# A Decidable Class of Nested Iterated Schemata*

Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier

Grenoble University (LIG/CNRS)

**Abstract.** Many problems can be specified by patterns of propositional formulae depending on a parameter, e.g. the specification of a circuit usually depends on the number of bits of its input. We define a logic whose formulae, called *iterated schemata*, allow to express such patterns. Schemata extend propositional logic with indexed propositions, e.g. $P_{\mathsf{i}}$, $P_{\mathsf{i}+1}$, $P_1$ or $P_{\mathsf{n}}$, and with generalized connectives, e.g. $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}}$ or $\bigvee_{\mathsf{i}=1}^{\mathsf{n}}$ where $\mathsf{n}$ is an (unbound) integer variable called a *parameter*. The expressive power of iterated schemata is strictly greater than propositional logic: it is even out of the scope of first-order logic. We define a proof procedure, called DPLL*, that can prove that a schema is satisfiable for at least one value of its parameter, in the spirit of the DPLL procedure [9]. But proving that a schema is unsatisfiable *for every value of the parameter*, is undecidable [1] so DPLL* does not terminate in general. Still, DPLL* terminates for schemata of a syntactic subclass called *regularly nested*.

## 1 Introduction

The specification of problems in propositional logic often leads to propositional formulae that depend on a parameter: the $n$-queens problem depends on $n$, the pigeonhole problem depends on the number of considered pigeons, a circuit depends on the number of bits of its input, etc. Consider for instance a specification of a ripple-adder circuit that takes as input two $\mathsf{n}$-bit vectors and computes their sum: $Adder \stackrel{\text{def}}{=} \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} Sum_{\mathsf{i}} \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} Carry_{\mathsf{i}} \wedge \neg C_1$ where $\mathsf{n}$ is the number of bits of the input, $Sum_{\mathsf{i}} \stackrel{\text{def}}{=} S_{\mathsf{i}} \Leftrightarrow (A_{\mathsf{i}} \oplus B_{\mathsf{i}}) \oplus C_{\mathsf{i}}$, $Carry_{\mathsf{i}} \stackrel{\text{def}}{=} C_{\mathsf{i}+1} \Leftrightarrow (A_{\mathsf{i}} \wedge B_{\mathsf{i}}) \vee (B_{\mathsf{i}} \wedge C_{\mathsf{i}}) \vee (A_{\mathsf{i}} \wedge C_{\mathsf{i}})$, $\oplus$ is the exclusive OR, $A_1, \ldots, A_{\mathsf{n}}$ (resp. $B_1, \ldots, B_{\mathsf{n}}$) are the bits of the first (resp. second) operand of the circuit, $S_1, \ldots, S_{\mathsf{n}}$ is the output (the **S**um), and $C_1, \ldots, C_{\mathsf{n}}$ are the intermediate **C**arries.

Presently, automated reasoning on such specifications requires that we give a concrete value to the parameter $\mathsf{n}$. Besides the obvious loss of generality, this instantiation hides the *structure* of the initial problem which can be however a useful information when reasoning about such specifications: the structure of the proof can in many cases be *guided* by the one of the original specification. This gave us the idea to consider parameterized formulae at the object level and to design a logic to reason about them.

Notice that schemata not only arise naturally from practical problems, but also have a deep conceptual interpretation, putting bridges between logic and

* This work has been partly funded by the project ASAP of the French *Agence Nationale de la Recherche* (ANR-09-BLAN-0407-01).

J. Giesl and R. Hähnle (Eds.): IJCAR 2010, LNAI 6173, pp. 293–308, 2010.

computation. As well as first-order logic abstracts from propositional logic via *quantification*, schemata allow to abstract via *computation*, in a complementary way. Indeed, a schema can be considered as a very specific algorithm taking as input a value for the parameter and generating a propositional formula depending on this value. So a schema can be seen as an algorithm whose codomain is the set of propositional formulae (its domain is the set of integers).

If we want to prove, e.g. that the implementation of a parameterized specification is correct, we need to prove that the corresponding schema is valid *for every value of the parameter*. As usual we actually deal with unsatisfiability: we say that a schema is *unsatisfiable* iff it is (propositionally) unsatisfiable for every value of its parameter. In [1] we introduced a first proof procedure for propositional schemata, called STAB. Notice that there is an easy way to systematically look for a counter-example: we can just enumerate all the values and check the satisfiability of the corresponding formula with a SAT solver. However this naive procedure does not terminate when the schema is unsatisfiable. On the other hand, STAB not only terminates (and much more efficiently) when the schema is satisfiable, but it can also terminate when the schema is unsatisfiable. However it still *does not terminate in general*, as we proved that the (un)satisfiability problem is undecidable for schemata [1]. Still, we proved that STAB terminates for a particular class of schemata, called *regular*, which is thus decidable.

An important restriction of the class of regular schemata is that it cannot contain nested iterations, e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} \bigvee_{\mathsf{j}=1}^{\mathsf{n}} P_{\mathsf{i}} \Rightarrow Q_{\mathsf{j}}$. Nested iterations occur frequently in the specification of practical problems. We take the example of a binary multiplier which computes the product of two bit vectors $A = (A_1, \ldots, A_{\mathsf{n}})$ and $B = (B_1, \ldots, B_{\mathsf{n}})$ using the following decomposition: $A.B = A.\sum_{\mathsf{i}=1}^{\mathsf{n}} B_{\mathsf{i}}.2^{\mathsf{i}-1} = \sum_{\mathsf{i}=1}^{\mathsf{n}} A.B_{\mathsf{i}}.2^{\mathsf{i}-1}$. The circuit is mainly an iterated sum: "$S^1 = 0$" $\wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} (B_{\mathsf{i}} \Rightarrow Add(S^{\mathsf{i}}, \text{"}A.2^{\mathsf{i}-1}\text{"}, S^{\mathsf{i}+1})) \wedge (\neg B_{\mathsf{i}} \Rightarrow (S^{\mathsf{i}+1} \Leftrightarrow S^{\mathsf{i}}))$ where $S^{\mathsf{i}}$ denotes the $\mathsf{i}^{th}$ partial sum (hence $S^{\mathsf{n}}$ denotes the final result) and $Add(x, y, z)$ denotes any schema specifying a circuit which computes the sum $z$ of $x$ and $y$ (for instance the previous *Adder* schema). We express "$A.2^{\mathsf{i}-1}$" by the bit vector $Sh^{\mathsf{i}} = (Sh^{\mathsf{i}}_1, \ldots, Sh^{\mathsf{i}}_{2\mathsf{n}})$ ($Sh$ for *Sh*ift): $(\bigwedge_{\mathsf{j}=1}^{\mathsf{n}} Sh^1_{\mathsf{j}} \Leftrightarrow A_{\mathsf{j}}) \wedge (\bigwedge_{\mathsf{j}=\mathsf{n}}^{2\mathsf{n}} \neg Sh^1_{\mathsf{j}}) \wedge (\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \neg Sh^{\mathsf{i}}_1 \wedge \bigwedge_{\mathsf{j}=1}^{2\mathsf{n}} (Sh^{\mathsf{i}}_{\mathsf{j}+1} \Leftrightarrow Sh^{\mathsf{i}}_{\mathsf{j}}))$ and "$S^1 = 0$" by $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \neg S^1_{\mathsf{i}}$. This schema obviously contains nested iterations.

STAB does not terminate in general on such specifications. We introduce in this paper a new proof procedure, called DPLL$^\star$, which is an extension of the DPLL procedure [9]. Extending DPLL to schemata is a complex task, because the formulae depend on an *unbounded* number of propositional variables (e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$ "contains" $P_1, \ldots, P_{\mathsf{n}}$). Furthermore, propagating the value given to an atom is not straightforward as in DPLL (in $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$ if the value of e.g. $P_2$ is fixed then we must propagate the assignment to $P_{\mathsf{i}}$ but only in the case where $\mathsf{i} = 2$). The main advantage of DPLL$^\star$ over STAB is that it can operate on subformulae occurring at a deep position in the schema. This feature is essential for handling nested iterations. DPLL$^\star$ is sound, complete for satisfiability detection and terminates on a class of schemata, called *regularly nested*.

The paper is organized as follows. Section 2 defines the syntax and semantics of iterated schemata. Section 3 presents the DPLL$^\star$ proof procedure. Section 4

deals with the detection of cycles in proofs, which is the main tool allowing termination. Section 5 presents the class of *regularly nested schemata*, for which DPLL$^\star$ terminates. Section 6 concludes the paper and overviews related works. Due to space restrictions, proofs are omitted or simply sketched (detailed proofs can be found in [2]).

## 2 Schemata of Propositional Formulae

Terms on the signature $\{0, s, +, -\}$ and on a countable set of integer variables $\mathcal{IV}$ are called *linear expressions*, whose set is written $\mathcal{LE}$. As usual we simply write $n$ for $s^n(0)$ $(n > 0)$ and $n.e$ for $e + \cdots + e$ ($n$ times). Linear expressions are considered modulo the usual properties of the arithmetic symbols (e.g. $s(0) + s(s(0)) - 0$ is assumed to be the same as $s(s(s(0)))$ and written 3). The set of first-order formulae built on $\mathcal{LE}$ and $=, <, >$ is called the set of *linear constraints* (or in short *constraints*), written $\mathcal{LC}$. If $C_1, C_2 \in \mathcal{LC}$, we write $C_1 \models C_2$ iff $C_2$ is a logical consequence of $C_1$. This relation is well known to be decidable see e.g. [7]. It is also well known that linear arithmetic admits quantifier elimination. Closed terms of $\Sigma$ (i.e. integers) are denoted by $n, m, i, j, k, l$, linear expressions by $e, f$, constraints by $C, C_1, C_2, \ldots$ and integer variables by $\mathsf{n}, \mathsf{i}, \mathsf{j}$ to make clear the distinction between integer variables and expressions of the meta-language.

To make technical details simpler, and w.l.o.g., only schemata in negation normal form (n.n.f.) are considered. A linear constraint *encloses* a variable $\mathsf{i}$ iff there exist $e_1, e_2 \in \mathcal{LE}$ s.t. $\mathsf{i}$ does not occur in $e_1, e_2$ and $C \models e_1 \leq \mathsf{i} \wedge \mathsf{i} \leq e_2$.

**Definition 1 (Schemata).** *For every $k \in \mathbb{N}$, let $\mathcal{P}_k$ be a set of symbols. The set $\mathfrak{P}$ of* formula patterns *(or, for short,* patterns*) is the smallest set such that:*

- $\top, \bot \in \mathfrak{P}$.
- *If $k \in \mathbb{N}$, $P \in \mathcal{P}_k$ and $e_1, \ldots, e_k \in \mathcal{LE}$ then $P_{e_1,\ldots,e_k} \in \mathfrak{P}$ and $\neg P_{e_1,\ldots,e_k} \in \mathfrak{P}$.*
- *If $\pi_1, \pi_2 \in \mathfrak{P}$ then $\pi_1 \vee \pi_2 \in \mathfrak{P}$ and $\pi_1 \wedge \pi_2 \in \mathfrak{P}$.*
- *If $\pi \in \mathfrak{P}$, $\mathsf{i} \in \mathcal{IV}$, $C \in \mathcal{LC}$ and $C$ encloses $\mathsf{i}$ then $\bigwedge_{\mathsf{i}|C} \pi \in \mathfrak{P}$ and $\bigvee_{\mathsf{i}|C} \pi \in \mathfrak{P}$.*

*A* schema *$S$ is a pair (written as a conjunction) $\pi \wedge C$, where $\pi$ is a pattern and $C$ is a constraint. $C$ is called the* constraint *of $S$, written $C_S$. $\pi$ is called its* pattern*, written $\Pi_S$.*

The first three items differ from propositional logic only in the atoms which we call *indexed propositions* ($e_1, \ldots, e_k$ are called *indices*). The real novel part is the last item. Patterns of the form $\bigwedge_{\mathsf{i}|C} \pi$ or $\bigvee_{\mathsf{i}|C} \pi$ are called *iterations*. $C$ is called the *domain* of the iteration. In [1] only domains of the form $e_1 \leq \mathsf{i} \wedge \mathsf{i} \leq e_2$ were handled, but as we shall see in Section 3, more general classes of constraints are required to define the DPLL$^\star$ procedure. If $C$ is unsatisfiable then the iteration is *empty*. Any occurrence of $\mathsf{i}$ in $\pi$ is *bound* by the iteration. A variable occurrence which is not bound is *free*. A variable which has free occurrences in a pattern is a *parameter* of the pattern. A pattern which is just an indexed proposition $P_{e_1,\ldots,e_k}$ is called an *atom*. An atom or the negation of an atom is called a *literal*.

In [1] a schema was just a pattern, however constraints appear so often that it is more convenient to integrate them to the definition of schema. Informally,

a pattern gives a "skeleton" with "holes" and the constraint specifies how the holes can be filled (this choice fits the abstract definition of schema in [8]). In the following we assume w.l.o.g. that $C_S$ entails $\mathsf{n}_1 \geq 0 \wedge \cdots \wedge \mathsf{n}_k \geq 0$ where $\mathsf{n}_1, \ldots, \mathsf{n}_k$ are the parameters of $\Pi_S$.

For instance, $S \stackrel{\text{def}}{=} P_1 \wedge \bigwedge_{1\leq \mathsf{i}\wedge \mathsf{i}\leq \mathsf{n}}(Q_\mathsf{i} \wedge \bigvee_{1\leq \mathsf{j}\leq \mathsf{n}+1\wedge \mathsf{i}\neq \mathsf{j}} \neg P_\mathsf{n} \vee P_{\mathsf{j}+1}) \wedge \mathsf{n} \geq 1$ is a schema. $P_1$, $Q_\mathsf{i}$, $P_\mathsf{n}$ and $P_{\mathsf{j}+1}$ are indexed propositions. $\bigvee_{1\leq \mathsf{j}\leq \mathsf{n}+1\wedge \mathsf{i}\neq \mathsf{j}} \neg P_\mathsf{i} \vee P_{\mathsf{i}+1}$ and $\bigwedge_{1\leq \mathsf{i}\wedge \mathsf{i}\leq \mathsf{n}}(Q_\mathsf{j} \wedge \bigvee_{1\leq \mathsf{j}\leq \mathsf{n}+1\wedge \mathsf{i}\neq \mathsf{j}} \neg P_\mathsf{i} \vee P_{\mathsf{i}+1})$ are the only iterations, of domains $1 \leq \mathsf{j} \leq \mathsf{n}+1 \wedge \mathsf{i} \neq \mathsf{j}$ and $1 \leq \mathsf{i} \leq \mathsf{n}$. $\mathsf{n}$ is the only parameter of $S$. Finally $\Pi_S$ is $P_1 \wedge \bigwedge_{1\leq \mathsf{i}\wedge \mathsf{i}\leq \mathsf{n}}(Q_\mathsf{i} \wedge \bigvee_{1\leq \mathsf{j}\leq \mathsf{n}+1\wedge \mathsf{i}\neq \mathsf{j}} \neg P_\mathsf{n} \vee P_{\mathsf{j}+1})$ and $C_S$ is $\mathsf{n} \geq 1$. Schemata are denoted by $S$, $S_1$, $S_2 \ldots$, parameters by $\mathsf{n}, \mathsf{n}_1, \mathsf{n}_2 \ldots$, bound variables by $\mathsf{i}, \mathsf{j}$. $\Delta_{\mathsf{i}|C}\, S$ and $\nabla_{\mathsf{i}|C}\, S$ denote generic iterations (i.e. $\bigvee_{\mathsf{i}|C}\, S$ or $\bigwedge_{\mathsf{i}|C}\, S$), $\triangle$ and $\triangledown$ denote generic binary connectives ($\vee$ or $\wedge$), $\Delta^{e_2}_{\mathsf{i}=e_1}\, S$ denotes $\Delta_{\mathsf{i}|e_1\leq \mathsf{i}\wedge \mathsf{i}\leq e_2}\, S$.

Let $S$ be a schema and $\Delta_{\mathsf{i}_1|C_1}\, S_1$, …, $\Delta_{\mathsf{i}_k|C_k}\, S_k$ be all the iterations occurring in $S$. Then $C_S \wedge C_1 \wedge \cdots \wedge C_k$ is called the *constraint context* of $S$, written Context($S$). Notice that Context($S$) loses the information on the *binding positions* of variables. This can be annoying if a variable name is bound by two different iterations or if it is both bound and free in the schema. So we assume that all schemata are such that this situation does not hold[1].

*Substitutions* on integer variables map integer variables to linear arithmetic expressions. We write $[e_1/\mathsf{i}_1, \ldots, e_k/\mathsf{i}_k]$ for the substitution mapping $\mathsf{i}_1, \ldots, \mathsf{i}_k$ to $e_1, \ldots, e_k$ respectively. The application of a substitution $\sigma$ to an arithmetic expression $e$, written $e\sigma$, is defined as usual. Substitution application is naturally extended to schemata (notice that bound variables are not replaced). A substitution is *ground* iff it maps integer variables to integers (i.e. ground arithmetic expressions). An *environment* $\rho$ of a schema $S$ is a ground substitution mapping all parameters of $S$ and such that $C_S\rho$ is true.

**Definition 2 (Propositional Realization).** *Let $\pi$ be a pattern and $\rho$ a ground substitution. The propositional formula $|\pi|_\rho$ is defined as follows:*

- $|P_{e_1,\ldots,e_k}|_\rho \stackrel{def}{=} P_{e_1\rho,\ldots,e_k\rho}$, $|\neg P_{e_1,\ldots,e_k}|_\rho \stackrel{def}{=} \neg P_{e_1\rho,\ldots,e_k\rho}$
- $|\top|_\rho \stackrel{def}{=} \top$, $|\bot|_\rho \stackrel{def}{=} \bot$, $|\pi_1 \wedge \pi_2|_\rho \stackrel{def}{=} |\pi_1|_\rho \wedge |\pi_2|_\rho$, $|\pi_1 \vee \pi_2|_\rho \stackrel{def}{=} |\pi_1|_\rho \vee |\pi_2|_\rho$
- $|\bigvee_{\mathsf{i}|C}\, \pi|_\rho \stackrel{def}{=} \bigvee \{|\pi[i/\mathsf{i}]|_{\rho\cup[i/\mathsf{i}]} | i \in \mathbb{Z}$ *s.t.* $C[i/\mathsf{i}]\rho$ *is valid*$\}$
- $|\bigwedge_{\mathsf{i}|C}\, \pi|_\rho \stackrel{def}{=} \bigwedge \{|\pi[i/\mathsf{i}]|_{\rho\cup[i/\mathsf{i}]} | i \in \mathbb{Z}$ *s.t.* $C[i/\mathsf{i}]\rho$ *is valid*$\}$

*When $\rho$ is an environment of a schema $S$, we define $|S|_\rho$ as $|\Pi_S|_\rho$. $|S|_\rho$ is called a* propositional realization *of $S$.*

Notice that $\top, \bot, \vee, \wedge, \neg$ on the right-hand sides of equations have their standard *propositional* meanings. $\bigvee$ and $\bigwedge$ on the right-hand side are meta-operators denoting respectively the *propositional* formulae $\cdots \vee \cdots \vee \cdots$ and $\cdots \wedge \cdots \wedge \cdots$ or $\bot$ and $\top$ when the sets are empty. In contrast, all those symbols on the left-hand side are *pattern* connectives.

We now make precise the semantics outlined in the introduction. Propositional logic semantics are defined as usual. A *propositional interpretation* is a function mapping every propositional variable to a truth value *true* or $false$.

[1] The rule *Emptiness* of the proof system defined in Section 3 does not preserve this property, but it is easily circumvented by renaming variables.

**Definition 3 (Semantics).** *Let $S$ be a schema. An* interpretation $\mathcal{I}$ of the schemata language *is a pair consisting of an environment $\rho_{\mathcal{I}}$ of $S$ and a propositional interpretation $\mathcal{I}_p$ of $|S|_{\rho_{\mathcal{I}}}$. A schema $S$ is* true *in $\mathcal{I}$ iff $|S|_{\rho_{\mathcal{I}}}$ is true in $\mathcal{I}_p$, in which case $\mathcal{I}$ is a* model *of $S$. $S$ is* satisfiable *iff it has a model.*

Notice that an iteration $\bigvee_{\mathsf{i}|C} \pi$ (resp. $\bigwedge_{\mathsf{i}|C} \pi$) where $C$ is equivalent to $\bot$ (i.e. the iteration is empty), is equivalent to $\bot$ (resp. $\top$).

*Example 1.* Let $S \stackrel{\text{def}}{=} P_1 \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} (P_{\mathsf{i}} \Rightarrow P_{\mathsf{i}+1}) \wedge \neg P_{\mathsf{n}+1} \wedge \mathsf{n} \geq 0$ (as usual, $S_1 \Rightarrow S_2$ is a shorthand for $\neg S_1 \vee S_2$). Then $|S|_{\mathsf{n}\mapsto 0} = P_1 \wedge \neg P_1$, $|S|_{\mathsf{n}\mapsto 1} = P_1 \wedge (P_1 \Rightarrow P_2) \wedge \neg P_2$, $|S|_{\mathsf{n}\mapsto 2} = P_1 \wedge (P_1 \Rightarrow P_2) \wedge (P_2 \Rightarrow P_3) \wedge \neg P_3$, etc. $S$ is clearly unsatisfiable.

The set of satisfiable schemata is recursively enumerable but not recursive [1]. Hence there cannot be a refutationally complete proof procedure for schemata.

The next definitions will be useful in the definition of DPLL$^\star$. Let $\phi$ be a propositional formula and $L$ a (propositional) literal. We say that $L$ occurs *positively* in $\phi$, written $L \sqsubset \phi$, iff there is an occurrence of $L$ in $\phi$ which is not in the scope of a negation.

**Definition 4.** *Let $S$ be a schema and $L$ a literal s.t. the parameters of $L$ are parameters of $S$. We write $L \sqsubset_\Box S$ iff for every environment $\rho$ of $S$, $|L|_\rho \sqsubset |S|_\rho$. We write $L \sqsubset_\Diamond S$ iff there is an environment $\rho$ of $S$ s.t. $|L|_\rho \sqsubset |S|_\rho$.*

*Example 2.* Consider $S$ as in Example 1. We have $P_1 \sqsubset_\Box S$, $P_{\mathsf{n}+1} \sqsubset_\Box S$, $P_2 \not\sqsubset_\Box S$. However $P_2 \sqsubset_\Diamond S$ and $P_2 \sqsubset_\Box (S \wedge \mathsf{n} \geq 1)$. Finally $P_0 \not\sqsubset_\Diamond S$ and $P_{\mathsf{n}+2} \not\sqsubset_\Diamond S$.

Suppose $L$ has the form $P_{e_1,\ldots,e_k}$ (resp. $\neg P_{e_1,\ldots,e_k}$). For a literal $L' \sqsubset S$ of indices $f_1, \ldots, f_k$, $\phi_L(L')$ denotes the formula $\exists \mathsf{i}_1 \ldots \mathsf{i}_n (C_{\mathsf{i}_1} \wedge \cdots \wedge C_{\mathsf{i}_n} \wedge e_1 = f_1 \wedge \cdots \wedge e_k = f_k)$ where $\mathsf{i}_1, \ldots, \mathsf{i}_n$ are all the bound variables of $S$ occurring in $f_1, \ldots, f_k$ and $C_{\mathsf{i}_1}, \ldots, C_{\mathsf{i}_n}$ are the domains of the iterations binding $\mathsf{i}_1, \ldots, \mathsf{i}_n$. $\phi_L(S)$ denotes $\bigvee\{\phi_L(P_{f_1,\ldots,f_k}) \mid P_{f_1,\ldots,f_k} \sqsubset S\}$ (resp. $\bigvee\{\phi_L(\neg P_{f_1,\ldots,f_k}) \mid \neg P_{f_1,\ldots,f_k} \sqsubset S\}$).

**Proposition 1.** *$L \sqsubset_\Box S$ iff $\forall \mathsf{n}_1, \ldots, \mathsf{n}_l (C_S \Rightarrow \phi_L(S))$ is valid, where $\mathsf{n}_1 \ldots \mathsf{n}_l$ are all the parameters of $S$. $L \sqsubset_\Diamond S$ iff $\exists \mathsf{n}_1, \ldots, \mathsf{n}_l (C_S \wedge \phi_L(S))$ is valid.*

Consider e.g. $S$ as in Example 1. For any $e$, $P_e \sqsubset_\Box S$ iff $\forall \mathsf{n} (\mathsf{n} \geq 0) \Rightarrow [e = 1 \vee \exists \mathsf{i} (1 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{n} \wedge e = \mathsf{i}) \vee \exists \mathsf{i} (1 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{n} \wedge e = \mathsf{i} + 1) \vee e = \mathsf{n} + 1]$ is valid. By decidability of linear arithmetic, both $\sqsubset_\Box$ and $\sqsubset_\Diamond$ are decidable. Besides, it is easy to compute the set $\mathcal{L}(S) \stackrel{\text{def}}{=} \{L \mid L \sqsubset_\Box S\}$ for a schema $S$.

## 3 A Proof Procedure: DPLL$^\star$

We provide now a set of (sound) deduction rules (in the spirit of the Davis-Putnam-Logemann-Loveland procedure for propositional logic [9]) complete w.r.t. satisfiability (we know that it is not possible to get refutational completeness). Compared to other proof procedures [1] DPLL$^\star$ allows to rewrite subformulae occurring at deep positions inside a schema — in particular occurring in the scope of iterated connectives: this is crucial to handle nested iterations. DPLL$^\star$ is a tableaux-like procedure: rules are given to construct a tree whose

root is the formula that one wants to refute. The formula is refuted iff all the branches are contradictory.

As usual with tableaux related methods, the aim of branching is to browse the possible interpretations of the schema. As a schema interpretation assigns a truth value to each atom and a number to each parameter, there are two branching rules: one for atoms, called *Propositional splitting* (this rule assigns a value to propositional variables, as the splitting rule in DPLL), and one for parameters, called *Constraint splitting*. However *Constraint splitting* does not give a value to the parameters, but rather restricts their values by refining the constraint of the schema (i.e. $C_S$), e.g. the parameter can be either greater or lower than a given integer, leading to two branches in the tableaux. Naturally, in order to analyze a schema, one has to investigate the contents of iterations. So a relevant constraint to use for the branching is the one that states the emptiness of some iteration. In the branch where the iteration is empty, we can replace it by its neutral element (i.e. $\top$ for $\bigwedge$ and $\bot$ for $\bigvee$), which is done by *Constraint splitting* (this may also entails the emptiness of some other iterations, and thus their replacement by their neutral elements too, this is handled by *Algebraic simplification*). Then in the branch where the iteration is not empty, we can unfold the iteration: this is done by the *Unfolding* rule.

Iterations might occur in the scope of other iterations. Thus their domains might depend on variables bound by the outer iterations. *Constraint splitting* is of no help in this case, indeed it makes a branching only according to the values of the *parameter*: bound variables are out of its scope. Hence we define the rule *Emptiness* that can make a "deep" branching, i.e. a branching not in the tree, but in the schema itself: it "separates" an iteration into two distinct ones, depending on the constraint stating the emptiness of the inner iteration, e.g. $\bigvee_{i=1}^{n}\bigvee_{j=3}^{i} P_i \wedge n \geq 2$ is replaced by $\bigvee_{i=3}^{n}\bigvee_{j=3}^{i} P_i \vee \bigvee_{i=1}^{2} \bot \wedge n \geq 2$.

*Constraint splitting* strongly affects the application of *Propositional splitting*. Indeed *Propositional splitting* only applies on atoms occurring in *all* instances of the schema (formalized by Definition 4), and we saw in Example 2 that this depends on the constraint. Once an atom $A$ is given the value true (resp. false) we can replace it by $\top$ (resp. $\bot$). But this is not as simple as in the propositional case as $A$ may occur in a realization of the schema without occurring in the schema itself (e.g. $P_1$ in $\bigwedge_{i=1}^{n} P_i$ $(\star)$), so we cannot just replace it by $\top$. The simplification is performed by the rule *Expansion* which wraps the indexed propositions that are more general than the considered atom ($P_i$ in $(\star)$) with an iteration whose domain is a disunification constraint stating that the proposition is distinct from the atom (for $(\star)$ this gives: $\bigwedge_{i=1}^{n}\bigwedge_{j|i\neq 1\wedge j=0} P_i$). The introduced iteration is very specific because the bound variable always equals 0 (this variable is not used but we assign it 0 to satisfy the condition in Definition 1 that it has to be enclosed by the domain). Whereas usual iterations shall be considered as "for loops", this one is an "if then else". It makes sense when *Emptiness* or *Constraint splitting* is applied: *if* the condition holds (i.e. if the wrapped proposition differs from the atom) *then* the contents of the iteration hold (i.e. we keep the indexed proposition) *else* the iteration is empty (i.e. we replace it by its neutral element). In $(\star)$, *Emptiness* applies: $\bigwedge_{i|1\leq i\leq n\wedge\exists j(i\neq 1\wedge j=0)}\bigwedge_{j|i\neq 1\wedge j=0} P_i \wedge \bigwedge_{i|1\leq i\leq n\wedge\forall j(i=1\vee j\neq 0)} \top$

(of course the domains can be simplified to allow reader-friendly presentation). Then *Algebraic simplification* gives: $\bigwedge_{\mathsf{i}|1\leq\mathsf{i}\leq\mathsf{n}\wedge\exists\mathsf{j}(\mathsf{i}\neq 1\wedge\mathsf{j}=0)} P_{\mathsf{i}}$, i.e. $\bigwedge_{\mathsf{i}=2}^{\mathsf{n}} P_{\mathsf{i}}$, as expected. This process may seem cumbersome, but it is actually a uniform and powerful way of propagating constraints about nested iterations along the schema.

Finally we may know that an iteration is empty without knowing which value of the bound variable satisfies the domain constraint. Then the *Interval splitting* rule adds some constraints on the involved expressions to ensure this knowledge.

We now define DPLL$^\star$ formally.

**Definition 5 (Tableau).** *A* tableau *is a tree $\mathcal{T}$ s.t. each node $\alpha$ in $\mathcal{T}$ is labeled with a pair $(S_{\mathcal{T}}(\alpha), \mathcal{L}_{\mathcal{T}}(\alpha))$ containing a schema and a finite set of literals.*

If $\alpha$ is the root of the tree then $\mathcal{L}_{\mathcal{T}}(\alpha) = \emptyset$ and $S_{\mathcal{T}}(\alpha)$ is called the *root schema*. The transitive closure of the child-parent relation is written $\prec$. For a set of literals $\mathcal{L}$, $\bigwedge_{\mathcal{L}}$ denotes the pattern $\bigwedge_{L\in\mathcal{L}} L$.

As usual a tableau is generated from another tableau by applying extension rules written $\frac{P}{C}$ (resp. $\frac{P}{C_1|C_2}$) where $P$ is the premise and $C$ (resp. $C_1, C_2$) the conclusion(s). Let $\alpha$ be a leaf of a tree $\mathcal{T}$, if the label of $\alpha$ matches the premise then we can *extend* the tableau by adding to $\alpha$ a child (resp. two children) labeled with $C\sigma$ (resp. $C_1\sigma$ and $C_2\sigma$), where $\sigma$ is the matching substitution. A leaf $\alpha$ is *closed* iff $\Pi_{S_{\mathcal{T}}(\alpha)}$ is equal to $\bot$ or $C_{S_{\mathcal{T}}(\alpha)}$ is unsatisfiable.

When used in a premise, $S[\pi]$ means that the schema $\pi$ occurs in $S$; when used in a conclusion, $S[\pi']$ denotes $S$ in which $\pi$ has been substituted with $\pi'$.

**Definition 6 (DPLL$^\star$ rules).** *The* extension rules *are:*

- Propositional splitting.

$$\frac{(S, \mathcal{L})}{(S, \mathcal{L} \cup P_{e_1,\ldots,e_k}) \;\big|\; (S, \mathcal{L} \cup \neg P_{e_1,\ldots,e_k})}$$

  *if either $P_{e_1,\ldots,e_k} \sqsubset_\square S$ or $\neg P_{e_1,\ldots,e_k} \sqsubset_\square S$, and neither $P_{e_1,\ldots,e_k} \sqsubset_\diamond \bigwedge_{\mathcal{L}} \wedge C_S$ nor $\neg P_{e_1,\ldots,e_k} \sqsubset_\diamond \bigwedge_{\mathcal{L}} \wedge C_S$.*
- Constraint splitting. *For $(\Delta, \varepsilon) \in \{(\bigwedge, \top), (\bigvee, \bot)\}$:*

$$\frac{(S[\Delta_{\mathsf{i}|C}\,\pi], \mathcal{L})}{(S[\Delta_{\mathsf{i}|C}\,\pi] \wedge \exists \mathsf{i} C, \mathcal{L}) \;\big|\; (S[\varepsilon] \wedge \forall \mathsf{i} \neg C, \mathcal{L})}$$

  *if $C_S \wedge \forall \mathsf{i} \neg C$ is satisfiable and free variables of $C$ other than* i *are parameters.*
- *Rewriting:*

$$\frac{(S_1, \mathcal{L})}{(S_2, \mathcal{L})}$$

  *where $C_{S_2} = C_{S_1}$ and $\Pi_{S_1} \to \Pi_{S_2}$ by the following rewrite system:*
  - Algebraic simplification. *For every pattern $\pi$:*

$$\neg\top \to \bot \quad \pi \wedge \top \to \pi \quad \pi \wedge \bot \to \bot \quad \bigwedge_{\mathsf{i}|C} \top \to \top \quad \pi \wedge \pi \to \pi$$

$$\neg\bot \to \top \quad \pi \vee \top \to \top \quad \pi \vee \bot \to \pi \quad \bigvee_{\mathsf{i}|C} \bot \to \bot \quad \pi \vee \pi \to \pi$$

$$\textit{if } \mathrm{Context}(S_1) \wedge \exists \mathsf{i} C \textit{ is unsatisfiable: } \bigwedge_{\mathsf{i}|C} \pi \to \top \quad \bigvee_{\mathsf{i}|C} \pi \to \bot$$

$$\textit{if } \mathrm{Context}(S_1) \Rightarrow \exists \mathsf{i} C \textit{ is valid and } \pi \textit{ does not contain } \mathsf{i}\text{:} \quad \Delta_{\mathsf{i}|C}\,\pi \to \pi$$

- Unfolding. *For* $(\vartriangle, \Delta) \in \{(\wedge, \bigwedge), (\vee, \bigvee)\}$*:*

$$\underset{\mathsf{i}|C}{\Delta}\, \pi \rightarrow \pi[e/\mathsf{i}] \vartriangle \underset{\mathsf{i}|C\wedge \mathsf{i}\neq e}{\Delta}\, \pi \qquad \textit{if } \mathrm{Context}(S_1) \Rightarrow C[e/\mathsf{i}] \textit{ is valid}$$

  *e can be chosen arbitrarily*[2]*.*
- Emptiness. *For* $(\vartriangle, \Delta) \in \{(\wedge; \bigwedge), (\vee; \bigvee)\}$, $(\nabla, \varepsilon) \in \{(\bigwedge; \top), (\bigvee; \bot)\}$*:*

$$\underset{\mathsf{i}|C}{\Delta}(\pi[\underset{\mathsf{i}'|C'}{\nabla}\, \pi']) \rightarrow \underset{\mathsf{i}|C\wedge\exists \mathsf{i}' C'}{\Delta}(\pi[\underset{\mathsf{i}'|C'}{\nabla}\, \pi']) \vartriangle \underset{\mathsf{i}|C\wedge\forall \mathsf{i}' \neg C'}{\Delta}(\pi[\varepsilon])$$

  *if* $\mathrm{Context}(S_1) \wedge \forall \mathsf{i}' \neg C'$ *is satisfiable and* $\mathsf{i}$ *occurs freely in* $C'$*.*
- Expansion.

$$P_{e_1,\ldots,e_k} \rightarrow \bigwedge_{\mathsf{i}|(e_1\neq f_1\vee\cdots\vee e_k\neq f_k)\wedge \mathsf{i}=0} P_{e_1,\ldots,e_k} \qquad \textit{if } P_{f_1,\ldots,f_k} \in \mathcal{L}$$

$$P_{e_1,\ldots,e_k} \rightarrow \bigvee_{\mathsf{i}|(e_1\neq f_1\vee\cdots\vee e_k\neq f_k)\wedge \mathsf{i}=0} P_{e_1,\ldots,e_k} \qquad \textit{if } \neg P_{f_1,\ldots,f_k} \in \mathcal{L}$$

  *if* $\mathrm{Context}(S_1) \wedge e_1 = f_1 \wedge \cdots \wedge e_k = f_k$ *is satisfiable.* $\mathsf{i}$ *is a fresh variable.*

– Interval splitting. *For* $k, l \in \mathbb{N}$, $\Delta \in \{\bigwedge, \bigvee\}$, $\lhd \in \{<, \leq, \geq, >\}$*:*

$$\frac{(S[\Delta_{\mathsf{i}|C\wedge k.\mathsf{i}\lhd e_1 \wedge l.\mathsf{i}\lhd e_2}\, \pi], \mathcal{L})}{(S[\Delta_{\mathsf{i}|C\wedge k.\mathsf{i}\lhd e_1}\, \pi] \wedge l.e_1 \lhd k.e_2, \mathcal{L}) \;\big|\; (S[\Delta_{\mathsf{i}|C\wedge l.\mathsf{i}\lhd e_2}\, \pi] \wedge l.e_1 \not\lhd k.e_2, \mathcal{L})}$$

  *if every free variable of* $C$ *is either* $\mathsf{i}$ *or a parameter, all variables of* $e_1, e_2$ *are parameters and* $k > 0$*,* $l > 0$*.*

A *derivation* is a (possibly infinite) sequence of tableaux $(\mathcal{T}_i)_{i\in I}$ s.t. $I$ is either $\mathbb{N}$ or $[0..k]$ for some $k \geq 0$, and s.t. for all $i > 0$, $\mathcal{T}_i$ is obtained from $\mathcal{T}_{i-1}$ by applying a rule. A derivation is *fair* iff no leaf can be indefinitely "freezed" i.e. either there is $i \in I$ s.t. $\mathcal{T}_i$ contains an irreducible, not closed, leaf or if for all $i \in I$ and every leaf $\alpha$ in $\mathcal{T}_i$ there is $j \geq i$ s.t. a rule is applied on $\alpha$ in $\mathcal{T}_j$.

**Theorem 1 (Soundness and Completeness w.r.t. Satisfiability).** *Consider a fair derivation* $(\mathcal{T}_i)_{i\in I}$*.* $\mathcal{T}_0$ *is satisfiable iff there is* $i \in I$ *s.t.* $\mathcal{T}_i$ *contains an irreducible and not closed leaf.*

The proof can be found in [2], which also contains a DPLL$^\star$ tableau example.

## 4 Looping Detection

The above extension rules do not terminate in general, but this is not surprising as the satisfiability problem is undecidable [1]. Non-termination comes from the fact that iterations can be infinitely unfolded (consider e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$), thus leading to infinitely many new schemata. However, often, newly obtained

[2] e.g. in Section 5 we choose the maximal integer fulfilling the desired property.

schemata have already been seen (up to some relation that remains to be defined) i.e. the procedure is *looping* (e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$ generates $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-1} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$, then $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-2} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$, then $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-k} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$, which are all equal up to a shift of $\mathsf{n}$). This is actually an algorithmic interpretation of a proof by mathematical induction. We now define precisely the notion of looping. We start with a very general definition:

**Definition 7 (Looping).** *Let $S_1, S_2$ be two schemata having the same parameters $\mathsf{n}_1, \ldots, \mathsf{n}_k$, we say that $S_1$* loops *on $S_2$ iff for every model $\mathcal{I}$ of $S_1$ there is a model $\mathcal{J}$ of $S_2$ s.t. $\rho_{\mathcal{J}}(\mathsf{n}_j) < \rho_{\mathcal{I}}(\mathsf{n}_j)$ for some $j \in 1..k$ and $\rho_{\mathcal{J}}(\mathsf{n}_l) \leq \rho_{\mathcal{I}}(\mathsf{n}_l)$ for every $l \neq j$. The induced relation among schemata is called the* looping relation.

For instance, if $S_1 = \bigvee_{\mathsf{i}=1}^{\mathsf{n}-1} P_{\mathsf{i}}$ and $S_2 = \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$, take any model $\mathcal{I}$ of $S_1$ and construct a model $\mathcal{J}$ of $S_2$ as follows: $\rho_{\mathcal{J}}(\mathsf{n}) \stackrel{\text{def}}{=} \rho_{\mathcal{I}}(\mathsf{n}) - 1$, and for every $i \in 1..\rho_{\mathcal{J}}(\mathsf{n})$, $\mathcal{J}_p(P_i) \stackrel{\text{def}}{=} \mathcal{I}_p(P_i)$. It is easy to see that $\mathcal{J}$ is indeed a model. Similarly $S_1 = \bigvee_{\mathsf{i}=2}^{\mathsf{n}} P_{\mathsf{i}}$ loops on $S_2 = \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$: take any model $\mathcal{I}$ of $S_1$ and build a model $\mathcal{J}$ of $S_2$ as follows: $\rho_{\mathcal{J}}(\mathsf{n}) \stackrel{\text{def}}{=} \rho_{\mathcal{I}}(\mathsf{n}) - 1$, and for every $i \in 1..\rho_{\mathcal{J}}(\mathsf{n})$, $\mathcal{J}_p(P_i) \stackrel{\text{def}}{=} \mathcal{I}_p(P_{i+1})$. In both cases, it is intuitive that if DPLL$^\star$ encounters $S_1$ after having seen $S_2$, then it will behave similarly as for $S_2$, hence the name of "looping". Notice that looping also applies e.g. with $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-1} P_{\mathsf{i}}$ and $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} Q_{\mathsf{i}}$, i.e. the name of symbols does not matter. Looping is undecidable (e.g. if $S_2 = \bot$ then $S_1$ loops on $S_2$ iff $S_1$ is unsatisfiable). It is trivially transitive.

**Definition 8.** *Let $\alpha, \beta$ be nodes in a tableau $\mathcal{T}$. $S\mathcal{L}_{\mathcal{T}}(\alpha)$ denotes the schema $S_{\mathcal{T}}(\alpha) \wedge \bigwedge_{\mathcal{L}_{\mathcal{T}}(\alpha)}$. Then $\beta$* loops *on $\alpha$ iff $S\mathcal{L}_{\mathcal{T}}(\beta)$ loops on $S\mathcal{L}_{\mathcal{T}}(\alpha)$.*

The *Looping* rule closes a leaf that loops on some existing node of the tableau. From now on, DPLL$^\star$ denotes the extension rules, plus the *Looping* rule. Theorem 1 still holds [2]. The notion of loop introduced in Definition 8 is undecidable, thus, in practice, we use decidable refinements of looping. Termination proofs work by showing that the set of schemata which are generated by the procedure is finite up to some looping refinement. We make precise this notion:

**Definition 9.** *A binary relation between schemata is a* looping refinement *iff it is a subset of the looping relation. Let $\mathcal{S}$ be a set of schemata and $\triangleright$ a looping refinement. A schema $S \in \mathcal{S}$ is a* $\triangleright$-maximal companion *w.r.t. $\mathcal{S}$ iff there is no $S' \in \mathcal{S}$ s.t. $S \triangleright S'$. The set of all $\triangleright$-maximal companions w.r.t. $\mathcal{S}$ is written $\mathcal{S}/\triangleright$. If it is finite then we say that $\mathcal{S}$ is* finite up to $\triangleright$.

### 4.1 Equality Up to a Shift

We now present perhaps the simplest refinement of looping. A *shiftable* is a schema, a linear constraint, a pattern, a linear expression or a tuple of those. The refinement is defined on shiftables (and not only on schemata) in order to handle those objects in a uniform way. This is useful in the termination proof of Section 5 (and even more in [2]).

**Definition 10.** *Let $s$, $s'$ be shiftables and $\mathsf{n}$ a variable. If $s' = s[\mathsf{n} - k/\mathsf{n}]$ for some $k > 0$, then $s'$ is* equal to $s$ up to a shift of $k$ on $\mathsf{n}$, *written $s' \rightrightarrows^{\mathsf{n}} s$ (or $s' \rightrightarrows^{\mathsf{n}}_k s$ when we want to make $k$ explicit).*

For instance, $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-1} P_{\mathsf{i}} \rightrightarrows_1^{\mathsf{n}} \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$ but $\bigvee_{\mathsf{i}=2}^{\mathsf{n}} P_{\mathsf{i}} \not\rightrightarrows^{\mathsf{n}} \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$. As examples of shiftables which are not schemata : $\mathsf{n}-2 \rightrightarrows_2^{\mathsf{n}} \mathsf{n}$ and $(\forall i \cdot \mathsf{n} \geq i) \rightrightarrows_1^{\mathsf{n}} (\forall i \cdot \mathsf{n}+1 \geq i)$.

The fact that we use syntactic equality makes the refinement less powerful but simpler to implement and easier to reason with. It is easy to check that $\rightrightarrows^{\mathsf{n}}$ is a looping refinement when restricted to schemata having $\mathsf{n}$ as a parameter. Furthermore $\rightrightarrows^{\mathsf{n}}$ is transitive but neither reflexive, nor irreflexive. It is irreflexive for shiftables containing $\mathsf{n}$, and reflexive for shiftables not containing $\mathsf{n}$.

We focus now on sets which are *finite up to equality up to a shift*, in short "$\rightrightarrows^{\mathsf{n}}$-finite": termination proofs go by showing that the set of all schemata possibly generated by DPLL$^\star$ is $\rightrightarrows^{\mathsf{n}}$-finite, thus ensuring that the *Looping* rule will eventually apply. To prove such results we need to reason by induction on the structure of a schema. To do this properly we need closure properties for $\rightrightarrows^{\mathsf{n}}$-finite sets i.e. if we know that two sets are $\rightrightarrows^{\mathsf{n}}$-finite, we would like to be able to combine them and preserve the $\rightrightarrows^{\mathsf{n}}$-finite property. This is generally not possible, e.g. for two $\rightrightarrows^{\mathsf{n}}$-finite sets of shiftables $\mathcal{S}_1$ and $\mathcal{S}_2$, the set $\mathcal{S}_1 \times \mathcal{S}_2$ (remember that shiftables are closed by tuple construction) is generally *not* $\rightrightarrows^{\mathsf{n}}$-finite. For instance take $\mathcal{S}_1 = \{P_{\mathsf{n}}, P_{\mathsf{n}-1}, P_{\mathsf{n}-2}, , \dots\}$ ($\mathcal{S}_1$ is $\rightrightarrows^{\mathsf{n}}$-finite with $\mathcal{S}_1/\rightrightarrows^{\mathsf{n}} = \{P_{\mathsf{n}}\}$) and $\mathcal{S}_2 = \{P_{\mathsf{n}}\}$ (which is finite and thus $\rightrightarrows^{\mathsf{n}}$-finite). Then $\{(P_{\mathsf{n}}, P_{\mathsf{n}}), (P_{\mathsf{n}-1}, P_{\mathsf{n}}), (P_{\mathsf{n}-2}, P_{\mathsf{n}}), \dots\}$ is not $\rightrightarrows^{\mathsf{n}}$-finite: indeed for every $i \in \mathbb{N}$, $(P_{\mathsf{n}-i}, P_{\mathsf{n}})$ is a maximal companion in $\mathcal{S}_1 \times \mathcal{S}_2$, there is thus an infinite set of maximal companions. Consequently $\mathcal{S}_1 \times \mathcal{S}_2$ is not $\rightrightarrows^{\mathsf{n}}$-finite. Hence we have to restrict our closure operators.

**Definition 11.** *Let* $\mathsf{n}$ *be a variable. A shiftable* $s$ *is* translated *w.r.t.* $\mathsf{n}$ *iff for every linear expression* $e$ *occurring in* $s$ *and containing* $\mathsf{n}$ *there is* $k \in \mathbb{Z}$ *s.t.* $e = \mathsf{n}+k$ *(i.e. neither* $k.\mathsf{n}$ *nor* $\mathsf{n}+\mathsf{i}$ *are allowed, where* $k \in \mathbb{Z}$, $k \neq 0$ *and* $\mathsf{i} \in \mathcal{IV}$*).*

*Assume that* $s$ *is translated w.r.t.* $\mathsf{n}$*. The* deviation *of* $s$ *w.r.t.* $\mathsf{n}$*, written* $\delta(s)$*, is defined as* $\delta(s) \stackrel{def}{=} \max\{k_1 - k_2 \mid k_1, k_2 \in \mathbb{Z}, \mathsf{n}+k_1, \mathsf{n}+k_2 \text{ occur in } s\}$. $\delta(s) \stackrel{def}{=} 0$ *if* $s$ *does not contain* $\mathsf{n}$*. Let* $k \in \mathbb{N}$*, we write* $\mathfrak{B}_k \stackrel{def}{=} \{s \mid \delta(s) \leq k\}$.

**Theorem 2.** *Let* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *be two sets of shiftables translated w.r.t. a variable* $\mathsf{n}$*. If* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are* $\rightrightarrows^{\mathsf{n}}$*-finite then, for any* $k \in \mathbb{N}$*,* $\mathcal{S}_1 \times \mathcal{S}_2 \cap \mathfrak{B}_k$ *is* $\rightrightarrows^{\mathsf{n}}$*-finite.*

Consequently (assume all shiftables are translated w.r.t. $\mathsf{n}$): $\{S_1 \vartriangle S_2 \mid S_1 \in \mathcal{S}_1, S_2 \in \mathcal{S}_2\} \cap \mathfrak{B}_k$, where $\vartriangle \in \{\wedge, \vee\}$, is $\rightrightarrows^{\mathsf{n}}$-finite if $\mathcal{S}_1$ and $\mathcal{S}_2$ are $\rightrightarrows^{\mathsf{n}}$-finite; and $\{(\bigwedge_{i|C} \Pi_S) \wedge C_S \mid S \in \mathcal{S}, C \in \mathcal{C}\} \cap \mathfrak{B}_k$ is $\rightrightarrows^{\mathsf{n}}$-finite if $\mathcal{S}$ and $\mathcal{C}$ are $\rightrightarrows^{\mathsf{n}}$-finite.

### 4.2 Refinement Extensions

We now define simple extensions that allow better detection. Consider for example the schema $S$ defined in Example 1. Using DPLL$^\star$ there is a branch which contains: $S' \stackrel{\text{def}}{=} P_1 \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}-1}(P_{\mathsf{i}} \Rightarrow P_{\mathsf{i}+1}) \wedge \neg P_{\mathsf{n}} \wedge \neg P_{\mathsf{n}+1} \wedge \mathsf{n} \geq 0 \wedge \mathsf{n}-1 \geq 0$. $S'$ loops on $S$ but $S'$ is not equal to $S$ up to a shift. However $\neg P_{\mathsf{n}+1}$ is pure in $S'$ (i.e. $P_{\mathsf{n}+1} \not\sqsubset S'$) so $\neg P_{\mathsf{n}+1}$ may be evaluated to true. Therefore we obtain $P_1 \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}-1}(P_{\mathsf{i}} \Rightarrow P_{\mathsf{i}+1}) \wedge \neg P_{\mathsf{n}} \wedge \mathsf{n} \geq 0 \wedge \mathsf{n}-1 \geq 0$, i.e. $S[\mathsf{n}-1/\mathsf{n}] \wedge \mathsf{n} \geq 0$. But $\mathsf{n}-1 \geq 0$ entails $\mathsf{n} \geq 0$ so we can remove $\mathsf{n} \geq 0$ and finally get $S[\mathsf{n}-1/\mathsf{n}]$. We now generalise this example, thereby introducing two new looping refinements: the *pure literal* extension and the *redundant constraint* extension.

As usual a literal $L$ is *(propositionally) pure* in a formula $\phi$ iff its complement does not occur positively in $\phi$. The pure literal rule is standard in propositional theorem proving: it consists in evaluating a literal $L$ to true in a formula $\phi$ if $L$ is pure in $\phi$. It is well-known that this operation preserves satisfiability. The notion of pure literal has to be adapted to schemata. The conditions on $L$ must be strengthened in order to take iterations into account. For instance, if $L = P_{\mathsf{n}}$ and $S = \bigvee_{\mathsf{i}=1}^{2\mathsf{n}} \neg P_{\mathsf{i}}$ then $L$ is not pure in $S$ since $\neg P_{\mathsf{i}}$ is the complement of $L$ for $\mathsf{i} = \mathsf{n}$ (and $1 \leq \mathsf{n} \leq 2\mathsf{n}$). However $P_{2\mathsf{n}+1}$ is pure in $S$ (since $2\mathsf{n}+1 \notin [1..2\mathsf{n}]$).

**Definition 12.** *A literal $L$ is* pure *in a schema $S$ iff for every environment $\rho$ of $S$, $|L|_\rho$ is propositionally pure in $|S|_\rho$.*

It is easily seen that $L$ is pure in $S$ iff $L^c \not\sqsubset_\diamond S$, thus by decidability of $\sqsubset_\diamond$, it is decidable to determine if a literal is pure or not.

The substitution of an indexed proposition $P_{e_1,\ldots,e_k}$ by a pattern $\pi'$ in a pattern $\pi$, written $\pi[\pi'/P_{e_1,\ldots,e_k}]$, is defined as follows: $P_{e_1,\ldots,e_k}[\pi'/P_{e_1,\ldots,e_k}] \stackrel{\text{def}}{=} \pi'$; $Q_{f_1,\ldots,f_k}[\pi'/P_{e_1,\ldots,e_k}] \stackrel{\text{def}}{=} Q_{f_1,\ldots,f_k}$ if $P \neq Q$ or $f_i \neq e_i$ for some $i \in [1..k]$; $(\pi_1 \vartriangle \pi_2)[\pi'/P_{e_1,\ldots,e_k}] \stackrel{\text{def}}{=} \pi_1[\pi'/P_{e_1,\ldots,e_k}] \vartriangle \pi_2[\pi'/P_{e_1,\ldots,e_k}]$; $(\Delta_{\mathsf{i}|C}\,\pi)[\pi'/P_{e_1,\ldots,e_k}] \stackrel{\text{def}}{=} \Delta_{\mathsf{i}|C}\,\pi[\pi'/P_{e_1,\ldots,e_k}]$. For a schema $S$, we set $S[\pi'/P_{e_1,\ldots,e_k}] \stackrel{\text{def}}{=} \Pi_S[\pi'/P_{e_1,\ldots,e_k}]$.

It is easy to show that for a literal $L$ which is pure in a schema $S$, if $S$ (resp. $S[\top/L]$) has a model $\mathcal{I}$ then $S[\top/L]$ (resp. $S$) has a model $\mathcal{J}$ s.t. $\rho_{\mathcal{I}}(\mathsf{n}) = \rho_{\mathcal{J}}(\mathsf{n})$ for every parameter $\mathsf{n}$ of $S$. A schema $S$ in which all pure literals have been substituted with $\top$ is written purified($S$).

**Definition 13.** *Let $\triangleright$ be a looping refinement. We call* the pure extension *of $\triangleright$ the relation $\triangleright'$: $S_1 \triangleright' S_2 \Leftrightarrow \text{purified}(S_1) \triangleright \text{purified}(S_2)$.*

$\triangleright'$ is easily proved to be a looping refinement.

Finally we describe the redundant constraint extension:

**Definition 14.** *Any normal form of a schema $S$ by the following rewrite rules:*

$$C_1 \wedge \cdots \wedge C_k \to C_1 \wedge \cdots \wedge C_{k-1} \quad \text{if } \{C_1, \ldots, C_{k-1}\} \models C_k$$
$$C \to \bot \quad \text{if } C \text{ is unsatisfiable}$$

*is called a* constraint-irreducible *schema of $S$.*

By decidability of satisfiability in linear arithmetic, it is easy to compute a constraint-irreducible schema of $S$.

**Definition 15.** *Let $\triangleright$ be a looping refinement. We call* the constraint-irreducible extension *of $\triangleright$ the relation $\triangleright'$ s.t. for all $S_1, S_2$, $S_1 \triangleright' S_2$ iff there exists $S_1'$ (resp. $S_2'$) a constraint-irreducible schema of $S_1$ (resp. $S_2$) s.t. $S_1' \triangleright S_2'$.*

Once again $\triangleright'$ is easily proved to be a looping refinement.

## 5 A Decidable Class: Regularly Nested Schemata

**Definition 16 (Regularly Nested Schema).** *An iteration $\Delta_{\mathsf{i}|C}\,\pi$ is* framed *iff there are two expressions $e_1, e_2$ s.t. $C \Leftrightarrow e_1 \leq \mathsf{i} \wedge \mathsf{i} \leq e_2$. $[e_1..e_2]$ is called the* frame *of the iteration. A schema $S$ is:*

- Monadic *iff all indexed propositions occurring in $S$ have only one index.*
- Framed *iff all iterations occurring in it are framed.*
- Aligned on $[e_1..e_2]$ *iff it is framed and all iterations have the same frame* $[e_1..e_2]$.
- Translated *iff it is translated (Def.11) w.r.t. every variable occurring in it.*
- Regularly Nested *iff it has a unique parameter* $\mathsf{n}$*, it is monadic, translated and aligned on* $[k..\mathsf{n}-l]$ *for some* $k, l \in \mathbb{Z}$.

Notice that regularly nested schemata allow the nesting of iterations. But they are too weak to express the binary multiplier presented in the Introduction (since only monadic propositions are considered). However $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \bigvee_{\mathsf{j}=1}^{\mathsf{n}} (P_{\mathsf{i}} \Rightarrow Q_{\mathsf{j}+1}) \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \neg Q_{\mathsf{i}+1} \wedge \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$, for instance, is a regularly nested schema: it is obviously monadic; all its iterations have the same domain $1 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{n}$ so they are clearly framed and aligned on $[1..\mathsf{n}]$ (so we have indeed the required alignment for $k = 1$ and $l = 0$); it is translated as no multiplication nor addition between variables occur inside the schema; finally it has of course only one parameter: $\mathsf{n}$. Also, though not needing the nesting of iterations the ripple-adder presented in the introduction is a regularly nested schema. So is a carry look-ahead adder or an arithmetic comparison circuit, or actually any circuit performing linear arithmetic computations on bit-vectors of arbitrary size. One can also express the inclusion of a finite automaton into another one, and similarly for alternating automata.

We divide *Constraint splitting* into two disjoint rules: *framed-Constraint splitting* (resp. *non framed-Constraint splitting*) denotes *Constraint splitting* with the restriction that $\Delta_{\mathsf{i}|C}\,\pi$ (following the notations of the rule) is framed (resp. not framed). We consider the following strategy $\mathfrak{S}$ for applying the extension rules on a regularly nested schema:

1. First only framed-*Constraint splitting* is applied until irreducibility.
2. Then all other rules except *Unfolding* are applied until irreducibility with the restriction that *Expansion* rewrites $P_{e_1}$ iff $e_1$ contains no variable other than the parameter of the schema.
3. Finally only *Unfolding* is applied until irreducibility, with the restriction that if the unfolded iteration is framed then $e$ (in the definition of *Unfolding*) is the upper bound of the frame. We then go back to 1.

For the *Looping* rule we use equality up to a shift with its pure and constraint-irreducible extensions. It is easy to prove that $\mathfrak{S}$ preserves completeness.

**Theorem 3.** $\mathfrak{S}$ *terminates on every regularly nested schema.*

*Proof.* (Sketch) We show that the set $\{S\mathcal{L}_{\mathcal{T}}(\alpha) \mid \alpha \text{ is a node of } \mathcal{T}\}$ — i.e. the set of schemata generated all along the procedure — is finite up to the constraint-irreducible and pure extensions of equality up to a shift. As $S\mathcal{L}_{\mathcal{T}}(\alpha) = \Pi_{S_{\mathcal{T}}(\alpha)} \wedge C_{S_{\mathcal{T}}(\alpha)} \wedge \bigwedge_{\mathcal{L}_{\mathcal{T}}(\alpha)}$, this set is equal to $\{\Pi_{S_{\mathcal{T}}(\alpha)} \wedge C_{S_{\mathcal{T}}(\alpha)} \wedge \bigwedge_{\mathcal{L}_{\mathcal{T}}(\alpha)} \mid \alpha \text{ is a node of } \mathcal{T}\}$. So the task can approximately be divided into four: prove that the set of patterns is finite up to a shift, prove that the set of constraints is finite up to a shift, prove that the set of partial interpretations is finite up to a shift and combine the three results thanks to Theorem 2.

Among those tasks, the hardest is the first one, because it requires an induction on the structure of $\Pi_{S_{\mathcal{T}}(\alpha)}$. For this induction to be achieved properly we need to "trace" the evolution under $\mathfrak{S}$ of every subpattern of $\Pi_{S_{\mathcal{T}}(\alpha)}$. A subpattern can be uniquely identified by its position. So we extend DPLL$^\star$ into T-DPLL$^\star$ (for **T**raced DPLL$^\star$), by adding to the pair $(S_{\mathcal{T}}(\alpha), \mathcal{L}_{\mathcal{T}}(\alpha))$ labelling nodes in DPLL$^\star$ a third component containing a set of positions of $\Pi_{S_{\mathcal{T}}(\alpha)}$. Along the execution of the procedure, this subpattern may be moved, duplicated, deleted, some context may be added around it, some of its subpatterns may be modified. Despite all those modifications, we are able to follow the subpattern thanks to the set of positions in the labels.

As usual, a position is a finite sequence of natural numbers, $\epsilon$ denotes the empty sequence, $s_1.s_2$ denotes the concatenation of $s_1$ and $s_2$ and $\leq$ denotes the prefix ordering. The *positions* of a pattern $\pi$ are defined as follows: $\epsilon$ is a position in $\pi$; if $p$ is a position in $\pi$ then $1.p$ is a position in $\neg\pi$, $\bigwedge_{i|C}\pi$ and $\bigvee_{i|C}\pi$; let $i \in \{1,2\}$, if $p$ is a position in $\pi_i$ then $i.p$ is a position in $\pi_1 \vee \pi_2$ and $\pi_1 \wedge \pi_2$.

For two sequences $s_1, s_2$ s.t. $s_2$ is a prefix of $s_1$, $s_2 \setminus s_1$ is the sequence s.t. $s_2.(s_2 \setminus s_1) = s_1$. In particular for two positions $p_1, p_2$ s.t. $p_2$ is a prefix of $p_1$, $p_2 \setminus p_1$ can be seen as the position *relatively to* $p_2$ of the subterm in position $p_1$.

**Definition 17 (T-DPLL$^\star$).** *A* T-DPLL$^\star$ tableau $\mathcal{T}$ *is the same as a* DPLL$^\star$ *tableau except that a node* $\alpha$ *is labeled with a triple* $(S_{\mathcal{T}}(\alpha), \mathcal{L}_{\mathcal{T}}(\alpha), \mathcal{P}_{\mathcal{T}}(\alpha))$ *where* $\mathcal{P}_{\mathcal{T}}(\alpha)$ *is a set of positions in* $\Pi_{S_{\mathcal{T}}(\alpha)}$. T-DPLL$^\star$ *keeps the behavior of* DPLL$^\star$ *for* $S_{\mathcal{T}}(\alpha)$ *and* $\mathcal{L}_{\mathcal{T}}(\alpha)$*, we only describe the additional behavior for* $\mathcal{P}_{\mathcal{T}}(\alpha)$ *as follows:* $p \to p_1, \ldots, p_k$ *means that* $p$ *is deleted and* $p_1, \ldots, p_k$ *are added to* $\mathcal{P}_{\mathcal{T}}(\alpha)$.

- *Splitting rules and the* Expansion *rewrite rule leave* $\mathcal{P}_{\mathcal{T}}(\alpha)$ *as is.*
- *Rewrite rules. We write* $q$ *for the position of the subpattern of* $\Pi_S$ *which is rewritten. We omit* Emptiness *as it never applies.*
  - Algebraic simplification. *For* $p > q$*:*

    | | |
    |---|---|
    | $p \to q.(1 \setminus (q \setminus p))$ | *for rules where* $\pi$ *occurs on both sides of the rewrite (following the notations of Definition 6), and if* $p$ *is the position of a subpattern of* $\pi$ |
    | $p \to \emptyset$ | *otherwise* |

  - Unfolding. *For* $p > q$ : $\quad p \to\ q.1.(q \setminus p),\ q.2.1.(q \setminus p)$

*(*Interval splitting *and* Emptiness *are untouched as they never apply when the input schema is regularly nested, see [2])*

From now on, $\mathcal{T}$ is a T-DPLL$^\star$ tableau whose root schema is regularly nested of parameter $\mathsf{n}$ and of alignment $[k..\mathsf{n} - l]$ for some $k, l \in \mathbb{Z}$.

The set $\{S\mathcal{L}_{\mathcal{T}}(\alpha) \mid \alpha \text{ is a node of } \mathcal{T}\}$ is actually *not* finite up to a shift. We have to restrict ourselves to a particular kind of nodes, called *alignment nodes*. Then $\{S\mathcal{L}_{\mathcal{T}}(\alpha) \mid \alpha$ is an *alignment* node of $\mathcal{T}\}$ will indeed be finite up to a shift. A node of $\mathcal{T}$ is an *alignment node* iff it is irreducible by step 2 of $\mathfrak{S}$. It is easy to check that every alignment node is framed and aligned on $[k..\mathsf{n} - l - j]$ for some $j > 0$. Thus every alignment node is regularly nested. Furthermore, by irreducility w.r.t. the *Propositional splitting* and *Expansion* rules, the parameter

$\mathsf{n}$ only occurs in the domain of the iteration (otherwise the corresponding literal would be added into $\mathcal{L}$). It can be shown (see [2] for details) that each of the steps 1-3 terminates. Thus every branch $b$ containing a node $\alpha$ is either finite or contains an alignment node $\beta \prec \alpha$, i.e. *an alignment node is always reached*.

Once this preliminary work is done, we can tackle the four aforementioned tasks. Finiteness (up to a shift) of $\{\Pi_{S_{\mathcal{T}}(\alpha)} \mid \alpha \text{ is an alignment node of } \mathcal{T}\}$ is proved by induction: T-DPLL$^\star$ enables to reason by induction and Theorem 2 enables to make use of the inductive hypotheses to prove each inductive case. Finiteness of $\{C_{S_{\mathcal{T}}(\alpha)} \mid \alpha \text{ is an alignment node of } \mathcal{T}\}$ is proved thanks to the constraint-irreducible extension of equality up to a shift. Finiteness of $\{\bigwedge_{\mathcal{L}_{\mathcal{T}}(\alpha)} \mid \alpha \text{ is an alignment node of } \mathcal{T}\}$ is proved thanks to the pure literal extension of equality up to a shift. Finally Theorem 2 enables to combine the three results.

See [2] for the detailed proof. □

## 6 Conclusion

We have presented a proof procedure, called DPLL$^\star$, for reasoning with propositional formula schemata. The main originality of our calculus is that the inference rules may apply at a deep position in a formula, a feature that is essential for handling nested iterations. A looping mechanism is introduced to improve the termination behavior. We defined an abstract notion of looping which is very general, then instantiated this relation into a more concrete version that is decidable, but still powerful enough to ensure termination in many cases.

We identified a class of schemata, called regularly nested schemata, for which DPLL$^\star$ always terminates. This class is much more expressive than the class of regular schemata handled in [1]. The principle of the termination proof is (to the best of our knowledge) original: we investigate how a given subformula is affected by the application of extension rules on the "global" schema. This is done by defining a "traced" version of the calculus in which additional information is provided concerning the evolution of a specific subformula (or set of subformulae, since a formula may be duplicated). This also required a thorough investigation of the properties of the looping relation. We think these ideas could be reused to prove termination of other calculi, sharing common features with DPLL$^\star$ (namely calculi that operate at deep levels inside a formula and that allow cyclic proofs).

We do not know of any similar work in automated deduction. Schemata have been studied in logic (see e.g. [8,12]) but our approach is different from these (essentially proof theoretical) works both in the particular kind of targeted schemata and in the emphasis on the automation of the proposed calculi. However one can find similarities with other works.

Iterations are closely related to fixed-point constructions, in particular in the (modal) $\mu$-calculus[3] [4] (with $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \phi$ translated into something like $\mu X.\phi \wedge X$). However the semantics are very different: that of iterated schemata is restricted to *finite models* (since every parameter is mapped to an integer, the obtained interpretation is finite), whereas models of the $\mu$-calculus may be infinite. Hence

[3] In which many temporal logics e.g. CTL, LTL, and CTL* can be translated.

the involved logic is very different from ours and actually simpler from a theoretical point of view: the $\mu$-calculus admits complete proof procedures and is decidable, whereas schemata enjoy none of those properties. The relation between schemata and the $\mu$-calculus is analogous to the relation between finite model theory [10] and classical first-order logic. The detailed comparison of all those formalisms is worth investigating but out of the scope of the present work.

One can also translate schemata into first-order logic by turning the iterations into (bounded) quantifications i.e. $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \phi$ (resp. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} \phi$) becomes $\forall \mathsf{i}(1 \leq \mathsf{i} \leq \mathsf{n} \Rightarrow \phi)$ (resp. $\exists \mathsf{i}(1 \leq \mathsf{i} \leq \mathsf{n} \wedge \phi)$). This translation is completed by quantifying universally on the parameters and by axiomatizing first-order linear arithmetic. Then one can use inductive theorem provers [6,5]. For such provers, the only decidability results that we know of are due to Kapur et al. (see e.g. [11]) but they do not apply to the formulae obtained by the above translation: they all deal with formulae of the form $\forall \boldsymbol{x}.\phi$ where $\phi$ is *quantifier-free*. However another translation is possible that is better suited to those results: inductive theorem provers are designed to prove results about recursively defined functions (e.g. $\forall x \cdot double(x) = x + x$ where *double* is defined by $double(0) \stackrel{\text{def}}{=} 0$ and $double(s(x)) = s(s(double(x))))$, and it happens that we can define iterations with recursive functions e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$ can be defined by $f(0) \stackrel{\text{def}}{=} \bot$ and $f(s(\mathsf{i})) \stackrel{\text{def}}{=} P(\mathsf{i}) \vee f(\mathsf{i})$, where $P$ is an *uninterpreted* symbol. And indeed ACL2 [3] is able to prove some *very simple* schemata translated this way. However such formulae still does not fit the shape required for the results in [11], due to the presence of uninterpreted symbol functions in the recursive definitions.

Future work includes the implementation of the DPLL$^\star$ calculus and the investigation of its practical performances. It would also be interesting to extend the termination result in Section 5 to non monadic schemata. Extension of the previous results to first-order logic is also planned.

## References


1. Aravantinos, V., Caferra, R., Peltier, N.: A Schemata Calculus For Propositional Logic. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 32–46. Springer, Heidelberg (2009)
2. Aravantinos, V., Caferra, R., Peltier, N.: A Decidable Class of Nested Iterated Schemata (extended version). Technical report, Laboratory of Informatics of Grenoble (2010), `http://arxiv.org/abs/1001.4251`
3. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, New York (1979)
4. Bradfield, J., Stirling, C.: Modal Mu-Calculi. In: Blackburn, P., van Benthem, J.F.A.K., Wolter, F. (eds.) Handbook of Modal Logic. Studies in Logic and Practical Reasoning, vol. 3. Elsevier Science Inc., New York (2007)
5. Bundy, A.: The Automation of Proof by Mathematical Induction. In: [13], pp. 845–911
6. Comon, H.: Inductionless induction. In: [13], ch. 14
7. Cooper, D.: Theorem proving in arithmetic without multiplication. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 7. Edinburgh University Press (1972)

8. Corcoran, J.: Schemata: the concept of schema in the history of logic. The Bulletin of Symbolic Logic 12(2), 219–240 (2006)
9. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. Communication of the ACM 5, 394–397 (1962)
10. Fagin, R.: Finite-Model Theory - A Personal Perspective. Theoretical Computer Science 116, 3–31 (1993)
11. Kapur, D., Subramaniam, M.: Extending Decision Procedures with Induction Schemes. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 324–345. Springer, Heidelberg (2000)
12. Orevkov, V.P.: Proof schemata in Hilbert-type axiomatic theories. Journal of Mathematical Sciences 55(2), 1610–1620 (1991)
13. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning (in 2 volumes). Elsevier, Amsterdam (2001)