Labelled Clauses

Tal Lev-Ami[1]*, Christoph Weidenbach[2], Thomas Reps[3]**, and Mooly Sagiv[1]

[1] School of Comp. Sci., Tel Aviv University, {tla,msagiv}@post.tau.ac.il
[2] Max-Planck-Institut für Informatik, Saarbrücken, weidenbach@mpi-inf.mpg.de
[3] Comp. Sci. Dept., University of Wisconsin, Madison, reps@cs.wisc.edu

Abstract. We add labels to first-order clauses to simultaneously apply superpositions to several proof obligations inside one clause set. From a theoretical perspective, the approach unifies a variety of deduction modes. These include different strategies such as set of support, as well as explicit case analysis, e.g., splitting. From a practical perspective, labelled clauses offer advantages in the case of related proof obligations resulting from multiple conjectures over the same axiom set or from a single conjecture that is a large conjunction. Here we can share clauses (e.g., the axioms and clauses deduced from them, share Skolem symbols), share deduced clause variants, and transfer lemmas between the different obligations. Motivated by software verification, we have created a prototype implementation of labelled clauses that supports multiple conjectures, and we provide convincing experiments for the benefits.

1 Introduction

Our work in program analysis and software verification using the TVLA system [10] has led us to explore the use of theorem provers for computing the effects of program statements [25]. Recently, we explored the possibility of using first-order automated theorem provers for the task [9]. A major obstacle in using existing automated theorem provers, such as E [18], Spass [21] and Vampire [16], is that of performance. TVLA requires many calls to the theorem prover to compute the effect of a single program statement. Therefore, the overall time required for analyzing even a simple program is prohibitive. This situation is common in other program analysis methods such as Cartesian Abstraction [1].

In the process of computing the effect of a program statement, TVLA generates multiple conjectures that share a common axiom set. However, current theorem provers can only attempt to prove a single conjecture at a time. Running the theorem prover multiple times, once for each conjecture, has three problems: (1) if the conjectures are proven sequentially, any inferences/reductions made between clauses from the axiom set are reconstructed and no synergies between the proof attempts can be exploited, (2) because not all the conjectures are

* Supported by an Adams Fellowship through the Israel Academy of Sciences and Humanities.

** Supported by ONR under grant N00014-01-1-0796 and by NSF under grants CCF-0540955 and CCF-0524051.

valid and first-order logic is only semi-decidable, we are required to use timeouts, which may cause us to give up on valid conjectures, (3) if a conjecture is validated by a proof attempt, there is no automatic way inside the prover to transfer the result to other proof attempts.

To overcome these obstacles, we devised algorithms to prove different conjectures simultaneously inside one proof attempt. We chose to modify the calculus used by the theorem prover to support deduction in parallel, i.e., for multiple conjectures, via a methodology of labelled clauses. The idea is to label each clause with the conjectures for which it is relevant, and update these labels during deduction. As a consequence, we solved the above problems: (1) inferences on axiom clauses are no longer duplicated but are shared — as are variant clauses derived from separate conjectures and reductions, (2) a fair strategy among all proof attempts (with one global timeout parameter) replaces the use of separate timeouts, and (3) valid conjectures can be naturally transferred to other proof attempts.

We have created a prototype implementation of labelled clauses for the purpose of proving multiple conjectures within the automated theorem prover Spass, and report on convincing experimental results in the context of our original application.

After having seen the success of labelled clauses for multiple conjectures, we believe that the methodology of labelled clauses has more potential in the context of first-order theorem proving. Labelled deductive systems (see e.g., [2]) as have been pushed forward by Dov Gabbay in the last fifteen years and have become recognized as a significant component of the logic culture, in particular in the context of non-classical logics. There labels are used on the one hand to bring the semantics into the syntax by naming possible worlds using labels (e.g., a Kripke structure) and on the other hand they can act as proof-theoretic resource labels. Our motivation of using labels is different. We suggest to use labels to study and implement variants of classical first-order-logic (superposition-based) theorem-proving calculi to eventually improve automation. We show that the methodology of labels carries over beyond proving multiple conjectures by instantiating the methodology for clausal splitting (see e.g., [20]), slicing (see e.g., [23]), and the set-of-support strategy (see e.g., [24]).

We use labels to summarize the derivation tree of a clause. For example, when using labelled clauses for clausal splitting, the labels represent the splits the clause depends on, and when using labelled clauses for multiple conjectures, the labels represent the conjectures for which the clause is valid. The purpose of the abstract framework described in Sect. 2 is to generalize the different applications of labelled clauses and give a common presentation for all of them. We believe that the abstract framework has interesting properties of its own, which we plan to investigate as future work.

1.1 Related Work

For the general methodology of labelled clauses there is a huge literature related to our approach in the sense that the work can be reformulated and explored via a labelled-clause discipline. A discussion of this general aspect is beyond the scope of this paper. Therefore, we concentrate on work related to our approach of proving multiple conjectures via labelled clauses.

Similar techniques for enhancing a theorem prover (for classical logic) to work with multiple goals have been developed by A. Voronkov, and are being incorporated into Vampire [19]. However, his approach is based on an extension of splitting, as described in [15].

A logical candidate for proving conjectures in parallel is to use additional predicates [6]. In our setting, this would mean replacing each negated conjecture $\neg\varphi_i$ with the disjunction $b_i \vee \neg\varphi_i$ where b_i is a fresh propositional variable (i.e., nullary relation). Now if the prover deduces the unit clause b_i, the conjecture φ_i is valid. The main problem with this approach is that it will also explore disjunctions between different b_i's, i.e., between different conjectures, which unnecessarily increases the search space by an exponential factor. Furthermore, the literals added to the clauses block any reductions from (derived) conjecture clauses in the axioms' clauses.

Symbolic decision procedures [8] are a technique for finding all possible disjunctions of conjectures for a given axiom set. The technique is limited to specific theories, such as uninterpreted functions and difference logic. Furthermore, because it is described in the context of the Nelson-Oppen method [13] for combining decision procedures, its usability in the case of quantifiers is limited.

In the Boolean satisfiability community there is a related idea of incremental solvers (see e.g., [22]). There, it is possible to add and remove clauses from the theorem prover without restarting it. On the other hand, labelled clauses allow us to attempt to prove multiple conjectures simultaneously.

1.2 Contributions

The main contributions of this paper are as follows:

- We introduce a deduction technique for working on multiple related proof attempts simultaneously. We have successfully created a first prototype implementation within Spass.
- We describe applications of the method in software verification, and provide experimental results to demonstrate the improvement.
- We propose the concept of superposition with labels as a general framework for the study of deduction techniques and their combination.
- We demonstrate several instantiations of this general framework for implementing different ideas, including clause splitting and slicing.

In Sect. 2, we present a labelled superposition calculus as an extension of the standard superposition calculus. In Sect. 3, we present several instantiations of the general calculus including one for multiple conjectures. In Sect. 4, we present applications of multiple conjectures in software verification and present experimental results from using our extension of Spass for handling multiple conjectures. We conclude in Sect. 5.

The tool as well as the input files used for the experiments are available at [11].

2 Superposition with Labels

For the presentation of the superposition calculus, we refer to the notation and notions of [20]. We write clauses in implication form $\varGamma \rightarrow \varDelta$ where comma to the left means conjunction and comma to the right disjunction. Upper Greek letters denote sequences of atoms $(\varGamma, \varDelta)$, lower Greek letter substitutions (σ, τ), lower Latin characters terms (s, r, t) and upper Latin characters atoms (A, B, E) where $\approx$ is used as the equality symbol. The notations $s[l]_p$ $(E[l]_p)$ expresses that the term s (the atom E) contains the term l at position p. Replacing a subterm at position p with a term r is denoted by $s[p/r]$ $(E[p/r])$.

We distinguish inference from reduction rules, where the clause(s) below the bar, the conclusions, of an inference rule are added to the current clause set, while the clause(s) below the bar of a reduction rule replace the clause(s) above the bar, the premises. For example,

$$\mathcal{I}\frac{\varGamma_1 \rightarrow \varDelta_1 \qquad \varGamma_2 \rightarrow \varDelta_2}{\varGamma_3 \rightarrow \varDelta_3} \qquad\qquad \mathcal{R}\frac{\varGamma'_1 \rightarrow \varDelta'_1 \qquad \varGamma'_2 \rightarrow \varDelta'_2}{\varGamma'_3 \rightarrow \varDelta'_3}$$

an application of the above inference adds the clause $\varGamma_3 \rightarrow \varDelta_3$ to the current clause set, while the above reduction replaces the clauses $\varGamma'_1 \rightarrow \varDelta'_1$, $\varGamma'_2 \rightarrow \varDelta'_2$ with the clause $\varGamma'_3 \rightarrow \varDelta'_3$. Note that reductions can actually be used to delete clauses, if there are no conclusions.

For the introduction and proofs of the properties of our label discipline, sorts, ordering or selection restrictions are not of importance. Therefore, we leave them out of the presentation of the superposition inference and reduction rules. They can all be added in a straightforward way.

To each superposition clause $\varGamma \rightarrow \varDelta$ we add a label m resulting in $m : \varGamma \rightarrow \varDelta$. Then the standard calculus rules are extended by conditions and operations on the labels. We use a binary operation $\circ$ to combine labels for inferences, and a binary operation $\bullet$ to combine labels for reductions. Both operations are commutative and associative. We use $\ominus$ as a special label to indicate when an inference or a reduction should be blocked.[4] Finally, a preorder, $\leq$, is used to define when labels are compatible for clause deletion.

The interpretation of the labels and label operators depends on the instantiation of the superposition-with-labels calculus. We will give examples in Section 3. In particular, the standard superposition calculus is obtained if all clauses are labelled by the set $\{1\}$ and $\circ$, $\bullet$, $\leq$, and $\ominus$ are instantiated by the standard set operations $\cap$, $\cap$, $\subseteq$, and $\emptyset$, respectively.

Below we present the extended inference rule superposition right. The missing binary inference rules superposition left, merging paramodulation, and ordered resolution are defined accordingly.

Definition 1 (Superposition Right). *The inference*

$$\mathcal{I}\frac{m_1 : \varGamma_1 \rightarrow \varDelta_1, l \approx r \qquad m_2 : \varGamma_2 \rightarrow \varDelta_2, s[l']_p \approx t}{(m_1 \circ m_2 : \varGamma_1, \varGamma_2 \rightarrow \varDelta_1, \varDelta_2, s[p/r] \approx t)\sigma}$$

[4] We require that, for any label m, $\ominus \circ m = \ominus$ and $\ominus \bullet m = \ominus$.

where
1. $m_1 \circ m_2 \neq \ominus$
2. σ *is the mgu of* l' *and* l
3. l' *is not a variable*
and the usual ordering/selection restrictions apply is called a superposition right inference.

For the unary inference rules equality resolution, ordered factoring, and equality factoring on a clause labelled with m, the resulting clause is labelled with $m \circ m$, and the rule has an extra condition that $m \circ m \neq \ominus$. For inference rules with multiple premises, such as hyper resolution, the label computations are straightforward extensions of the binary case.

Definition 2 (Ordered Hyper Resolution). *The inference*

$$\mathcal{I}\frac{m_1 \,:\, E_1, \ldots, E_n \rightarrow \Delta \qquad m_{2,i} \,:\rightarrow \Delta_i, E_i' \quad (1 \leq i \leq n)}{(m_1 \circ m_{2,1} \circ \ldots \circ m_{2,n} \,:\rightarrow \Delta, \Delta_1, \ldots, \Delta_n)\sigma}$$

where
1. $m_1 \circ m_{2,1} \circ \ldots \circ m_{2,n} \neq \ominus$
2. σ *is the simultaneous mgu of* $E_1, \ldots, E_n, E_1', \ldots, E_n'$*,*
and the usual ordering restrictions apply is called an ordered hyper resolution inference.

Below we present matching replacement resolution, weak contextual rewriting, and subsumption deletion as examples of how to extend reduction rules by labels. In contrast to the standard superposition calculus, one or more premises of a reduction rule are typically retained.

The reason for this is illustrated by the application of labelled clauses to splitting (see Sect. 3.2), where the label describes the clause splittings on which a given clause depends. Here it is clear that when a reduction is performed from a clause C_1 to a clause C_2 that depends on fewer splittings, we must keep C_1.

Definition 3 (Matching Replacement Resolution). *The reduction*

$$\mathcal{R}\frac{m_1 \,:\, \Gamma_1 \rightarrow \Delta_1, E_1 \quad m_2 \,:\, \Gamma_2, E_2 \rightarrow \Delta_2}{\begin{array}{c} m_1 \,:\, \Gamma_1 \rightarrow \Delta_1, E_1 \\ m_2 \,:\, \Gamma_2, E_2 \rightarrow \Delta_2 \\ m_1 \bullet m_2 \,:\, \Gamma_2 \rightarrow \Delta_2 \end{array}}$$

where
1. $m_1 \bullet m_2 \neq \ominus$
2. $E_1\sigma = E_2$
3. $(\Gamma_1 \rightarrow \Delta_1)\sigma$ *subsumes* $\Gamma_2 \rightarrow \Delta_2$*, where all variables in the co-domain of* σ *are treated as constants for both clauses*
is called matching replacement resolution.

This presentation of the matching replacement resolution rule is non-standard, because the parent clause $m_2 : \Gamma_2, E_2 \rightarrow \Delta_2$ is kept. However, in many applications it can then be subsequently deleted by subsumption deletion (see below). For example, in the case of simulating the standard calculus where all labels are identical to $\{1\}$, the clause $\{1\} : \Gamma_2, E_2 \rightarrow \Delta_2$ is always subsumed by $\{1\} \cap \{1\} : \Gamma_2 \rightarrow \Delta_2$, yielding the standard rule.

We present a variant of the rewriting rule; other variants are modified similarly. Two versions of the rule are supplied. The first one is a reduction that can be used when the resulting label is smaller than the parent clause to be deleted. In the second version, the clause is simplified, but the parent clause is not deleted.

Definition 4 (Weak Contextual Rewriting). *The reductions*

$$\mathcal{R}\frac{m_1 : \Gamma_1 \rightarrow \Delta_1, s \approx t \qquad m_2 : \Gamma_2 \rightarrow \Delta_2, E[s']_p}{\begin{array}{c} m_1 : \Gamma_1 \rightarrow \Delta_1, s \approx t \\ m_1 \bullet m_2 : \Gamma_2 \rightarrow \Delta_2, E[p/t\sigma] \end{array}}$$

and

$$\mathcal{R}\frac{m_1 : \Gamma_1 \rightarrow \Delta_1, s \approx t \qquad m_2 : \Gamma_2 \rightarrow \Delta_2, E[s']_p}{\begin{array}{c} m_1 : \Gamma_1 \rightarrow \Delta_1, s \approx t \\ m_2 : \Gamma_2 \rightarrow \Delta_2, E[s']_p \\ m_1 \bullet m_2 : \Gamma_2 \rightarrow \Delta_2, E[p/t\sigma] \end{array}}$$

where

1. $m_1 \bullet m_2 \neq \ominus$
2. $s\sigma = s'$
3. $\Gamma_1\sigma \subseteq \Gamma_2$, $\Delta_1\sigma \subseteq \Delta_2$
4. *For the first variant,* $m_1 \bullet m_2 \leq m_2$
5. *For the second variant* $m_1 \bullet m_2 \not\leq m_2$

and the usual ordering restrictions apply are called weak contextual rewriting.

The unary simplification rules, such as trivial literal elimination or condensation, are handled similarly to unary inference rules; i.e., given a clause labelled with m, the resulting clause is labelled with $m \bullet m$ and the rule has an extra condition that $m \bullet m \neq \ominus$.

For rules that actually delete clauses, such as tautology deletion and subsumption deletion, we need to guarantee the compatibility of labels, as shown for subsumption deletion below. Tautology deletion is never blocked by the labels because $\leq$ is reflexive.

Definition 5 (Subsumption Deletion). *The reduction*

$$\mathcal{R}\frac{m_1 : \Gamma_1 \rightarrow \Delta_1 \qquad m_2 : \Gamma_2 \rightarrow \Delta_2}{m_1 : \Gamma_1 \rightarrow \Delta_1}$$

where

1. $m_2 \leq m_1$
2. $\Gamma_2 \rightarrow \Delta_2$ *is subsumed by* $\Gamma_1 \rightarrow \Delta_1$

is called subsumption deletion.

3 Instantiations

3.1 Multiple Labelled Conjectures

The starting point are n different proof obligations $\Psi_1, \ldots, \Psi_n$ (dependent or independent) with respect to a theory Φ. We want to check for which i $\Phi \models \Psi_i$ holds. Labels are subsets of $\{1, \ldots, n\}$. All clauses resulting from Φ receive the label $\{1, \ldots, n\}$, and all clauses resulting from Ψ_i receive the label $\{i\}$. The label indicates for which proof obligations the clause may be used. The operations $\circ$, $\bullet$, $\leq$, and $\ominus$ are instantiated by the standard set operations $\cap$, $\cap$, $\subseteq$, and $\emptyset$, respectively.

Proposition 1. *From $\Phi \wedge \neg\Psi_i$ we can derive $\rightarrow \square$ by superposition iff we can derive $\{i\} : \rightarrow \square$ by labelled superposition.*

Proof. (Sketch) By induction on the length of the superposition derivation doing a case analysis over the different rules. For all labelled inference and simplification rules it holds that the conclusion of a rule is labelled with a number i iff all premises are labelled with i. If a clause can be removed by labelled subsumption deletion, there exists a more general clause labelled with a superset, i.e., subsumption deletion can be applied to the standard subproofs.

3.1.1 Refinements Labelled subsumption deletion is actually weaker than subsumption deletion. For example, in the form shown in Definition 5 it does not enable subsumption of axiom clauses by conjecture clauses. Furthermore, the calculus considered so far for multiple labelled conjectures does not consider sharing of clauses resulting (from inferences) from different conjectures. Both problems can be overcome by adding the following rules to the calculus.

Definition 6 (Join). *The reduction*

$$\mathcal{R}\frac{m_1 : \Gamma_1 \rightarrow \Delta_1 \qquad m_2 : \Gamma_2 \rightarrow \Delta_2}{m_1 \cup m_2 : \Gamma_1 \rightarrow \Delta_1}$$

where

1. $m_1 \neq \emptyset$ and $m_2 \neq \emptyset$
2. $\Gamma_2 \rightarrow \Delta_2$ and $\Gamma_1 \rightarrow \Delta_1$ are variants[5]
is called join.

Definition 7 (Subsumption Separation). *The reduction*

$$\mathcal{R}\frac{m_1 : \Gamma_1 \rightarrow \Delta_1 \qquad m_2 : \Gamma_2 \rightarrow \Delta_2}{\begin{array}{c} m_1 : \Gamma_1 \rightarrow \Delta_1 \\ m_2 \setminus m_1 : \Gamma_2 \rightarrow \Delta_2 \end{array}}$$

where

1. $m_1 \cap m_2 \neq \emptyset$
2. $m_2 \not\subseteq m_1$
3. $\Gamma_2 \rightarrow \Delta_2$ is subsumed by $\Gamma_1 \rightarrow \Delta_1$
is called subsumption separation.

[5] equal with respect to variable renaming

Subsumption separation can be built into other reduction rules that rely on subsumption. For example, when $m_2 \nsubseteq m_1$ we can turn matching replacement resolution, Definition 3, into a reduction, keeping the first parent clause and reducing the second parent clause to $m_2 \setminus m_1 : \Gamma_2, E_2 \rightarrow \Delta_2$.

Note that using labels allow us to perform reductions of the axiom clauses using conjecture clauses. This would not be possible when using extra predicates for handling multiple conjectures.

When adding the Join rule, the correctness claim needs to be refined as well.

Proposition 2. *From $\Phi \wedge \neg\Psi_i$ we can derive $\rightarrow \Box$ by superposition iff we can derive $m : \rightarrow \Box$ by labelled superposition for some label m that contains i.*

When deriving an empty clause with a label that contains a certain conjecture, we know that the conjecture is valid given the axiom set. It is sometimes useful to then clausify the conjecture and add the result as new axioms to the axiom set so that it can be used by the other conjectures not yet proven.

Reusing Skolem Functions. We can use the labels to devise a better Skolemization process for our setting that results in more clauses being shared between the different proof obligations and thereby avoids duplicate work. Our experimental results is Sect. 4.3 show that this is valuable in practice. The idea is that two Skolem functions of the same arity can be merged if all the clauses they appear in have disjoint labels: any inference between such clauses is blocked, and thus the terms containing the merged functions will never be unified.

In general, we would want to merge Skolem functions in a way that allows the Join rule to share the most active clauses. Lacking a way to predict this, we use a heuristic that ensures that at least some clauses will be shared as a result of the process. The heuristic searches for clauses that will be variants after Skolem functions are merged. The heuristic guarantees that the merge is allowed by maintaining for each symbol the set of conjectures it is currently used in.

Example 1 *Let conjecture 1 be $\forall v \,.\, p(v) \vee q(v)$ and conjecture 2 be $\forall v \,.\, p(v) \vee \neg q(v)$. The usual Skolemization of the negated conjectures would result in the following four clauses: $\{1\} : p(c_1) \rightarrow, \{1\} : q(c_1) \rightarrow, \{2\} : p(c_2) \rightarrow, \{2\} : \rightarrow q(c_2)$. However, by sharing Skolem constants the following clauses suffice: $\{1, 2\} : p(c_1) \rightarrow, \{1\} : q(c_1) \rightarrow, \{2\} : \rightarrow q(c_1)$.*

Note that although $\{1\} : q(c_1) \rightarrow$ and $\{2\} : \rightarrow q(c_1)$ are resolvable if labels were ignored (resulting in the empty clause), they will never be resolved because their labels are disjoint. As the number and complexity of the conjectures increases, more of the shared structures can be used.

Conjunction Conjectures. When trying to prove a conjecture composed of a large conjunction of formulas, the standard practice of negation and then Skolemization creates large clauses and is typically intractable. Instead, the conjunction can be split into a conjecture per conjunct and processed using the refined calculus above. Now, proving a contradiction for *all* the labels is equivalent to proving the validity of the original formula.

Proposition 3. *From $\Phi \wedge \neg(\bigwedge_{1\leq i\leq n} \Psi_i)$ we can derive $\rightarrow \Box$ by superposition iff we can derive $\{1,\ldots,n\} \,:\rightarrow \Box$ by labelled superposition for multiple labelled conjectures.*

Theory Consistency. When proving that $\Phi \rightarrow \Psi$ is valid, it is possible that the reason for the validity is that Φ is inconsistent. In the framework of the standard superposition calculus we cannot distinguish between the two cases without inspecting proofs. However, it is easy to support such a consistency check in the context of labelled superposition. We simply add to the label of the axioms a new number 0 resulting in the extended set $\{0,\ldots,n\}$ for n conjectures. Then the theory is inconsistent iff we can derive $m \,:\rightarrow \Box$ for some label m that contains 0.

3.1.2 Implementation We have a prototype implementation of the labelled clauses calculus for multiple conjectures within the Spass theorem prover (available at [11]). Labels were implemented using bit-vectors attached to each clause. The inference and reduction rules were modified to correctly maintain the labels during derivations. The Join rule and Skolem function reuse were also implemented.

The most challenging part in modifying Spass to support labelled clauses is updating the forward reduction rules, which can now generate many clauses instead of only reducing the given clause. We used a naive approach for implementing these reductions and yet the result is still effective, as can be seen in Sect. 4.3. Backward reduction rules were modified to perform separation to correctly handle, for example, reduction of axiom clauses by conjecture clauses.

We can prevent one conjecture from starving the rest by once in a while selecting a clause from a conjecture that was missing from the labels of the recently selected given clauses. We have implemented this idea as a runtime option in the new version, but do not yet have experimental results concerning its effectiveness.

3.2 Labelled Splitting

Let us consider how labels can be used to model splitting with a single conjecture. For splitting we use a different type of label: sequences of (overlined) clauses with an extra label $\ominus$. We use the labels to record the path in the derivation tree of splits required to generate the clause. We say that $m_1 \leq m_2$ when m_2 is a prefix of m_1, or $m_1 = \ominus$. The combine operations are simply the greatest lower bound of $\leq$, i.e., $m_1 \circ m_2 = m_1$ if $m_1 \leq m_2$, $m_1 \circ m_2 = m_2$ if $m_2 \leq m_1$, and $m_1 \circ m_2 = \ominus$ otherwise (we define $\bullet$ to be the same is $\circ$). Initially, all clauses are labelled with the empty sequence, denoted by ϵ.

In addition to the labelled superposition rules the extra rules implementing labelled splitting are:

Definition 8 (Splitting). *The inference*

$$\mathcal{I}\,\frac{m \,:\, \Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2}{\begin{array}{c} m.\overline{\Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2} \,:\, \Gamma_1 \rightarrow \Delta_1 \\ m.\overline{\Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2} \,:\, \Gamma_2 \rightarrow \Delta_2 \end{array}}$$

where

1. $vars(\Gamma_1 \rightarrow \Delta_1) \cap vars(\Gamma_2 \rightarrow \Delta_2) = \emptyset$
2. $\Delta_1 \neq \emptyset$ *and* $\Delta_2 \neq \emptyset$

is called splitting.

Definition 9 (Branch Closing). *The inference*

$$\mathcal{I}\frac{m.C \; : \rightarrow \square \qquad m.\overline{C} \; : \rightarrow \square}{m \; : \rightarrow \square}$$

is called branch closing.

Proposition 4. *From a clause set N we can derive a contradiction by superposition with splitting iff we can derive $\epsilon \; : \rightarrow \square$ by labelled superposition with splitting.*

Proof. (Sketch) By induction on the length of the superposition derivation, via a case analysis over the different rules. The Splitting rule needs special consideration: we have to show that no inference (reduction) between two clauses $m.C.m_1 \; : \; \Gamma_1 \rightarrow \Delta_1$ and $m.\overline{C}.m_2 \; : \; \Gamma_2 \rightarrow \Delta_2$ is possible. This holds by the definitions of the two combination operations ($\circ$ and $\bullet$): combining any two sequences $m.C.m_1$ and $m.\overline{C}.m_2$ results in $\ominus$. On the other hand, the combination of $m.C$ (or $m.\overline{C}$) and any prefix of m results in $m.C$ ($m.\overline{C}$), which corresponds to the standard clause set copy and separation of the standard splitting rule.

3.2.1 Refinements

Once the labels are available and the different branches of the derivation tree spanned by the splitting rule can be investigated simultaneously, it is easy to define and employ the well-known refinements for splitting and tableau proofs. By studying the labels of clauses used in the derivation of an empty clause, refinements like branch condensing (implemented in Spass, splittings from the most recent backtracking empty clause that did not contribute to the current empty clause can be removed from the label) or the generation of more suitable "backtracking clauses" (see e.g., [14]) are straightforward to integrate. The lemma-generation rules invented in the context of tableau [7], for example, the *folding up rule* can also be integrated.

An obvious refinement of splitting is to add the negation of the first clause part $\Gamma_1 \rightarrow \Delta_1$ to the second part, labelled with the second part (or the other way round). For example, applying this refined version of splitting to the clause $C = m \; : \rightarrow A, B$ yields $m.C \; : \rightarrow A$, $m.\overline{C} \; : \rightarrow B$, and $m.\overline{C} \; : \; A \rightarrow$.

3.2.2 Implementation

In addition to the above mentioned refinements, implementing Splitting via labelled clauses seems to be less effort and as efficient as a standard implementation via a depth-first search and clause copy, as it is done, e.g., in Spass. For example, branch closing becomes a standard inference/reduction rule application of the calculus while in the context of the depth-search algorithm implemented in Spass it is a procedure running orthogonal to the standard saturation process, complicating the overall implementation.

3.3 Strategies

Labelled clauses can also be used to model well-known strategies and new strategies. For example, the set-of-support strategy forbids inferences between axiom clauses. This can be easily established by the labelled superposition framework using the label set $\{0, 1, \ominus\}$ and labelling all axiom clauses with 0, all conjecture clauses with 1. We instantiate $m_1 \leq m_2$ to always be true and use the following combination operations:

$\circ$	0	1	$\ominus$
0	$\ominus$	1	$\ominus$
1	1	1	$\ominus$
$\ominus$	$\ominus$	$\ominus$	$\ominus$

$\bullet$	0	1	$\ominus$
0	0	1	$\ominus$
1	1	1	$\ominus$
$\ominus$	$\ominus$	$\ominus$	$\ominus$

Thus any inferences between clauses for the axiom set are blocked, but reductions are allowed, keeping the 0 label.

3.4 Slicing

When trying to prove a conjecture in a given time limit, it can be a good heuristic to try different strategies for a fixed period of time. For example, strategies might differ in the selection strategy for negative literals (no selection, always select a negative literal, etc.) or the heuristic to pick the next clauses for inferences (lightest weight, heaviest weight, lowest age). Labelled superposition offers here a framework where these runs can be done simultaneously and benefit from each other.

We simply run the same conjecture simultaneously with different labels for the different strategies. Given n strategies, we give each one a number from $\{1, \ldots, n\}$. Initially we label all clauses with $\{1, \ldots, n\}$. We extend the labelled superposition calculus of Section 2 to also consider the strategy label, say m' for inference computation. Then the newly generated clause gets the label $m_1 \circ m_2 \circ m'$ and we have to check the condition $m_1 \circ m_2 \circ m' \neq \ominus$. Nevertheless, common clauses can be shared via the Join rule as in the case of multiple labelled conjectures.

Because the reduction (and simplification) combinator $\bullet$ is different from the inference combinator, it can be used to implement different strategies concerning reductions and simplifications between clauses generated by different strategies. This provides a high potential for synergy between the different proof attempts. Note that because different strategies may use different orderings, by considering the strategy label at inference rule level, the ordering used by the rule conditions can also be chosen by that label.

4 Applications in Software Verification

We present two applications that motivated our interest in labelled clauses. Both concern the application of *abstract interpretation* [4] in software verification.

Abstract interpretation provides a way to obtain information about the possible states that a program reaches, without actually running the program on specific inputs; instead, the program is "run abstractly", using descriptors that represent collections of many states. Each different family of descriptors (or "abstract domain") that is used can typically be characterized by means of a restricted class of formulas (presented in some normal form). Abstract interpretation provides a way to synthesize loop invariants (albeit only ones expressible in a given class of formulas), and hence sidesteps the need for a user to supply loop invariants.

4.1 Cartesian Abstraction

Predicate abstraction [5] abstracts a program into a Boolean program that conservatively simulates (i.e., overapproximates) all potential executions of the original program. Thus, every safety property that holds for the Boolean program is guaranteed to hold for the original program. A predicate p in predicate abstraction is defined by a closed formula ψ_p that evaluates to true or false for each concrete program state. An *abstract state* is a conjunction of (possibly negated) predicates; it represents the set of concrete states that satisfy each (possibly negated) predicate. A set of abstract states corresponds to a disjunction of the individual abstract states. The best approximation of a set of concrete states C is the strongest set of abstract states such that every $c \in C$ is satisfied (or, equivalently, the strongest Boolean combination of predicates that holds for every $c \in C$).

The operational semantics of a statement *st* in a program can be defined using a formula τ_{st} over the pre-state and the post-state. A post-state S' can be the result of a given pre-state S if τ_{st} holds for $S \uplus S'$. The predicate p' defined using formula ψ_p, but applied to the post-state, is called the *primed version* of p. The effect of *st* on an abstract pre-state A can be computed using a theorem prover by checking which combinations of primed-predicate formulas are satisfiable when conjoined with $A \wedge \tau_{st}$.

A less-costly, but less-precise, variant of predicate abstraction is called Cartesian Abstraction [1]. Instead of using a Boolean combination of predicates to represent a set of states C, this uses a single conjunction of (possibly negated) predicates. Thus, a predicate p should appear in abstract state A_C (i) positively when ψ_p holds for every $c \in C$; (ii) negatively when ψ_p does not hold for any $c \in C$; and (iii) otherwise should not appear.

With Cartesian Abstraction, it is possible to compute the effect of a statement via a set of validity queries to a theorem prover. In each query, the axiom set contains the pre-state A, the transformer τ_{st}, and any background theory we have. The conjecture is either a primed predicate or a negated primed predicate. This makes computing transformers for Cartesian Abstraction a good target for our method. All the primed predicates and their negations can be added as conjectures to the same axiom set, and the effect of the statement can be computed using a single call to the theorem prover; any shared structure between the different predicates or similarities among the proofs of the different conjectures will be exploited by the theorem prover.

4.2 Shape Analysis and Canonical Abstraction

Shape analysis aims to discover information (in a conservative way) about the possible "shapes" of heap-allocated data structures to which a program's pointer variables can point. The method of Canonical Abstraction [17] provides a family of abstract domains for capturing certain classes of memory configurations. Canonical Abstraction uses first-order (FO) logic with transitive closure (TC), in which individuals are heap-cells and predicate symbols of the vocabulary are interpreted via relations on heap-cells. A background theory is used to formalize the properties of these predicates (e.g. a pointer variable can point to at most one location). TVLA [10] is a parametric framework for building analyzers for canonical-abstraction-based abstract domains.

In Canonical Abstraction, the heap is partitioned according to conjunctions of unary predicates. Only universal information about these partitions is saved; that is, if formulas such as $node_1(v)$, $node_2(v)$, etc. are conjunctions of unary predicates that characterize the various subsets that partition the heap-cells, and p is a binary predicate symbol from the vocabulary, we can only record information of the form

$$\forall v_1, v_2 . node_i(v_1) \wedge node_j(v_2) \Rightarrow [\neg] p(v_1, v_2). \tag{1}$$

To control the abstraction, the designer of an abstraction (or an automatic tool for discovering abstractions [12]) can define auxiliary unary predicates with FO+TC formulas that are added to the background theory.

Similar to Sect. 4.1, the operational semantics of a statement is defined using a formula over the pre-state and the post-state; to determine the effect of a statement, we need to determine which formulas in form Eq. (1) hold for the primed predicates.[6] In this setting, it is again possible to compute the effect of a statement via a set of validity queries, which may be answered *en masse* by a theorem prover using our method.[7] This allows the theorem prover to exploit similarities among the proofs of the different conjectures.

4.3 Experiments

We have integrated the prototype implementation of labelled clauses for multiple conjectures in SPASS into TVLA, i.e., the validity queries required to compute the result of applying a transformer are performed by SPASS. In this section, we present a comparison between the performance of the improved version and running the original SPASS version 2.2 on each conjecture sequentially. The results presented here are from the queries generated by the analysis of several heap-manipulating programs, including reversal of a singly-linked list, insert sort of

[6] See [9] for methods that can be used to handle formulas that involve the TC operator.

[7] One technical point: case splits are performed externally to the theorem prover (and before the theorem prover is called), by the "focus" operation of [17], hence we do not have to be concerned about disjunctions of the different conjectures.

a singly-linked list, insertion into a doubly-linked list, and the mark phase of a simple mark-and-sweep garbage collector.

To make the comparison fair, we removed three types of queries: (i) any conjecture that is not valid, (ii) queries in which the common axioms are inconsistent, and (iii) queries in which there is only one valid conjecture. When also considering invalid conjectures, the advantage of the labelled-clauses method is even more apparent because the sequential approach has to wait for a timeout for each invalid conjecture. Similarly, when considering queries with an inconsistent axiom set, the improved version can detect that the original axioms are inconsistent, while the sequential approach will have to prove it again and again for each conjecture. Finally, for a single conjecture the labelled-clauses version behaves almost exactly the same as the original version, both in terms of the clauses generated and the time required.

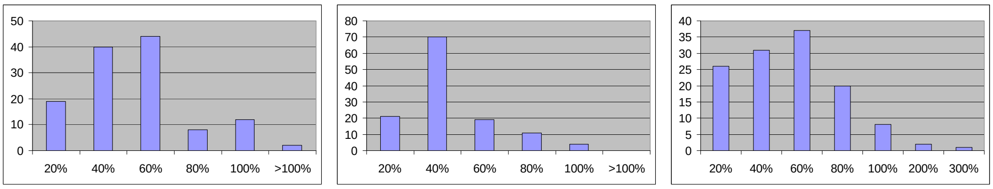
The test set includes 125 queries, with a minimum of 2 conjectures, a maximum of 32 conjectures, and an average of 9.3 conjectures per query. To improve the statistical significance of the results, we took to the test set only queries in which the total time taken by the sequential prover was less than one second. We compare the provers according to three criteria: the number of derived clauses, the number of kept clauses (i.e., which were not forward subsumed), and the time. Table 1 has a comparison of the maximum, minimum, and average values for these three criteria for the two provers.

Table 1. Comparison between the labelled-clauses Spass and running the original Spass sequentially.

Criteria	Sequential			Labelled Clauses		
	Min.	Max.	Avg.	Min.	Max.	Avg.
Derived	3569	408837	83657.1	1408	140679	33088.5
Kept	3882	70463	22672.1	1366	26238	7357.2
Time (sec)	1.0	74.0	8.1	0.2	29.0	3.8

Fig. 1 presents a histogram for each criterion showing the values of the labelled-clauses prover as a percentage of the appropriate value for the sequential prover. In most cases we have a least 2-fold improvement in all criteria. The number of kept clauses never increases. The number of derived clauses increases in one case, but only by 6%. There are three examples in which the sequential version was faster; we believe that this is a result of the quality of the current prototype implementation, because the number of derived and kept clauses in these cases is lower in the labelled-clauses version.

Because our conjectures are based on formulas in the normal form shown in Eq. (1), the negated conjectures are sets of ground unit clauses. This makes the technique of reusing Skolem constants between conjectures very lucrative. We have checked the improvements brought about by the Join rule and by reusing the Skolem constants. Introducing the Join rule to a vanilla implementation of the labelled-clauses approach makes it run 1.7 times faster. Reusing Skolem constants added, on average, an extra speedup of 1.7, which produced about a 3-fold speedup in total. When considering only cases in which 10 or more conjectures are used, the average speedup is 7-fold.



(a) (b) (c)

Fig. 1. Histograms for each criterion showing the percentage of the values for the labelled-clauses version compared with the sequential version: (a) derived clauses, (b) kept clauses, (c) running time.

5 Conclusion

We suggested the methodology of labelled clauses for the study and implementation of superposition-based classical first-order-logic calculi. We believe that using labelled clauses as an extension of ordinary ones will become as fruitful for advances in automated theorem proving as it is in the context of non-classical logics.

Our work on labelled clauses offers new possibilities to study saturation and tableau-like calculi in a common framework. This might also be fruitful for advances in theorem-proving search strategies, as, e.g., suggested by Bonacina [3].

We have shown how to instantiate the general framework of labelled clauses for several interesting cases, including clause splitting and slicing. For the case of proving multiple conjectures simultaneously, we have also implemented the calculus as an extension of the Spass theorem prover and report on convincing experimental results in the context of software verification.

We believe that there are other techniques in the world of theorem proving that can benefit from the idea of labelled clauses. Investigating them is the subject of future work. In particular, the combination of theorem proving techniques represented in the labelling methodology simplifies to the combination of the labelling disciplines.

References

1. T. Ball, A. Podelski, and S.K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *TACAS*, pages 268–283, 2001.
2. David Basin, Marcello D'Agostino, Dov M. Gabbay, Seán Matthews, and Luca Viganò, editors. *Labelled Deduction*, volume 17 of *Applied Logic Series*. Kluwer, 2000.

3. Maria Paola Bonacina. Towards a unified model of search in theorem-proving: subgoal-reduction strategies. *J. Symb. Comput.*, 39(2):209–255, 2005.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, pages 238–252, 1977.
5. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
6. C. Green. Theorem-proving by resolution as a basis for question-answering systems. In *Machine Intelligence 4*, pages 183–205, 1969.
7. Reiner Hähnle. Tableaux and related methods. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 6, pages 103–177. Elsevier, 2001.
8. S. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In *CAV*, pages 24–38, 2005.
9. T. Lev-Ami, N. Immerman, T. W. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE*, pages 99–115, 2005.
10. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.
11. T. Lev-Ami, C. Weidenbach, T. Reps, and M. Sagiv. Experimental version of SPASS for multiple conjectures. Available at "http://www.cs.tau.ac.il/˜tla/SPASS", 2007.
12. A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *CAV*, pages 519–533, 2005.
13. C.G. Nelson and D.C. Oppen. A simplifier based on efficient decision algorithms. In *POPL*, pages 141–150, 1978.
14. R. Nieuwenhuis and A. Oliveras. Decision procedures for SAT, SAT modulo theories and beyond. the barcelogictools. In *LPAR*, pages 23–46, 2005.
15. A. Riazanov and A. Voronkov. Splitting without backtracking. In *IJCAI*, pages 611–617, 2001.
16. Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.
17. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, pages 217–298, 2002.
18. Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
19. A. Voronkov. Personal communication. 2007.
20. Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 27, pages 1965–2012. Elsevier, 2001.
21. Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topic. SPASS version 2.0. In *CADE*, pages 275–279, 2002.
22. J. Whittemore, J. Kim, and K.A. Sakallah. SATIRE: A new incremental satisfiability engine. In *DAC*, pages 542–545, 2001.
23. A. Wolf. Strategy selection for automated theorem proving. In *AIMSA*, pages 452–465, 1998.
24. L. Wos, G.A. Robinson, and D.F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4):536–541, 1965.
25. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, pages 530–545, 2004.