X)))$
- PNT12 $\forall X. (\text{isrecord}(X) \wedge \text{isrecord}(\text{next}(X))$
 $\rightarrow \text{data}(\text{split2}(X)) = \text{data}(\text{next}(X)))$
- PNT13 $\forall X. (\text{isrecord}(X) \wedge \text{isrecord}(\text{next}(X))$
 $\rightarrow \text{next}(\text{split2}(X)) = \text{split2}(\text{next}(\text{next}(X))))$

were used to axiomatize singly linked lists with pointers and two recursive functions *split1* and *split2* (computing the sublists of odd-numbered and even-numbered elements of a list) for the following problems:

- (74) 20.02 + $\forall X, Y. (\text{isrecord}(X) \wedge \text{isrecord}(\text{next}(X)) \wedge Y = \text{next}(\text{next}(X))$
 $\wedge (2 \cdot \text{length}(\text{split2}(Y)) = \text{length}(Y) - 1$
 $\vee 2 \cdot \text{length}(\text{split2}(Y)) = \text{length}(Y))$
 $\rightarrow (2 \cdot \text{length}(\text{split2}(X)) = \text{length}(X) - 1$
 $\vee 2 \cdot \text{length}(\text{split2}(X)) = \text{length}(X)))$

$$(75) \quad 17.18 + \quad \forall X, Y. (isrecord(X) \wedge isrecord(next(X)) \wedge Y = next(next(X)) \\ \wedge length(split1(Y)) + length(split2(Y)) = length(Y) \\ \rightarrow length(split1(X)) + length(split2(X)) = length(X))$$

7.3 Discussion

Summarizing the previous section, we see that SPASS+T solves 63 out of the 75 problems in our list. In order to check to what extent the mechanisms implemented in SPASS+T contribute to these proofs, we have gradually disabled various features and re-run the tests. The following table shows the number of proofs found for several combinations of features:

SMT procedure	Theory instantiation	Arithmetic simplification	Auxiliary axioms	Problems solved (out of 75)
+	+	+	+	63
+	+	-	+	52
+	-	+	+	56
-	-	+	+	39
-	-	-	+	12
+	+	+	-	53

How does SPASS+T compare to a system like Simplify (Detlefs, Nelson, and Saxe [4])? The automatic theorem prover Simplify was developed as the proof engine of ESC/Java and ESC/Modula-3. It handles universal quantifiers in the input by computing (hopefully) relevant instances in a similar way as the *theory instantiation* rule of SPASS+T and analyzing the resulting formulas using SAT checking and a Nelson-Oppen combination of decision procedures. Simplify (version 1.5.4) solves 40 out of the 75 problems in our list. The difference to SPASS+T is partially due to the fact that Simplify solves only four out of 23 problems that involve existential quantifiers, namely (41) and (51)–(53). Somewhat surprisingly, however, Simplify fails also for (55), (56), (59), and (68)–(74). On the other hand, due to the highly optimized implementation and the reduced search space, all problems that Simplify does solve are solved in less than one second.

It is perhaps illustrative to have a closer look at those problems that SPASS+T failed to prove. SPASS+T has some support for solving non-ground equations, but it makes no attempt to be complete in this domain, and in particular it does not use any kind of theory unification. This accounts for most of the missed proofs in the TPTP list and in theorems (1)–(14), as well as for (22), (25), and (40). It may look strange that SPASS+T succeeds for (2) and (4), but fails for (1) and (3), but the explanation is easy: (2) and (4) are proved by superposition with INT01. We could add the symmetric version of INT01, so that (1) and (3) become provable as well, but having both INT01 and its symmetric version in the clause set destroys termination and seems to create more problems than it solves. Similarly, (7) and (8) can be handled by superposition with INT14 or INT16 plus elementary equation solving, whereas (9) would require a fully-fledged Diophantine equation solver.

Theorem (19) is rather difficult for an automated system because for proving it one has to “invent” a term that does not occur in the problem, namely $f(f(Y) + 1)$, and

then use transitivity. Moreover, as one might expect, SPASS+T has no chance to prove the induction theorem (26). On the other hand, an extension of the *theory instantiation* rule would enable SPASS+T to prove the pigeonhole-like formula $\forall X, Y. (f(X) = f(Y) \rightarrow X = Y) \wedge 6 < f(3) \wedge f(3) < 9 \wedge 6 < f(4) \wedge f(4) < 9 \rightarrow f(5) < 6 \vee 9 < f(5)$ from the TPTP list. Two changes are required: The inference rule *theory instantiation II* has to take an arbitrary number of premises

$$\frac{\text{ground}(s_1) \dots \text{ground}(s_k) \quad L[t_1, \dots, t_k] \vee D}{\text{ground}(u_1) \dots \text{ground}(u_n) \quad \mathbf{export}(D')}$$

moreover it has to be applicable even if the terms t_i occur below a negated equation symbol. So far, we have not implemented this modification, since the extended rule would be extremely prolific.

Even though the CCO axioms have a simple operational reading, it seems to be difficult for automated provers to detect this fact, as one can see from the poor results for straightforward computation tasks like (71) or (73). Simplify even exceeds the 600 second time limit for the simpler variants (69), (70), and (72).

There is one odd case left: Surprisingly, SPASS+T succeeds to prove formula (37), if *theory instantiation* or *integer ordering expansion* are switched off, but fails to find the proof if both inference rules are switched on. It turns out that, in the latter case, SPASS+T triggers a bug in CVC Lite.

7.4 Download

A current snapshot of SPASS+T for Linux including the benchmark problems can be downloaded from <http://www.mpi-inf.mpg.de/~uwe/software/>.

8 Outlook

The development of SPASS+T is ongoing work. More effort has to be spent on fine-tuning the theory instantiation rule and the arithmetic simplification rules. A more challenging extension is the implementation of subsumption checks for clauses with theory literals, which might turn SPASS+T into a decision procedure for some classes of formulas. We are also looking into using SPASS+T with non-numeric base theories, such as arrays, lists, or bitvectors.

Acknowledgments: We are grateful to Christoph Weidenbach and Thomas Hillenbrand for helpful discussions.

References

- [1] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing*, 5(3/4):193–212, 1994.
- [2] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the Cooperating Validity Checker. In Rajeev Alur and Doron A. Peled, editors, *Computer*

- Aided Verification, 16th International Conference, CAV 2004*, LNCS 3114, pages 515–518, Boston, MA, USA, 2004. Springer-Verlag.
- [3] Robert S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In Jean E. Hayes, Donald Michie, and Judith Richards, editors, *Machine Intelligence 11: Logic and the acquisition of knowledge*, chapter 5, pages 83–124. Oxford University Press, 1988.
 - [4] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
 - [5] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004*, LNCS 3114, pages 175–188, Boston, MA, USA, 2004. Springer-Verlag.
 - [6] Harald Ganzinger, Viorica Sofronie-Stokkermans, and Uwe Waldmann. Modular proof systems for partial functions with weak equality. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning: Second International Joint Conference, IJCAR 2004*, LNAI 3097, pages 168–182, Cork, Ireland, 2004. Springer-Verlag. Corrected version at <http://www.mpi-sb.mpg.de/~uwe/paper/PartialFun-bibl.html>.
 - [7] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005*, LNCS 3576, pages 476–490, Edinburgh, Scotland, UK, 2005. Springer-Verlag.
 - [8] Stephan Schulz and Geoff Sutcliffe. <http://www.cs.miami.edu/~tptp/TPTP/Proposals/IntegerArithmetic.p>, March 15, 2006.
 - [9] Geoff. Sutcliffe and Christian B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
 - [10] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobalt, and Dalibor Topić. SPASS version 2.0. In Andrei Voronkov, editor, *Automated Deduction – CADE-18, 18th International Conference on Automated Deduction*, LNAI 2392, pages 275–279, Copenhagen, Denmark, 2002. Springer-Verlag.