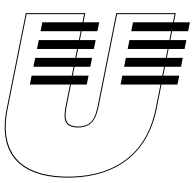

UNIVERSITÄT
KOBLENZ · LANDAU

The Model Evolution Calculus

Peter Baumgartner, Cesare Tinelli

1/2003

Fachberichte
INFORMATIK

Universität Koblenz-Landau
Institut für Informatik, Rheinau 1, D-56075 Koblenz

E-mail: researchreports@infko.uni-koblenz.de,
WWW: http://www.uni-koblenz.de/fb4/

The Model Evolution Calculus

Peter Baumgartner
Institut für Informatik
Universität Koblenz-Landau
peter@uni-koblenz.de

Cesare Tinelli
Department of Computer Science
The University of Iowa
tinelli@cs.uiowa.edu

February 20, 2003

Abstract

The DPLL procedure is the basis of some of the most successful propositional satisfiability solvers to date. Although originally devised as a proof-procedure for first-order logic, it has been used almost exclusively for propositional logic so far because of its highly inefficient treatment of quantifiers, based on instantiation into ground formulas. The recent FDPLL calculus by Baumgartner was the first successful attempt to lift the procedure to the first-order level without resorting to ground instantiations. FDPLL lifts to the first-order case the core of the DPLL procedure, the splitting rule, but ignores other aspects of the procedure that, although not necessary for completeness, are crucial for its effectiveness in practice. In this paper, we present a new calculus loosely based on FDPLL that lifts these aspects as well. In addition to being a more faithful litfing of the DPLL procedure, the new calculus contains a more systematic treatment of *universal literals*, one of FDPLL's optimizations, and so has the potential of leading to much faster implementations.

Keywords: DPLL procedure, first-order logic, sequent calculi, model generation.

Contents

1 Introduction

In propositional satisfiability the DPLL procedure, named after its authors: Davis, Putnam, Logemann, and Loveland [DP60, DLL62], is the dominant method for building (complete) SAT solvers. Its popularity is due to its simplicity, its polynomial space requirements, and the fact that, as a search procedure, it is amenable to powerful but also relatively inexpensive heuristics for reducing the search space. Thanks to these heuristics and to very careful engineering, the best SAT solvers today can successfully attack real-world problems with hundreds of thousands of variables and of clauses [MMZ+01, GN02]. These solvers are so powerful that many developers of automated reasoning-based tools are starting to use them as back-ends to solve *first-order* satisfiability problems, albeit often in an incomplete way, by means of ingenious domain specific translations into propositional logic [JNR01, Jac00, SSB02].

Interestingly, the DPLL procedure was actually devised in origin as a proof-procedure for first-order logic. Its treatment of quantifiers is highly inefficient, however, because it is based on enumerating all possible ground instances of an input formula's clause form, and checking the propositional satisfiability of each of these ground instances one at a time. Because of its primitive treatment of quantifiers the DPLL procedure, which predates Robinson's resolution calculus by a few years, was quickly overshadowed by resolution as the method of choice for automated first-order reasoning, and its use has been confined to propositional satisfiability ever since.[1]

Given the great success of DPLL-based SAT solvers today, two natural research questions arise. One is whether the DPLL procedure can be properly lifted to the first-order level—in the sense first-order resolution lifts propositional resolution, say. The other is whether those powerful search heuristics that make DPLL so effective at the propositional level can be successfully adapted to the first-order case. We answer the first of these two questions affirmatively in this paper, providing a complete lifting of the DPLL procedure to first-order clausal logic by means of a new sequent calculus, the *Model Evolution* calculus, or $\mathcal{ME}$ for short. We believe that the $\mathcal{ME}$ calculus can be used to answer the second question affirmatively as well, although that will be the subject of our future work.

The recent FDPLL calculus by Baumgartner [Bau00] was the first successful attempt to lift the DPLL procedure to the first-order level without resorting to ground instantiations. FDPLL lifts to the first-order case the core of the DPLL procedure, the splitting rule, but ignores another major aspect, *unit propagation* [ZS96], that although not necessary for its completeness is absolutely crucial to its effectiveness in practice. The calculus described in this paper lifts this aspect as well. While the $\mathcal{ME}$ calculus borrows many fundamental ideas from FDPLL and generalizes it, it is not an extension of FDPLL proper but of DPLL [Tin02], a simple

[1] But see Section 5 for a brief overview of first-order reasoning systems that use the procedure to help them focus their search.

sequent calculus for propositional logic modeling the main features of the DPLL procedure. As we will see, the Model Evolution calculus is a direct lifting of DPLL in the sense that it consists of appropriate first-order versions of DPLL's rules, plus two additional rules specific to the first-order case.

A very useful feature of the DPLL procedure—and of most propositional proof procedures for that matter—is that it is able to provide a (Herbrand) model of the input formula whenever that formula is satisfiable. The procedure, and by extension the DPLL calculus, generates this model incrementally as it goes. The Model Evolution calculus can be seen as lifting this model generation process at the first-order level. We could say that the purpose of the Model Evolution calculus is, like the DPLL calculus, to construct a Herbrand model of a given set Φ of clauses, if any such model exists. As in DPLL, this model is built incrementally during a derivation.

At any step of a derivation the calculus maintains a *context* Λ, that is, a finite set of (possibly non-ground) literals. The context Λ is a finite—and compact—representation of a Herbrand *preinterpretation*.[2] The preinterpretation I_Λ induced by Λ serves in turn as a candidate model for Φ. This preinterpretation might not be a model of Φ because it is not an interpretation to start with, or because it does not satisfy some clauses in Φ. The purpose of the main rules of the calculus is to detect each of these situations and either *repair* I_Λ, by modifying Λ, so that it becomes an interpretation that satisfies all clauses of Φ, or recognize that I_Λ is unrepairable and fail. In addition to these rules, the calculus contains a number simplification rules whose purpose is, again like in DPLL, to simplify the clause set and, as a consequence, to speed up the computation.

We call our calculus *Model Evolution* calculus because it starts with a default candidate model, one that satisfies no positive literals, and "evolves it" as needed until it becomes an actual model of the input clause set Φ, or until it is clear that Φ has no models at all. The DPLL calculus does exactly the same thing, but for ground formulas only. The Model Evolution calculus simply extends this behavior to non-ground formulas as well. An important by-product of this model evolution process is that terminating derivations of a satisfiable clause set Φ, when they exist, produce a context whose induced interpretation is indeed a model of Φ. This makes the calculus well-suited for all applications in which it is important to also provide counter-examples to invalid statements, as opposed to simply proving their invalidity.

The Model Evolution calculus is refutationally sound and complete: an input clause set Φ is unsatisfiable iff the calculus (finitely) fails to find a model for Φ. The calculus is obviously non-terminating for arbitrary, satisfiable input sets. With some of these clause sets, the calculus might go on repairing their candidate model forever, without ever turning it into an actual model. The calculus is however

[2] This is a set of ground literals some of whose subsets are Herbrand interpretations in the standard sense. See later for a formal definition.

terminating for the class of ground clauses (of course), and for the class of clauses resulting from the translation of conjunctions of Bernays-Schönfinkel formulas into clause form.[3] The termination for ground clause sets is a direct consequence of the fact that with such inputs the Model Evolution calculus reduces to the DPLL calculus, as we will show. The reasons for termination for Bernays-Schönfinkel formulas. are similar to those given in [Bau00] for FDPLL.

As mentioned, the Model Evolution calculus is already a significant improvement over FDPLL because it is a more faithful litfing of the DPLL procedure, having additional rules for simplifying the current clause set and the current context. Another advantage over FPDLL is that it contains a more systematic and general treatment of *universal literals*, one of FDPLL's optimizations. As we will see, adding universal literals to a context imposes stronger restrictions on future modification of that context. This has the consequence of greatly reducing the non-determinism in the calculus, and hence the potential of leading to much faster implementations.

The paper is organized as follows. After some formal preliminaries, given below, we briefly describe in Section 2 the DPLL procedure, and define the DPLL calculus, a declarative version of the procedure. We then define and discuss the Model Evolution calculus, in Section 3, showing how it extends DPLL. We prove the calculus' correctness in Section 4. Then we show in Section 5 how the calculus compares to other calculi in related work. We conclude the paper in Section 6 with directions for further research. The more technical results needed in Section 4 are proved in detail in the appendix.

1.1 Formal Preliminaries

In this paper, we use two disjoint, infinite sets of variables: a set X of *universal* variables, which we will refer to just as variables, and another set V, which we will always refer to as *parameters*. The reason for having two types of variables will be explained later. We will use, possibly with subscripts, u, v to denote elements of V, x, y to denote elements of X, and w to denote elements of $V \cup X$. We fix a signature Σ throughout the paper. We denote by Σ^{sko} the expansion of Σ obtained by adding to Σ an infinite number of (Skolem) constants not already in Σ. By Σ-term (Σ^{sko}-term) we mean a term of signature Σ (Σ^{sko}) over $X \cup U$. In the following, we will simply say "term" to mean a Σ^{sko}-term. If t is a term we denote by $\mathcal{V}ar(t)$ the set of t's variables and by $\mathcal{P}ar(t)$ the set of t's parameters. A term t is *ground* iff $\mathcal{V}ar(t) = \mathcal{P}ar(t) = \emptyset$. Two terms are *variable-disjoint (parameter-disjoint)* iff they have no variables (parameters) in common. They are *disjoint* iff they are both variable- and parameter-disjoint. We extend the above notation and terminology to literals and clauses in the obvious way.

We adopt the usual notion of substitution over Σ^{sko}-expressions or sets thereof. We also use the standard notion of unifier and of most general unifier. We will

[3] Such clauses contain no function symbols, but no other restrictions apply.

denote by $\{w_1 \mapsto t_1, \ \ldots, \ w_n \mapsto t_n\}$ the substitution σ such that $w_i\sigma = t_i$ for all $i = 1, \ldots, n$ and $w\sigma = w$ for all $w \in X \cup V \setminus \{w_1, \ldots, w_n\}$. Also, we will denote by $\mathcal{D}om(\sigma)$ the set $\{w_1, \ldots, w_n\}$ and by $\mathcal{R}an(\sigma)$ the set $\{w_1\sigma, \ldots, w_n\sigma\}$.

If σ is a substitution and W a subset of $X \cup V$, the restriction of σ to W, denoted by $\sigma_{|W}$ is the substitution that maps every $w \in W$ to $w\sigma$ and every $w \in (V \cup X) \setminus W$ to itself. A substitution ρ is a *renaming on* $W \subseteq (V \cup X)$ iff $\rho_{|W}$ is a bijection of W onto W. For instance $\rho := \{x \mapsto u, v \mapsto u, u \mapsto v\}$ is a renaming on V. Note however that ρ is not a renaming on $V \cup X$ as it maps both x and v to u. We call a substitution simply a *renaming* if it is a renaming on $V \cup X$. We call a substitution σ *parameter-preserving*, or *p-preserving* for short, if it is a renaming on V. Similarly, we call σ *variable-preserving* if it is a renaming on X. Note that a renaming is parameter-preserving iff it is variable-preserving. For example, the renaming $\{x \mapsto y, y \mapsto x, u \mapsto v, v \mapsto u\}$ is both variable- and parameter-preserving, wheres the renaming $\{x \mapsto v, v \mapsto x\}$ is neither variable-preserving nor parameter-preserving.

If s and t are two terms, we say that s *is more general than* t, and write $s \gtrsim t$, iff there is a substitution σ such that $s\sigma = t$. We say that s is *a variant of* t, and write $s \approx t$, iff $s \gtrsim t$ and $t \gtrsim s$ or, equivalently, iff there is a renaming ρ such that $s\rho = t$. We write $s \gtrsim\!\!\!\!\!\!\not\ \ t$ if $s \gtrsim t$ but $s \not\approx t$ We say that s *is parameter-preserving more general than* t, and write $s \geq t$, iff there is a parameter-preserving substitution σ such that $s\sigma = t$. When $s \geq t$ we will also say that t is *a p-instance of* s. Since the empty substitution is parameter-preserving and the composition of two parameter-preserving substitutions is also parameter preserving, it is immediate that the relation $\geq$ is, like $\gtrsim$, both reflexive and transitive. We say that s is *a parameter-preserving variant, or p-variant, of* t, and write $s \simeq t$, iff $s \geq t$ and $t \geq s$; equivalently, iff there is a parameter-preserving renaming ρ such that $s\rho = t$.[4] We write $s \gneq t$ if $s \geq t$ but $s \not\simeq t$. Note that both $\simeq$ and $\approx$ are equivalence relations.

All of the above about substitutions is extended from terms to *literals*, that is, atomic formulas or negated atomic formulas, in the obvious way. We denote literals in general by the letters K, L. We denote by $\overline{L}$ the complement of a literal L. As usual, a *clause* is a disjunction $L_1 \vee \cdots L_n$ of zero or more literals. We denote clauses by the letters C and D and the empty clause by $\Box$. We will write $L \vee C$ to denote a clause obtained as the disjunction of a (possibly empty) clause C and a literal L. When convenient, with a slightly abose of notation, we will treat a clauses as the set of its literals.

A *Skolemizing substitution* is a substitution θ with $\mathcal{D}om(\theta) \subseteq X$ that replaces each variable in $\mathcal{D}om(\theta)$ by a fresh Skolem constant and every remaining element of $X \cup V$ by itself. A *Skolemizing substitution for a literal* L *(clause* C*)* is a Skolemizing substitution θ with $\mathcal{D}om(\theta) = \mathcal{V}ar(L)$ ($\mathcal{D}om(\theta) = \mathcal{V}ar(C)$). We write L^{sko} (C^{sko}) to denote the result of applying to L (C) some Skolemizing substitution

[4] Note that we could have just as well defined s to be a *variable-preserving variant* of t when $s\rho = t$ for some parameter-preserving renaming ρ. The reason is that, as observed above, parameter preserving renamings are also variable-preserving, and vice versa.

for L (C).

We call a *(Herbrand) preinterpretation* any set I of ground Σ^{sko}-literals that contains L or $\overline{L}$ or both for every ground Σ^{sko}-literal L. A *(Herbrand) interpretation* is a Herbrand preinterpretation that contains a literal L if and only if it does not contain its complement $\overline{L}$. Satisfiability of literals and clauses in a Herbrand interpretation I is defined as usual. The interpretation I satisfies (or is a model of) a ground literal L, written $I \models L$, iff $L \in I$; I satisfies a ground clause C, iff $I \models L$ for some L in C; I satisfies a clause C, iff $I \models C'$ for all ground instances C' of C; I satisfies a clause set Φ, iff $I \models C$ for all $C \in \Phi$ in C. The interpretation I *falsifies* a literal L (a clause C) if it does not satisfy L (C). Sometimes we will also say that a clause C is valid in I to mean that $I \models C$.

2 The DPLL Calculus

The DPLL procedure can be used to decide the satisfiability of ground (or propositional) formulas in conjunctive normal form, or, more precisely but equivalently, the satisfiability of finite sets of ground clauses. The three essential operations of the procedure are unit resolution with backward subsumption, unit subsumption, and recursive reduction to smaller problems. The procedure can be roughly described as follows.[5]

Given an input clause set Φ, whose satisfiability is to be checked, apply *unit propagation* to it, that is, close Φ under unit resolution with backward subsumption, and eliminate in the process (a) all non-unit clauses subsumed by a unit clause in the set and (b) all unit clauses whose (only) atom occurs only once in the set. If the closure Φ^* of Φ contains the empty clause, then fail. If Φ^* is the empty set, then succeed. Otherwise, choose an arbitrary literal L from Φ^* and check recursively, and separately, the satisfiability of $\Phi^* \cup \{L\}$ and of $\Phi^* \cup \{\overline{L}\}$, succeeding if and only if one of the two subsets is satisfiable.

The essence of this procedure can be captured by a sequent calculus, the DPLL calculus, first described [Tin02], consisting of the derivation rules below. The calculus manipulates sequents of the form $\Lambda \vdash \Phi$, where Λ, the *context* of the sequent, is a finite set of ground literals and Φ is a finite (multi)set of ground clauses.[6]

$$\textsf{Split}\quad \frac{\Lambda \vdash \Phi,\ L \vee C}{\Lambda, L \vdash \Phi,\ L \vee C \qquad \Lambda, \overline{L} \vdash \Phi,\ L \vee C}\quad \text{if}\ \begin{cases} C \neq \Box, \\ L \notin \Lambda, \\ \overline{L} \notin \Lambda \end{cases}$$

[5] See the original papers [DP60, DLL62], among others, for a more complete description.
[6] As customary, we write $\Lambda, L \vdash \Phi, C$, say, to denote the sequent $\Lambda \cup \{L\} \vdash \Phi \cup \{C\}$.

$$\textsf{Assert}\quad \frac{\Lambda \vdash \Phi,\ L}{\Lambda, L \vdash \Phi,\ L}\ \text{if}\ \begin{cases} L \notin \Lambda, \\ \overline{L} \notin \Lambda \end{cases} \qquad \textsf{Subsume}\quad \frac{\Lambda,\ L \vdash \Phi,\ L \vee C}{\Lambda,\ L \vdash \Phi}$$

$$\textsf{Empty}\quad \frac{\Lambda \vdash \Phi, \square}{\Lambda \vdash \square}\ \text{if}\ \Phi \neq \emptyset \qquad \textsf{Resolve}\quad \frac{\Lambda, \overline{L} \vdash \Phi,\ L \vee C}{\Lambda, \overline{L} \vdash \Phi,\ C}$$

The intended goal of the calculus is to derive a sequent of the form $\Lambda \vdash \emptyset$ from an initial sequent $\emptyset \vdash \Phi_0$, where Φ_0 is a clause set to be checked for satisfiability. If that is possible, then Φ_0 is satisfiable; otherwise, Φ_0 is unsatisfiable. Informally, the purpose of the context Λ is to store incrementally a set of *asserted literals*, i.e., a set of literals in Φ_0 that must or can be true for Φ_0 to be satisfiable. When $\Lambda \vdash \emptyset$ is derivable from $\emptyset \vdash \Phi_0$, the context Λ is indeed a witness of Φ_0's satisfiability as it describes a (Herbrand) model of Φ_0: one that satisfies an atom p in Φ_0 iff p occurs positively in Λ.

The context is grown by the Assert and the Split rules. The Assert rule models the fact that every literal occurring as a unit clause in the the current clause set must be satisfied for the whole clause set to be satisfied. The Split rule corresponds to the decomposition in smaller subproblems of the DPLL procedure. This rule is the only *don't-know* non-deterministic rule of the calculus. It is used to guess the truth value of an *undetermined* literal L in the clause set Φ of the current sequent $\Lambda \vdash \Phi$, where by undetermined we mean such that neither L nor $\overline{L}$ is in the context Λ. The guess allows the continuation of the derivation with either the sequent $\Lambda,\ L \vdash \Phi$ or with the sequent $\Lambda,\ \overline{L} \vdash \Phi$.

The other two main operations of the DPLL procedure, unit resolution with backward subsumption and unit subsumption, are modeled respectively by the Resolve and the Subsume rule. The Resolve rule removes from a clause all literals whose complement has been asserted—which corresponds to generating the simplified clause by unit resolution and then discarding the old clause by backward subsumption. The Subsume rule removes all clauses that contain an asserted literal—because all of these clauses will be satisfied in any model in which the asserted literal is true.

The DPLL calculus is easily proven sound, complete and terminating. It is not hard to show that the calculus maintains its completeness even if one constrains the Split rule to split only on positive literals.[7] In other words, there is no loss of completeness if Split is replaced by the rule:

$$\textsf{Split'}\quad \frac{\Lambda \vdash \Phi,\ L \vee C}{\Lambda, L \vdash \Phi,\ L \vee C \qquad \Lambda, \overline{L} \vdash \Phi,\ L \vee C}\ \text{if}\ \begin{cases} L \text{ is positive}, \\ C \neq \square, \\ L \notin \Lambda, \\ \overline{L} \notin \Lambda \end{cases}$$

[7] This fact is known in the SAT literature and is used as an optimization in a number of DPLL-based SAT solvers.

Another change that does not alter the calculus in any fundamental way—and is actually more faithful to the way the DPLL procedure is usually implemented—is the replacement of the Empty rule by the following, more powerful rule:

$$\textsf{Close}\quad \frac{\Lambda \vdash \Phi,\ L_1 \vee \cdots \vee L_n}{\Lambda \vdash \square}\quad \text{if } \begin{cases} \Phi \neq \emptyset \text{ or } n > 0, \\ \overline{L_1}, \ldots, \overline{L_n} \in \Lambda \end{cases}$$

Note that Close reduces to the Empty rule given earlier if $L_1 \vee \cdots \vee L_n$ has no literals (if $n = 0$). The reason Close does not really change the calculus is that every application of Close can be simulated by n applications of Resolve followed by one application of Empty. Interestingly, with Close the Resolve rule becomes superfluous for completeness.

We mention the Split' and Close rules here because they will facilitate our comparison between the Model Evoution calculus and DPLL.

3 The Model Evolution Calculus

The Model Evolution calculus is a direct lifting of the DPLL calculus to the first-order level. The lifting is achieved with a suitable first-order version of the rules Split', Assert, Subsume, Resolve and Close of DPLL, plus the addition of two extra rules, Commit and Compact, specific to the first-order case. Of these two extra rules, Compact is just a simplification rule like Resolve and Subsume, while Commit is analogous to a rule with the same name in FDPLL.

Similarly to DPLL, the derivation rules of the Model Evolution calculus apply to and produce sequents of the form $\Lambda \vdash \Phi$. This time, however, Λ is finite set of literals possibly with variables or with parameters, called again a context, and Φ is a set of clauses possibly with variables.

As mentioned in the introduction, the context Λ in a sequent $\Lambda \vdash \Phi$ determines a (pre)interpretation I_Λ which is meant to be a model of Φ. The purpose of the main rules of the calculus is to recognize when I_Λ is not a model of Φ—either because it is not even an interpretation or because it falsifies a clause in Φ—and *repair* it so that it can become one. The repairs are both localized and incremental, and based on the computation of most general unifiers. The progressive repair process or *evolution* of the candidate model starts with a default interpretation and continues until an actual model is found or no further repairs are possible. The calculus is non-deterministic because in some cases the current interpretation can be repaired in two alternative ways, neither of which can be ruled out a priori. With an initial sequent $\Lambda_0 \vdash \Phi_0$ then, this gives rise to a search space of possible evolution sequences for I_{Λ_0}, the initial candidate model for Φ_0.

We will show that when Φ_0 is unsatisfiable and Λ_0 is just $\{\neg v\}$ all these alternative sequences are finitely failed—making the calculus complete. We will also show that, conversely, if all evolution sequences for $I_{\{\neg v\}}$ are finitely failed, then Φ_0 is guaranteed to be unsatisfiable—making the calculus sound as well. In the process,

we will also show that non-failed finite sequences that cannot be grown any longer end with a context whose candidate model is indeed a model of Φ_0.

3.1 Contexts and Interpretations

The defining aspect of the calculus, modeled after FDPLL, is the way contexts are extended to the first-order case, and the rôle they play in driving the derivation and the model generation process. Therefore, we start our description of the calculus with them.

Definition 3.1 (Context) *A* context *is a set of the form* $\{\neg v\} \cup S$ *where* $v \in V$ *and* S *is a finite set of literals each of which is either parameter-free or variable-free.*

A context is then a set of of literals that do not have both variables and parameters in them, plus a *pseudo-literal* of the form $\neg v$. The role of $\neg v$ will become clear later.

Where L is a literal and Λ a context, we will write $L \in_{\approx} \Lambda$ if L is a variant of a literal in Λ, will write $L \in_{\simeq} \Lambda$ if L is a p-variant of a literal in Λ, and will write $L \in_{\geq} \Lambda$ if L is a p-instance of a literal in Λ.

We will work only with *non-contradictory* contexts in this paper.

Definition 3.2 (Contradictory) *A literal* L *is* contradictory with *a context* Λ *iff* $L\sigma = \overline{K}\sigma$ *for some* $K \in_{\simeq} \Lambda$ *and some parameter-preserving substitution* σ*. A context* Λ *is* contradictory *iff it contains a literal that is contradictory with* Λ*.*

Example 3.3 *Let* $\Lambda := \{\neg v,\ p(x_1, g(y_1)),\ \neg q(v_1)\}$*. Then* $\neg p(h(x), u)$*,* $\neg p(v, u)$*, and* $q(y)$ *are all contradictory with* Λ*. However,* $q(f(v))$ *and* $r(x)$*, say, are not. (Recall that* x, x_1, y_1 *are variables while* v, v_1, u *are parameters.)*

A non-contradictory context induces a (unique) preinterpretation by means of the next two notions.

Definition 3.4 (Most Specific Generalization) *Let* L *be a literal and* Λ *a context. A literal* K *is* a most specific generalization (msg) of L in Λ *iff* $K \gtrsim L$ *and there is no* $K' \in \Lambda$ *such that* $K \gtrsim K' \gtrsim L$*.*

Definition 3.5 (Productivity) *Let* L *be a literal,* C *a clause, and* Λ *a context. A literal* K produces L in Λ *iff*

1. *K is an msg of L in Λ, and*
2. *there is no* $K' \in_{\geq} \Lambda$ *such that* $K \gtrsim \overline{K'} \gtrsim L$*.*

The context Λ produces L *iff it contains a literal* K *that produces* L *in* Λ*. The context* Λ produces C *iff it produces one of* C*'s literals.*

Example 3.6 *Let* $\Lambda := \{\neg v,\ p(v_1, g(u_1)),\ \neg p(v_1, g(v_1)),\ q(h(u), v),\ \neg q(u, g(v))\}$. *The literals*

$$\neg p(v, u),\ p(v, g(u)),\ p(x, g(a)),\ \neg p(a, g(a))$$

are all produced by Λ. *On the other hand,*

$$p(v, u),\ \neg p(v, g(u)),\ \neg p(x, g(a)),\ p(a, g(a))$$

are not. Note though that both $q(h(u), g(v))$ *and* $\neg q(h(u), g(v))$ *are produced by* Λ.

It is not difficult to show that there are effective—and simple—unification-based procedures to test whether two literals are contradictory and whether a literal produces another in a given context.

Contexts and interpretations satisfy a number of general properties that are useful for our calculus. We present and discuss these properties next, deferring their proof to the appendix.

Definition 3.7 (Induced Preinterpretation) *Let* Λ *be a non-contradictory context. The* preinterpretation induced by Λ, *denoted by* I_Λ, *is the set of all ground* Σ^{sko}*-literals produced by* Λ.

Induced preinterpretation are indeed pre interpretations.

Proposition 3.8 *Let* Λ *a non-contradictory context. Then,* I_Λ *is a preinterpretation.*

Recalling that each literal in a context is either parameter-free or variable-free (or both), one way to understand how a non-contradictory context Λ induces a preinterpretation I_Λ is the following.

Let us say that a ground literal L in I_Λ is *true in* I_Λ if $L \in I_\Lambda$ and $\overline{L} \notin I_\Lambda$, is *false in* I_Λ if $\overline{L} \in I_\Lambda$ and $L \notin I_\Lambda$, and is *over-defined in* I_Λ otherwise. Where K is a literal, let us say that a literal L is *immediately below* K *in* Λ iff K is an msg of L or of $\overline{L}$.

Now, if Λ contains a parameter-free literal K, then all the ground instances of K will be true in I_Λ without exception.[8] If Λ contains a variable-free literal K, then all the ground instances of K will be true in I_Λ except for those whose complement is an instance of a literal in Λ that is parameter-free or is immediately below K in Λ.

It should be clear now that the purpose of the pseudo-literal $\neg v$ in a context Λ is to provide a *default* truth-value to those ground literals whose value is not determined by the rest the context. In fact, consider a ground literal L such that neither L nor $\overline{L}$ is produced by $\Lambda \setminus \{\neg v\}$. If L is negative, then it is true in I_Λ

[8] It is obvious that the instances are not false. They are not over-defined either because otherwise Λ would be contradictory, as we will show.

because it is produced by $\neg v$. If L is positive, then it is false in I_Λ because its complement is produced by $\neg v$.

The preinterpretation induced by a context may not be an interpretation in general. It becomes one exactly when Λ is *consistent*.

Definition 3.9 (Consistent) *A context Λ is* consistent wrt. a literal L *iff Λ does not produce both L and $\overline{L}$; Λ is just* consistent *iff it is consistent wrt. every literal L.*

Proposition 3.10 *Let Λ be a non-contradictory context. Then, I_Λ is an interpretation iff Λ is consistent.*

There is a simple necessary condition for the inconsistency of a non-contradictory context Λ, typified by Example 3.6 earlier: there exist two variable-free $K, L \in_\simeq \Lambda$ such that K and $\overline{L}$ unify and neither $K \gtrsim \overline{L}$ nor $\overline{L} \gtrsim K$. One of the derivation rules of the $\mathcal{ME}$ calculus uses this fact to recognize that the context Λ in the current sequent is inconsistent. Specifically, it looks for a literal $L \in \Lambda$ and a literal $K \in_\simeq \Lambda$ with K disjoint with L such that K and $\overline{L}$ have a most general unifier σ and neither $K \gtrsim \overline{L}$ nor $\overline{L} \gtrsim K$. Each pair of literals like L and K above is a source of inconsistency for Λ, which we call *a connection*. The calculus attempts to eliminate all connections (L, K) from the current context by adding $L\sigma$ or its complement to the context, provided that the addition does not result into a contradictory context.

As we mentioned, even if a context of a sequent $\Lambda \vdash \Phi$ is consistent, its induced intepretation may falsify a clause of Φ. This situation is detectable through the computation of *context unifiers*.

Definition 3.11 (Context Unifier) *Let Λ be a context and*

$$C = L_1 \vee \cdots \vee L_m \vee L_{m+1} \vee \cdots \vee L_n$$

a parameter-free clause, where $0 \leq m \leq n$. A substitution σ is a context unifier of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ *iff there are fresh variants $K_1, \ldots, K_n \in_\simeq \Lambda$ such that*

1. *σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,*
2. *for all $i = 1, \ldots, m$, $(\mathcal{P}ar(K_i))\sigma \subseteq V$,*
3. *for all $i = m+1, \ldots, n$, $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$.*

We say, in addition, that σ is productive *iff K_i produces $\overline{L_i}\sigma$ in Λ for all $i = 1, \ldots, n$.*

A context unifier σ of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ is admissible (for Split) *iff for all distinct $i, j = m+1, \ldots, n$, $L_i\sigma$ is parameter- or variable-free and $\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$.*

Note that each context unifier has a unique remainder. If σ is a context unifier of a clause C with remainder D we call each literal of D *a remainder literal* of σ.

Example 3.12 *Let* $\Lambda := \{\neg v,\ p(v_1, u_1),\ \neg p(x_1, g(x_1)),\ q(v_2, g(v_2))\}$ *and* $C_1 = r(x) \vee \neg p(x, y)$. *Then, the substitutions*

$$\begin{aligned} \sigma_1 &:= \{v \mapsto r(x),\ v_1 \mapsto x,\ u_1 \mapsto y\} \\ \sigma_2 &:= \{v \mapsto r(v_1),\ x \mapsto v_1,\ u_1 \mapsto y\} \end{aligned}$$

are both context unifiers of C_1 *against* Λ *with respective remainders* $r(x) \vee \neg p(x, y)$ *and* $\neg p(v_1, y)$. *While both* σ_1 *and* σ_2 *are productive, neither of them is admissible; the first because its remainder literals are not variable-disjoint, the second because its remainder literal contains both variables and parameters. By contrast,*

$$\sigma_3 := \{v \mapsto r(v_1),\ x \mapsto v_1,\ y \mapsto u_1\}$$

is a context unifier of C_1 *against* Λ, *this time with remainder* $\neg p(v_1, u_1)$, *that is both productive and admissible.*

Consider now the clause $C_2 = \neg p(x, y) \vee \neg q(x, y)$. *The substitution*

$$\sigma_4 := \{v_1 \mapsto v_2,\ u_1 \mapsto g(u_2),\ x \mapsto v_2,\ y \mapsto g(u_2)\}$$

is a context unifier of C_2 *against* Λ *with remainder* $\neg p(v_2, g(v_2))$. *This context unifier is admissible but it is not productive because the literal* $p(v_1, u_1)$ *of* Λ *chosen to unify with* $\overline{\neg p(x, y)}$ *does not produce* $\overline{\neg p(x, y)}\sigma_4 = p(v_2, g(v_2))$.

We point out for later comparisons with the DPLL calculus that when, in Definition 3.11, C is ground and $\neg v$ is the only non-ground literal of Λ, the substitution σ is a context unifier of C against Λ with remainder $(L_{m+1}\sigma \vee \cdots \vee L_n\sigma) = (L_{m+1} \vee \cdots \vee L_n)$ iff (i) for all $i = 1, \ldots, m$, $K_i = \overline{L_i}$ and (ii) for all $i = m+1, \ldots, n$, L_i is a positive literal occurring neither positively nor negatively in Λ while K_i is a p-variant of $\neg v$.

Admissible context unifiers are fundamental in the Model Evolution calculus. In fact, with a context Λ and a clause C, the existence of an admissible context unifier of C against Λ is a sign that I_Λ might not a model of C. This is because it is possible to compute an admissible context unifier of C against Λ whenever Λ is consistent and I_Λ falsifies C. The discovery by the calculus of an admissible context unifier σ of C against the current context Λ prompts a modification of Λ that involves adding a literal of $C\sigma$, with the goal of making C valid in the new I_Λ. This literal is chosen only among the remainder literals of σ, the reason being essentially that non-remainder literals can be ignored with no loss of completeness.

Note that while the existence of an admissible context unifier σ of C against Λ is necessary for the unsatisfiability of C in I_Λ, it is not sufficient unless σ is also productive. As a matter of fact, for completeness the calculus needs to add to the

context only remainder literals of admissible unifiers that are also productive. For greater flexibility, however, we allow it to add remainder literals of non-productive admissible unifiers as well. The reason is mostly practical and twofold: first, when implementing the calculus, insisting on computing only productive context unifiers can be considerably more expensive than computing context unifiers that are usually, although not always, productive; second, sometimes "repairing" candidate models with remainder literals from non-productive context unifiers can produce more constrained contexts, as illustrated in the example that follows.

Example 3.13 *Let* $\Lambda := \{\neg v,\ p(u),\ \neg q(g(y))\}$ *and* $C := p(x) \vee q(x)$*. The substitution*

$$\sigma \quad := \quad \{v \mapsto p(g(y)),\ x \mapsto g(y)\}$$

is a context unifier of C *against* Λ *with remainder* $p(g(y))$*, but it is not productive. As a matter of fact,* I_Λ *satisfies* C*, and so* $C\sigma$*, because* Λ *it produces every ground instance of* $p(x)$*. However, having the universal literal* $p(g(y))$ *in* Λ *along with* $p(u)$ *considerably constraints further repairs involving instances of* $p(u)$*, as we explain in Section 3.2.*

Productivity issues aside, it is important to observe at this point is that although context unifiers for a given clause C and context Λ are easily computable (they are just simultaneous most general unifiers), they are not unique and may not be admissible. Nevertheless, the calculus does not need to search for admissible context unifiers. For completeness purposes *any* admissible context unifier of C against Λ will do. Furthermore, and more important, admissible context unifiers are easily derived from non-admissible ones. In fact, let σ be a context unifier of C against Λ with remainder D. If σ has a remainder literal L that contains both variables and parameters or shares variables with another remainder literal, one can compose σ with a substitution that moves the variables of L to fresh parameters (and fixes everything else). It easy to see that a repeated application of this process leads to an admissible context unifier $\sigma\rho$ of C whose remainder is included in $D\rho$. For instance, the non-admissible contexts unifiers σ_1 and σ_2 in Example 3.12 can both be turned into the admissible σ_3 by this kind of process.

Now, while the choice of an admissible context unifier over another is irrelevant for completeness, some context unifiers are better than others for efficiency purposes. A context unifier with an empty remainder for instance is always preferrable to one with an non-empty remainder, because it lets the calculus stop the derivation right away, as we will see. In absense of those, context unifiers with parameter-free remainder literals are in general preferrable over context unifiers with variable-free remainder literals only. As we explain later, the addition of a parameter-free literal to a context imposes more constraints on later additions than the addition of a variable-free literal, leading in principle to shorter derivations.

3.2 Parameters vs. Variables

Before moving to describe the rules of the Model Evolution calculus, it is important to clarify the respective rôles that parameters and variables play in the calculus. We said that the calculus manipulates sequents of the form $\Lambda \vdash \Phi$, where Φ is a clause set and Λ is a context providing a candidate model for Φ.

Each derivation in the calculus starts with a sequent of the form $\neg v \vdash \Phi_0$, where Φ_0 contain only standard clauses, i.e. clauses with no parameters—but possibly with variables. Similarly, all sequents generated during a derivation have clause sets consisting of standard clauses only. Variables then can appear both in clause sets and in contexts. Parameters instead can appear only in contexts.

The rôle of variables within a clause is the usual one: they stand for all ground terms. In contrast, the rôle of variables and parameters within a context is to constrain, in different ways, how a candidate model can be repaired. To describe this it is helpful to ignore at first the issue of consistency. Let us assume then that the context of the current sequent $\Lambda \vdash \Phi$ is consistent—so that the candidate model induced by Λ is at least an interpretation.

When Λ is consistent, the current interpretation I_Λ needs repairing only if it falsifies a clause C in Φ. As we observed earlier, in that case there is an admissible context unifier σ of C against Λ. If every instance of C falsified by I_Λ is also an instance of $C\sigma$, to make C valid in I_Λ it is enough to modify Λ so that I_Λ satisfies $C\sigma$. One way to do that is to pick from $C\sigma$ a literal $L\sigma$ that is not contradictory with Λ, and *assert* it by adding it to Λ. The goal is to make the unit clause $L\sigma$ valid in I_Λ, which then makes $C\sigma$ valid as well. Now recall that, since σ is admissible, the added literal will not contain both parameters and variables.

If $L\sigma$ is a parameter-free literal, a *univesal literal* in FDPLL terminology, the assertion of $L\sigma$ cannot be retracted. No repairs that involve making instances of $L\sigma$ false will be allowed from that point on. Intuitively, this is justified by the fact that, because of the way we define context unifiers, when $L\sigma$ is parameter-free every model of $C\sigma$ that falsifies a ground instance of $C\sigma \setminus L\sigma$ satisfies all ground instances of $L\sigma$. That is the case, for instance, if $C\sigma$ has the form $P(f(x)) \vee Q(y, y) \vee R(y)$, where x and y are distinct variables, and $L\sigma$ is $P(f(x))$.

If $L\sigma$ is variable-free, the calculus is not sure that every model of $C\sigma$ that falsifies a ground instance of $C\sigma \setminus L\sigma$ satisfies all ground instances of $L\sigma$. Therefore, the assertion of $L\sigma$ is provisional; it can be (partially) retracted later. When adding $L\sigma$ to the context, the calculus is in essence making the assumption that there is a model of $C\sigma$ that satisfies all ground instances of $L\sigma$. This assumption, however, is just a working hypothesis, subject to be revised when evidence against it is found. This might happen if the calculus later adds to the current context Λ' a literal $\overline{L}\sigma'$, for some context unifier σ', in order to fix some other problem with the current interpretation, and it happens that $C\sigma'$ is an instance of $C\sigma$. After the addition, the new induced interpretation satisfies only those instances of $L\sigma$ that are not an instance of $\overline{L}\sigma'$. At that point, the clause $C\sigma$ may not be valid anymore because

its instance $C\sigma'$ may now be falsified. If that is case, the calculus will detect this and will try to make $C\sigma'$ valid (thereby restoring the validity of $C\sigma$) by looking in $C\sigma'$ for a literal other than $L\sigma'$ that can be added to the context, as explained earlier for $L\sigma$.

We point out that, since literals are never removed from a context, once a variable-free literal L has been asserted it can be retracted only partially—in the sense that only some, not all, of its ground instances can be made false in the current interpretation. Hence it would be more accurate to say that when the calculus adds L to the current context it is assuming that (i) definitely one ground instance of L is true and (ii) possibly all ground instances of L are true. Note that this is consistent with the fact that ground literals are both parameter- and variable-free: no matter how one looks at them, once they are asserted they cannot be retracted.

When Λ is not consistent the reasoning above still applies, but with the difference that I_Λ is only a preinterpretation—which intuitively means that I_Λ makes at least one literal L and its complement $\overline{L}$ both valid. In that case, I_Λ must be repaired so that it commits to either L or $\overline{L}$. As we mentioned earlier, a necessary condition for the existence of L is that L is a common instance of two variable-free literals K_1 and $\overline{K_2}$ such that $K_1, K_2 \in_\simeq \Lambda$. It is possible to repair I_Λ then by computing the most general common instance K of K_1 and $\overline{K_2}$ and adding either K or $\overline{K}$ to Λ—as long as the added literal is not contradictory with Λ.

3.3 Derivation Rules

Having explained the main concepts and ideas behind the calculus, we can now describe and discuss its derivation rules. While doing that we will also make comparisons with the rules of the DPLL calculus. We will show that, modulo a negligible technical difference, the Model Evolution calculus reduces precisely to DPLL when the input clause set is ground.[9] The technical difference is simply that, contrary to DPLL, contexts in our calculus contain the pseudo literal $\neg v$. We will see that, except for that, the two calculi operate on the same kind of sequents in the ground case, and stepwise simulate each other.

$$\textsf{Split}\quad \frac{\Lambda \vdash \Phi,\ C \vee L}{\Lambda,\ L\sigma \vdash \Phi,\ C \vee L \qquad \Lambda,\ (\overline{L}\sigma)^{\text{sko}} \vdash \Phi,\ C \vee L}\quad \text{if } (*)$$

$$\text{where } (*) = \begin{cases} C \neq \square, \\ \sigma \text{ is an admissible context unifier of } C \vee L \text{ against } \Lambda \\ \text{with remainder literal } L\sigma, \\ \text{neither } L\sigma \text{ nor } (\overline{L}\sigma)^{\text{sko}} \text{ is contradictory with } \Lambda \end{cases}$$

[9] More precisely, it reduces to the version of DPLL that uses the rules Split' and Close, described at the end of Section 2, in place of Split and Empty, respectively.

We say that the clause $C \vee L$ above is the *selected clause*, the literal L is the *selected literal*, and σ is the *context unifier* Split.

The Split rule is the analog of the Split' rule in DPLL. Together with Commit, described later this the only (*don't-know*) non-deterministic rule of the calculus, the one that drives the search for a model for the input clause set. Split is the rule that discovers when the current candidate model falsifies one of the clauses in the current clause set. It does that by computing a context unifier σ with non-empty remainder for a clause with at least two literals. Once it finds σ, Split attempts to repair the candidate model by selecting a remainder literal $L\sigma$ and adding either $L\sigma$ or its complement to the context. The reason for adding the complement of $L\sigma$ in alternative to $L\sigma$ is of course that the current clause set may have no models that satisfy $L\sigma$. Obviously, the addition of $L\sigma$'s complement to the context will not make the selected clause $C \vee L$ valid in the new candidate model. But it will make sure that no context unifier σ' of $C \vee L$ has $L\sigma'$ in it remainder, forcing the calculus to select other remainder literals, if any, to make $C \vee L$ valid.

Note that Split does not quite add the complement of $L\sigma$: when $L\sigma$ is parameter-free it adds a Skolemized version of $\overline{L\sigma}$.[10] This is in accordance to our treatment of parameter-free literals in contexts as universal sentences.

In the ground case—that is, when both $\Lambda \setminus \{\neg v\}$ and $\Phi \cup \{C \vee L\}$ are ground—the Split rule reduces exactly to the Split' rule of DPLL. To see that it is enough to recall that in the ground case, if $L\sigma = L$ is a remainder literal of a context unifier σ of $C \vee L$ against Λ, then L is positive and such that neither L nor $\overline{L}$ occurs in Λ. Moreover, L (respectively, $\overline{L}$) is contradictory with Λ, in the sense of Definition 3.2, iff $\overline{L} \in \Lambda$ (respectively, $L \in \Lambda$).

$$\textsf{Commit} \quad \frac{\Lambda, K, L \vdash \Phi}{\Lambda, K, L, L\sigma \vdash \Phi \qquad \Lambda, K, L, \overline{L}\sigma \vdash \Phi} \quad \text{if } (*)$$

$$\text{where } (*) = \begin{cases} \sigma \text{ is an mgu of } \overline{L} \text{ and of a fresh p-variant of } K, \\ \mathcal{V}ar(K) = \mathcal{V}ar(L) = \mathcal{V}ar(L)\sigma = \emptyset, \\ \text{neither } L\sigma \text{ nor } \overline{L}\sigma \text{ is contradictory with } \Lambda \cup \{K, L\} \end{cases}$$

We say that the pair (L, K) above is the *selected connection* of Commit.

The Commit rule is the one that detects an inconsistency in the current context.[11] When Commit applies to the sequent $\Lambda, L \vdash \Phi$ with selected connection (L, K), both $L\sigma$ and $\overline{L}\sigma$ are produced by $\Lambda \cup \{L\}$. Each conclusion of Commit removes this anomaly by making the new context produce either just $L\sigma$ or just $\overline{L}\sigma$. Observe that Commit never applies with selected connection (L, K) such that $K \gtrsim \overline{L}$ or

[10] When $L\sigma$ is variable-free the Skolemization step is vacuous.
[11] See the observation after Proposition A.3.

$\overline{L} \gtrsim K$. In fact, if $K \gtrsim \overline{L}$ then $\overline{L}\sigma$ coincides with $\overline{L}$, which makes it contradictory with $\Lambda \cup \{K, L\}$; if $\overline{L} \gtrsim K$ then $L\sigma$ coincides with $\overline{K}$ which makes it contradictory with $\Lambda \cup \{K, L\}$.

There is no rule corresponding to Commit in DPLL. However, Commit never applies in the ground case. In fact, then, $\overline{L}$ and a p-variant of K in Commit's precondition have a most general unifier only if they are identical, or either K or L has the form $\neg v$. Now, in the first case, $\overline{L}\sigma = \overline{L}$, which is obviously contradictory with $\Lambda \cup \{K, L\}$; In the second case, either $K \gtrsim \overline{L}$ or $\overline{L} \gtrsim K$. In both cases then, the rule does not apply.

$$\textsf{Assert} \quad \frac{\Lambda \quad \vdash \Phi,\ L}{\Lambda, L \vdash \Phi,\ L} \quad \text{if} \quad \begin{cases} \text{there is no } K \in \Lambda \text{ s.t. } K \geq L, \\ L \text{ is not contradictory with } \Lambda \end{cases}$$

We say that the clause L above is the *selected unit clause* of Assert.

As in DPLL, the Assert rule is extremely useful in reducing the non-determinism of the calculus. Every candidate model of a clause set $\Phi \cup \{L\}$ *must* make L valid in order to become a model of $\Phi \cup \{L\}$. The Assert rule achieves just that by adding L to the context. Note that since L is parameter-free, its addition to the context is not retractable. Also note that the rule does not apply if the (permament) validity of L has been already established. This is the case when Λ contains a—necessarily parameter-free—literal K such that $K \geq L$. The rule does not apply also if L is contradictory with Λ. In that case, however, the candidate model is unrepairable. Other rules will detect that and cause the calculus to stop working on $\Lambda \vdash \Phi,\ L$.

In the ground case, Assert reduces exactly to its namesake in DPLL. The reason is that, then, $K \geq L$ iff $K = L$, and L is not contradictory with Λ iff $\overline{L} \in \Lambda$.

$$\textsf{Subsume} \quad \frac{\Lambda,\ K \vdash \Phi,\ L \vee C}{\Lambda,\ K \vdash \Phi} \quad \text{if} \quad K \geq L.$$

We say that the clause $L \vee C$ above is the *selected clause* of Subsume.

The purpose of Subsume is the same as in DPLL: get rid of clauses that are valid in the current candidate model, and are guaranteed to stay so. These are exactly those clauses one of whose literals is a p-instance of a—necessarily parameter-free—literal in the current context. Although Subsume is not needed for completeness, it is very useful in practice because it reduces the size of the current clause set.

In the ground case, the Subsume rule reduces to its namesake in DPLL because, then, $K \geq L$ iff $K = L$.

$$\textsf{Resolve} \quad \frac{\Lambda,\ \overline{K} \vdash \Phi,\ L \vee C}{\Lambda,\ \overline{K} \vdash \Phi,\ C} \quad \text{if} \quad \begin{cases} \text{there is an mgu } \sigma \text{ of } L \text{ and } K', \\ \text{a fresh p-variant of } K, \text{ s.t.} \\ (\mathcal{P}ar(K'))\sigma \subseteq V \text{ and } C\sigma = C \end{cases}$$

We say that the clause $L \vee C$ above is the *selected clause* and L is the *selected literal* of Resolve.

This rule is similar to Subsume in that it is not needed for completeness but is useful to reduce the complexity of the current clause set. Since Resolve is in a sense dual to Subsume, it would be reasonable to expect its precondition to be just $K \geq L$. This precondition, however, is a special case of the one provided. The given precondition makes Resolve more widely applicable, allowing for more frequent simplifications. Observe that Resolve is a special case of unit resolution (with backward subsumption): the one in which the resolvent of a unit clause K and a clause $L \vee C$ is exactly C—as opposed to a proper instance of C.

In the ground case, the Resolve rule as well reduces to its namesake in DPLL. To see why it is enough to observe that in that case Resolve's precondition holds iff σ is the empty subtitution and $K' = K = L$.

$$\textsf{Compact} \quad \frac{\Lambda,\ K,\ L \vdash \Phi}{\Lambda,\ K \quad \vdash \Phi} \quad \text{if} \ \ K \geq L$$

We say that the literal L above is the *selected literal* and the literal K is the *subsuming literal* of Compact.[12]

The Compact rule is another simplication rule that is not nedeed for completeness but is useful in practice. To understand the rule's rationale it is important to know that, the way the calculus is defined, Compact's precondition holds only if K is a parameter-free literal. As discussed in a previous section, parameter-free context literals stand for all their instances, with no exception. This means that when a parameter-free literal K is added to a context, all literals in the context that are an instance of K become superfluous. The purpose of Compact is to eliminate these superfluous literals.

There is no rule in DPLL corresponding to Compact. However, it is easy to see that Compact never applies in the ground case.

$$\textsf{Close} \quad \frac{\Lambda \vdash \Phi,\ C}{\Lambda \vdash \square} \quad \text{if} \ \begin{cases} \Phi \neq \emptyset \text{ or } C \neq \square, \\ \text{there is a context unifier of } C \text{ against } \Lambda \\ \text{with an empty remainder} \end{cases}$$

We say that the clause C above is the *selected clause* of Close.

The idea behind Close is that when its precondition holds there is no way to repair the current candidate model to make it satisfy C. The replacement of the current close set by the empty clause signals that the calculus has given up on that candidate

[12] The literals K and L are meant to be distinct.

model. Note that, because of Resolve, it is possible for the calculus to generate a sequent containing an empty clause among other clauses. The Close rule recognizes such sequents and applies to them as well. To see that it is enough to observe that, for any context Λ, the empty substitution is a context unifier of $\Box$ against Λ with an empty remainder.

In the ground case, the Close rule reduces to its namesake in DPLL, because then C has a context unifier against Λ with an empty remainder iff $\overline{L} \in \Lambda$ for every literal L of C.

3.4 Derivations

As customary in sequent calculi, derivations in the Model Evolution calculus are defined formally in terms of *derivation trees*, where each node corresponds to a particular application of a derivation rule, and each of the node's children corresponds to one of the conclusions of the rule.

Definition 3.14 (Derivation Tree) *A* derivation tree (in $\mathcal{ME}$) *is a labeled tree inductively defined as follows:*

1. *a one-node tree is a derivation tree iff its root is labeled with a sequent of the form $\Lambda \vdash \Phi$, where Λ is a context and Φ is a clause set;*
2. *A tree $\mathbf{T}'$ is a derivation tree iff it is obtained from a derivation tree $\mathbf{T}$ by adding to a leaf node N in $\mathbf{T}$ new children nodes $N_1, \ldots, N_m$ so that the sequents labeling $N_1, \ldots, N_m$ can be derived by applying a rule of the calculus to the sequent labeling N. In this case, we say that $\mathbf{T}'$* is derived from $\mathbf{T}$.

We say that a derivation tree $\mathbf{T}$ is a *derivation tree of a clause set* Φ iff its root node tree is labeled with $\neg v \vdash \Phi$.

Let us call a non-leaf node in a derivation tree a Split *node* if the sequents labelling its children are obtained by applying the Split rule to the sequent labeling the node. (Similarly for nodes to which other rules are applied.) Observe that every non-leaf node in a derivation tree has only one child unless it is a Split or a Commit node, in which case it has two children. When it is convenient and it does not cause confusion, we will identify the nodes of a derivation tree with their labels.

Definition 3.15 (Open, Closed) *A branch in a derivation tree is* closed *if its leaf is labeled by a sequent of the form $\Lambda \vdash \Box$; otherwise, the branch is* open. *A derivation tree is* closed *if each of its branches is closed, and it is* open *otherwise.*

We say that a derivation tree (of a clause set Φ) is a *refutation tree (of Φ)* iff it is closed.

In the rest of the paper, the letters i and n will denote finite ordinal numbers, whereas the letter κ will denote an ordinal smaller than or equal to the first infinite ordinal. For every κ then, we will denote a possibly infinite sequence $a_0, a_1, a_2, \ldots$ of κ elements by $(a_i)_{i<\kappa}$.

Definition 3.16 (Derivation) *A* derivation (in $\mathcal{ME}$) *is a possibly infinite sequence of derivation trees* $(\mathbf{T}_i)_{i<\kappa}$*, such that for all* i *with* $0 < i < \kappa$*,* $\mathbf{T}_i$ *is derived from* $\mathbf{T}_{i-1}$*.*

We say that a derivation $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa}$ is a *derivation of a clause set* Φ iff $\mathbf{T}_0$ is a one-node tree with label $\{\neg v\} \vdash \Phi$. We say that $\mathcal{D}$ is a *refutation of* Φ iff $\mathcal{D}$ is finite and ends with a refutation tree of Φ.

We show in the next sections that the Model Evolution calculus is sound and complete in the following sense: for all sets Φ_0 of Σ-clauses with no parameters, Φ_0 is unsatisfiable iff Φ_0 has a refutation in the calculus.

To prove the calculus' completeness we will introduce the notion of an *exhausted branch*, in essence, a derivation tree branch that cannot be extended any further by the calculus. A by-product of the completeness proof will be to show that the interpretation induced by the context in the leaf of an open exhausted branch is a model of the clause set in the branch's root. This means that whenever a derivation of a clause set Φ_0 produces a tree with an open exausted branch, it is possible not only to state that Φ_0 is satisfiable, but also to provide (a finite description) of a model of Φ_0.

4 Correctness of the Calculus

In this section, we prove the soundness and completeness of the Model Evolution calculus.

4.1 Soundness

To prove that the calculus is sound we will first prove that each of its derivation rules preserves a particular notion of satisfiability that we call *a-satisfiability*, after [Bau00].

Let us fix a constant a from the signature $\Sigma^{\text{sko}} \setminus \Sigma$ and consider the substitution $\alpha := \{v \mapsto a \mid v \in V\}$.[13] Given a literal L, we denote by L^a the literal $L\alpha$. Note that L^a is ground if, and only if, L is variable-free. Similarly, given a context Λ, we denote by Λ^a the set of *unit clauses* obtained from Λ by removing the pseudo-literal $\neg v$, replacing each literal L of Λ with L^a, and considering it as a unit clause. Finally, if σ is a substitution, we denote by σ^a the composed substitution $\sigma\alpha$. We point out for later that for all literals L and substitutions σ such that $(\mathcal{P}ar(L))\sigma \subseteq V$ (which includes all parameter-preserving substitutions), $L\sigma^a = L^a\sigma^a$.

We say that a sequent $\Lambda \vdash \Phi$ is *a-(un)satisfiable* iff the clause set $\Lambda^a \cup \Phi$ is (un)satisfiable in the standard sense—that is, has no (Herbrand) model.

Lemma 4.1 *For each rule of the* $\mathcal{ME}$ *calculus, if the premise of the rule is a-satisfiable, then one of its conclusions is a-satisfiable as well.*

[13] Strictly speaking, α is not a substitution in the standard sense because $\mathcal{D}om(\alpha)$ is not finite. But this will cause no problems here.

Proof. We prove the claim only for the rules Split, Resolve, and Close. The proof for Commit is very similar to that for Split. For the other rules the claim holds trivially.

Split) The premise of Split has the form $\Lambda \vdash \Psi$, while its conclusions have respectively the form $\Lambda, K \vdash \Psi$ and $\Lambda, \overline{K}^{\mathrm{sko}} \vdash \Psi$. Suppose that $\Lambda \vdash \Psi$ is a-satisfiable. Now let $\mathbf{x} := (x_1, \ldots, x_n)$ be an enumeration of all the variables of K and note that K and K^a have exactly the same variables. Then consider the unit clause K^a (or, more explicitly, $\forall \mathbf{x}\, K^a$) and its negation $\neg\forall \mathbf{x}\, K^a$. Clearly, one of the two sets

$$S_1 := \Lambda^a \cup \{K^a\} \cup \Psi \quad \text{and} \quad S_2 := \Lambda^a \cup \{\neg\forall \mathbf{x}\, K^a\} \cup \Psi$$

must be satisfiable. If S_1 is satisfiable, we have immediately that $\Lambda, K \vdash \Psi$ is a-satisfiable. If S_2 is satisfiable, then its Skolem form $\Lambda^a \cup \{(\overline{K^a})^{\mathrm{sko}}\} \cup \Psi$ is also satisfiable. Since $(\overline{K^a})^{\mathrm{sko}} = (\overline{K}^{\mathrm{sko}})^a$, as one can easily see, we then have that $\Lambda, \overline{K}^{\mathrm{sko}} \vdash \Psi$ is a-satisfiable.

Resolve) The premise of Resolve has the form $\Lambda \vdash \Phi,\ L \vee C$, while its conclusion has the form $\Lambda \vdash \Phi,\ C$, and there is a most general unifier σ of $\{K, \overline{L}\}$ for some $K \in_\simeq \Lambda$ such that (i) $(\mathcal{P}ar(K))\sigma \subseteq V$, and (ii) $C\sigma = C$. Suppose $\Lambda \vdash \Phi,\ L \vee C$ is a-satisfiable, which means that $\Lambda^a \cup \Phi \cup \{L \vee C\}$ is satisfiable. It is easy to see that because of point (i) above and the fact that L is parameter-free, σ^a is a unifier of $\{K^a, \overline{L}\}$. Observing that $K^a \in_\simeq \Lambda^a$, it follows by the soundness of standard resolution that $\Lambda^a \cup \Phi \cup \{L \vee C,\ C\sigma^a\}$ is also satisfiable. By point (ii) above and the fact that C is parameter-free, we have that $C\sigma^a = (C\sigma)^a = C^a = C$. But this entails that $\Lambda^a \cup \Phi \cup \{C\}$ is satisfiable, and so $\Lambda \vdash \Phi,\ C$ is a-satisfiable.

Close) The premise of Close has the form $\Lambda \vdash \Phi, C$, while its conclusion has the form $\Lambda \vdash \Box$, and there is a context unifier σ of C against Λ with an empty remainder. As $\Lambda \vdash \Box$ is a-unsatisfiable, we must show that $\Lambda \vdash \Phi,\ C$ is a-unsatisfiable as well. We show that by proving that $\Lambda^a \cup \{C\}$ is unsatisfiable.

Let $C = L_1 \vee \cdots \vee L_n$ for some $n \geq 0$. Since σ is a context unifier σ of C against Λ with an empty remainder, we know that there are fresh variants $K_1, \ldots, K_n \in_\simeq \Lambda$ such that σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$, and $(\mathcal{P}ar(K_i))\sigma \subseteq V$ for all $i = 1, \ldots, n$. Let us fix the literals $K_1, \ldots, K_n$.

Clearly, σ^a is a simultaneous unifier of $\{K_1, \overline{L_1}\},\ \ldots,\ \{K_n, \overline{L_n}\}$. By an earlier observation we know that $K_i\sigma^a = K_i^a\sigma^a$ for all $i = 1, \ldots, n$. It follows that σ^a is a simultaneous unifier of

$$\{K_1^a, \overline{L_1}\},\ \{K_2^a, \overline{L_2}\},\ \ldots,\ \{K_n^a, \overline{L_n}\}.$$

This entails that $\{K_1^a, \ldots,\ K_n^a,\ L_1 \vee \cdots \vee L_n\}$ is unsatisfiable. From the fact that $K_1^a, \ldots, K_n^a \in_\simeq \Lambda^a$ it then immediately follows that $\Lambda^a \cup \{C\}$ is unsatisfiable. □

Proposition 4.2 (Soundness) *For all sets Φ_0 of parameter-free Σ-clauses, if Φ_0 has a refutation tree $\mathbf{T}$, then Φ_0 is unsatisfiable.*

Proof. Let $\mathbf{T}$ be a refutation tree of Φ_0. We prove by structural induction on refutation trees that the root $\neg v \vdash \Phi_0$ of $\mathbf{T}$ is a-unsatisfiable. The claim will then follow from the immediate fact that the sequent $\neg v \vdash \Phi_0$ is a-unsatisfiable iff Φ_0 is unsatisfiable.

Base) If $\mathbf{T}$ consists of the single node $\neg v \vdash \Phi_0$, the only way for $\mathbf{T}$ to be a refutation tree is that Φ_0 be $\{\Box\}$. But then $\neg v \vdash \Phi_0$ is trivially a-unsatisfiable.

Step) If $\mathbf{T}$ has more than one node, let M be the root note of $\mathbf{T}$. It is easy to see that for every child node N of M, the subtree of $\mathbf{T}$ rooted at N is a refutation tree of N. Therefore, we can assume by induction that all the children nodes of M are a-unsatisfiable. But then we can conclude that M is also a-unsatisfiable by the contrapositive of Lemma 4.1. □

4.2 Fairness

As customary, we will prove the completeness of the calculus with respect to *fair derivations*. The specific notion of fairness that we adopt is defined formally in the following. For that, it will be convenient to describe a tree $\mathbf{T}$ as the pair $(\mathbf{N}, \mathbf{E})$, where $\mathbf{N}$ is the set of the nodes of $\mathbf{T}$ and $\mathbf{E}$ is the set of the edges of $\mathbf{T}$.

Each derivation $\mathcal{D}$ in the Model Evolution calculus determines a *limit tree* wrt. to all the derivation trees in $\mathcal{D}$.

Definition 4.3 (Limit Tree) *Let $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa}$ be a derivation, where $\mathbf{T}_i = (\mathbf{N}_i, \mathbf{E}_i)$ for all $i < \kappa$. We say that*

$$\mathbf{T} \;:=\; (\bigcup_{i<\kappa} \mathbf{N}_i, \bigcup_{i<\kappa} \mathbf{E}_i)$$

is the limit tree of $\mathcal{D}$.

It is easy to show that a limit tree of a derivation $\mathcal{D}$ is indeed a tree. But note that it will not be a derivation tree unless $\mathcal{D}$ is finite.

Definition 4.4 (Persistency) *Let $\mathbf{T}$ be the limit tree of some derivation, and let $\mathbf{B} = (N_i)_{i<\kappa}$ be a branch in $\mathbf{T}$ with κ nodes. Let $\Lambda_i \vdash \Phi_i$ be the sequent labeling node N_i, for all $i < \kappa$. We define the following sets of* persistent context literals *and* persistent clauses*, respectively:*

$$\Lambda_{\mathbf{B}} := \bigcup_{i<\kappa} \bigcap_{i\leq j<\kappa} \Lambda_j \qquad\qquad \Phi_{\mathbf{B}} := \bigcup_{i<\kappa} \bigcap_{i\leq j<\kappa} \Phi_j$$

In words, a context literal is persistent in the considered branch $\mathbf{B}$ iff it appears in the context of some node and in the context of all the node's descendants (and similarly for persistent clauses).

Although, strictly speaking, $\Lambda_{\mathbf{B}}$ is not a context because it may be infinite, for the purpose of the completeness proof we treat it as one. We note that all the definitions introduced in Section 3.1 can be applied without change to $\Lambda_{\mathbf{B}}$ as well.

Fair derivations in the $\mathcal{ME}$ calculus are defined in terms of *exhausted branches*.

Definition 4.5 (Exhausted branch) *Let* $\mathbf{T}$ *be a limit tree, and let* $\mathbf{B} = (N_i)_{i<\kappa}$ *be a branch in* $\mathbf{T}$ *with* κ *nodes. For all* $i < \kappa$*, let* $\Lambda_i \vdash \Phi_i$ *be the sequent labeling node* N_i*. The branch* $\mathbf{B}$ *is* exhausted *iff for all* $i < \kappa$ *all of the following hold:*

(i) For all $C \in \Phi_{\mathbf{B}}$*, if* Split *is applicable to* $\Lambda_i \vdash \Phi_i$ *with selected clause* C*, productive context unifier* σ*, then there is a* $j \geq i$ *with* $j < \kappa$ *such that* Λ_j *produces* $C\sigma$*.*

(ii) For all $L, K \in \Lambda_{\mathbf{B}}$*, if* Commit *is applicable to* $\Lambda_i \vdash \Phi_i$ *with selected connection* (L, K) *and mgu* σ*, then there is a* $j \geq i$ *with* $j < \kappa$ *such that* Λ_j *is consistent wrt.* $L\sigma$*.*

(iii) For all unit clauses $L \in \Phi_{\mathbf{B}}$*, if* Assert *is applicable to* $\Lambda_i \vdash \Phi_i$ *with selected unit clause* L*, then there is a* $j \geq i$ *with* $j < \kappa$ *such that* $L \in_{\geq} \Phi_j$*.*

(iv) For all $C \in \Phi_{\mathbf{B}}$*,* Close *is not applicable to* $\Lambda_i \vdash \Phi_i$ *with selected clause* C*.*

(v) $\Phi_i \neq \{\Box\}$*.*

It is worth noticing that Point (i) in Definition 4.5 does *not* require that Split be eventually applied with selected clause C and context unifier σ, for the branch to be exhausted. It only requires that the *intended effect* of applying Split with selected clause C and context unifier σ, namely that $C\sigma$ is permanently produced, be eventually achieved. A similar observation can be made about Point (ii) and the intended effect of applying Commit with selected connection (L, K), namely that the inconsistency generated by L and K is permanently resolved, and about Point (iii) and the effect of applying Assert with selected unit clause L, namely that a literal more general (wrt. $\geq$) than L is permanently added to the context.

Definition 4.6 (Fairness) *A limit tree of a derivation is* fair *iff it is a refutation tree or it has an exhausted branch. A derivation is* fair *iff its limit tree is fair.*

We point out that fair derivations as defined above do exist and are computable for any set of (parameter-free) Σ-clauses. A proof of this fact can be given by adapting a technique used in [Bau00] to show the computability of fair derivations in FDPLL. Moreover, and similarly to FDPLL, fair derivations need not be searched. As we will see, the calculus is *proof convergent*, that is, if a set Φ of Σ-clauses is unsatisfiable, then *every* fair derivation of Φ is a refutation.

4.3 Completeness

For the rest of this section, let Φ be a set of parameter-free Σ-clauses and assume that $\mathcal{D}$ is a fair derivation of Φ that is not a refutation. Observe that $\mathcal{D}$'s limit tree must have at least one exhausted branch. We denote this branch by $\mathbf{B} = (N_i)_{i<\kappa}$. Then, by $\Lambda_i \vdash \Phi_i$, we will always mean the sequent labeling the node N_i in $\mathbf{B}$, for all $i < \kappa$. (As a consequence, we will also have that $\Lambda_0 = \{\neg v\}$ and $\Phi_0 = \Phi$.)

Quite often we will appeal to the following compactness property of $\Lambda_{\mathbf{B}}$. By definition, $L \in \Lambda_{\mathbf{B}}$ holds iff there is an $i < \kappa$ such that $L \in \Lambda_j$ for all $j \geq i$ with $j < \kappa$. Similarly, if $L \in_{\simeq} \Lambda_{\mathbf{B}}$ then, by definition, $L \simeq K$ for some literal $K \in \Lambda_{\mathbf{B}}$. Again, there is an $i < \kappa$ such that $K \in \Lambda_j$ for all $j \geq i$ with $j < \kappa$, which entails that $L \in_{\simeq} \Lambda_j$, for all $j \geq i$ with $j < \kappa$. More generally then, if $L_1, \ldots, L_n \in \Lambda_{\mathbf{B}}$ (or $L_1, \ldots, L_n \in_{\simeq} \Lambda_{\mathbf{B}}$) for some $n \geq 0$, then there is an $i < \kappa$ such that $L_1, \ldots, L_n \in \Lambda_j$ (or $L_1, \ldots, L_n \in_{\simeq} \Lambda_j$) for all $j \geq i$ with $j < \kappa$.[14]

We provide below only a sketch of the completeness proof by proving just the main results. A complete proof of all the auxiliary results stated here (and used in the proof the main results) can be found in the appendix.

4.3.1 Evolving Contexts

Derivations are about about stepwise modifications of sequents. This section contains lemmas mainly describing bounds the contexts in sequents can evolve within. For instance, it is impossible to derive a sequent with a context that contains two (or more) p-variants of the same literal.

Lemma 4.7 *For all $i < \kappa$, Λ_i is not contradictory.*

Being non-contradictory is a fundamental property of the contexts manipulated by the calculus. Lemma 4.7 essentially says that rule applications produce non-contradictory contexts from non-contradictory contexts. The next lemma extends the previous one to the limit case.

Lemma 4.8 *$\Lambda_{\mathbf{B}}$ is not contradictory.*

Lemma 4.9 *The sequent $\Lambda,\ L \vdash \Psi$ is not derivable from $\Lambda \vdash \Psi$ if $L \in_{\geq} \Lambda$ or $\overline{L} \in_{\geq} \Lambda$.*

Lemma 4.9 expresses that the inference rules of the model evolution calculus never add a literal to a context in presence of a more general literal (wrt. $\geq$). One could say that this fact expresses a kind of "loop check".

Lemma 4.10 *Let $i < \kappa$ and $L \in \Lambda_i$. For every j with $i \leq j < \kappa$ there is a $K \in \Lambda_j$ such that $K \geq L$.*

In the course of the development of a branch, a literal in a sequent's context may be deleted by means of the Compact rule. Such a deletion is only possible in the presence of a p-subsuming literal, which takes the rôle of the deleted literal. The p-subsuming literal itself may be deleted later, in a similar way. Lemma 4.10 is a formal statement of this process.

[14] It is easy to see that this index i can be determined by taking the maximum of the i-indices associated individually to the literals $L_1, \ldots, L_n$, as just described.

Lemma 4.11 *For any two different literals $K, L \in \bigcup_{i<\kappa} \Lambda_i$ it holds that $K \not\simeq L$.*

Lemma 4.11 states that if some context in the branch $\mathbf{B}$ contains a literal K, then neither this nor any other context can contain a p-variant L of it. This holds even if K is deleted at some point along the branch. Occasionally, we will use the the following specialization of the lemma.

Lemma 4.12 *For all $i < \kappa$ and for any two different literals $K, L \in \Lambda_i$ it holds that $K \not\simeq L$.*

Lemma 4.13 *For all $i < \kappa$ and $L \in \Lambda_i$ there is a $K \in \Lambda_{\mathbf{B}}$ such that $K \geq L$.*

Lemma 4.13 essentially states that the set of persistent context literals of the branch $\mathbf{B}$ contains generalizations of all the context literals along $\mathbf{B}$.

Lemma 4.14 *For all $k < \kappa$ and $L \in \Lambda_k \setminus \Lambda_{\mathbf{B}}$, there is an $i > k$ such that for all j with $i \leq j < \kappa$, $L \not\in_{\sim} \Lambda_j$.*

Notice that this lemma is not trivial. That $L \notin \Lambda_{\mathbf{B}}$ holds is consistent with adding L to a sequent's context, deleting it later again, and repeating this forever. The lemma guarantees that this cannot happen. It could be strengthened and express that L is entered at some timepoint, say, k, and L is present in each context up to timepoint $i-1 \geq k$, *and L (or any p-variant of L) is not contained in any subsequent context*. However, for the purpose of the completeness proof this stronger formulation is not needed.

Lemma 4.15 *Let K, L be two literals. If K produces L in $\Lambda_{\mathbf{B}}$, then for all $i < \kappa$ there is no $K' \in \Lambda_i$ and no p-preserving substitution σ such that $K \gtrsim \overline{K'}\sigma \gtrsim L$.*

Lemma 4.16 *Let K, L be two literals with $K \in \Lambda_{\mathbf{B}}$. If K produces L in $\Lambda_{\mathbf{B}}$, then there is an i such that for all $j \geq i$ with $j < \kappa$, $K \in \Lambda_j$ and K produces L in Λ_j.*

This important lemma takes productivity wrt. the limit to a finite ordinal.

Lemma 4.17 *If $\Lambda_{\mathbf{B}}$ does not produce a literal L, then there is an i such that for all $j \geq i$, Λ_j does not produce L.*

This lemma is the counterpart of Lemma 4.16, this time about non-productivity.

4.3.2 Properties of Inference Rules

The following lemmas provide sufficient conditions for the applicability of the main rules of the calculus to a given context. We will refer to these conditions to prove the completeness of the calculus.

The first lemma considers the case of Commit, whose purpose is to repair an interpretation when consistency is violated in the corresponding context, which happens when both a literal and its complement are produced by the context.

Lemma 4.18 (Commit Applicability) *Let $\Lambda \vdash \Psi$ be a sequent with a non-contradictory context Λ, and let $K, K' \in \Lambda$ be variable-free literals. If K produces L in Λ and K' produces $\overline{L}$ in Λ, then* Commit *is applicable to $\Lambda \vdash \Psi$ with selected connection (K, K') and some most general unifier σ. Moreover, $K \gtrsim K\sigma \gtrsim L$ and $K' \gtrsim K'\sigma \gtrsim \overline{L}$.*

As for Commit, a similar result is needed to characterize conditions under which Split will be applicable. Its proof is more complex and will use the next two lemmas. The first one shows how unification can be used to identify clause instances that are false in the interpretation induced by the currently context.

Lemma 4.19 (Lifting Lemma) *Let Λ be a non-contradictory context. Let $C = L_1 \vee \cdots \vee L_n$ be a Σ-clause and γ a grounding substitution for C. If Λ produces $\overline{L_1}\gamma, \ldots, \overline{L_n}\gamma$, then there are fresh variants $K_1, \ldots, K_n \in_\simeq \Lambda$ and a substitution σ such that*

1. *σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,*
2. *for all $i = 1, \ldots, n$, $L_i \gtrsim L_i\sigma \gtrsim L_i\gamma$,*
3. *for all $i = 1, \ldots, n$, K_i produces $\overline{L_i}\sigma$ in Λ.*

In Section 3 we mentioned that the calculus does not need to search for admissible context unifiers, and that any context unifier can be composed with a renaming substitution, determined deterministically, such that the resulting context unifier is admissible. This fact is expressed by the following lemma.

Lemma 4.20 (Existence of Admissible Context Unifiers) *Let Λ be a context, C a clause and σ a context unifier of C against Λ. Then, there is a renaming ρ such that $\sigma' := \sigma\rho$ is an admissible context unifier of C against Λ.*

It should be mentioned that the purpose of this lemma is just to show the existence of an admissible context unifier based on a possibly non-admissible context unifier. A realistic implementation would compute a "more clever" renaming, one that maximizes the parameter-free literals in the resulting remainder. For completeness purposes, however, *any* renaming that yields an admissibile context unifier will do, as it will be clear from the proof of Proposition 4.25.

The following lemma applies (in particular) to remainders of admissible context unifiers. It expresses, roughly, if each remainder literal *individually* is contradictory with a given context, then there is a substitution, δ, that allows to consider the context as closed by the thus instantiated clause.

Lemma 4.21 *Let Λ be a context, $L_1 \vee \cdots \vee L_n$ be a clause, where $n \geq 0$, such that for all distinct $i, j = 1, \ldots, n$, L_i is parameter- or variable-free and $\mathcal{V}ar(L_i) \cap \mathcal{V}ar(L_j) = \emptyset$. If for all $i = 1, \ldots, n$, L_i is contradictory with Λ then there are fresh literals $K_1, \ldots, K_n \in_\simeq \Lambda$ and a substitution δ such that the following holds:*

1. *δ is a simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,*
2. *for all $i = 1, \ldots, n$, $\mathcal{D}om(\delta) \cap \mathcal{P}ar(L_i) = \emptyset$, (i.e. δ does not move any single parameter in the given clause)*
3. *for all $i = 1, \ldots, n$, $(\mathcal{P}ar(K_i))\delta \subseteq V$.*

Now we can turn to the lemma stating conditions under which the Split rule is applicable. Roughly, Split is applicable if its selected clause admits a context unifier, it does not overlap with Assert, i.e. the selected clause must contain at least two literals, and Close is not applicable with the selected clause.

Lemma 4.22 (Split Applicability) *Let $\Lambda \vdash \Psi$, C be a sequent with a non-contradictory context Λ, where C contains at least two literals. If all context unifiers of C against Λ have a non-empty remainder, and σ is a context unifiers of C against Λ with a remainder not produced by Λ, then* Split *is applicable to $\Lambda \vdash \Psi$, C with selected clause C and context unifier σ.*

Finally, it remains the case of Assert, which is as follows.

Lemma 4.23 (Assert Applicability) *Let $\Lambda \vdash \Psi$, L be a sequent with a non-contradictory context Λ. If all context unifiers of L against Λ have a non-empty remainder and there is an instance of L that is not produced by Λ, then* Assert *is applicable to $\Lambda \vdash \Psi$, L with selected unit clause L.*

4.3.3 Main Result

In the first lemma below we show that the set $\Lambda_{\mathbf{B}}$ of persistent context literals of any exhausted branch $\mathbf{B}$ of a limit tree is a consistent context. This property is essential for viewing $\Lambda_{\mathbf{B}}$ as a representation of an interpretation. Building on this result, in the subsequent lemma we show that the interpretation induced by $\Lambda_{\mathbf{B}}$ is a model of the clause set at the root of the limit tree.

Lemma 4.24 *$\Lambda_{\mathbf{B}}$ is consistent.*

As with finite contexts, Lemma 4.24 guarantees that the preinterpretation $I_{\Lambda_{\mathbf{B}}}$ induced by $\Lambda_{\mathbf{B}}$ is in fact an interpretation.

Proof. Suppose to the contrary that $\Lambda_{\mathbf{B}}$ is not consistent. This means there is a literal L such that $\Lambda_{\mathbf{B}}$ produces both L and $\overline{L}$. Let $K, K' \in \Lambda_{\mathbf{B}}$ be two literals such that K produces L in $\Lambda_{\mathbf{B}}$ and K' produces $\overline{L}$ in $\Lambda_{\mathbf{B}}$.

From the application of Lemma 4.16 to K and L on the one side, and to K' and $\overline{L}$ on the other side, we conclude there is an i such that for all $j \geq i$ it holds $K, K' \in \Lambda_j$, K produces L in Λ_j, and K' produces $\overline{L}$ in Λ_j.

By Lemma 4.7, Λ_i is not contradictory. By Lemma A.6 both K and K' are variable-free. Therefore, Lemma 4.18 can be applied to conclude that Commit is

applicable to the node N_i, which is labeled with $\Lambda_i \vdash \phi_i$, with selected connection (K, K') and some most general unifier σ. Moreover, Lemma 4.18 gives us $K \gtrsim K\sigma \gtrsim L$ and $K' \gtrsim K'\sigma \gtrsim \overline{L}$.

By Definition 4.5-(ii), there is a $j \geq i$ such that Λ_j is consistent wrt. $K\sigma$. This means Λ_j does not produce $K\sigma$ or Λ_j does not produce $\overline{K}\sigma$. It suffices to consider the former case, because the proof for the latter case is similar for reasons of symmetry.

Recall that for $K \in \Lambda_j$ it holds $K \gtrsim K\sigma \gtrsim L$. The literal K is thus a candidate msg of L in Λ_j. Let $K''' = K$ if K is such an msg, otherwise chose $K''' \in \Lambda_j$ such that $K \gtrsim K'''$ and K''' is an msg of L in Λ_j.

Above we concluded that Λ_j does not produce $K\sigma$. Since K''' is an msg of L in Λ_j, there must be a literal $K'' \in \Lambda_j$ and a p-preserving substitution σ' such that $K''' \gtrsim \overline{K''}\sigma' \gtrsim K\sigma$. Recall that the application of Lemma 4.18 above gave us $K\sigma \gtrsim L$, and that $K \gtrsim K'''$ holds. Together this entails $K \gtrsim \overline{K''}\sigma' \gtrsim L$. But then, using Lemma 4.15 in the contrapositive direction conclude that K does not produce L in $\Lambda_{\mathbf{B}}$. A plain contradiction to what was derived near the beginning of the proof. □

With Lemma 4.24, we are now ready to prove the following fundamental proposition, which expresses that the calculus computes a model for the persistent clauses.

Proposition 4.25 *If $\square \notin \Phi_{\mathbf{B}}$, then $I_{\Lambda_{\mathbf{B}}}$ is a model of $\Phi_{\mathbf{B}}$.*

Proof. From Lemmas 4.8, 4.24 and Proposition A.3 we to know that $I_{\Lambda_{\mathbf{B}}}$ is an interpretation. Now, suppose ad absurdum that $\Phi_{\mathbf{B}}$ does not contain the empty clause, but $I_{\Lambda_{\mathbf{B}}}$ is not a model of $\Phi_{\mathbf{B}}$. This means that there is a ground instance $C\gamma$ of a clause $C = L_1 \vee \cdots \vee L_n$ with $n \geq 1$ from $\Phi_{\mathbf{B}}$ that is not satisfied by $I_{\Lambda_{\mathbf{B}}}$. It follows by definition of $I_{\Lambda_{\mathbf{B}}}$ that $\Lambda_{\mathbf{B}}$ produces $\overline{L_1}\gamma, \ldots, \overline{L_n}\gamma$. We distinguish two complementary cases, depending on whether $n = 1$ or $n > 1$, and show that they both lead to a contradiction.

($n = 1$) In this case, C consists of the single literal L_1. Since $\Lambda_{\mathbf{B}}$ produces $\overline{L_1}\gamma$, and $\Lambda_{\mathbf{B}}$ is consistent by Lemma 4.24, we know that $\Lambda_{\mathbf{B}}$ does not produce $L_1\gamma$. By Lemma 4.17 then, we can conclude that there is an i such that

$$\text{for all } j \geq i,\ \Lambda_j \text{ does not produce } L_1\gamma. \tag{1}$$

Because L_1 is a (unit) clause from $\Phi_{\mathbf{B}}$, there is a i' such that $L_1 \in \Phi_{j'}$ for all $j' \geq i'$. Without loss of generality assume that $i \geq i'$ (otherwise i' can be used instead of i in the sequel).

By of Definition 4.5-(iv), Close is not applicable to $\Lambda_i \vdash \Phi_i$ with selected clause L_1. Since $L_1 \in \Phi_i$, this entails that all context unifiers of L_1 against Λ_i have a non-empty remainder. Together with (1), this implies by Lemma 4.23 that Assert

is applicable to $\Lambda_i \vdash \Phi_i$ with selected unit clause L_1. According to Definition 4.5-(iii) then, there is a $j \geq i$ with $j < \kappa$ and an $L \in \Phi_j$ such that $L \geq L_1$. Recall that clauses in sequents are parameter-free. It is easy to show then, with L_1 being parameter-free, L must be parameter-free as well. Moreover, $L \geq L_1 \geq L_1\gamma$. But then, we have by Lemma A.5 that Λ_j produces $L_1\gamma$, in contradiction to (1) above.

($n > 1$) By the lifting lemma (Lemma 4.19), there are fresh p-variants $K_1, \ldots, K_n \in_\simeq \Lambda_\mathbf{B}$ and a substitution σ such that

1. σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,
2. for all $k = 1, \ldots, n$, $L_k \gtrsim L_k\sigma \gtrsim L_k\gamma$,
3. for all $k = 1, \ldots, n$, K_k produces $\overline{L_k}\sigma$ in $\Lambda_\mathbf{B}$.

Clearly, by Definition 3.11, σ is a productive context unifier of C against $\Lambda_\mathbf{B}$ with some remainder D. By Lemma 4.20 then, an admissible context unifier of C against $\Lambda_\mathbf{B}$ can be obtained as $\sigma' = \sigma\rho$, for some renaming ρ.

Let $k \in \{1, \ldots, n\}$ and observe that a literal K produces a literal L in a context Λ iff K produces a variant of L in Λ. From the fact that K_k produces $\overline{L_k}\sigma$ in $\Lambda_\mathbf{B}$, we have that K_k produces $\overline{L_k}\sigma'$ in $\Lambda_\mathbf{B}$ as well. Given that $K_n \in_\simeq \Lambda_\mathbf{B}$, we have that $\Lambda_\mathbf{B}$ produces $\overline{L_k}\sigma'$ and so, because of its consistency, it cannot produce $L_k\sigma'$.

By applying Lemma 4.17 to every $L_k\sigma'$ individually, and taking the maximum of the indices i mentioned in the lemma's statement, we conclude that there is an i such that

$$\text{for all } k = 1, \ldots, n \text{ and all } j \geq i,\ \Lambda_j \text{ does not produce } L_k\sigma'. \tag{2}$$

By assumption, C is a clause of $\Phi_\mathbf{B}$. Hence, there is a i' such that $C \in \Phi_{j'}$ for all $j' \geq i'$. Without loss of generality suppose that $i \geq i'$ (otherwise i' can be used instead of i in the sequel).

Because of Definition 4.5-(iv), Close is not applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C. Therefore, all context unifiers of C against Λ_i must have a non-empty remainder.By (2), Λ_i does not produce $L_k\sigma'$ (for all $k = 1, \ldots, n$), and so, in particular, Λ_i does not produce any remainder literal of σ'. By Lemma 4.22 then, Split is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C and productive context unifier σ'. Because of Definition 4.5-(i), there is a $j \geq i$ such that Λ_j produces $C\sigma$. This means Λ_j produces $L_k\sigma'$, for some $k \in \{1, \ldots, n\}$, in contradiction to (2) above. □

The completeness of the calculus is a consequence of Proposition 4.25. We state it here in its contrapositive form to underline the model computation ability of $\mathcal{ME}$.

Theorem 4.26 (Completeness) *Let $\mathcal{D}$ be a fair derivation of Φ with limit tree $\mathbf{T}$. If $\mathbf{T}$ is not a refutation tree, then Φ is satisfiable; more specifically, for every exhausted branch $\mathbf{B}$ of $\mathbf{T}$, $I_{\Lambda_\mathbf{B}}$ is a model of Φ.*

Let $\top$ be the universally true clause. For every clause $C \in \Phi$, we define $C^0 := C$, and for all $i > 0$

$$C^i := \begin{cases} D & \text{if } C^{i-1} \text{ is of the form } L \vee D \text{ and Resolve is applied with selected clause } C^{i-1} \text{ and selected literal } L \text{ to } \Lambda_{i-1} \vdash \Phi_{i-1} \text{ to obtain } \Lambda_i \vdash \Phi_i \\ \top & \text{if } C^{i-1} \text{ is of the form } L \vee D \text{ and Subsume is applied with selected clause } C^{i-1} \text{ and selected literal } L \text{ to } \Lambda_{i-1} \vdash \Phi_{i-1} \text{ to obtain } \Lambda_i \vdash \Phi_i \\ C^{i-1} & \text{otherwise} \end{cases}$$

Observe that for all $i \geq 0$, $\{C^i \mid C \in \Phi\} = \Phi_i \cup \{\top\}$.

Proof. From Lemmas 4.8, 4.24 and Proposition A.3 we to know that $I_{\Lambda_\mathbf{B}}$ is an interpretation. Let C be any clause in Φ. It is enough to show that $I_{\Lambda_\mathbf{B}}$ is a model of C. Now, it is easy to see that there is a smallest j such that $C^i = C^{i-1}$ for all $i > j$, which means that C^j is either $\top$ or a persistent clause of $\mathbf{B}$. Let us fix that j. We show below by induction on i that $I_{\Lambda_\mathbf{B}}$ is a model of C^i for all $i \leq j$, from which it will immediately follow that $I_{\Lambda_\mathbf{B}}$ is a model of $C = C^0$.

($i = j$) If C^i is $\top$, $I_{\Lambda_\mathbf{B}}$ is trivially a model of C^i. Hence assume that C^i is a persistent clause of $\mathbf{B}$, that is, $C_i \in \Phi_\mathbf{B}$. By Proposition 4.25, it is enough to show that $\Phi_\mathbf{B}$ does not contain the empty clause. Assume by contradiction that it does, that is that $\Phi_\mathbf{B} = \Phi', \square$ for some clause set Φ'.

That $\Phi' = \emptyset$ holds is impossible by Definition 4.5-(v). If $\Phi' \neq \emptyset$, there must be an i such that $\Lambda_i \vdash \Phi_i$ has the form $\Lambda_i \vdash \Phi'_i, \square$ for some non-empty clause set Φ'_i. But then, since the empty substitution is certainly a context unifier of $\square$ against Λ_i with an empty remainder, Close is applicable to $\Lambda_i \vdash \Phi'_i, \square$ with selected clause $\square$, which is impossible by Definition 4.5-(iv). It follows that $I_{\Lambda_\mathbf{B}}$ is a model of C^j.

($i < j$) Assume by induction hypothesis that $I_{\Lambda_\mathbf{B}}$ is a model of C^{i+1}, and consider the following three cases, depending on the definition of C^{i+1}.
(i) If $C^i = C^{i+1}$, we can conclude immediately that $I_{\Lambda_\mathbf{B}}$ is a model of C^i.
(ii) If C^i is of the form $L \vee D$ and Resolve is applied with selected literal L to $\Lambda_i \vdash \Phi_i$ to obtain $\Lambda_{i+1} \vdash \Phi_{i+1}$, then $C^{i+1} = D$. It follows immediately that $I_{\Lambda_\mathbf{B}}$ is a model of C^i.
(iii) If C^i is of the form $L \vee D$ and Subsume is applied with selected clause C^i to $\Lambda_i \vdash \Phi_i$ to obtain $\Lambda_{i+1} \vdash \Phi_{i+1}$, then $C^{i+1} = \top$. By the definition of Subsume, there is a $K \in \Lambda_i$ such that $K \geq L$. By Lemma 4.13, there is a $K' \in \Lambda_\mathbf{B}$ such that $K' \geq K$. It follows that there is a $K' \in \Lambda_\mathbf{B}$ such that $K' \geq L$. Recalling that $C \in \Phi$ is parameter-free and that, by definition, C^i is a sub-clause of C, we have that C^i, and so L, is parameter-free. From the fact that $K' \geq L$, it follows that K' is also parameter-free and that $K' \geq L\gamma$, for any grounding substitution γ. Now, since $K' \in \Lambda_\mathbf{B}$ and $K' \geq L\gamma$, we have by Lemma A.5 that $\Lambda_\mathbf{B}$ produces $L\gamma$. From

the consistency of $\Lambda_{\mathbf{B}}$ it follows that $I_{\Lambda_{\mathbf{B}}}$ satisfies $L\gamma$. Because $L\gamma$ was an arbitrary ground instance of L, we can deduce that $I_{\Lambda_{\mathbf{B}}}$ is a model of L, and so of C^i. □

When the branch $\mathbf{B}$ in Theorem 4.26 is finite, $\Lambda_{\mathbf{B}}$ coincides with the context Λ_n, say, in $\mathbf{B}$'s leaf. From a model computation perspective, this is a very important fact because it means that a model of the original clause set—or rather, a finite representation of it, Λ_n—is readily available at the end of the derivation; it does not have to be computed from the branch, as in other model generation calculi.

The calculus is proof confluent [Bib82]: any derivation of an unsatisfiable clause set extends to a refutation. In fact, because of the strong completeness result in Theorem 4.26, the calculus satisfies an even stronger property, which we refer to as *proof convergence*.

Corollary 4.27 (Proof Convergence) *Let Φ be a a parameter-free clause set over the signature Σ. If Φ is unsatisfiable, then every fair derivation of Φ is a refutation.*

In practical terms, the above corollary implies that as long as a derivation strategy guarantees fairness, the order of application of the rules of the calculus is irrelevant for proving an input clause set unsatisfiable, giving to the $\mathcal{ME}$ calculus the same flexibility enjoyed by the the DPLL calculus at the propositional level.

5 Related work

Approaches that have features in common with $\mathcal{ME}$ come from the following four categories: *first-order DPLL methods*, *instance-based methods*, *Resolution methods* and *Tableau methods*.

5.1 First-Order DPLL Methods

A "lifted" version of the DPLL method has been described in the early textbook on automated reasoning by Chang and Lee [CL73]. It uses the device of pseudosemantic trees, which, like $\mathcal{ME}$, realize splits at the non-ground level. Nethertheless, the pseudosemantic tree method is very different: in sharp contrast to $\mathcal{ME}$, a variable is treated rigidly there, i.e. as a placeholder for a (one) not-yet-known term.[15] The Section 5.4 below discusses rigid variable methods, and what is said there applies to the method in [CL73] as well.

The closest relative or the $\mathcal{ME}$ calculus is the FDPLL calculus developed by one of us [Bau00]. As said in the introduction, $\mathcal{ME}$ is loosely based on FDPLL. More precisely, the $\mathcal{ME}$ calculus can be specialized to the core FDPLL calculus by (a) removing the Subsume, the Resolve and the Compact inference rules (these

[15] However the term "rigid" is not used there, as it was not introduced at the time the book [CL73] was written.

rules are optional in $\mathcal{ME}$), and (b), speaking in terms of definition of the Split rule, use only admissible context unifiers such that $C\sigma$ is variable-free (using only such admissible context unifiers preserves completeness). In terms of the present paper, the core calculus of FDPLL does not have simplification rules and does not deal with variables – it just uses parameters. However, in [Bau00] an extension of the core FDPLL calculus to include reasoning with variables is sketched. As in $\mathcal{ME}$, mixed literals are not allowed, and so the literals used there for splits are of the same type – variable free or parameter-free (or both). Nethertheless, $\mathcal{ME}$ is much stronger than that version of FDPLL. Expressed in $\mathcal{ME}$ terms, the rules mentioned under (a) above are still not available in FDPLL, and the admissible context unifiers are now of the kind (b) above, or such that the remainder always consists of exactly one literal, and furthermore each non-remainder literal is confronted with a parameter-free literal from the context. In resolution terminology, FDPLL mimics unit-resulting resolution.

The impact of the restricted capabilities of FDPLL can be seen by looking at some examples. For instance, if the current context is just $\Lambda = \{\neg v\}$ and there is a given clause $P(x) \vee Q(y) \vee R(z)$, FDPLL will consider the clause instance $P(u) \vee Q(v) \vee R(w)$ and split based on its literals, which contain *parameters*. In contrast, $\mathcal{ME}$ will in essence carry out a case analysis according to the three literals $P(x)$, $Q(y)$ and $R(z)$, which are *parameter-free* and hence are implicitly universally quantified. (As explained in Section 3, a parameter-free literal in a context puts more constraints on later additions of literals than a variable-free literal, leading in principle to shorter derivations.)

As another example consider the context $\Lambda = \{\neg v, P(u, v), Q(x, a, z)\}$ and the clause $\neg P(x, x) \vee \neg Q(x, y, z) \vee R(y, z)$. Based on the admissible context unifier[16] $\sigma = \{u \mapsto v,\ x \mapsto v,\ y \mapsto a\}$, the Split rule is applicable, and it will split on the sole remainder literal $R(a, z)$. A comparable inference step is not possible with FDPLL, as one of the involved context literals is not parameter-free.

In conclusion, due to the presence of the simplification inference rules Subsume, Resolve and Compact, and due to the better treatment of variables, the $\mathcal{ME}$ calculus improves significantly on FDPLL.

5.2 Instance-Based Methods

Besides the FDPLL calculus, $\mathcal{ME}$ is related to the family of *instance-based methods*. Proof search in instance-based methods relies on maintaining a set of instances of input clauses and analyzing it for satisfiability until completion. We point out that $\mathcal{ME}$ is *not* an instance-based method in this sense, as clause instances are used only temporarily within the Split inference rule and can be forgotten after the split has been carried out.

The contemporary stream of research on instance-based methods was initiated with the Hyperlinking calculus [LP92], The Hyperlinking calculus (HL) is based on

[16] For simplicity of presentation, no fresh variants are taken.

the idea of steadily growing a set of instances of input clauses in an intelligent way, which is regularly tested for propositional unsatisfiability by an integrated DPLL procedure. As a conceptual difference to $\mathcal{ME}$, HL *includes* a DPLL procedure but does not (directly) *extend* it to first-order clause logic.

The current successor of HL is the Ordered Semantic Hyperlinking calculus (OSHL) [PZ97, PZ00]. OSHL has many interesting features, for instance "semantical guidance" by assuming a procedural representation of an (any) interpretation, that just has to be capable to decide if a given ground literal is true in the interpretation. As in $\mathcal{ME}$, the main operation in OSHL is to detect an instance of a clause that is false in a current interpretation, and then repair the interpretation (in the sense given here). However, unlike as in $\mathcal{ME}$, the repairs are carried out through *ground literals*.

Another method in this category is the Confluent Connection Calculus (CCC) by Baumgartner, Furbach and Eisinger [BEF99]. Unlike all connection methods known at the time it was introduced, CCC enjoys the important property of proof convergence. (See [Bib82] for more information on connection methods in general). CCC is similar to HL but is formulated as a connection method and does not rely on a DPLL method behind. A conceptual difference of CCC to all other instance based methods mentioned here is, that substitutions are applied globally, to the whole clause set derived so far, and thus CCC behaves in this respect much like rigid variable tableaux calculi (see below). Unlike $\mathcal{ME}$, the completeness proof is not based on a model-generation argument but relied on the Herbrand theorem instead. This makes it hard to identify semantically justified refinements.

Some instance-based calculi have been formulated within the (clausal) tableau framework. The initial work in this direction is Billon's disconnection method [Bil96]. The calculus described in [Bau98] relates to the disconnection method much like the hyper resolution calculus relates to the resolution calculus.

The disconnection method has been picked up by Letz and Stenz for further improvements. The disconnection calculus, as they call it, uses clausal tableau as the primary data structure. The tableau structure represents an exhaustive search through all possible connections between literals in clauses; the (single) inference rule extends the current tableau by two clause instances found via a connection on the branch. In [LS01] improvements have been mentioned and a dedicated inference rule for deriving unit clauses has been sketched. Interestingly, all variables in a derived unit clause have to be identified for soundness reasons. Various forms of equality handling have been sketched for the disconnection calculus [LS02] (without completeness proof, however), and an efficient implementation has been built [Ste02].

Two variants of an instance-based method are described by Hooker *et al.* [HRCS02]. One of them, the "Primal Approach" seems to be very similar to the disconnection method (see above) although, unfortunately, the relation with this method is not made explicit in [HRCS02]. The other variant, the "Dual Approach", differs from the former by the presence of *auxiliary* clauses of the form $K \rightarrow L$ generated during

the proof search, where (K, L) is a connection of literals occurring in the current clause set. No simplification mechanisms have been described, like for instance those based on unit propagation rules. Both methods compare to $\mathcal{ME}$ in the same way as the disconnection method, discussed above.

A significant difference between the instance based methods (we are aware of) and the $\mathcal{ME}$ calculus is that the former maintain a growing set of instances of input *clauses*, while $\mathcal{ME}$ does maintain a growing set of instances of input *literals*: the current context. However, contexts grow more slowly than sets of clause instances, which may lead to an (at least) exponential advantage for $\mathcal{ME}$ regarding space consumption. As a drastic example, consider a clause C of the form $P_1(x_1) \vee \cdots \vee P_n(x_n)$ and assume a signature that includes m constants. There are clearly more than m^n different instances of C, and there seems no principle way to avoid including that many of them in the set of instances of input clauses. This is, because by nature of instance-based methods, clause subsumption cannot be used.

5.3 Resolution Methods

Resolution calculi are conceptually very different to the $\mathcal{ME}$ calculus, which makes a comparison difficult. However, a common feature concern model generation: contemporary completeness proofs for resolution calculi are typically of the model-generation style: in the proofs it is argued that any saturated clause set that does not contain the empty clause is satisfiable, and a construction is supplied how to extract a model then (see [BG01]). However, this is a *conceptual construction*, and in order to actually extract a model from a failed refutation some non-trivial postprocessing is necessary (but see [GMW97]). Typically, a model is computed by enumerating all true *ground* literals, thereby interleaving this enumeration with calls to the resolution procedure again in order to determine the "next" ground literal [FL93, FL96]. On the other side, it should be said that some of the strongest decision procedures for subclasses of clause logic are based on resolution, however without outputting a model.

5.4 Tableau Methods

(Clausal) tableau methods that are also instance-based methods have been discussed above. Clausal tableau calculi different to those are also related to $\mathcal{ME}$, as, in general, tableau calculi share with $\mathcal{ME}$ the property of encoding in an exhausted open branch a model of the given clause set. For the purpose of comparison , it is useful to classify the former with regard to their treatment of variables: *rigid* or *ground-level oriented*.

In tableau calculi with *rigid variables*, a variable in a tableau is a placeholder for a (one) not-yet-known term (see e.g. [Fit90] for a basic version). The meaning of rigid variables can also be captured by constraints [Pel99, Gie01, vE01, see]. Although (most) tableau calculi are proof confluent, practically usable fair strategies

to achieve proof convergence (cf. Corollary 4.27) are hardly known (but see [Bec00]). A drawback of the rigid variable approach is that useful redundancy mechanisms are very hard to find, for instance, one that would in general prevent to have an unbounded number of variants of the same literal along a branch. For instance, given the unit clause $P(x)$, there seems no simple justification for not enumerating variants $P(x), P(x'), P(x''), \ldots$ of $P(x)$ along a branch. In fact, in general, one variant will not be enough, unlike to $\mathcal{ME}$, and it is difficult to (automatically) determine sharp bounds on the number of variants required for completeness. See [Gie02] for discussion on various options one has in the design of rigid tableau calculi, including constraint based approaches.

Ground-level tableau calculi avoid the problems with rigid variables by recurring to the propositional level. While analytic tableau with the "classical" γ-rule do not seem a suitable basis to build competitive theorem provers, there are structural refinements for clause logic that are also related to hyper resolution [MB88, BFN96, e.g.] or to Ordered Semantic Hyper Tableaux [YP02]. However, these methods suffer from an, in general unavoidable, don't-know nondeterminism. It may lead to an enumeration of the whole Herbrand base along a branch.

Compared to $\mathcal{ME}$, tableau calculi (also the instance-based tableau calculi) branch on subformulas, or, the literals of a clause in the clausal logic case, as opposed to complementary literals like $\mathcal{ME}$ does. For the propositional case it is easy to see that branching on complementary literals is more general than branching on clauses. In fact, each branching on a clause with n literals can be simulated by n splits with complementary literals. Furthermore, some improvements like factoring (see [LMG94]) are *automatically* realized by the branching on complementary literals approach. A systematic investigation on how this fact exactly carries over to the first-order case—i.e. $\mathcal{ME}$ vs. certain clausal tableau calculi—is left for future work.

6 Conclusions

In this paper we have introduced the Model Evolution ($\mathcal{ME}$) calculus. The $\mathcal{ME}$ calculus extends (the propositional part of) the DPLL procedure to first-order clause logic by supplying unification-based, first-order versions of DPLL's inference rules. Compared to its most immediate predecessor, FDPLL [Bau00], $\mathcal{ME}$ is a more faithful lifting of DPLL to first-order clause logic, as it also includes first-order versions of the propositional DPLL inference rules for unit propagation, which are not present in FDPLL.

Further Work

Various directions for further work are conceivable. The following list is ordered from the more concretely reachable to the more remote and research intensive.

Mixed Literals. The inference rules of the $\mathcal{ME}$ calculus make sure that only literals that are parameter-free or variable-free are inserted into contexts. "Mixed" literals with parameters and variables presently occur in $\mathcal{ME}$ only temporarily, during the computation of branch unifiers. Although reasoning with mixed literals seems a novel feature, and not realized in related calculi, its full potential is not exploited in the current version of the $\mathcal{ME}$ calculus. A conceivable improvement would involve admitting mixed literals in contexts, allowing then single variables to be singled out as universal, as opposed to entire literals as it is now.

This will be a non-trivial extension, however. While it is straightforward to modify the formal framework of this paper to admit mixed literals, some properties will be lost, or at least it is not clear if they are preserved under the relaxation. For instance, we found neither a proof of nor a counterexample against Proposition 3.8 for (non-contradictory) contexts with possibly mixed literals. This proposition, however, is a fundamental one: it implies that every ground atom will receive a truth value, and it cannot be dispensed with. With a modified definition of productivity, we were able to prove Proposition 3.8, but it was unclear if consistency of contexts would be achievable (by Commit applications).

Strategies. The $\mathcal{ME}$ calculus is proof convergent (cf. Corollary 4.27), and so the order of rule applications does not matter. This *don't-care* nondeterminism can be exploited to have the calculus stepwise simulate certain other calculi such as, e.g., the propositional logic oriented Tableau calculi [Bau98, YP02] or the Hyper Tableaux calculi in [Bau98]. Beyond the simulation as such, we expect that improved versions of some clausal tableau calculi can be obtained this way.

$\exists\forall$-Based Splitting. From a logical point of view, when used with a variable-free literal, $p(u)$ say, the Split rule in $\mathcal{ME}$ is a cut on a tautology of the form $\exists x\ p(x) \vee \exists x\ \neg p(x)$. It seems possible to build a complete calculus that instead splits on the tautology $\forall x\ p(x) \vee \exists x\ \neg p(x)$. In concrete, the resulting Split rule would choose between $p(x)$ and $\neg p(c)$ for some fresh Skolem constant c. However, the constant c would now have to fill two rôles: that of a constant proper, for the purpose of closing branches, and that of parameter (in sense of this paper) for the purpose of identifying falsified clauses. We speculate, however, that the resulting calculus would behave rather differently from the $\mathcal{ME}$ calculus presented here, due to the non-symmetrical nature of its Split rule.

$\mathcal{ME}$ as a decision procedure. A deduction system capable of deciding relevant classes of formulas is usually of greater practical interest than a mere refutation system (e.g. to disprove false "theorems" in a software verification context).

The $\mathcal{ME}$ calculus is guaranteed to terminate for clauses resulting from the translation to clausal form of conjunctions of Bernays-Schönfinkel formulas[17] and hence

[17] Such clauses contain no function symbols, but no other restrictions apply.

gives a decision procedure[18]. The same holds for many instance-based methods (see Section 5.2), but, interestingly, not for any known refinement of the resolution calculus. On the other side, there are refinements of the resolution calculus as decision procedure (see [FLHT01]) for classes of formulas that are not obviously decidable by instance-based methods or by the $\mathcal{ME}$ calculus. For instance, the guarded fragment of first-order logic, which is of high practical relevance as it covers (basic) description logic, was shown to be decidable by a refinement of resolution [dN98]. To decide the clause sets that result from the translation of formulas from the guarded fragment, ordered resolution can be used. The ordering refinements chosen will guarantee that the literals in the resolvent have a term complexity (using a certain measure) that does not exceed that of the parent clauses. It seems promising to pick up that strategy and attempt an $\mathcal{ME}$ decision procedure based on it.

Combining $\mathcal{ME}$ with Resolution. The $\mathcal{ME}$ calculus and, say, ordered resolution calculi are based on rather different principles. It can therefore be expected that an $\mathcal{ME}$ proof procedure spawns a very different search space than typical resolution proof procedures. Furthermore, we speculate that $\mathcal{ME}$ will find proofs that are difficult for resolution and vice versa. This conjectured property, however, motivates the design of a *combined* calculus, one that has inference rules from both worlds. Ideally, the resulting calculus would instantiate to both its parents, but would supply means to specify a "mixed" mode, where derivations use both resolution and $\mathcal{ME}$ inference rules to the favor of tuning the search space.

Indeed, some attempts have been made along these lines. For the DPLL method, its combination with restricted forms of (propositional) resolution have been proposed. For instance, in [DFW02] it is considered to apply resolution inferences such that the resolvent subsumes one (or both) of its parent clauses.

We are aware of only three proposals for integrating splitting techniques into first-order resolution systems. The theorem prover Otter [McC94] includes a heuristically controlled binary split rule that branches with two complementary ground literals, and an n-ary split rule that constructs one case for each literal in a ground clause. Both types of splitting are realized by forking the current Otter process on the operating system level. A tighter integration of a split rule, is realized in the SPASS prover [WAB+99]. The technique there permits, essentially, to break a clause with variable disjoint subclauses into these subclauses and search for refutations with them individually. A similar effect is achieved in the Vampire prover through a certain transformation of the input clause set [RV00].

That not much work has been done else on integrating splitting techniques into resolution systems is due to the reported incompatibility of the state-of-the-art resolution *implementations* with splitting: splitting involves backtracking to a previously derived clause set, which is difficult to implement in the saturation based

[18] This is an easy consequence of the fact that there are no two p-variants of a literal on any sequent derivable by $\mathcal{ME}$ (cf. Lemma 4.12).

resolution theorem provers. We consider it as an open problem if a more powerful splitting rule than the ones currently used, like the $\mathcal{ME}$ splitting rule, can help to build a faster resolution prover.

From the technical side, an extension of the $\mathcal{ME}$ calculus by resolution facilities seems not so far away: $\mathcal{ME}$ maintains a set of current clauses, and it would be a simple exercise to just add to $\mathcal{ME}$ resolution inference rules. The challenge, however, is to design the fairness appropriately, such that the $\mathcal{ME}$ and the resolution inference rules can be flexibly balanced.

Equational Theories and Equality. In many theorem proving applications, a proper treatment of equational theories or equality is mandatory. In principle, there seems to be nothing against a modern treatment of equality in $\mathcal{ME}$ by means of a superposition-style inference rule and of simplification rules based on rewriting [BG98].

Nonmonotonic Extensions. One of us is currently working on a calculus for a first-order clause logic with a default negation operator [BGHS03]. The underlying representation of interpretations has some resemblance with the one used for $\mathcal{ME}$. An interesting question is whether $\mathcal{ME}$ itself can be modified to accommodate such logic. The resulting calculus would be useful promising for applications demanding nonmonotonic knowledge representation languages.

A Appendix

This appendix contains auxiliary lemmas, their proofs, and proofs of the results stated in the main part of this paper. It is structured in three parts. Section A.1 is a collection of results about contexts in general, not necessarily about contexts as they evolve in derivations. Exactly the latter is the subject of Section A.2. The subsequent Section A.3 then contains lemmas stating conditions under which the mandatory inference rules of $\mathcal{ME}$ are applicable. The results collected up to then were employed in Section 4.3 to prove our main theorem, the completeness of $\mathcal{ME}$.

A.1 Properties of Contexts

The first lemma is not concerned with contexts; it will be needed, however, for some proofs below.

Lemma A.1 *For any literal L, the sets $\{K \mid K \geq L\} / \simeq$ and $\{K \mid K \gtrsim L\} / \simeq$ are finite.*

That is, for a given literal L, there are only finitely many more general literals wrt. $\geq$ of L modulo p-variantship, and similarly for $\gtrsim$. A similar result formulated in terms of $\gtrsim$ and $\approx$ is proven in [Ede85].

Proof. Since $K \gtrsim L$ whenever $K \geq L$, it is enough to show that the set $\{K \mid K \gtrsim L\} / \simeq$ is finite.

In the following argumentation we will prove the claim using a tree representation of literals: if L is a literal, its tree representation is the (ordered) tree, the root of which is labeled with the predicate symbol of the literal, inner nodes are labeled with function symbols, and leaf nodes are labeled with constant or variable symbols, all in the obvious way.

Recall that $K \geq L$ means there is a (p-preserving) substitution σ such that $K\sigma = L$. This means that the tree for L is obtained by replacing each variable leaf node x in the tree for K by the tree for $x\sigma$, and similarly for parameters. Clearly, the number of nodes in K is less than or equal to the number of nodes in L.

Now let $\{x_1, \ldots, x_n\} \subset X$ and $\{u_1, \ldots, u_n\} \subset V$ be finite sets of any n pairwise different variables and any n pairwise different parameters, respectively,where n is the number of nodes of the tree representation of L.

Let K be any literal such that $K \gtrsim L$ and such that K contains variables and parameters from the finite sets just mentioned only. Because the number of nodes in K is less than or equal to the number of nodes in L, it follows together with the assumed finite sets of variables and parameters that only finitely many such literals $K \gtrsim L$ exist. Let $\mathbf{K}$ be the finite set of all these literals.

Notice that $\mathbf{K}$ is finite also if the signature under consideration contains *infinitely* many function symbols. This holds, because K cannot contain any function symbol not occurring in L (because then K could not be instantiated to L), and there occur only finitely many function symbols in L.

Clearly, every literal K with $K \gtrsim L$ is a $\simeq$-variant of some literal in $\mathbf{K}$. Therefore, with $\mathbf{K}$ being finite, so is $\{K \mid K \gtrsim L\} / \simeq$. □

Proposition 3.8 *Let Λ be a non-contradictory context. Then, I_Λ is a preinterpretation.*

This proposition expresses, in other words, that a non-contradictory context Λ produces *at least* one of L or $\overline{L}$, for any ground literal L. In connection with the subsequent Proposition A.3 then, and provided that Λ is consistent, it follows that Λ produces *exactly* one of L or $\overline{L}$, for any ground literal L.

Proof. Let L be any literal, not necessarily ground. We will show that Λ produces L or Λ produces $\overline{L}$. From this, the claim follows immediately.

Due to the presence of the pseudo-literal $\neg v$ in Λ, Λ must contain an msg of L or of $\overline{L}$ in Λ (viz., $\neg v$ if there is no other such msg). Now let $K \in \Lambda$ be such an msg of L or of $\overline{L}$ in Λ. Beyond having K chosen as an msg, we may also assume having chosen K such that there is no literal $K' \in \Lambda$ with $K \gtrsim \overline{K'} \gtrsim L$ or $K \gtrsim \overline{K'} \gtrsim \overline{L}$. This is possible, because there are only finitely many literals K' (modulo renaming) such that $\overline{K'} \gtrsim L$ or $K' \gtrsim L$ (cf. Lemma A.1), and because the relation $\gtrsim$ is a strict, partial ordering, and hence does not admit cycles.

For reasons of symmetry we consider in the sequel only the case that $K \in \Lambda$ is a msg of L in Λ and that there is no $K' \in \Lambda$ such that $K \gtrsim \overline{K'} \gtrsim L$.

Now, if K produces L in Λ the proof is complete. Otherwise, there must be a literal $K' \in \Lambda$ and a p-preserving substitution σ such that $K \gtrsim \overline{K'\sigma} \gtrsim L$. We consider two complementary cases, the first one of which will be impossible to hold, however.

In the first case K' is variable-free. Since σ is p-preserving this implies trivially $K' \simeq K'\sigma$. But then, from $K \gtrsim \overline{K'\sigma} \gtrsim L$ we conclude immediately $K \gtrsim \overline{K'} \gtrsim L$. This, however contradicts the the choice of K. Hence this case is impossible.

Thus, the second case must hold, where K' is not variable-free. This entails that K' is parameter-free. Trivially then, $\overline{K'} \gtrsim L$ implies $\overline{K'} \geq L$, and $K' \geq \overline{L}$ follows trivially, too. But then, Lemma A.5 gives immediately that Λ produces $\overline{L}$.

In sum, in any case we have now shown that Λ produces L or Λ produces $\overline{L}$. □

Proposition A.3 *Let Λ be a non-contradictory context. Then, I_Λ is an interpretation iff Λ is consistent.*

Proof. The only-if part being trivial, we turn immediately to the if-part. Hence assume that Λ is consistent.

Let L be any ground literal. We have to show that either $L \in I_\Lambda$ or $\overline{L} \in I_\Lambda$ (but not both). By definition, $L \in I_\Lambda$ iff Λ produces L. Since Λ is consistent, Λ cannot produce both L and $\overline{L}$.

It remains to show that Λ produces L or Λ produces $\overline{L}$. Since Λ is consistent, it is not contradictory. But then the claim follows immediately with Lemma 3.8. □

Lemma A.4 *Let Λ be a context and L a literal. If $\overline{L} \in_\geq \Lambda$, then L is contradictory with Λ.*

Since the calculus works on non-contradictory contexts only, this Lemma implies that no context will contain a literal and an instance of it (wrt. $\geq$) with complementary sign.

Proof. Suppose that $K \in \Lambda$ and $K \geq \overline{L}$ holds. We have to show that L is contradictory with Λ.

We distinguish two cases. In the first case, K is parameter-free. Let $K' \simeq K$ be a fresh p-preserving variant of K. Clearly, K' is parameter-free as well, and with $K \geq \overline{L}$ it follows $K' \geq \overline{L}$. Let σ be a substitution such that $K'\sigma = \overline{L}$. Clearly, σ may be assumed to move only the variables of K' (but no parameters). Since K' is fresh, σ will not modify L, and so $L = L\sigma$ follows. Altogether then $K'\sigma = \overline{L}\sigma$. Since $K' \in_\simeq \Lambda$, L is contradictory with Λ.

In the second case, K is variable-free. That $K \geq \overline{L}$ holds means there is a p-preserving substitution σ such that $K\sigma = \overline{L}$. Since σ is a renaming on V and K contains no variables, $K \simeq K\sigma$ follows. Since $K \in \Lambda$ it follows $K\sigma \in_\simeq \Lambda$. Since we know $K\sigma = \overline{L}$ from above, L is contradictory with Λ. □

Lemma A.5 *Let Λ be a non-contradictory context and $K \in \Lambda$ a parameter-free literal. Then, for any literal L with $K \geq L$,*

(i) Λ produces L, and

(ii) Λ does not produce $\overline{L}$.

Observe that if K is parameter-free, then $K \geq L$ iff $K \gtrsim L$. Hence, in the lemma statement, $K \geq L$ could be replaced by $K \gtrsim L$. But then, equivalently, the lemma expresses that a (non-contradictory) context containing a parameter-free literal K produces every instance L of K. In other words, Lemma A.5 corresponds to the fact that all instances of a parameter-free literal in a non-contradictory context are true in the (pre)interpretation induced by the context.

Note that Lemma A.5 does not say that K itself produces L. In fact, this is not true in general, as one can easily see by considering $\Lambda = \{\neg v, P(x,y), P(u,u)\}$, $K = P(x,y)$ and $L = P(a,a)$.

Proof. Let L be a literal such that $K \geq L$. First we show that Λ produces L.

Let $K' \in \Lambda$ be a msg of L in Λ, such that $K \gtrsim K' \gtrsim L$ holds (the case $K = K'$ is possible). We are going to show that moreover K' produces L in Λ.

Suppose to the contrary that this is not the case. Then there is a literal $K'' \in \Lambda$ and a p-preserving substitution σ such that $K' \gtrsim \overline{K''\sigma} \gtrsim L$. From $K \gtrsim K'$ it follows that $K \gtrsim \overline{K''\sigma} \gtrsim L$. Since K is parameter-free this is equivalent to $K \geq \overline{K''\sigma} \gtrsim L$.

We distinguish two cases. If K'' is variable-free, consider $\sigma' = \sigma_{|V}$, which is a p-preserving renaming. We also need the substitution σ^{-1}, which is a p-preserving renaming, too. Observe that from $K \geq \overline{K''\sigma}$ it follows $K\sigma^{-1} \geq \overline{K''\sigma}\sigma^{-1} = \overline{K''}$. Since K'' is parameter-free, it holds $K''\sigma' = K''\sigma$. Since K is parameter-free, $K = K\sigma'^{-1}$ holds as well. With $K\sigma^{-1} \geq \overline{K''}$ we get $K \geq \overline{K''}$. Since $K \in \Lambda$, this means $\overline{K''} \in_{\geq} \Lambda$. By Lemma A.4 then, K'' is contradictory with Λ. Since $K'' \in \Lambda$, Λ itself is contradictory, against the assumption that is is not.

In the second case, if K'' is not variable-free, it must be parameter-free. Above we derived the chain $K \geq \overline{K''\sigma} \gtrsim L$. Assume that L is disjoint with both K and K'' (if this is not the case, let L itself denote an appropriate variant, and the chain will still hold). Since K'' is parameter-free, the chain entails $K \geq \overline{K''\sigma} \geq L$. Because of $\overline{K''\sigma} \geq L$, there is a substitution δ such that $\overline{K''\sigma}\delta = L$. Let $\sigma'' := \sigma\delta_{|\mathcal{V}ar(K'')}$, and it will hold $\overline{K''\sigma''} = L$. Now let $K' \in_{\simeq} \Lambda$ be a fresh p-variant of K. Since K is parameter-free, K' will be parameter-free as well. From $K \geq L$ and $K' \simeq K$ we conclude that there is a substitution σ' such that $K'\sigma' = L$. We may assume that σ' is already restricted to $\mathcal{V}ar(K')$. Since L is disjoint with K'', σ'' will not affect L, i.e. $L = L\sigma''$ holds. From $K'\sigma' = L$ it follows $K'\sigma'\sigma'' = L$. Since K' is fresh, σ' will not affect K'', i.e. $K'' = K''\sigma'$ holds. With $\overline{K''}\sigma'' = L$ it follows $\overline{K''}\sigma'\sigma'' = L$, and with $K'\sigma'\sigma'' = L$ conclude $\overline{K''}\sigma'\sigma'' = K'\sigma'\sigma''$. Since both σ' and σ'' move only variables, $\sigma'\sigma''$ is trivially p-preserving. Since $K' \in_{\simeq} \Lambda$ and $\overline{K''}\sigma'\sigma'' = K'\sigma'\sigma''$ holds, K'' is contradictory with Λ. Since $K'' \in \Lambda$, Λ itself is contradictory, which

plainly contradicts the lemma statement in this case, too. Hence, the assumption that K' does not produce L in Λ must be retracted, and the first claim of the lemma follows.

It remains to show that Λ does not produce $\overline{L}$ in Λ. Let $K' \in \Lambda$ be any literal such that $K' \gtrsim \overline{L}$. (If such a literal does not exist, Λ cannot produce $\overline{L}$). We will show that K' does not produce $\overline{L}$ in Λ.

Now, if not only $K' \gtrsim \overline{L}$ but the strong claim $K' \gtrsim\!\!\!\!\!\!\approx \overline{L}$ holds, then K' cannot produce $\overline{L}$. This is, because K and δ prevent K' from doing so, as is seen by the chain $K' \gtrsim\!\!\!\!\!\!\approx \overline{K}\delta = \overline{L} \gtrsim \overline{L}$. Hence, suppose from now on $K' \approx \overline{L}$.

From $K \geq L$, as given, it follows $K \gtrsim L$, and with $K' \approx \overline{L}$ we get $K \gtrsim \overline{K'}$. Since K is parameter-free this is equivalent to $K \geq \overline{K'}$. Since $K \in \Lambda$, this means in other words $\overline{K'} \in_{\geq} \Lambda$. By Lemma A.4 then, K' is contradictory with Λ. Furthermore, since $K' \in \Lambda$, Λ itself is contradictory. However, Λ was given as non-contradictory in the lemma statement. From this contradiction it follows that the case $K' \approx \overline{L}$ cannot hold. Thus we have excluded all possibilities for K' to produce $\overline{L}$ in Λ, which remained to be shown. □

Lemma A.6 *Let Λ be a non-contradictory context, $K, K' \in \Lambda$, and L a literal. If K produces L in Λ and K' produces $\overline{L}$ in Λ, then both K and K' are variable-free.*

This lemma will be important to prove that the Commit inference rule is applicable whenever it should be, namely if a contexts produces both a literal and its complement.

Proof. For reasons of symmetry it suffices to prove that K is variable-free. That K produces L in Λ means in particular that $K \gtrsim L$. Now, if K were not variable-free, it would be parameter-free. In this case $K \gtrsim L$ means the same as $K \geq L$. But then, with Lemma A.5 conclude that Λ does not produce $\overline{L}$, contradicting what was given in the lemma statement. □

Lemma A.7 *Let Λ be a context and K, K', L literals. If K produces L in Λ and $K \gtrsim K' \gtrsim L$ then K produces K' in Λ.*

Proof. Suppose that K produces L in Λ and $K \gtrsim K' \gtrsim L$. We will show that K produces K' in Λ.

Clearly, K is a msg of K' in Λ (for, if it were not, K would not be a msg of L either, contradicting that K produces L in Λ). Now, if K would not produce K' in Λ, then there is a $K'' \in_{\geq} \Lambda$ such that $K \gtrsim\!\!\!\!\!\!\approx \overline{K''} \gtrsim K'$. But with $K' \gtrsim L$ it would follow $K \gtrsim\!\!\!\!\!\!\approx \overline{K''} \gtrsim L$, and so K would not produce L in Λ either.

□

The next two lemmas complement each other. Their prerequisites mean that L is contradictory with Λ, as witnessed by a literal $K \in_{\simeq} \Lambda$. The first lemma considers the case that L is parameter-free, and the second lemma considers the case that L is variable-free. Both lemmas express how a literal $K' \simeq K$ can take the rôle of K.

Lemma A.8 *Let Λ a context, $K \in_\simeq \Lambda$, L a parameter-free literal, and σ a p-preserving substitution such that $L\sigma = \overline{K}\sigma$. For all literals K' disjoint with L such that $K' \simeq K$, there is a p-preserving substitution σ' such that $L\sigma' = \overline{K'}\sigma'$ and $\mathcal{D}om(\sigma') \cap V = \emptyset$.*

Proof. Let K' be any p-preserving variant of K that is disjoint with L. This means there is a p-preserving renaming δ such that $K'\delta = K$. Let $\delta' = \delta_{|\mathcal{V}ar(K') \cup V}$. Clearly, δ' is p-preserving (but not necessarily a renaming on X) and it holds $K'\delta' = K$. With $L\sigma = \overline{K}\sigma$ we therefore get $L\sigma = \overline{K'}\delta'\sigma$. Since L is parameter-free and disjoint with K', it is easy to see that $L = L\delta'$ holds. Thus, from $L\sigma = \overline{K'}\delta'\sigma$ it follows $L\delta'\sigma = \overline{K'}\delta'\sigma$.

Let $\rho = (\delta'\sigma)_{|V}$ be the restriction of $\delta'\sigma$ to parameters. Since both δ' and σ are p-preserving, $\delta'\sigma$ is p-preserving, and so ρ is a renaming on V. Therefore, the substitution ρ^{-1} exists, and from $L\delta'\sigma = \overline{K'}\delta'\sigma$ it follows trivially $L\delta'\sigma\rho^{-1} = \overline{K'}\delta'\sigma\rho^{-1}$. In other words, setting $\sigma' := \delta'\sigma\rho^{-1}$ proves the first claim of the lemma.

Concerning the second claim, note that by construction of ρ it holds $u\rho = u\delta'\sigma$, for any parameter u, and so $u = u\rho\rho^{-1} = u\delta'\sigma\rho^{-1}$ follows immediately. In other words $\mathcal{D}om(\delta'\sigma\rho^{-1}) \cap V = \emptyset$, which is equivalent to $\mathcal{D}om(\sigma') \cap V = \emptyset$. Finally notice that $\delta'\sigma\rho^{-1} = \sigma'$ trivially is p-preserving, as claimed in the lemma statement. □

Lemma A.9 *Let Λ be a context, $K \in_\simeq \Lambda$, L a variable-free literal, and σ a p-preserving substitution such that $L\sigma = \overline{K}\sigma$. (In other words, L is contradictory with Λ). For all literals K' such that $K' \simeq K$, there is a p-preserving substitution σ' such that $L = \overline{K'}\sigma'$.*

Proof. Let K' be any p-preserving variant of K. This means there is a p-preserving renaming ρ' such that $K'\rho' = K$.

Let $\rho = \sigma_{|V}$ be the restriction of σ to parameters. Since σ is given as a p-preserving substitution, ρ is a renaming on V. Therefore, the substitution ρ^{-1} exists, and it holds trivially $L\sigma\rho^{-1} = \overline{K}\sigma\rho^{-1}$.

Since each substitution ρ', σ and ρ^{-1} is p-preserving, $\rho'\sigma\rho^{-1}$ is p-preserving, too. Since L is variable-free, it follows $L\sigma = L\rho$, and so $L\sigma\rho^{-1} = L\rho\rho^{-1} = L$. Together, thus, $L = \overline{K}\sigma\rho^{-1}$. Using the equality $K'\rho' = K$ from above, it follows $L = \overline{K}\rho'\sigma\rho^{-1}$. Therefore, setting $\sigma' := \rho'\sigma\rho^{-1}$ proves the lemma. □

A.2 Evolving Contexts

Quite often we will appeal to the following compactness property of $\Lambda_\mathbf{B}$. By definition, $L \in \Lambda_\mathbf{B}$ holds iff there is an $i < \kappa$ such that $L \in \Lambda_j$ for all $j \geq i$ with $j < \kappa$. Similarly, if $L \in_\simeq \Lambda_\mathbf{B}$ then, by definition, $L \simeq K$ for some literal $K \in \Lambda_\mathbf{B}$. Again, there is an $i < \kappa$ such that $K \in \Lambda_j$ for all $j \geq i$ with $j < \kappa$, which entails that $L \in_\simeq \Lambda_j$, for all $j \geq i$ with $j < \kappa$. More generally then, if $L_1, \ldots, L_n \in \Lambda_\mathbf{B}$ (or

$L_1, \ldots, L_n \in_\simeq \Lambda_\mathbf{B}$) for some $n \geq 0$, then there is an $i < \kappa$ such that $L_1, \ldots, L_n \in \Lambda_j$ (or $L_1, \ldots, L_n \in_\simeq \Lambda_j$) for all $j \geq i$ with $j < \kappa$.[19]

Lemma 4.7 *For all $i < \kappa$, Λ_i is not contradictory.*

Proof. The proof is by induction on i. For the base we have $\Lambda_0 = \{\neg v\}$ and this set is trivially not contradictory. For the induction step we take as induction hypothesis the claim of the lemma. Observe that each inference rule that extends a context includes as an applicability condition that the resulting context(s) is (are) not contradictory. With this observation the induction step follows immediately. □

Lemma 4.8 *$\Lambda_\mathbf{B}$ is not contradictory.*

Proof. Suppose that $\Lambda_\mathbf{B}$ is contradictory. Then there are literals $L \in \Lambda_\mathbf{B}$ and $K \in_\simeq \Lambda_\mathbf{B}$ and there is a p-preserving substitution σ such that $L\sigma = \overline{K}\sigma$. By the compactness property, there is a j such that both $L \in \Lambda_j$ and $K \in_\simeq \Lambda_j$. By virtue of the substitution σ, Λ_j is contradictory then. However, this is impossible by Lemma 4.7. □

Lemma 4.9 *The sequent $\Lambda,\ L \vdash \Psi$ is not derivable from $\Lambda \vdash \Psi$ if $L \in_\geq \Lambda$ or $\overline{L} \in_\geq \Lambda$.*

Proof. It suffices to consider potential applications of the Split, the Assert or of the Commit inference rule to Λ, because these are the only rules that can extend a context.

Split) Recall that the Split rule is applicable only if neither K nor $\overline{K}^{\text{sko}}$ is contradictory with Λ, where K is the remainder literal to split with. We consider two cases, corresponding to the case that the literal L in the lemma statement is K or is $\overline{K}^{\text{sko}}$. In both cases we will show that Split is not applicable by showing that K or $\overline{K}^{\text{sko}}$ is contradictory with Λ.

In the first case the literal L in the lemma statement is K. That $K \in_\geq \Lambda$ holds means there is a literal $K' \in \Lambda$ and a p-preserving substitution σ such that $K'\sigma = K$.

If K is variable-free, since σ is p-preserving and hence a renaming on the parameters, $K'\sigma = K$ is equivalent to $K' \simeq K$. With $K' \in \Lambda$ it follows by definition that $K \in_\simeq \Lambda$. Since K is variable-free this implies trivially $\overline{K}^{\text{sko}} = \overline{K}$. But then, by taking the empty substitution one sees that $\overline{K}$ is contradictory with Λ. If K is not variable-free, K must be parameter-free (this follows immediately from the definition of admissible context unifier and the fact that K is a remainder literal). Let μ be the Skolemizing substitution used, i.e. the substitution μ such that $K\mu = K^{\text{sko}}$.

[19] It is easy to see that this index i can be determined by taking the maximum of the i-indices associated individually to the literals $L_1, \ldots, L_n$, as just described.

From $K'\sigma = K$ it follows trivially that $K'\sigma\mu = K\mu$. Since $\sigma\mu$ is p-preserving, we then have that $K' \geq K\mu$. With $K\mu = K^{\text{sko}}$ we get $K' \geq K^{\text{sko}}$, or equivalently $K' \geq \overline{K^{\text{sko}}}$. Since $K' \in \Lambda$, in other words, $\overline{K^{\text{sko}}} \in_{\geq} \Lambda$. But now, by Lemma A.4, $\overline{K}^{\text{sko}}$ is contradictory with Λ, too. This completes the proof for the first case.

In the second case the literal L in the lemma statement is $\overline{K}^{\text{sko}}$. That $\overline{K}^{\text{sko}} \in_{\geq} \Lambda$ holds means there is a literal $K' \in \Lambda$ and a p-preserving substitution σ such that $K'\sigma = \overline{K}^{\text{sko}}$.

If K is variable-free, the proof is almost identical to the one above: since σ is p-preserving and hence a renaming on the parameters, $K'\sigma = \overline{K}^{\text{sko}}$ is equivalent to $K' \simeq \overline{K}^{\text{sko}}$. With $K' \in \Lambda$ it follows by definition $\overline{K}^{\text{sko}} \in_{\simeq} \Lambda$. Since K is variable-free this implies trivially $\overline{K}^{\text{sko}} = \overline{K}$. But then, by taking the empty substitution one sees that K is contradictory with Λ. If K is not variable-free, as said above, K must be parameter-free. Let μ be the Skolemizing substitution used, i.e. the substitution μ such that $K\mu = K^{\text{sko}}$. It can be written as $\mu = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$, where $x_1, \ldots, x_n$ are all the variables occurring in K, and $a_1, \ldots, a_n$ are fresh constants. Now, because the constants $a_1, \ldots, a_n$ are fresh, none of them will occur in K. This means that we can consider the "substitution" $\mu' = \{a_1 \mapsto x_1, \ldots, a_n \mapsto x_n\}$ and have that $K = K\mu\mu' = K^{\text{sko}}\mu'$. From $K'\sigma = \overline{K}^{\text{sko}}$ it follows trivially $K'\sigma\mu' = \overline{K}^{\text{sko}}\mu'$. Together with $K = K^{\text{sko}}\mu'$ we get easily $K'\sigma\mu' = \overline{K}$. Since $K' \in \Lambda$, in other words, $\overline{K} \in_{\geq} \Lambda$. But now, by Lemma A.4, K is contradictory with Λ, too. This completes the proof for the second case.

Together, thus, K or $\overline{K}^{\text{sko}}$ is contradictory with Λ, which remained to be shown.

Assert) The proof is immediate from the applicability condition from Assert, which explicitly demands that there is no $K \in \Lambda$ such that $K \geq L$.

Commit) Because of Commit's applicatibility conditions, the literal L in the lemma's statement must be variable-free. The proof for this case is similar to the subcase in the proof for of Split, where the literal K mentioned there, and hence also $\overline{K}^{\text{sko}}$, is variable-free. □

In the course of the development of a branch, a literal in a sequent's context may be deleted by means of the Compact rule. Such a deletion is only possible in presence of a p-subsuming literal, which takes the rôle of the deleted literal. This process may continue and is formalized in the following definition.

Definition A.13 *Let K be a literal. For all $i < \kappa$, if $L \in \Lambda_i$ then the* trace of L from Λ_i *is the sequence $(L^j)_{i \leq j < \kappa}$, where $L^j := L$ if $j = i$, and for all $j > i$,*

$$L^j := \begin{cases} K & \text{if Compact is applied with selected literal } L^{j-1} \text{ and} \\ & \text{subsuming literal } K \text{ to } \Lambda_{j-1} \vdash \Phi_{j-1} \text{ to obtain } \Lambda_j \vdash \Phi_j \\ L^{j-1} & \text{otherwise} \end{cases}$$

Lemma 4.10 *Let $i < \kappa$ and $L \in \Lambda_i$. For every j with $i \leq j < \kappa$ there is a $K \in \Lambda_j$ such that $K \geq L$.*

Proof. Consider the trace $(L^j)_{i \leq j < \kappa}$ of L from Λ_i. Each literal L^j from the trace, where $i \leq j < \kappa$, is contained in Λ_j, and $L^j \geq L$ holds by construction of the trace. □

Lemma 4.11 *For any two different literals $K, L \in \bigcup_{i<\kappa} \Lambda_i$, it holds that $K \not\simeq L$.*

Proof. Assume by way of contradiction that $K \simeq L$ for some different literals $K, L \in \bigcup_{i<\kappa} \Lambda_i$. Notice first that not both K and L can be pseudo-literals, i.e. of the form $\neg v$, because the context Λ_0 contains exactly one such pseudo-literal and the calculus has no inference rules to add (or to delete) those. If only one of K and L is a pseudo-literal, the lemma holds trivially. Hence, from now on assume that neither K nor L is a pseudo-literal.

There must be finite ordinals j and k such that $L \in \Lambda_j$ and $K \in \Lambda_k$. W.l.o.g. assume that $j \geq k$. Furthermore j and k may be chosen minimal, i.e. $L \notin \Lambda_{j-1}$ and $K \notin \Lambda_{k-1}$. This means that some inference rule is applied to the node N_{j-1} that extends the context Λ_{j-1} with L to obtain Λ_j (and similarly for K). Observe that the inference rules of $\mathcal{ME}$ extend the given context by at most one literal. In particular, K cannot have been added to Λ_{j-1} by the considered inference rule application. Thus, not only $j \geq k$ but also $j > k$ must hold.

But then, by Lemma 4.10 there is a literal $K' \in \Lambda_{j-1}$ such that $K' \geq K$. Together with $K \simeq L$ it follows immediately that $K' \geq L$. However, according to Lemma 4.9 the considered inference rule application that extends Λ_{j-1} by L is not possible. A plain contradiction. Thus, we must have $K \not\simeq L$. □

Lemma 4.12 *For all $i < \kappa$ and for any two different literals $K, L \in \Lambda_i$ it holds that $K \not\simeq L$.*

Proof. Because $\Lambda_i \subseteq \bigcup_{i<\kappa} \Lambda_i$ the result follows trivially from Lemma 4.11. □

Lemma 4.13 *For all $i < \kappa$ and $L \in \Lambda_i$ there is a $K \in \Lambda_{\mathbf{B}}$ such that $K \geq L$.*

Proof. Consider the trace $(L^j)_{i \leq j < \kappa}$ of L from Λ_i. All its consecutive different elements L_j and L_{j+1} are those where the Compact rule is applied to the sequent $\Lambda_j \vdash \Phi_j$ labeling the node N_j. That Compact is applied means $L_{j+1} \geq L_j$. Of course, both $L_{j+1} \in \Lambda_j$ and $L_j \in \Lambda_j$ must hold as well. By Lemma 4.12 we can conclude that $L_{j+1} \not\simeq L_j$. Together with $L_{j+1} \geq L_j$ this entails that $L_{j+1} \gneq L_j$.

In other words, the considered consecutive different elements from the trace determine a sequence of increasing literals wrt. $\gneq$. With Lemma A.1 it follows immediately that this sequence is finite. If the sequence is non-empty, let K be its last element. Otherwise, let $K = L$. In both cases it is easy to see that $K \in \Lambda_{\mathbf{B}}$ and $K \geq L$. □

Lemma 4.14 *For all $k < \kappa$ and $L \in \Lambda_k \setminus \Lambda_{\mathbf{B}}$, there is an $i > k$ such that for all j with $i \leq j < \kappa$, $L \not\simeq \Lambda_j$.*

Proof. Let $k < \kappa$ and $L \in \Lambda_k \setminus \Lambda_{\mathbf{B}}$. Since $L \notin \Lambda_{\mathbf{B}}$ there must be an $i > k$ such that $L \notin \Lambda_i$. Let i be the smallest such index. We prove that $L \not\simeq \Lambda_j$ for all j with $i \leq j < \kappa$ by induction on j.

$j = i$) From the minimality of i we know tha t Compact rule is applied to $\Lambda_{i-1} \vdash \Phi_{i-1}$ with selected literal L, and that L does not occur in Λ_i. By Lemma 4.12, there is no literal $K \in \Lambda_{i-1}$ different from L such that $K \simeq L$. This implies that $L \not\simeq \Lambda_i$.

$j \mapsto j + 1$) Let $K \in \Lambda_{i-1}$ be the subsuming literal of the Compact rule application mentioned in the proof of the case $j = i$. By definition of the Compact inference rule $K \geq L$ holds. By Lemma 4.10 there is a literal $K' \in \Lambda_j$ such that $K' \geq K$. Together it follows $K' \geq L$. Let $L' \simeq L$ be any variant, and it will hold $K' \geq L'$. But then, according to Lemma 4.9 an inference rule that would extend Λ_j with L' is not applicable. With $L' \simeq L$ this implies $L \not\simeq \Lambda_{j+1}$. Now, together with the induction hypothesis the result for the induction step follows immediately. □

Lemma 4.15 *Let K, L be two literals. If K produces L in $\Lambda_{\mathbf{B}}$, then for all $i < \kappa$ there is no $K' \in \Lambda_i$ and no p-preserving substitution σ such that $K \gtrsim \overline{K'}\sigma \gtrsim L$.*

Proof. We prove the lemma by proving its contrapositive. Suppose then that there is a literal $K' \in \Lambda_i$, for some i, and some substitution σ such that $K \gtrsim \overline{K'}\sigma \gtrsim L$. Then, from Lemma 4.13 it follows that there is a $K'' \in \Lambda_{\mathbf{B}}$ such that $K'' \geq K'$. Let σ' be the p-preserving substitution such that $K''\sigma' = K'$. From this, $K \gtrsim \overline{K''}\sigma'\sigma \gtrsim L$ follows immediately. Because both σ' and σ are p-preserving, $\sigma'\sigma$ is p-preserving as well. Hence $K \in \Lambda_{\mathbf{B}}$ does not produce L in $\Lambda_{\mathbf{B}}$. □

Lemma 4.16 *Let K, L be two literals with $K \in \Lambda_{\mathbf{B}}$. If K produces L in $\Lambda_{\mathbf{B}}$, then there is an i such that for all $j \geq i$ with $j < \kappa$, $K \in \Lambda_j$ and K produces L in Λ_j.*

Proof. Since $K \in \Lambda_{\mathbf{B}}$ there is a k such that $K \in \Lambda_j$, for all $j \geq k$. However, there is no guarantee that k is the index i we are looking for. Informally, it might be the case that a literal K' with the property $K \gtrsim K' \gtrsim L$ is added to the context Λ_k, or to some successor context, and deleted later again. In both cases, K' prevents K from producing L in the contexts containing K'. We will show that this process of adding and deleting literals like K' can happen only *finitely* often. Clearly, any index i at some timepoint after this process is finished will have the desired property.

More formally, let

$$M = \{K' \mid \text{there is an } m \geq k \text{ such that } K' \in \Lambda_m \text{ and } K \gtrsim K' \gtrsim L\}$$

be those literals that prevent K from being a msg of L in Λ_m, for some $m \geq k$.

Since $K' \gtrsim L$ for each $K' \in M$, the set $M \,/\, \simeq$ must be finite by Lemma A.1. Moreover, M is trivially a subset of $\bigcup_{i<\kappa} \Lambda_i$. Therefore Lemma 4.11 is applicable, and it gives us $K'_1 \not\simeq K'_2$ for any two different literals $K'_1, K'_2 \in M$. This means that each element of $M \,/\, \simeq$ is a singleton. In sum, $M \,/\, \simeq$ is a finite set of equivalence classes Therefore M itself is finite.

No literal K' from M is persistent, i.e. $K' \notin \Lambda_{\mathbf{B}}$ holds for each $K' \in M$. This is the case because otherwise $K \in \Lambda_{\mathbf{B}}$ would not produce L in $\Lambda_{\mathbf{B}}$ as assumed.

By applying Lemma 4.14 to each literal $K' \in M$, we can conclude from the finiteness of M that there is an index $i \geq k$ such that for all $j \geq i$ and for each $K' \in M$ it holds $K' \not\simeq \Lambda_j$ (k was chosen at the very beginning of the proof). In other words, K is a msg of L in Λ_j for all $j \geq i$.

We are given that $K \in \Lambda_{\mathbf{B}}$ produces L in $\Lambda_{\mathbf{B}}$. Since K is a msg of L in Λ_j for all $j \geq i$, as just derived, we can immediately conclude by Lemma 4.15 that K produces L in Λ_j for all $j \geq i$. To complete the proof it is enough to recall that $K \in \Lambda_j$ for all $j \geq k$ with $j < \kappa$ and that $i \geq k$. □

Lemma 4.17 *If $\Lambda_{\mathbf{B}}$ does not produce a literal L, then there is an i such that for all $j \geq i$, Λ_j does not produce L.*

Proof. Suppose that $\Lambda_{\mathbf{B}}$ does not produce L. Let

$$M = \{K \mid \text{there is a } m \geq 0 \text{ such that } K \in \Lambda_m \text{ and } K \text{ is a msg of } L \text{ in } \Lambda_m\} \ .$$

Only those literals are candidates to produce L in some eventually derived context. Since $K \gtrsim L$ for each $K \in M$, the set $M \,/\, \simeq$ must be finite by Lemma A.1. Moreover, M is trivially a subset of $\bigcup_{i<\kappa} \Lambda_i$. Therefore Lemma 4.11 is applicable, and it gives us $K_1 \not\simeq K_2$ for any two different literals $K_1, K_2 \in M$. This means that each element of $M \,/\, \simeq$ is a singleton. In sum, $M \,/\, \simeq$ is a finite set of finite equivalence classes Therefore M itself is finite.

Now let $K \in M$. By construction of M, $K \in \Lambda_m$ for some $m \geq 0$. We prove that there is an index i_K such that for all $j \geq i_K$, $K \notin \Lambda_j$ or Λ_j does not produce L in Λ_j. Since M is finite, this proof can be applied to all literals in $K \in M$ and the maximum of all the respective timepoints i_K exists. Let i be that timepoint. Because M contains all msg's of L in all contexts derived so ever, but, for all $K \in M$ and all $j \geq i$, $K \notin M$ or K does not produce L in Λ_j, it follows that Λ_j does not produce L, for all $j \geq i$, and so the proof will be complete. We consider two complementary cases.

If K is not persistent, i.e. $K \notin \Lambda_{\mathbf{B}}$, then from Lemma 4.14 it follows immediately that there is a i_K such that $K \notin \Lambda_j$, for all $j \geq i_K$.

If K is persistent, i.e. $K \in \Lambda_{\mathbf{B}}$, recall the given assumption that $\Lambda_{\mathbf{B}}$ does not produce L. This means that (i) K is not a msg of L in $\Lambda_{\mathbf{B}}$ or (ii) there is a literal $K' \in_{\geq} \Lambda_{\mathbf{B}}$ such that $K \gtrsim \overline{K'} \gtrsim L$. If (i) holds, because $K \gtrsim L$ by construction of M, there is a literal $K'' \in \Lambda_{\mathbf{B}}$ such that $K \gtrsim K'' \gtrsim L$. Since K'' is persistent, there is an index i_K such that $K'' \in \Lambda_j$, for all $j \geq i_K$. With $K \gtrsim K''$ it follows that K

does not produce L in Λ_j, for all $j \geq i_K$. If (ii) holds, there is a literal $K' \in \Lambda_{\mathbf{B}}$ and a p-preserving substitution σ such that $K \gtrsim \overline{K'}\sigma \gtrsim L$. Since K' is persistent, there is an index i_K such that $K' \in \Lambda_j$, for all $j \geq i_K$. With $K \gtrsim \overline{K'}\sigma \gtrsim L$ it follows that K does not produce L in Λ_j, for all $j \geq i_K$. □

A.3 Properties of Inference Rules

Lemma 4.18 (Commit Applicability) *Let $\Lambda \vdash \Psi$ be a sequent with a non-contradictory context Λ, and let $K, K' \in \Lambda$ be variable-free literals. If K produces L in Λ and K' produces $\overline{L}$ in Λ, then* Commit *is applicable to $\Lambda \vdash \Psi$ with selected connection (K, K') and some most general unifier σ. Moreover, $K \gtrsim K\sigma \gtrsim L$ and $K' \gtrsim K'\sigma \gtrsim \overline{L}$.*

Proof. Let Λ, K and L be as mentioned in the lemma statement. Observe that K produces L in Λ iff any variant of K produces any variant of L in Λ, and similarly for K' and $\overline{L}$. This follows immediately from the definition of productivity (Definition 3.5). Therefore, we can assume with no loss of generality that K, K' and L are all pairwise disjoint.

That K produces L in Λ means in particular that $K \gtrsim L$, and that K' produces $\overline{L}$ in Λ means in particular $K' \gtrsim \overline{L}$. By definition, there are substitutions γ and γ' such that $K\gamma = L$ and $K'\gamma' = \overline{L}$. Because of the assumption that L is disjoint with K and with K', γ' need not move a parameter or variable occurring in L. This implies that $L = L\gamma$, and, together with $K\gamma = L$ that $K\gamma\gamma' = L$. Similarly, since K and K' are given as parameter disjoint, γ need not act on the parameters of K'. This implies $K' = K'\gamma$, and, together with $K'\gamma' = \overline{L}$ it follows $K'\gamma\gamma' = \overline{L}$.

From $K\gamma\gamma' = L$ and $K'\gamma\gamma' = \overline{L}$, as just derived, conclude $K\gamma\gamma' = \overline{K'}\gamma\gamma'$. In other words, $\gamma\gamma'$ is a unifier of K and $\overline{K'}$. Hence there is a most general unifier σ of K and $\overline{K'}$ and a substitution δ such that $K\sigma = \overline{K'}\sigma$ and $\sigma\delta = \gamma\gamma'$ hold. Any standard unification algorithm will not introduce parameters (or even variables) beyond those occurring in K or K', and so $K\sigma$ will be variable-free, as demanded by Commit. For later use, note that from $K\gamma\gamma' = L$ and $\sigma\delta = \gamma\gamma'$ it follows $K'\sigma\delta = L$. Similarly, from $K'\gamma\gamma' = \overline{L}$ it follows $K'\sigma\delta = \overline{L}$.

The substitution σ is the one taken to prove that neither $K\sigma$ nor $\overline{K\sigma}$ is contradictory with Λ, as demanded by Commit. It also proves the "moreover" part in the lemma statement. For reasons of symmetry, it suffices to prove only that $K\sigma$ is not contradictory with Λ.

We first rule out the possibility that $K\sigma \approx K$ holds. Suppose to the contrary that $K\sigma \approx K$ holds. Because K is given as variable-free and $K\sigma$ was assumed to be variable-free further above, $K\sigma \approx K$ is equivalent to $K\sigma \simeq K$. Now we need a further case analysis: in the first case $K'\sigma \approx K'$ holds. As with K and $K\sigma$, both K' and $K'\sigma$ are variable-free for the same reasons, and therefore $K'\sigma \approx K'$ is equivalent to $K'\sigma \simeq K'$. Buth then, the chain $K \simeq K\sigma = \overline{K'}\sigma \simeq \overline{K'}$ holds true. With $K' \in_{\simeq} \Lambda$ it follows that K is contradictory with Λ, and so Λ itself is contradictory. However,

Λ was given as non-contradictory, and so the case $K'\sigma \approx K'$ is impossible. Hence we consider the complementary case, $K'\sigma \not\approx K'$. Then, with $K' \gtrsim K'\sigma$, which holds trivially, it follows $K' \gtrsim\!\!\!\!\not\approx K'\sigma$. Above we noted that $K'\sigma\delta = \overline{L}$ holds. Together with $K' \gtrsim\!\!\!\!\not\approx K'\sigma$ we get the chain $K' \gtrsim\!\!\!\!\not\approx K'\sigma \gtrsim \overline{L}$. With $K\sigma = \overline{K'}\sigma$ this chain is equivalent to the chain $K' \gtrsim\!\!\!\!\not\approx \overline{K}\sigma \gtrsim \overline{L}$. Recall that we currently assume $K\sigma \approx K$, which implied $K\sigma \simeq K$. However, from $K \in \Lambda$ it holds by definition $K\sigma \in_{\simeq} \Lambda$, which implies trivially $K\sigma \in_{\geq} \Lambda$. With $K' \gtrsim\!\!\!\!\not\approx \overline{K}\sigma \gtrsim \overline{L}$ it follows immediately that K' does not produce $\overline{L}$ in Λ. This contradicts what is given in the lemma statement, and hence the second case, that $K'\sigma \not\approx K'$ holds, is impossible, too. In sum, the assumption that $K\sigma \approx K$ holds led to a contradiction in both (complementary) subcases.

From now on we may therefore assume that $K\sigma \not\approx K$. As said above, it suffices to show that $K\sigma$ is not contradictory with Λ. Suppose, to the contrary, that $K\sigma$ is contradictory with Λ. This means there is a literal $K'' \in_{\simeq} \Lambda$ and a p-preserving substitution σ' such that $K\sigma\sigma' = \overline{K''}\sigma'$. Since $K\sigma$ is variable-free, Lemma A.9 can be applied to conclude that there is a p-preserving substitution σ'' such that $K\sigma = \overline{K''}\sigma''$.

Because $K \gtrsim K\sigma$ holds trivially, together with $K\sigma \not\approx K$ it follows $K \gtrsim\!\!\!\!\not\approx K\sigma$. From above there is a substitution δ such that $K\sigma\delta = L$. Therefore it holds $K \gtrsim\!\!\!\!\not\approx K\sigma \gtrsim L$. Using the just derived identity $K\sigma = \overline{K''}\sigma''$, we get $K \gtrsim\!\!\!\!\not\approx \overline{K''}\sigma'' \gtrsim L$. From the fact $K'' \in_{\simeq} \Lambda$ and that σ'' is p-preserving it follows that $K''\sigma'' \in_{\geq} \Lambda$. But then, K does not produce L in Λ, contradicting what was given in the lemma statement. Hence, the assumption that $K\sigma$ is contradictory with Λ leads to a contradiction, which remained to be shown. □

Lemma 4.19 (Lifting Lemma) *Let Λ be a non-contradictory context. Let $C = L_1 \vee \cdots \vee L_n$ be a Σ-clause and γ a grounding substitution for C. If Λ produces $\overline{L_1}\gamma,\ldots,\overline{L_n}\gamma$, then there are fresh variants $K_1,\ldots,K_n \in_{\simeq} \Lambda$ and a substitution σ such that*

1. *σ is a most general simultaneous unifier of $\{K_1,\overline{L_1}\},\ldots,\{K_n,\overline{L_n}\}$,*
2. *for all $i = 1,\ldots,n$, $L_i \gtrsim L_i\sigma \gtrsim L_i\gamma$,*
3. *for all $i = 1,\ldots,n$, K_i produces $\overline{L_i}\sigma$ in Λ.*

Proof. Let $i \in \{1,\ldots,n\}$ and assume that Λ produces $\overline{L_i}\gamma$. Then, there are literals $K_i' \in \Lambda$ such that K_i' produces $\overline{L_i}\gamma$ in Λ. Let $K_i \simeq K_i'$ be fresh variants of K_i'. It is easy to see that $K_i \in_{\simeq} \Lambda$ produces $\overline{L_i}\gamma$ in Λ. Because all the K_i's are fresh, they are pairwise disjoint, and each K_i is disjoint from C.

By definition of productivity, $K_i \gtrsim \overline{L_i\gamma}$, that is, there is a substitutions π_i such that $K_i\pi_i = \overline{L_i\gamma}$. Since K_i is variable disjoint from C, we can assume that π_i moves only the variables and the parameters of K_i. Now, since K_i is disjoint from K_j for $j \in \{1,\ldots,n\}$ distinct from i, and π_i is a ground substitution for K_i, we have that

$K_i\pi_i = K_i\pi$ where $\pi := \pi_1 \cdots \pi_i \cdots \pi_n$. Since $L_i\gamma$ is ground, it follows immediately that $L_i\gamma = L_i\gamma\pi$.

We may assume that all variables moved by γ occur in C only (otherwise restrict γ respectively). Together with the assumptions made it follows that $K_i = K_i\gamma$, which implies trivially that $K_i\pi = K_i\gamma\pi$.

Putting together all results obtained so far together, we get that $K_i\gamma\pi = \overline{L_i}\gamma\pi$ for all $i = 1, \ldots, n$. In other words, $\gamma\pi$ is a simultaneous unifier of $\{K_1, \overline{L_1}\}$ $\ldots$, $\{K_n, \overline{L_n}\}$. It follows that $\{K_1, \overline{L_1}\}$ $\ldots$, $\{K_n, \overline{L_n}\}$ admits a simultaneous mgu σ, which proves item 1 in the statement of the lemma.

Now, to prove item 2 observe that since $L_i\gamma$ is ground, $L_i\gamma\pi = L_i\gamma$. Since σ is a more general substitution than $\gamma\pi$ we know that $\gamma\pi = \sigma\delta$ for some substitution δ. It follows that $L_i\sigma\delta = L_i\gamma\pi = L_i\gamma$. In other words, $L_i\sigma \gtrsim L_i\gamma$. But then $L \gtrsim L_i\sigma \gtrsim L_i\gamma$ as desired.

To prove item 3 first observe that $K_i \gtrsim \overline{L_i}\sigma$ because $K_i\sigma = \overline{L_i}\sigma$. By item 2 we then have that $K_i \gtrsim \overline{L_i}\sigma \gtrsim \overline{L_i}\gamma$. Recalling that the literal K_i produces $\overline{L_i}\gamma$ in Λ, it follows by Lemma A.7 that K_i produces $\overline{L_i}\sigma$ in Λ as well. □

Lemma 4.20 (Existence of Admissible Context Unifiers) *Let Λ be a context, C a clause and σ a context unifier of C against Λ. Then, there is a renaming ρ such that $\sigma' := \sigma\rho$ is an admissible context unifier of C against Λ.*

Proof. Let $C = L_1 \vee \cdots \vee L_n$ for some $n \geq 0$. By Definition 3.11 of context unifier, for all $i = 1, \ldots, n$ there is a $K_i \in_\simeq \Lambda$ such that $K_i\sigma = \overline{L_i}\sigma$. Moreover, there is an $m \in \{1, \ldots, n\}$ such that $(\mathcal{P}ar(K_i))\sigma \subseteq V$ for all $i = 1, \ldots, m$ and $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$ for all $i = m+1, \ldots, n$.

We are going to construct a renaming substitution ρ as stated. Let $x_1, \ldots, x_k$ be the variables such that $\{x_1, \ldots, x_k\} = \mathcal{V}ar(L_{m+1}\sigma \vee \cdots \vee L_n\sigma)$, i.e. all variables occurring in the remainder. Define $\rho := \{x_1 \mapsto u_1, \ldots, x_k \mapsto u_k, u_1 \mapsto x_1, \ldots, u_k \mapsto x_k, \}$, where $u_1, \ldots, u_k$ are pairwise different and fresh parameters[20].

Clearly, ρ is a renaming. It remains to show that $\sigma\rho$ is admissible for Split. Recall that $(\mathcal{P}ar(K_i))\sigma \subseteq V$ holds, for $i = 1, \ldots, m$. By construction, all the parameters moved by ρ are fresh parameters, none of which therefore can occur in K_i. In other words, $(\mathcal{P}ar(K_i))\rho = \mathcal{P}ar(K_i)$ holds, which entails $(\mathcal{P}ar(K_i))\sigma\rho = (\mathcal{P}ar(K_i))\sigma$. (However, $(\mathcal{P}ar(K_i))\sigma\rho \not\subseteq V$, for $i = m+1, \ldots, n$, will in general not hold). Therefore, there is a m' with $m \leq m' \leq n$ such that $(\mathcal{P}ar(K_i))\sigma\rho \subseteq V$, for $i = 1, \ldots, m'$ and $(\mathcal{P}ar(K_i))\sigma\rho \not\subseteq V$, for $i = m'+1, \ldots, n$.

None of the remainder literals $K_i\sigma\rho$, for $i = m+1, \ldots, n$, contains a single variable. Hence they all are variable-free, and the disjointness requirement in the definition of admissible context unifier is trivially satisfied. This concludes the proof of existence of a renaming ρ as claimed. □

[20] That is, every variable in the remainder is renamed by ρ to a parameter. From a practical point of view this is absurd, and it is better to compute a renaming that keeps as many variables in the remainder as possible. For the purpose of the completeness proof, however, the renaming ρ as constructed will do.

Lemma 4.21 *Let Λ be a context, $L_1 \vee \cdots \vee L_n$ be a clause, where $n \geq 0$, such that for all distinct $i, j = 1, \ldots, n$, L_i is parameter- or variable-free and $\mathcal{V}ar(L_i) \cap \mathcal{V}ar(L_j) = \emptyset$. If for all $i = 1, \ldots, n$, L_i is contradictory with Λ then there are fresh literals $K_1, \ldots, K_n \in_{\simeq} \Lambda$ and a substitution δ such that the following holds:*

1. *δ is a simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,*
2. *for all $i = 1, \ldots, n$, $\mathcal{D}om(\delta) \cap \mathcal{P}ar(L_i) = \emptyset$, (i.e. δ does not move any single parameter in the given clause)*
3. *for all $i = 1, \ldots, n$, $(\mathcal{P}ar(K_i))\delta \subseteq V$.*

Proof. Let Λ and $L_1 \vee \cdots \vee L_n$ be as stated, and such that the condition in the lemma result is satisfied. The conclusions, items 1–3, are proven by induction on n.

Base) If $n = 0$ then the result follows trivially by choosing for δ the empty substitution.

Step) Suppose $n > 0$ and consider the clause $L_1 \vee \cdots \vee L_{n-1}$. Clearly, for all $i, j = 1, \ldots, n-1$, L_i is parameter- or variable-free and $\mathcal{V}ar(L_i) \cap \mathcal{V}ar(L_j) = \emptyset$ holds. Therefore, by the induction hypothesis there are literals $K_1, \ldots, K_{n-1} \in_{\simeq} \Lambda$ and a substitution δ' such that

1. δ' is a simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_{n-1}, \overline{L_{n-1}}\}$,
2. for all $i = 1, \ldots, n-1$, $\mathcal{D}om(\delta') \cap \mathcal{P}ar(L_i) = \emptyset$,
3. for all $i = 1, \ldots, n-1$, $(\mathcal{P}ar(K_i))\delta' \subseteq V$.

Since L_n is contradictory with Λ, there is a literal $K \in_{\simeq} \Lambda$ and a p-preserving substitution σ such that $L_n\sigma = \overline{K}\sigma$. Let K_n be a fresh p-variant of K. We distinguish two complementary cases. In both cases we will show there is a substitution σ' such that $\delta := \delta'\sigma'$ proves the induction step.

If L_n is parameter-free, then by Lemma A.8 there is a p-preserving substitution σ' such that $L_n\sigma' = \overline{K_n}\sigma'$ and $\mathcal{D}om(\sigma') \cap V = \emptyset$ (K_n is fresh and hence disjoint with L_n, and so the lemma can indeed be applied). From the latter conclusion and items 2 and 3 from the induction hypothesis, items 2 and 3 for the induction step follow immediately (recall we set $\delta := \delta'\sigma'$).

Since K_n is fresh, the substitution δ' will not modify K_n, i.e. $K_n = K_n\delta'$ holds (the substitution δ' is a unifier for just L_i and $\overline{K_i}$, for $i = 1, \ldots, n-1$ and hence need not modify the fresh literal K_n). Together with $L_n\sigma' = \overline{K_n}\sigma'$ it follows $L_n\sigma' = \overline{K_n}\delta'\sigma'$. Since L_n does not contain any parameter, does not share any variable with the literals $L_1, \ldots, L_{n-1}$, and all the literals $K_1, \ldots, K_n$ are fresh, it is safe to assume that δ' will not modify L_n. In other words, it holds $L_n = L_n\delta'$. Together with $L_n\sigma' = \overline{K_n}\delta'\sigma'$ from above conclude $L_n\delta'\sigma' = \overline{K_n}\delta'\sigma'$.

If L_n is variable-free, then by Lemma A.9, there is a p-preserving substitution σ'' such that $L_n = \overline{K_n}\sigma''$. Let $\sigma' := \sigma''_{|\mathcal{V}ar(K_n)\cup\mathcal{P}ar(K_n)}$. From this, items 2 and 3 from the induction hypothesis, items 2 and 3 for the induction step follow immediately (recall we set $\delta := \delta'\sigma'$).

From $L_n = \overline{K_n}\sigma''$ and the definition of σ' it follows trivially $L_n = \overline{K_n}\sigma'$. From the induction hypothesis, item 2, it follows that δ' does not move any parameter in L_n that occurs also in $L_1 \vee \cdots \vee L_{n-1}$. Furthermore, it is safe to assume that δ' does not move any other parameter in L_n, (δ' needs just move the parameters in $K_1, \ldots, K_{n-1}$, which, by freshness assumptions, are disjoint with L_n). In other words, $L_n = L_n\delta'$ will hold. Using the fact that K_n is fresh, hence disjoint with L_n, and the definition of σ' it follows trivially $L_n = L_n\sigma'$. Altogether then $L_n = L_n\delta' = L_n\sigma' = L_n\delta'\sigma'$, and from $L_n = \overline{K_n}\sigma'$ we get $L_n\delta'\sigma' = \overline{K_n}\sigma'$. As already argued for in the first case above, since K_n is fresh, it will hold $K_n = K_n\delta'$. With $L_n\delta'\sigma' = \overline{K_n}\sigma'$ it follows $L_n\delta'\sigma' = \overline{K_n}\delta'\sigma'$.

This concludes the case analysis. Note that in both cases we have shown that there is a substitution σ' such that $L_n\delta'\sigma' = \overline{K_n}\delta'\sigma'$. From the induction hypothesis we know that δ' is a simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_{n-1}, \overline{L_{n-1}}\}$. Hence $\delta'\sigma'$ is trivially a simultaneous unifier of these literals, too. Together, thus, item 1 for the induction step is shown (recall we set $\delta := \delta'\sigma'$). Also we have shown in the case analysis that items 2 and 3 for the induction step hold for $\delta'\sigma'$. Hence the proof is complete. □

Lemma 4.22 (Split Applicability) *Let $\Lambda \vdash \Psi, C$ be a sequent with a non-contradictory context Λ, where C contains at least two literals. If all context unifiers of C against Λ have a non-empty remainder, and σ is a context unifiers of C against Λ with a remainder not produced by Λ, then* Split *is applicable to $\Lambda \vdash \Psi, C$ with selected clause C and context unifier σ.*

Proof. Suppose the condition of the lemma statement holds. The proof of the conclusion consists of two parts: in a first part, we will show that there is a remainder literal that is not contradictory with Λ. Then, in a second part we will show that for each remainder literal L, $\overline{L}^{\text{sko}}$ is not contradictory with Λ. This will immediately give a proof that Split is applicable to $\Lambda \vdash \Psi, C$ with selected clause C, context unifier σ and that mentioned remainder literal.

Let $C = L_1 \vee \cdots \vee L_m \vee L_{m+1} \vee \cdots \vee L_m$, where $0 \leq m \leq n$ (and $n \geq 2$), where the remainder D is $(L_{m+1} \vee \cdots \vee L_m)\sigma$. Suppose, to the contrary of the statement for the first part that every literal $L_j\sigma$, for $j = m+1, \ldots, n$ is contradictory with Λ. Since σ is admissible, all prerequisites to apply Lemma 4.21 are satisfied. By this lemma then, there are fresh literals $K_{m+1}, \ldots, K_n \in_{\simeq} \Lambda$ and there is a simultaneous unifier δ of $\{K_{m+1}, \overline{L_{m+1}\sigma}\}, \ldots, \{K_n, \overline{L_n}\sigma\}$ (item 1) such that for all $j = m+1, \ldots, n$, it holds $\mathcal{D}om(\delta) \cap \mathcal{P}ar(L_j) = \emptyset$ (item 2), and $(\mathcal{P}ar(K_j))\delta \subseteq V$ (item 3). We may assume that δ is restricted so that each parameter moved by it occurs in some literal K_j, where $m+1 \leq j \leq n$. Otherwise restrict δ respectively by excluding

from its domain all the parameters that do not occur in any K_j, and items 1–3 will still hold. In particular, δ will still be a simultaneous unifier as stated in item 1, because the unrestricted δ does not move the parameters in L_j anyway.

In the sequel the index j always ranges from $m+1, \ldots, n$.

Since each literal K_j is fresh, we may assume that σ does not modify K_j, i.e. $K_j = K_j\sigma$ holds. Therefore, δ is a simultaneous unifier of $\{K_{m+1}\sigma, \overline{L_{m+1}\sigma}\}$, $\ldots$, $\{K_n\sigma, \overline{L_n\sigma}\}$. Equivalently, $\sigma\delta$ is a simultaneous unifier of $\{K_{m+1}, \overline{L_{m+1}}\}$, $\ldots$, $\{K_n, \overline{L_n}\}$.

Furthermore, from $(\mathcal{P}ar(K_j))\delta \subseteq V$ and $K_j = K_j\sigma$ it follows $(\mathcal{P}ar(K_j))\sigma\delta \subseteq V$.

We are given that σ is an (admissible) context unifier. This means in particular that σ is a simultaneous unifier of $\{K_1, \overline{L_1}\}$, $\ldots$, $\{K_m, \overline{L_m}\}$. Trivially, $\sigma\delta$ is a simultaneous unifier of these literals as well.

Above we assumed that δ is restricted so that each parameter moved by it occurs in some literal K_j, where $m+1 \leq j \leq n$. Since each literal K_j is fresh, δ will not move any parameter in any literal $K_i\sigma$, for all $i = 1, \ldots, m$. Since σ is a context unifier, we know $(\mathcal{P}ar(K_i))\sigma \subseteq V$, for all $i = 1, \ldots, m$. Together this implies $(\mathcal{P}ar(K_i))\sigma\delta \subseteq V$.

Summing up, there is a simultaneous unifier $\sigma\delta$ (of $\{K_1, \overline{L_1}\}$, $\ldots$, $\{K_n, \overline{L_n}\}$ – we will omit in the sequel the mentioning of these pairs if just these are meant) such that $(\mathcal{P}ar(K_i))\sigma\delta \subseteq V$, for all $i = 1, \ldots, n$.

However, there is no guarantee that $\sigma\delta$ will be a simultaneous most general unifier. We will show next that a simultaneous most general unifier exists, that, moreover will be a context unifier of C against Λ with empty remainder, contradicting the lemma statement.

Since $\sigma\delta$ is a simultaneous unifier, there is a most general simultaneous unifier σ' and a substitution δ' such that $\sigma'\delta' = \sigma\delta$. (The same arguments as in the proof of the Lifting Lemma, Lemma 4.19, can be applied to show this). However, there is no guarantee that $(\mathcal{P}ar(K_i))\sigma' \subseteq V$, for all $i = 1, \ldots, n$. But it must hold $(\mathcal{P}ar(K_i))\sigma' \subseteq X \cup V$, for all $i = 1, \ldots, n$, because otherwise there would be a parameter u in some literal K_i, where $1 \leq i \leq n$ and that would be moved to a term $u\sigma' \notin X \cup V$, which implies $u\sigma'\delta' \notin V$. However, we know $u\sigma'\delta' = u\sigma\delta \in V$.

Let $x_1, \ldots, x_k$ be all the variables in $\mathcal{P}ar(K_1)\sigma' \cup \cdots \cup \mathcal{P}ar(K_n)\sigma'$ and define the renaming

$$\rho = \{x_1 \mapsto u_1, \ldots, x_k \mapsto u_k, u_1 \mapsto x_1, \ldots, u_k \mapsto x_k\} \ ,$$

where $u_1, \ldots, u_k$ are fresh parameters. By this construction, each variable in $(\mathcal{P}ar(K_i))\sigma'$ is moved to a parameter, and because $u_1, \ldots, u_k$ are fresh, each parameter in $(\mathcal{P}ar(K_i))\sigma'$ is moved to itself, for all $i = 1, \ldots, n$. This proves $(\mathcal{P}ar(K_i))\sigma'\rho \subseteq V$, for all $i = 1, \ldots, n$. Furthermore, with σ' being a most general simultaneous unifier and ρ being a renaming, $\sigma'\delta'$ is a most general simultaneous unifier, too. (And it holds $(\sigma'\delta')(\delta'^{-1}\delta') = \sigma\delta$). In other words, $\sigma'\delta'$ is a context unifier of C against Λ with empty remainder. Since this plainly contradicts what is given in the lemma statement, the assumption that every literal $L_j\sigma$, for $j = m+1, \ldots, n$, is

contradictory with Λ must be withdrawn. Hence, as claimed, there is a remainder literal $L\sigma$ that is not contradictory with Λ. This completes the first part of the proof.

For the second part, let $L \in D$ be any remainder literal. We have to show that $\overline{L}^{\text{sko}}$ is not contradictory with Λ. Suppose to the contrary that $\overline{L}^{\text{sko}}$ is contradictory with Λ. Then, there is a $K \in_{\simeq} \Lambda$ and a p-preserving substitution σ such that $\overline{L}^{\text{sko}}\sigma = \overline{K}\sigma$.

Because of Skolemization, $\overline{K}^{\text{sko}}$ is variable-free. Now, if K is variable-free as well, we take $\sigma' := \sigma_{|V}$ and it holds $\overline{L}^{\text{sko}}\sigma' = \overline{K}\sigma'$. Because σ' is a p-preserving renaming, σ'^{-1} exists, and it follows $\overline{L}^{\text{sko}} = \overline{L}^{\text{sko}}\sigma'\sigma'^{-1} = \overline{K}\sigma'\sigma'^{-1} = \overline{K}$. This implies trivially $L^{\text{sko}} = K$. That L contains variables is impossible, because then L^{sko} would contain (one or more) Skolem constants, which are fresh, but the context literal K cannot contain any of these Skolem constants, and so $L^{\text{sko}} = K$ would be impossible. Hence it holds $L^{\text{sko}} = L$, and therefore $L = K$. Since we have $K \in_{\simeq} \Lambda$, which is the same as $L \in_{\simeq} \Lambda$, Λ trivially produces L. However, since we are given that Λ does not produce any literal in D, hence in particular Λ does not produce L, the case that K is variable-free is impossible.

Now that we know that K is not variable-free, by property of contexts, K must be parameter-free. Recall that L is variable-free. From Lemma A.9 then it follows there is a p-preserving substitution σ' such that $\overline{L}^{\text{sko}} = \overline{K}\sigma'$. This implies $L^{\text{sko}} = K\sigma'$. Let $\mu = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ be the Skolemizing substitution used, for some $n \geq 0$. Now, because the constants $a_1, \ldots, a_n$ are fresh, none of then will occur in K. This means, we can consider the "substitution" $\mu' = \{a_1 \mapsto x_1, \ldots, a_n \mapsto x_n\}$ and it will hold $L = L^{\text{sko}}\mu' = K\sigma'\mu'$. The substitution $\sigma'\mu'$ instantiates the variables of K, which implies $K \geq L$. But then, by Lemma A.5, Λ produces L. As above, since we are given that Λ does not produce any literal in D, hence in particular Λ does not produce L, the case that K is parameter-free is impossible as well.

In sum, we now now that K is neither variable- nor parameter-free, which contradicts a fundamental property of contexts. Therefore, the assumption that $\overline{L}^{\text{sko}}$ is contradictory with Λ is false, and so no remainder literal is contradictory with Λ. Since this is all that remained to be proven, the proof is complete now. □

Lemma 4.23 (Assert Applicability) *Let $\Lambda \vdash \Psi$, L be a sequent with a non-contradictory context Λ. If all context unifiers of L against Λ have a non-empty remainder and there is an instance of L that is not produced by Λ, then* Assert *is applicable to $\Lambda \vdash \Psi$, L with selected unit clause L.*

Proof. Suppose that Λ does not produce $L\sigma$ and that there is no context unifier of L against Λ with empty remainder. To show that Assert is applicable as stated, we first have to show that there is no literal $K \in \Lambda$ such that $K \geq L$. Suppose there were such a literal K. Recall that the clauses in the sequents are parameter-free.

With L being therefore parameter-free, it follows easily that $L \geq L\sigma$. Together with $K \geq L$ conclude $K \geq L\sigma$.

But then, Lemma A.5 can be applied to conclude that Λ produces $L\sigma$, plainly contradicting to what was supposed. Therefore, there is no literal $K \in \Lambda$ such that $K \geq L$.

Finally, we have to show that L is not contradictory with Λ. Suppose, to the contrary, there is a literal $K \in_{\simeq} \Lambda$ and a p-preserving substitution δ such that $L\delta = \overline{K}\delta$. Let $K' \in_{\simeq} \Lambda$ be a fresh p-preserving variant of K. As said above, L is parameter-free. By Lemma A.8 then, there is a p-preserving substitution δ' such that $L\delta' = \overline{K'}\delta'$ such that $\mathcal{D}om(\delta') \cap V = \emptyset$. Since δ' is a unifier of L and $\overline{K'}$, it is not difficult to see that there is a mgu σ' of L and $\overline{K'}$ such that $\mathcal{D}om(\sigma') \cap V = \emptyset$ holds as well (roughly, any mgu not having this property must rename, say by a substitution ρ, the offending parameters to variables, because otherwise it would be impossible that σ' could be appended with some substitution δ'' to give $\sigma'\delta'' = \delta'$. But then $\sigma'\rho^{-1}$ will be a mgu such that $\mathcal{D}om(\sigma'\rho^{-1}) \cap V = \emptyset$ holds and can be used instead of σ').

Since $\mathcal{D}om(\sigma') \cap V = \emptyset$ trivially entails $(\mathcal{P}ar(K'))\sigma' \subseteq V$, we have just shown that σ' is a context unifier of L against Λ with a non-empty remainder. This, however, plainly contradicts what was supposed above. Altogether, all applicability conditions for applying Assert as stated have been shown. □

References

[Bau98] Peter Baumgartner. Hyper Tableaux — The Next Generation. In Harry de Swaart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 60–76. Springer, 1998.

[Bau00] Peter Baumgartner. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In David McAllester, editor, *CADE-17 – The 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 200–219. Springer, 2000.

[Bec00] Bernhard Beckert. Depth-first proof search without backtracking for free-variable clausal tableaux. In Peter Baumgartner and Hantao Zhang, editors, *FTP 2000 – Third International Workshop on First-Order Theorem Proving*, volume 5-2000 of *Technical Report*, pages 44–55. Universitt Koblenz-Landau, 2000.

[BEF99] Peter Baumgartner, Norbert Eisinger, and Ulrich Furbach. A confluent connection calculus. In Ganzinger [Gan99], pages 329–343.

[BFN96] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper Tableaux. In *Proc. JELIA 96*, number 1126 in Lecture Notes in Artificial Intelligence. European Workshop on Logic in AI, Springer, 1996.

[BG98] Leo Bachmair and Harald Ganzinger. Chapter 11: Equational Reasoning in Saturation-Based Theorem Proving. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction. A Basis for Applications*, volume I: Foundations. Calculi and Refinements, pages 353–398. Kluwer Academic Publishers, 1998.

[BG01] L. Bachmair and H. Ganzinger. Resolution Theorem Proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. North Holland, 2001.

[BGHS03] Peter Baumgartner, Margret Gross-Hardt, and Alex Sinner. Living Book – Deduction, Slicing, and Interaction. Fachberichte Informatik 2–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.

[Bib82] W. Bibel. *Automated Theorem Proving*. Vieweg, 1982.

[Bil96] Jean-Paul Billon. The Disconnection Method. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Theorem Proving with Analytic Tableaux and Related Methods*, number 1071 in Lecture Notes in Artificial Intelligence, pages 110–126. Springer, 1996.

[CL73] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[DFW02] Lyndon Drake, Alan M. Frisch, and Toby Walsh. Adding resolution to the dpll procedure for boolean satisfiability. In *The Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*, Cincinnati, 2002.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[dN98] Hans de Nivelle. Resolution decides the guarded fragment. Research Report CT-1998-01, ILLC, 1998.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[Ede85] Elmar Eder. Properties of Substitutions and Unifications. *Journal of Symbolic Computation*, 1(1), March 1985.

[EF02] Uwe Egly and Christian G. Fermüller, editors. *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2002, Copenhagen, Denmark, July 30 - August 1, 2002, Proceedings*, volume 2381 of *Lecture Notes in Computer Science*. Springer, 2002.

[Fit90] M. Fitting. *First Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer, 1990.

[FL93] Christian Fermueller and Alexander Leitsch. Model building by resolution. In E. Brger, G. Jger, H. Kleine-Bning, S. Martini, and M.M. Richter, editors, *Computer Science Logic – CSL'92*, volume 702 of *Lecture Notes in Computer Science*, pages 134–148. Springer Verlag, Berlin, Heidelberg, New-York, 1993.

[FL96] Christian Fermller and Alexander Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 6(2):173–230, 1996.

[FLHT01] C. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution Decision Procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1791–1850. Noth-Holland, 2001.

[Gan99] Harald Ganzinger, editor. *Automated Deduction – CADE-16*, volume 1632 of *Lecture Notes in Artificial Intelligence*, Trento, Italy, 1999. Springer.

[Gie01] Martin Giese. Incremental closure of free variable tableaux. In Rajeev Gor, Alexander Leitsch, and Tobias Nipkov, editors, *CADE-18 – The 18th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence. Springer, 2001. To appear.

[Gie02] Martin Giese. A model generation style completeness proof for constraint tableaux with superposition. In Uwe Egly and Christian G. Fermu"ller, editors, *Proc. Intl. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, Copenhagen, Denmark*, volume 2381 of *LNCS*. Springer-Verlag, 2002.

[GMW97] Harald Ganzinger, Christoph Meyer, and Christoph Weidenbach. Soft typing for ordered resolution. In W. McCune, editor, *Automated Deduction — CADE 14*, LNAI 1249, pages 321–335, Townsville, North Queensland, Australia, July 1997. Springer-Verlag.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[HRCS02] J.N. Hooker, G. Rago, V. Chandru, and A. Shrivastava. Partial Instantiation Methods for Inference in First Order Logic. *Journal of Automated Reasoning*, 28:371–396, 2002.

[Jac00] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the ACM SIGSOFT Conference on Foundations of Software Engineering*, pages 130–139, 2000.

[JNR01] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. Technical report, Compaq SRC, July 2001.

[LMG94] R. Letz, K. Mayr, and C. Goller. Controlled Integrations of the Cut Rule into Connection Tableau Calculi. *Journal of Automated Reasoning*, 13, 1994.

[LP92] S.-J. Lee and D. Plaisted. Eliminating Duplicates with the Hyper-Linking Strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.

[LS01] Reinhold Letz and Gernot Stenz. Proof and Model Generation with Disconnection Tableaux. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*. Springer, 2001.

[LS02] Reinhold Letz and Gernot Stenz. Integration of Equality Reasoning into the Disconnection Calculus. In Egly and Fermüller [EF02], pages 176–190.

[MB88] Rainer Manthey and François Bry. SATCHMO: a theorem prover implemented in Prolog. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the 9^{th} Conference on Automated Deduction, Argonne, Illinois, May 1988*, volume 310 of *Lecture Notes in Computer Science*, pages 415–434. Springer, 1988.

[McC94] William W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, National Laboratory, Argonne, IL, 1994.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[Pel99] N. Peltier. Pruning the search space and extracting more models in tableaux. *Logic Journal of the IGPL*, 7(2):217–251, 1999.

[PZ97] David A. Plaisted and Yunshan Zhu. Ordered Semantic Hyper Linking. In *Proceedings of Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[PZ00] David A. Plaisted and Yunshan Zhu. Ordered Semantic Hyper Linking. *Journal of Automated Reasoning*, 25(3):167–217, 2000.

[RV00] Alexandre Riazanov and Andrei Voronkov. Splitting without backtracking. Technical Report CSPP-10, University of Manchester, 2000.

[SSB02] O. Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with sat. In *Proceedings of the Computer Aided Verification conference (CAV'02)*, 2002.

[Ste02] Gernot Stenz. DCTP 1.2 - System Abstract. In Egly and Fermüller [EF02], pages 335–340.

[Tin02] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence*. Springer, 2002. (to appear).

[vE01] Jan van Eijck. Constrained Hyper Tableaux. In L. Fribourg, editor, *Computer Science Logic*, pages 232–246, 2001. 15th International Workshop, CSL 2001 (LNCS 2142).

[WAB+99] Christoph Weidenbach, Bijan Afshordel, Uwe Brahm, Christian Cohrs, Thorsten Engel, Enno Keen, Christian Theobalt, and Dalibor Topić. System description: SPASS version 1.0.0. In Ganzinger [Gan99], pages 378–382.

[YP02] Adnan Yahya and David Plaisted. Ordered Semantic Hyper-Tableaux. *Journal of Automated Reasoning*, 29(1):17–57, 2002.

[ZS96] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.

Available Research Reports (since 1998):

2003

1/2003 *Peter Baumgartner, Cesare Tinelli.* The Model Evolution Calculus.

2002

12/2002 *Kurt Lautenbach.* Logical Reasoning and Petri Nets.

11/2002 *Margret Gro-Hardt.* Processing of Concept Based Queries for XML Data.

10/2002 *Hanno Binder, Jrme Diebold, Tobias Feldmann, Andreas Kern, David Polock, Dennis Reif, Stephan Schmidt, Frank Schmitt, Dieter Zbel.* Fahrassistenzsystem zur Untersttzung beim Rckwrtsfahren mit einachsigen Gespannen.

9/2002 *Jrgen Ebert, Bernt Kullbach, Franz Lehner.* 4. Workshop Software Reengineering (Bad Honnef, 29./30. April 2002).

8/2002 *Richard C. Holt, Andreas Winter, Jingwei Wu.* Towards a Common Query Language for Reverse Engineering.

7/2002 *Jrgen Ebert, Bernt Kullbach, Volker Riediger, Andreas Winter.* GUPRO – Generic Understanding of Programs, An Overview.

6/2002 *Margret Gro-Hardt.* Concept based querying of semistructured data.

5/2002 *Anna Simon, Marianne Valerius.* User Requirements – Lessons Learned from a Computer Science Course.

4/2002 *Frieder Stolzenburg, Oliver Obst, Jan Murray.* Qualitative Velocity and Ball Interception.

3/2002 *Peter Baumgartner.* A First-Order Logic Davis-Putnam-Logemann-Loveland Procedure.

2/2002 *Peter Baumgartner, Ulrich Furbach.* Automated Deduction Techniques for the Management of Personalized Documents.

1/2002 *Jrgen Ebert, Bernt Kullbach, Franz Lehner.* 3. Workshop Software Reengineering (Bad Honnef, 10./11. Mai 2001).

2001

13/2001 *Annette Pook.* Schlussbericht “FUN - Funkunterrichtsnetzwerk”.

12/2001 *Toshiaki Arai, Frieder Stolzenburg.* Multiagent Systems Specification by UML Statecharts Aiming at Intelligent Manufacturing.

11/2001 *Kurt Lautenbach.* Reproducibility of the Empty Marking.

10/2001 *Jan Murray.* Specifying Agents with UML in Robotic Soccer.

9/2001 *Andreas Winter.* Exchanging Graphs with GXL.

8/2001 *Marianne Valerius, Anna Simon.* Slicing Book Technology — eine neue Technik fr eine neue Lehre?.

7/2001 *Bernt Kullbach, Volker Riediger.* Folding: An Approach to Enable Program Understanding of Preprocessed Languages.

6/2001 *Frieder Stolzenburg.* From the Specification of Multiagent Systems by Statecharts to their Formal Analysis by Model Checking.

5/2001 *Oliver Obst.* Specifying Rational Agents with Statecharts and Utility Functions.

4/2001 *Torsten Gipp, Jrgen Ebert.* Conceptual Modelling and Web Site Generation using Graph Technology.

3/2001 *Carlos I. Chesevar, Jrgen Dix, Frieder Stolzenburg, Guillermo R. Simari.* Relating Defeasible and Normal Logic Programming through Transformation Properties.

2/2001 *Carola Lange, Harry M. Sneed, Andreas Winter.* Applying GUPRO to GEOS – A Case Study.

1/2001 *Pascal von Hutten, Stephan Philippi.* Modelling a concurrent ray-tracing algorithm using object-oriented Petri-Nets.

2000

8/2000 *Jrgen Ebert, Bernt Kullbach, Franz Lehner (Hrsg.).* 2. Workshop Software Reengineering (Bad Honnef, 11./12. Mai 2000).

7/2000 *Stephan Philippi.* AWPN 2000 - 7. Workshop Algorithmen und Werkzeuge fr Petrinetze, Koblenz, 02.-03. Oktober 2000 .

6/2000 *Jan Murray, Oliver Obst, Frieder Stolzenburg.* Towards a Logical Approach for Soccer Agents Engineering.

5/2000 *Peter Baumgartner, Hantao Zhang (Eds.).* FTP 2000 – Third International Workshop on First-Order Theorem Proving, St Andrews, Scotland, July 2000.

4/2000 *Frieder Stolzenburg, Alejandro J. Garca, Carlos I. Chesevar, Guillermo R. Simari.* Introducing Generalized Specificity in Logic Programming.

3/2000 *Ingar Uhe, Manfred Rosendahl.* Specification of Symbols and Implementation of Their Constraints in JKogge.

2/2000 *Peter Baumgartner, Fabio Massacci.* The Taming of the (X)OR.

1/2000 *Richard C. Holt, Andreas Winter, Andy Schrr.* GXL: Towards a Standard Exchange Format.

1999

10/99 *Jrgen Ebert, Luuk Groenewegen, Roger Sttenbach.* A Formalization of SOCCA.

9/99 *Hassan Diab, Ulrich Furbach, Hassan Tabbara.* On the Use of Fuzzy Techniques in Cache Memory Managament.

8/99 *Jens Woch, Friedbert Widmann.* Implementation of a Schema-TAG-Parser.

7/99 *Jrgen Ebert, and Bernt Kullbach, Franz Lehner (Hrsg.).* Workshop Software-Reengineering (Bad Honnef, 27./28. Mai 1999).

6/99 *Peter Baumgartner, Michael Khn.* Abductive Coreference by Model Construction.

5/99 *Jrgen Ebert, Bernt Kullbach, Andreas Winter.* GraX – An Interchange Format for Reengineering Tools.

4/99 *Frieder Stolzenburg, Oliver Obst, Jan Murray, Bjrn Bremer.* Spatial Agents Implemented in a Logical Expressible Language.

3/99 *Kurt Lautenbach, Carlo Simon.* Erweiterte Zeitstempelnetze zur Modellierung hybrider Systeme.

2/99 *Frieder Stolzenburg.* Loop-Detection in Hyper-Tableaux by Powerful Model Generation.

1/99 *Peter Baumgartner, J.D. Horton, Bruce Spencer.* Merge Path Improvements for Minimal Model Hyper Tableaux.

1998

24/98 *Jrgen Ebert, Roger Sttenbach, Ingar Uhe.* Meta-CASE Worldwide.

23/98 *Peter Baumgartner, Norbert Eisinger, Ulrich Furbach.* A Confluent Connection Calculus.

22/98 *Bernt Kullbach, Andreas Winter.* Querying as an Enabling Technology in Software Reengineering.

21/98 *Jrgen Dix, V.S. Subrahmanian, George Pick.* Meta-Agent Programs.

20/98 *Jrgen Dix, Ulrich Furbach, Ilkka Niemel .* Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations.

19/98 *Jrgen Dix, Steffen Hlldobler.* Inference Mechanisms in Knowledge-Based Systems: Theory and Applications (Proceedings of WS at KI ’98).

18/98 *Jose Arrazola, Jrgen Dix, Mauricio Osorio, Claudia Zepeda.* Well-behaved semantics for Logic Programming.

17/98 *Stefan Brass, Jrgen Dix, Teodor C. Przymusinski.* Super Logic Programs.

16/98 *Jrgen Dix.* The Logic Programming Paradigm.

15/98 *Stefan Brass, Jrgen Dix, Burkhard Freitag, Ulrich Zukowski.* Transformation-Based Bottom-Up Computation of the Well-Founded Model.

14/98 *Manfred Kamp.* GReQL – Eine Anfragesprache fr das GUPRO–Repository – Sprachbeschreibung (Version 1.2).

12/98 *Peter Dahm, Jrgen Ebert, Angelika Franzke, Manfred Kamp, Andreas Winter.* TGraphen und EER-Schemata – formale Grundlagen.

11/98 *Peter Dahm, Friedbert Widmann.* Das Graphenlabor.

10/98 *Jrg Jooss, Thomas Marx.* Workflow Modeling according to WfMC.

9/98 *Dieter Zbel.* Schedulability criteria for age constraint processes in hard real-time systems.

8/98 *Wenjin Lu, Ulrich Furbach.* Disjunctive logic program = Horn Program + Control program.

7/98 *Andreas Schmid.* Solution for the counting to infinity problem of distance vector routing.

6/98 *Ulrich Furbach, Michael Khn, Frieder Stolzenburg.* Model-Guided Proof Debugging.

5/98 *Peter Baumgartner, Dorothea Schfer.* Model Elimination with Simplification and its Application to Software Verification.

4/98 *Bernt Kullbach, Andreas Winter, Peter Dahm, Jrgen Ebert.* Program Comprehension in Multi-Language Systems.

3/98 *Jrgen Dix, Jorge Lobo.* Logic Programming and Nonmonotonic Reasoning.

2/98 *Hans-Michael Hanisch, Kurt Lautenbach, Carlo Simon, Jan Thieme.* Zeitstempelnetze in technischen Anwendungen.

1/98 *Manfred Kamp.* Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools — A Generic Approach.