TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Interner Bericht

Induction of Linear Temporal Logic Programs

Robert Kolter

Juni 2004

330/04

FACHBEREICH
INFORMATIK

Postfach 3049 · D-67653 Kaiserslautern

Induction of Linear Temporal Logic Programs

Robert Kolter

Department of Computer Science

Technical University of Kaiserslautern

Postfach 3049, 67653 Kaiserslautern, Germany

kolter@informatik.uni-kl.de

June 2004

330/04

We propose a framework for the synthesis of temporal logic programs which are formulated in a simple temporal logic programming language from both positive and negative examples. First we will prove that results from the theory of first order inductive logic programming carry over to the domain of temporal logic. After this we will show how programs formulated in the presented language can be generalized or specialized in order to satisfy the specification induced by the sets of examples.

Keywords: Logic Programming, ILP, Theorem Proving, Temporal Logic

Contents

1 Introduction

The field of Inductive Logic Programming (ILP) is an active research area. However, to the best of our knowledge, the results of this field have not been applied to temporal logic programming by now. We will present a simple programming language and show how programs written in this language can be induced from sets of positive and negative examples. The programming language which we will use is linear in the sense that it only allows the usage of one temporal operator $\circ$, the so called *Next-State-Operator*.
We will proceed as follows:

- We will define fundamental concepts used throughout this report. This includes first order logic, logic programming, automata theory and temporal logic.
- After this we will define the programming language of interest. We will see that this language is very simple but one can model many interesting concepts within this language. After the introduction of this language we will introduce a proof procedure for it. This procedure is a slight generalization of a well known procedure from logic programming.
- After these basic sections we will show how programs and clauses can be generalized and specialized with respect to certain orderings. This gives a way to define refinement operators which specialize and generalize programs and clauses.
- The last part of the report will be a section which introduces a complexity measure for certain classes of programs. We will see how complicated the task of synthesizing a program from positive and negative examples really is.

The basic task to be performed by an ILP system is the following: given two sets $\mathcal{E}^+$ ane $\mathcal{E}^-$ of so called *positive* and *negative* examples, find a program P which has the following properties:

- $P \models e$ for every $e \in \mathcal{E}^+$ and
- $P \not\models e$ for every $e \in \mathcal{E}^-$.

That is, every positive example has to be a logical consequence of the program and no negative example is allowed to be a consequence. This setting has been studied for the case of first order logic for quite a long time. But for temporal logic this approach is new (at least for first order languages since for propositional temporal logic there has been some research in *system refinement*, see e.g. [Pnu92]).

2 Preliminaries

First order logic deals with the task of proving that *something is a logical consequence of something else.* In formal terms this means that one is given a set of *premises* or *axioms* (written as formulas) and he/she is asked to prove that a given theorem (which is again written as a formula) is a logical consequence of these axioms. To formalize this we will have to define a certain kind of syntax, i.e. a language in which everything we use is formulated. We will therefore first define signatures. Signatures can be seen as the *letters* from which the words of our logical language can can be built. After that we define terms, atoms and formulas in general. The last part of the syntax will be concerned with *substitutions*, i.e. mappings which replace certain symbols in a formula with other symbols. The definition of syntax will be complete after that.
All results from this and the following sections will be stated without proving them. The reader which is interested in a more in-depth introduction to first order logic is referred to [Sho67], [Rau02] and [SH61].

2.1 Signatures

As we have mentioned before, the first logical objects which we have to define are signatures. They allow us to form terms and formulas by stating which symbols can be used and which cannot. We will have to deal with two different kind of symbols: function symbols and predicate symbols. While function symbols will be used to build terms, predicate symbols will be used to build atoms. Every formula will then be built from atoms and terms using several connectives and quantifiers.

Definition 2.1 (Signature) *A signature is a tuple* $sig = (F, Pr, V, \alpha)$ *where* F *is a finite set of function-symbols,* Pr *is a finite set of predicate-symbols and* V *is a countably infinite set of variable-symbols. Furthermore* α *is a mapping from* $F \cup Pr \cup V \to \mathbb{N}$ *(called the arity-function) with the following properties:*

1. $\alpha(f) \in \mathbb{N}$ *for every* $f \in F$,
2. $\alpha(p) \in \mathbb{N}$ *for every* $p \in Pr$ *and*
3. $\alpha(x) = 0$ *for every* $x \in V$.

A signature $sig = (\{f_1, \ldots, f_{n_1}\}, \{p_1, \ldots, p_{n_2}\}, \{x_1, \ldots, x_i, \ldots, \}, \alpha)$ will also be written as

$$((f_1, \alpha(f_1)), \ldots, (f_{n_1}, \alpha(f_{n_1})); (p_1, \alpha(p_1)), \ldots, (p_{n_2}, \alpha(p_{n_2})), \{x_1, \ldots, x_i, \ldots\})$$

or

$$((f_1, \alpha(f_1)), \ldots, (f_{n_1}, \alpha(f_{n_1})); (p_1, \alpha(p_1)), \ldots, (p_{n_2}, \alpha(p_{n_2})))$$

if it is clear, which set of variable symbols will be used.

2.2 Terms and Formulas

The first objects which can be built from a signature are *terms*. Terms will be the elements which are the possible values[1] of the variable symbols. We have three different kinds of terms:

- the variables,
- the constants[2] and
- the terms of the form $f(t_1, \ldots, t_n)$.

The next definition formalizes this.

Definition 2.2 (Terms) *Let $sig = (F, Pr, V, \alpha)$ be a signature. Then the set $\mathfrak{T}(sig)$ of terms which are built using symbols from sig is defined inductively as follows:*

1. *For every $x \in V : x \in \mathfrak{T}(sig)$,*
2. *for every constant symbol $c \in F$ (i.e.every $f \in F$ with $\alpha(f) = 0$): $c \in \mathfrak{T}(sig)$,*
3. *if $f \in F$ is a function symbol with $\alpha(f) = n > 0$ and $t_1, \ldots, t_n \in \mathfrak{T}(sig)$ are terms, then $f(t_1, \ldots, t_n) \in \mathfrak{T}(sig)$ and*
4. *no other strings are terms over sig.*

Let us illustrate this in an example.

Example 2.1 *Assume that $sig = ((f, 1), (g, 2); ; \{x_1, x_2, x_3, \ldots)$, i.e. at this moment we don't bother about predicate symbols. Then*

- x_2,
- $f(x_1)$,
- $f(f(x_1), x_2)$
- *and* $g(f(g(f(g(f(x_1), x_2)), x_3)), x_1)$

are terms while $g(x_1)$ is not a term.

The next more complicated kind of objects are the so called *atoms*. Atoms are the *smallest* formulas which can be built from a signature. No connectives, negations or quantifiers are allowed. Formally:

Definition 2.3 (Atoms) *Let $sig = (F, Pr, V, \alpha)$ be a signature. Then the set $\mathfrak{A}(sig)$ of atoms which are built using symbols from sig is defined inductively as follows:*

[1] w.r.t.. a fixed way to define the meaning of a formula

[2] We denote function symbols f with $\alpha(f) = 0$ as constants. Some authors have chosen a different approach but this is uncritical as soon as we have defined semantics

1. *If $p \in Pr$ is a predicate symbol with $\alpha(p) = 0$*[3]*, then $p \in \mathfrak{A}(sig)$,*
2. *if $p \in Pr$ is a predicate symbol with $\alpha(p) = n > 0$ and $t_1, \ldots, t_n \in \mathfrak{T}(sig)$ are terms, then $p(t_1, \ldots, t_n) \in \mathfrak{A}(sig)$ and*
3. *no other strings are atoms over sig.*

Again we will illustrate this definition in an example.

Example 2.2 *Let sig be the signature from example 2.1 with the only difference that Pr is now not empty but it consists of two symbols p and q with $\alpha(p) = 1$ and $\alpha(q) = 2$. Then $p(x_1)$ and $q(f(x_2), x_3)$ are atoms while $q(x_1)$ is not an atom.*

The next two definitions are standard. While atoms and negated atoms are referred to as *literals*, general formulas are built from literals using the *connectives* $\wedge, \vee, \rightarrow$ and $\leftrightarrow$ as well as the *quantifiers* $\forall$ and $\exists$.

Definition 2.4 (Literals) *Let $sig = (F, Pr, V, \alpha)$ be a signature. Then the set $\mathfrak{L}(sig)$ of atoms which are built using symbols from sig is defined as follows:*

1. *If $\varphi \in \mathfrak{A}(sig)$ is an atom, then $\varphi \in \mathfrak{L}(sig)$,*
2. *if $\varphi \in \mathfrak{A}(sig)$ is an atom, then $\neg\varphi \in \mathfrak{L}(sig)$ and*
3. *no other strings are literals over sig.*

So we have

$$\mathfrak{L}(sig) = \mathfrak{A}(sig) \cup \{\neg\varphi | \varphi \in \mathfrak{A}(sig)\}$$

Definition 2.5 (Formulas) *Let $sig = (F, Pr, V, \alpha)$ be a signature. Then the set $\mathfrak{F}(sig)$ of formulas which are built using symbols from sig is defined inductively as follows:*

1. *If $\varphi \in \mathfrak{L}(sig)$ is a literal, then $\varphi \in \mathfrak{F}(sig)$,*
2. *if $\varphi, \psi \in \mathfrak{F}(sig)$ are formulas, then $\neg\varphi, (\varphi \vee \psi), (\varphi \wedge \psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi) \in \mathfrak{F}(sig)$,*
3. *if $\varphi \in \mathfrak{F}(sig)$ is a formula and $x \in V$ is a variable symbol, then $\forall x\varphi \in \mathfrak{F}(sig)$ and $\exists x\varphi \in \mathfrak{F}(sig)$ and*
4. *no other strings are formulas over sig.*

Example 2.3 *Again consider the signature from example 2.2. Then $\neg q(x_1, x_2)$ and $p(f(x_3))$ are literals. Furthermore $\forall x_1(p(x_1) \rightarrow \exists x_2(q(x_2, f(x_3)))$ is a formula.*

[3] p is then called a *propositional* variable

So for a fixed signature *sig* with a nonempty set Pr of predicate symbols we have

$$\mathfrak{A}(sig) \subset \mathfrak{L}(sig) \subset \mathfrak{F}(sig)$$

We will follow the conventions that brackets in formulas are omitted as often as possible, so $\varphi \circ \psi$ represents the formula $(\varphi \circ \psi)$ for every connective $\circ$.
If $\psi = Qx\varphi$ is a formula (for some $Q \in \{\forall, \exists\}$ and some $x \in V$) then the formula φ is called the *scope* of the variable x in ψ If $Q = \forall$, then x is said to be *universally quantified* in ψ while if $Q = \exists$ x is called *existentially quantified* in ψ. Variables which are neither existentially nor universally quantified in a formula ψ are called *free* in ψ.

Example 2.4 *If* $\varphi = \forall x_1 \forall x_2 (p(x_1) \rightarrow q(x_3, x_3))$ *then* x_1 *and* x_2 *are universally quantified in* φ, *while* x_3 *is free in* φ. *If* $\psi = \forall x_1 \forall x_2 (p(x_1) \rightarrow \exists q(x_3, x_3))$ *then* x_1, x_2 *are universally quantified in* ψ *while* x_3 *is existentially quantified in* ψ.

In the sequel we will refer to the set $quan(\varphi)$ as the set of all variables in a formula φ, which are quantified in φ (existentially or universally quantified), while $free(\varphi)$ represents the set of variables which are free in φ. Furthermore $var(\varphi)$ denotes the set of all variables in φ. Since every variable in φ in φ is quantified or free in φ we have $var(\varphi) = quan(\varphi) \cup free(\varphi)$.

2.3 Substitutions and Unification

Substitutions are operations which can be applied to objects and which will yield objects again. The action a substitutions carried out is the replacement of variables by terms. For this let $sig = (F, Pr, V, \alpha)$ be a fixed signature.

Definition 2.6 (Substitution) *A substitution is a mapping* $\theta : V \rightarrow \mathfrak{T}(sig)$.

If θ is a substitution, the extension $\hat{\theta}$ of θ from V to $\mathfrak{T}(sig)$ is defined in the obvious way:

- If $t = x \in V$ then $\hat{\theta}(t) = \theta(x)$ and
- if $t(f(t_1, \ldots, t_n) \in \mathfrak{T}(sig)$, then $\hat{\theta}(t) = f(\hat{\theta}(t_1), \ldots, \hat{\theta}(t_n))$.

This extension $\hat{\theta}$ can now be extended further to a mapping $\bar{\theta}$ defined on $\mathfrak{F}(sig) \cup \mathfrak{T}(sig)$:

- If $\varphi = t \in \mathfrak{T}(sig)$, then $\bar{\theta}(t) = \hat{\theta}(t)$,
- if $\varphi = p(t_1, \ldots, t_n) \in \mathfrak{A}(sig)$, then $\bar{\theta}(\varphi) = p(\hat{\theta}(t_1), \ldots, \hat{\theta}(t_n))$,
- if $\varphi = \neg p(t_1, \ldots, t_n) \in \mathfrak{L}(sig)$, then $\bar{\theta}(\varphi) = \neg p(\bar{\theta}(t_1), \ldots \bar{\theta}(t_n))$ and
- if $\varphi = \psi_1 \circ \psi_2$ for $\psi_1, \psi_2 \in \mathfrak{F}(sig)$ and $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$, then $\bar{\theta}(\varphi) = \bar{\theta}(\psi_1) \circ \bar{\theta}(\psi_2)$.

From now on we will identify $\theta, \hat{\theta}$ and $\bar{\theta}$ and we will write θ for a substitution, no matter on witch set it is defined.
We will write substitutions as sets, i.e. if a substitution θ replaces variables $x_1, \ldots, x_n$ with terms $t_1, \ldots, t_n$ we will write $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$. Each pair x_i/t_i is called a *binding*. A substitution $\sigma = x_1/t_1, \ldots, x_n/t_n\}$where each t_i is a variable is called a *variable substitution*. A variable substitution with the property that for each $i \in \{1, \ldots, n\}$ there is a j with $x_i = t_j$ is called a *renaming substitution* or a *renaming* for short.
We will also use the following notation: if ψ is a formula and θ is a substitution, then $\psi\theta$ stands for the formula $\theta(\psi)$.

Example 2.5 *Consider the formulas* $\varphi_1 = p(x_1, a, x_2) \rightarrow q(x_2, x_1)$ *and* $\varphi_2 = p(x_1, x_1, x_2) \leftrightarrow p(x_2, x_2, x_1)$ *and the substitution* $\theta = \{x_1/f(a), x_2/a\}$. *then we have*

$$\begin{aligned} \theta(\varphi_1) &= p(f(a), a, a) \rightarrow q(a, f(a)) \\ \theta(\varphi_2) &= p(f(a), f(a), a) \leftrightarrow p(a, a, f(a)) \end{aligned}$$

In later chapters we will need the composition of substitutions, i.e. a way to build *bigger* substitutions from *smaller* ones. Therefore we define the composition right here.

Definition 2.7 (Composition of Substitutions) *Let* $\theta_1 = \{x_1/t_1, \ldots, x_n/t_n\}$ *and* $\theta_2 = \{x'_1/t'_1, \ldots, x'_m/t'_m\}$ *be substitutions. Then the composition* $\theta = \theta_1 \circ \theta_2$ *is defined as the substitution which is formed from*

$$\{x_1/t_1\theta_2, \ldots, x_n/t_n\theta_2, \ldots, x'_1/t'_1, \ldots, x'_m/t'_m\}$$

after all bindings of the form $x_i/t_i\theta_2$ *with* $x_i = t_i\theta_2$ *and all bindings of the form* x'_i/t'_i *with* $x'_i \in \{x_1, \ldots, x_n\}$ *have been deleted.*

We denote the substitution which is given by $\emptyset$ with ε and call it the *empty substitution*.
A special class of substitutions turns out to be important. This is the class of substitutions which makes two syntactical different objects (terms, formulas) identical. This means that for $\psi_1, \psi_2 \in \mathfrak{T}(sig) \cup \mathfrak{F}(sig)$ with $\psi_1 \neq \psi_2$ there is a substitution σ such that $\sigma(\psi_1) = \sigma(\psi_2)$.

Definition 2.8 (Unifier) *Let* ψ_1 *and* ψ_2 *be two logical objects.* ψ_1 *and* ψ_2 *are called unifiable if there is a substitution* σ *with* $\sigma(\psi_1) = \sigma(\psi_2)$. σ *is then called a unifier.*

This definition can be extended to sets of objects: if S is a set of objects with $S = \{o_1, \ldots, o_n\}$ then σ is a unifier for S if σ unifies every pair (o_i, o_j).
In general there is more than one unifier for two unifiable objects ψ_1 and ψ_2. However we can point out certain unifiers which are minimal in a way that they do not change more variables as necessary.

Definition 2.9 (Most general Unifier) *Let S be a set of logical objects. A substitution σ is called a most general unifier for S if σ is a unifier for S and for every unifier θ for S there exists a substitution θ' such that $\theta = \sigma \circ \theta'$. If σ is a most general unifier for ψ_1 and ψ_2 we write $\sigma = mgu(\psi_1, \psi_2)$.*

The calculation of unifiers for two or more objects will turn out to be very important. So it is necessary to have efficient algorithms to perform this task. The two most important algorithms are described in [PW78] and [MM82, Apt97]. While the first one is a linear algorithm, the second one is easier to understand although it is in general not linear in runtime. Nevertheless if UNIFY denotes the problem of calculating a unifier for two objects, we have the following theorem.

Theorem 2.1 UNIFY $\in \mathcal{O}(n)$.

2.4 Semantics of First Order Logic

In the last section we have defined the syntax of formulas. This means we have defined what a formula is and what is not. Semantics on the other side is concerned with the meaning of a formula. Therefore we will give a meaning to the symbols in a signature. This includes interpreting the function symbols with functions over a given set (the *universe*) and interpreting the atomic formulas with predicates over the set of terms incudes by the signature and the universe). As we will see, every formula can be evaluated in this way.

Definition 2.10 (Structure) *Let $sig = (F, Pr, V, \alpha)$ be a signature. A structure for sig is a tuple $\mathcal{A} = (U_\mathcal{A}, F_\mathcal{A}, P_\mathcal{A})$ with the following properties:*

1. *$U_\mathcal{A}$ is a nonempty set which we will call the universe of $\mathcal{A}$,*
2. *for every symbol $f \in F$ there is a function $f_\mathcal{A} : U_\mathcal{A}^{\alpha(f)} \to U_\mathcal{A}$ in $F_\mathcal{A}$ and*
3. *for every symbol $p \in Pr$ there is a relation $p_\mathcal{A} : U_\mathcal{A}^{\alpha(p)} \to \{0,1\}$ in $P_\mathcal{A}$.*

Given a formula φ we can evaluate this formula by interpreting the symbols with a structure and interpreting the free variables with elements from the set of terms over the universe of this structure. So an *interpretation* consists of a structure and a mapping which maps free variables to terms.

Definition 2.11 (Interpretation) *Let $\varphi \in \mathfrak{F}(sig)$ be a formula over a signature sig. An interpretation for φ is a tuple $\mathcal{J} = (\mathcal{A}, w)$ where $\mathcal{A}$ is a structure for sig and $w : free(\varphi) \to \mathfrak{T}(sig)$ is a mapping. We define the value of objects inductively as follows:*

1. *If $x \in V$ is a variable, then $\mathcal{J}(x) = w(x)$,*
2. *if $\psi = p(t_1, \ldots, t_n)$, then $\mathcal{J}(\psi) = p_\mathcal{A}(\mathcal{J}(t_1), \ldots, \mathcal{J}(t_n)) \in \{0,1\}$,*
3. *if $\psi = \neg p(t_1, \ldots, t_n)$, then $\mathcal{J}(\psi) = 1 - \mathcal{J}(\psi)$,*

4. *if* $\psi = \psi_1 \wedge \psi_2$, *then* $\mathcal{J}(\psi) = \min\{\mathcal{J}(\psi_1), \mathcal{J}(\psi_2)\}$,

5. *if* $\psi = \psi_1 \vee \psi_2$, *then* $\mathcal{J}(\psi) = \max\{\mathcal{J}(\psi_1), \mathcal{J}(\psi_2)\}$,

6. *if* $\psi = \psi_1 \rightarrow \psi_2$, *then* $\mathcal{J}(\psi) = \max\{\mathcal{J}(\neg\psi_1), \mathcal{J}(\psi_2)\}$,

7. *if* $\psi = \psi_1 \leftrightarrow \psi_2$, *then* $\mathcal{J}(\psi) = \min\{\mathcal{J}(\psi_1 \rightarrow \psi_2), \mathcal{J}(\psi_2 \rightarrow \psi_1)\}$,

8. *if* $\psi = \forall x\varphi$, *then* $\mathcal{J}(\psi) = 1$ *if for all* $a \in U_{\mathcal{A}}$ *we have* $\mathcal{J}_x^a(\varphi) = 1$ *and*

9. *if* $\psi = \exists x\varphi$, *then* $\mathcal{J}(\psi) = 1$ *if for one* $a \in U_{\mathcal{A}}$ *we have* $\mathcal{J}_x^a(\varphi) = 1$.

Here $\mathcal{J}_x^a$ *denotes the interpretation which emerges from* $\mathcal{J}$ *if* x *is interpreted with* a.

Definition 2.12 (Model) 1. *Let* ψ *be a formula and let* $\mathcal{J}$ *be an interpretation. If* $\mathcal{J}(\psi) = 1$ *we write* $\mathcal{J} \models \psi$ *and* $\mathcal{J}$ *is then called a model for* ψ. *If* $\mathcal{J}(\psi) = 0$ *we write* $\mathcal{J} \not\models \psi$.

2. *Let* Ψ *be a set of formulas and let* $\mathcal{J}$ *be an interpretation. We write* $\mathcal{J} \models \Psi$ *if* $\mathcal{J} \models \psi$ *for every* $\psi \in \Psi$. *Again* $\mathcal{J}$ *is called a model in this case.*

3. *A formula* ψ *is called satisfiable if there is an interpretation* $\mathcal{J}$ *with* $\mathcal{J} \models \psi$.

4. *A formula* ψ *is called valid or a tautology if* $\mathcal{J} \models \psi$ *for every interpretation* $\mathcal{J}$. *In this case we also write* $\models \psi$.

5. *A formula* ψ *is called unsatisfiable if* $\mathcal{J} \not\models \psi$ *for every interpretation* $\mathcal{J}$.

6. *A set of formulas* Ψ *is called satisfiable if there is an interpretation* $\mathcal{J}$ *with* $\mathcal{J} \models \Psi$.

7. *A set of formulas* Ψ *is called unsatisfiable if* $\mathcal{J} \not\models \Psi$ *for every interpretation* $\mathcal{J}$.

For a formula ψ we denote the set of models with $\mathrm{Md}(\psi)$, i.e. we have $\mathrm{Md}(\psi) = \{\mathcal{J} | \mathcal{J} \models \psi\}$. Similarly for a set of formulas Ψ we define

$$\begin{aligned} \mathrm{Md}(\Psi) &= \{\mathcal{J} | \mathcal{J} \models \psi \text{ for all } \psi \in \Psi\} \\ &= \bigcap_{\psi \in \Psi} \mathrm{Md}(\psi) \end{aligned}$$

Now as the semantics of formulas is defined we can reason about logical consequences. Therefore we assume that any set Ψ of formulas (so called *axioms*) is given together with a formula ψ. Then ψ is called a *logical consequence* of Ψ if every model of ψ.

Definition 2.13 (Logical Consequence) *Let* Ψ *be a set of formulas and let* ψ *be a formula. We call* ψ *a logical consequence of* Ψ *if* $Md(\Psi) \subseteq Md(\psi)$. *In this case we write* $\Psi \models \psi$. *If* Ψ *consists of a single formula* ψ_0 *we write* $\psi_0 \models \psi$ *and if* $\Psi = \emptyset$ *we write* $\models \psi$.

The last point in this definition is the same as in the definition of tautologies: a formula is a tautology if it is a consequence of the empty set. This is because the empty set is considered to be fulfilled by every interpretation.
Two formulas are considered to be equivalent if they have the same models. This also means that they do imply each other. This is formalized in the next definition.

Definition 2.14 (Logical Equivalence) *Let ψ_1 and ψ_2 be two formulas. Then ψ_1 and ψ_2 are said to be logical equivalent if $\psi_1 \models \psi_2$ and $\psi_2 \models \psi_1$. If ψ_1 and ψ_2 are logical equivalent, we write $\psi_1 \equiv \psi_2$. If ψ_1 and ψ_2 are not logical equivalent we write $\psi_1 \not\equiv \psi_2$.*

Of course one directly has that $\mathrm{Md}(\psi_1) = \mathrm{Md}(\psi_2)$ if $\psi_1 \equiv \psi_2$. Some equivalences are quite useful for later chapters, so we mention them here.

Theorem 2.2 (Important Equivalences) *For all ψ_1, ψ_2, ψ_3 it holds that:*

$$\begin{aligned}
\psi_1 \rightarrow \psi_2 &\equiv \neg\psi_1 \vee \psi_2 \\
\psi_1 \wedge (\psi_2 \wedge \psi_3) &\equiv (\psi_1 \wedge \psi_2) \wedge \psi_3 \\
\psi_1 \vee (\psi_2 \vee \psi_3) &\equiv (\psi_1 \vee \psi_2) \vee \psi_3 \\
\neg(\psi_1 \wedge \psi_2) &\equiv \neg\psi_1 \vee \neg\psi_2 \\
\neg(\psi_1 \vee \psi_2) &\equiv \neg\psi_1 \wedge \neg\psi_2
\end{aligned}$$

The last two equations in theorem 2.2 are usually referred to as *De-Morgan's laws*.
Now as we have defined a semantic consequence-relation ($\models$) we mention a useful tool for proving that a formula is a logical consequence of a set of axioms. The following lemma holds..

Lemma 2.1 *Let Ψ be a set of universally quantified formulas and let ψ be a formula. Then $\Psi \models \psi$ if and only if $\Psi \cup \{\neg\psi\}$ is unsatisfiable.*

Theorem 2.3 (Finiteness Theorem) *Let Ψ be any set of formulas. Then Ψ is is satisfiable if and only if every finite subset of Ψ is satisfiable.*

Theorem 2.3 can be reformulated in such a way that it turns out more useful.

Theorem 2.4 (Compactness Theorem) *Let Ψ be any set of formulas. Then Ψ is unsatisfiable if and only if there is a finite subset of Ψ is unsatisfiable.*

Combining theorem 2.4 and lemma 2.1 gives an outline of a procedure for proving that a formula is a logical consequence of a set of axioms: We negate the formula and prove that the union of the set of axioms and the negated formula is unsatisfiable. In the case that the formula is indeed a logical consequence of the axioms there is a finite subset from which this contradiction can be derived.

2.5 Introduction to Graph Theory

In several chapters we will need the concept of graphs which is well known from mathematics and theoretical computer science. Therefore our treatment will be rather short. A reader which is more interested in this topic is referred to [Die00] or [Wes01]. Roughly speaking, a graph consists of *nodes* and *edges* connecting these nodes. The following definition will make this precise.

Definition 2.15 (Graph) *A graph is a tuple $G = (V, E)$ where V is a finite set of nodes (or vertices) and $E \subseteq V \times V$ is a relation (the set of edges).*

If $G = (V, E)$ is a graph and $v \in V$ is a node from V, we define two attributes for v, the so called *indegree* and the so called *outdegree* of v, that is the number of incoming edges and the number of outgoing edges. Furthermore we will define an important special case of nodes.

Definition 2.16 (Indegree, Outdegree, Root, Terminal node) *Let $G = (V, E)$ be a graph and let $v \in V$ be a node in G. Then the indegree* $\textsc{Indeg}(v)$ *and the outdegree* $\textsc{Outdeg}(v)$ *of v are defined as follows:*

$$\textsc{Indeg}(v) = |\{v' \in V | (v', v) \in E\}|$$
$$\textsc{Outdeg}(v) = |\{v' \in V | (v, v') \in E\}|$$

Furthermore v is called a root if $\textsc{Indeg}(v) = 0$ *and a terminal node if* $\textsc{Outdeg}(v) = 0$.

These concepts allow us to figure out special kinds of graphs: G is called *rooted* if it contains at least one root, *multirooted* if it contains at least two roots and *directed* if for every $e = (v_1, v_2) \in E$ we have that not necessarily $(v_2, v_1) \in E$.

Definition 2.17 (Path, Subpath, Cycle) *Let $G = (V, E)$ be a graph. A path in G is a sequence $\pi = \{v_i\}_{i=1,\ldots,n}$ of nodes $v_i \in V$ such that for every $i \in \{1, \ldots, n-1\}$ we have that $(v_i, v_{i+1}) \in E$.*
A path $\psi = \{v_{i_0}, \ldots, v_{i_m}\}$ in G is a subpath of π if there is an index $j \in \{1, \ldots, n\}$ with $v_j = v_{i_0}$ and for every $k = 1, \ldots, m$ we have that $v_{j+k} = v_{i_0+k}$. ψ is called a proper subpath is ψ is a subpath and $m < n$. A path $\pi = \{v_i\}_{i=1,\ldots,n}$ is called maximal if there is no $v \in V$ with $(v_n, v) \in E$.
A path $\pi = \{v_i\}_{i=1,\ldots,n}$ is called a cycle if $v_i = v_n$. G contains a cycle if there is a path π in G which contains a subpath which is a cycle. G is called acyclic if it does not contain a cycle.

We will illustrate the introduced concepts in an example.

Example 2.6 *Consider the graph G from figure 1. Here v_1 is a root and v_8 is a terminal node since* $\textsc{Indeg}(v_1) = 0$ *and* $\textsc{Outdeg}(v_8) = 0$. *An example for a path is $\pi_1 = \{v_1, v_2, v_6\}$. This path is not maximal since it is a proper subpath of $\pi_2 = \{v_1, v_2, v_6, v_8\}$. Furthermore G contains the cycle $\{v_2, v_4, v_7, v_5, v_2\}$.*

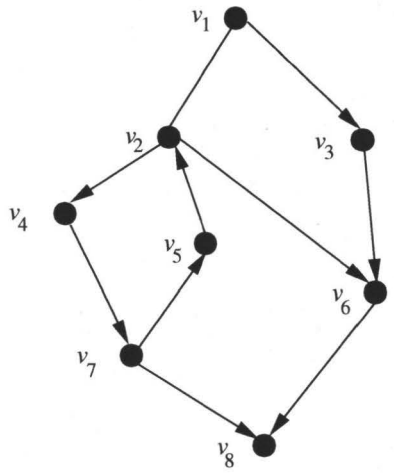

Figure 1: A Graph

2.6 Foundations of Automata Theory

To conclude this chapter we will review the foundations of automata theory and the theory of formal languages in this section. The material presented here is standard, so we will just give the necessary definitions and theorems and illustrate them in some examples. The reader which is interested in a more in-depth treatment of the subject is referred to [Hop69] and [HU79].

Many authors define two kinds of automata: deterministic ones and nondeterministic ones. We will just define the second class (i.e. nondeterministic automata) since deterministic automata are a special case of nondeterministic automata. Therefore assume that a finite set Σ is given. Σ is called the *alphabet*. Every finite sequence $w = \sigma_1 \dots \sigma_n$ of symbols from the alphabet Σ is then called a *word* over Σ. If $n = 0$ this word is denoted by ε and is called the *empty word*. The number $n \in \mathbb{N}$ is called the *length* of the word w and is often denoted by $n = |w|$. So ε is the uniquely determined word w over Σ with $|w| = 0$. The set of all words of positive length over Σ is denoted by Σ^+. Furthermore we set

$$\begin{aligned} \Sigma^n &:= \{w \in \Sigma^+ \mid |w| = n\} \quad \text{for every } n \in \mathbb{N} \\ \Sigma^* &:= \Sigma^+ \cup \{\varepsilon\} \\ &= \bigcup_{n>0} \Sigma^n \cup \{\varepsilon\} \end{aligned}$$

Using these concepts we define the *concatenation* of two words as a mapping *cat* from the set $\Sigma^* \times \Sigma^*$ to Σ^* as follows:

$$\begin{aligned} cat : \Sigma^* \times \Sigma^* &\to \Sigma^* \\ (w_1, w_2) &\mapsto cat(w_1, w_2) = w_1 w_2 \end{aligned}$$

One easily verifies that the following facts hold:

1. for every $w_1, w_2 \in \Sigma^* : |cat(w_1, w_2)| = |w_1| + |w_2|$ and
2. for every $w \in \Sigma^* : cat(w, \varepsilon) = cat(\varepsilon, w) = w$

To simplify notations we will write $w_1 w_2$ for $cat(w_1, w_2)$ from now on.
In the sequel we will be concerned with some subsets of Σ^*. Every such subset L is called a *language* over the alphabet Σ. We will see that certain languages can be characterized in terms of finite automata.

Definition 2.18 (Nondeterministic Finite Automaton) *A (nondeterministic) finite automaton is a tuple*

$$\mathcal{A} = (Q, \Sigma, \Delta, Q_0, Q_f)$$

where Q is a nonempty set (the set of states), Σ is an alphabet, $\Delta : Q \times \Sigma \to 2^Q$ is a (partial) relation (the transition relation) and Q_0 and Q_f are subsets of Q. Q_0 is called the set if initial states and Q_f is called the set of final states.

Deterministic automata are a special class of nondeterministic automata.

Definition 2.19 (Deterministic Finite Automaton) *Let $\mathcal{A} = (Q, \Sigma, \Delta, Q_0, Q_f)$ be a finite automaton. $\mathcal{A}$ is called deterministic if $|Q_0| = 1$ and for every $q \in Q$ and every $\sigma \in \Sigma$ we have $|\Delta(q, \sigma)| = 1$.*

In the case of a deterministic automaton we write δ instead of Δ and q_0 instead of $Q_0 = \{q_0\}$. δ is then called the *transition function* of $\mathcal{A}$.
Automata can be depicted as graphs. States $q \in Q$ are represented as circles which are labeled with q and if q_1, q_2 are states from Q and $\sigma \in \Sigma$ is any symbol from the underlying alphabet such that $q_1 \in \Delta(q_1, \sigma)$ there is an arc which is labeled with σ from the circle which is labeled with q_1 to the circle which is labeled with q_2. Circles for initial states are marked with an incoming arrow and circles for final states are marked grey.

Example 2.7 *Consider the automaton $\mathcal{A} = (\{q_0, q_1, q_2\}, \{a, b\}, \Delta, \{q_0, q_2\}, \{q_1\})$ with*

$$\begin{aligned}
\Delta(q_0, a) &= \{q_1, q_2\} \\
\Delta(q_0, b) &= \{q_0\} \\
\Delta(q_1, a) &= \{q_0, q_2\} \\
\Delta(q_1, b) &= \{q_0, q_2\} \\
\Delta(q_2, a) &= \{q_2\} \\
\Delta(q_2, b) &= \{q_0, q_1\}
\end{aligned}$$

A graph for $\mathcal{A}$ is depicted in figure 2.

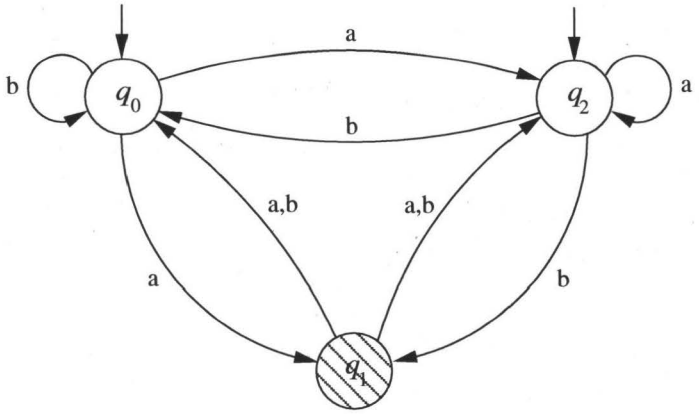

Figure 2: Nondeterministic finite automaton for example 2.7

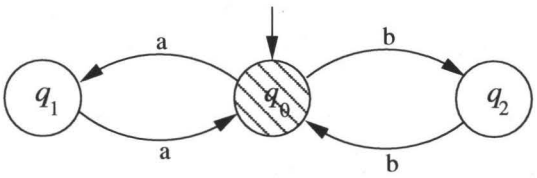

Figure 3: Deterministic finite automaton for example 2.8

Example 2.8 *Consider the automaton* $\mathcal{A} = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_0\})$ *which is determi-nistic. The transition function* δ *is given by*

$$\begin{aligned} \delta(q_0, a) &= q_1 \\ \delta(q_0, b) &= q_2 \\ \delta(q_1, a) &= q_0 \\ \delta(q_2, b) &= q_0 \end{aligned}$$

Its graph is depicted in figure 3. Note that the transition function δ *is partial since* $\delta(q_1, b)$ *and* $\delta(q_2, a)$ *are both not defined.*

In the sequel we will write $\Delta(q, \sigma) \downarrow$ if there is a set $Q' \in 2^Q$ such that $\Delta(q, \sigma) = Q'$. If no such Q' exists we write $\Delta(q, \sigma) \uparrow$. In the case of a deterministic automaton we write $\delta(q, \sigma) \downarrow$ if there is a $q' \in Q$ with $\delta(q, \sigma) = q'$ and $\delta(q, \sigma) \uparrow$ if no such q' exists. It is easily seen that Δ and δ can be extended to a a relation (resp. function) from $Q \times \Sigma^*$ to 2^Q (resp. Q). We refer the reader to [HU79] for details about this construction. From now on we assume that this construction has been carried out.
Using the above extension we define the *language accepted by* $\mathcal{A}$ as the set of all paths

through the automaton which result in a (set of) final state(s):

$$\begin{aligned} L(\mathcal{A}) &= \{w \in \Sigma^* | \Delta(q_0, w) \downarrow \ \& \ \Delta(q_0, w) \subseteq Q_f\} \text{ for some } q_0 \in Q_0 \\ L(\mathcal{A}) &= \{w \in \Sigma^* | \delta(q_0, w) \downarrow \ \& \ \delta(q_0, w) \in Q_f\} \end{aligned}$$

The so called *regular* languages are the most important class of languages. They are exactly the languages which are accepted by a finite automaton. Note that it makes no difference if the accepting automaton is deterministic or not: every nondeterministic finite automaton with n states can be converted into a deterministic one with at most 2^n states which accepts the same language.
The last concept which we will introduce here is the concept of the *product automaton.* This concept is needed to build *bigger* automata from *smaller* ones. So now assume that automata

$$\begin{aligned} \mathcal{A}_1 &= (Q_1, \Sigma, \delta_1, q_{0,1}, Q_{f,1}) \\ \mathcal{A}_2 &= (Q_2, \Sigma, \delta_2, q_{0,2}, Q_{f,2}) \end{aligned}$$

are given. Note that they are defined over the same alphabet Σ. Then the product automaton $\mathcal{A}_1 \times \mathcal{A}_2$ is defined as the automaton which accepts the strings which are accepted both by $\mathcal{A}_1$ and $\mathcal{A}_2$. This is yielded by defining

$$\mathcal{A}_1 \times \mathcal{A}_2 = (Q_1 \times Q_2, \Sigma, \delta, (q_{0,1}, q_{0,2}), Q_{f,1} \times Q_{f,2})$$

with

$$\delta\left((q^{(1)}, q^{(2)}), \sigma\right) = \begin{cases} (\delta_1(q^{(1)}, \sigma), \delta_2(q^{(2)}, \sigma)) & \text{if } \delta_1(q^{(1)}, \sigma) \downarrow, \delta_2(q^{(2)}, \sigma) \downarrow \\ \uparrow & \text{else} \end{cases}$$

Similarly one defines the so called *synchronous product* of two automata. The difference in the following definition is that the resulting automaton does not accept the intersection of the two accepted languages but the union. So we set

$$A_1 \| A_2 = (Q_1 \times Q_2, \Sigma, \delta, (q_{0,1}, q_{0,2}), Q_{f,1} \times Q_{f,2})$$

with

$$\delta\left((q^{(1)}, q^{(2)}), \sigma\right) = \begin{cases} (\delta_1(q^{(1)}, \sigma), \delta_2(q^{(2)}, \sigma)) & \text{if } \delta_1(q^{(1)}, \sigma) \downarrow, \delta_2(q^{(2)}, \sigma) \downarrow \\ (\delta(q^{(1)}, \sigma), q^{(2)}) & \text{if } \delta_1(q^{(1)}, \sigma) \downarrow, \delta_2(q^{(2)}, \sigma) \uparrow \\ (q^{(1)}, \delta(q^{(2)}, \sigma)) & \text{if } \delta_1(q^{(1)}, \sigma) \uparrow, \delta_2(q^{(2)}, \sigma) \downarrow \\ \uparrow & \text{else} \end{cases}$$

2.7 Basics from Temporal Logic

Up till now we were only dealing with *primitive* first order logic. This means that given an interpretation $\mathcal{J}$ and a formula ψ we have either $\mathcal{J} \models \psi$ or $\mathcal{J} \not\models \psi$ regardless

of the context. Now we will introduce the concept of *time* into the language of first order logic. This definition is given in analogy to the definition of first order predicate logic in chapter 2.4. So we will also deal with signatures, terms and formulas. Indeed the definitions of signatures, terms and variables remain identical to the case of first order logic. The only point in which temporal logic differs from first order logic is the fact that so called *temporal Operations* will be allowed. Let us now make this precise.

Definition 2.20 *Let sig be a signature. Then the language $\mathcal{L}_{temp}(sig)$ of temporal formulas over sig is defined inductively as:*

1. *Each formula $\psi \in \mathfrak{F}(sig)$ is in $\mathcal{L}_{temp}(sig)$ and*
2. *for every formula $\psi \in \mathfrak{F}(sig)$ we have $\circ^n\psi \in \mathcal{L}_{temp}(sig)$[4] (for every $n \in \mathbb{N}$).*

This definition is a restriction of common temporal logic languages, since only one temporal operator (namely the operator $\circ$) is allowed. But we will see that this language us expressive enough to derive interesting and practical relevant results.
We will also need some concept of substitution. Recall that a substitution is a mapping which maps variables to terms. We will extend this concept to the class of all temporal formulas. So if θ is a substitution and $\varphi \in \mathcal{L}_{\text{temp}}(\text{sig})$ is a temporal formula, then: if $\varphi = \circ\psi$, then

$$\varphi\theta = \circ(\psi\theta)$$

Similarly for finite sets of formulas $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ we have

$$\Phi\theta = \{\varphi_1\theta, \ldots, \varphi_n\theta\}.$$

At the moment we won't bother about the formal semantics of the operator $\circ$ and and will just state the informal meaning:

If $\mathcal{J}$ is an interpretation which is a model of φ at the next point of time, then $\mathcal{J}$ is a model of $\circ\varphi$ at the present point of time.

We will now define the semantics of temporal languages. Therefore we need the concept of a *temporal state*. A temporal state can be seen as the interpretation of a temporal logic formula at a fixed point of time. Therefore the definition of a temporal state will be the same as the definition of a first order logic interpretation (see page 9).

Definition 2.21 (Temporal State) *Let $sig = (F, Pr, V, \alpha)$ be a signature. A temporal state over sig is a tuple $I = (U, F_I, P_I, w_I)$ with the following properties:*

1. *$J := (U_I, F_I, P_I)$ is a structure for sig and*
2. *$w_I : V \to U_I$ is a function that maps variable symbols to values from the universe of this structure.*

[4]The term $\circ^n$ denotes the string $\underbrace{\circ\cdots\circ}_{n-\text{times}}$.

The evaluation of a first order logic object o (a term or a formula) is obvious:

1. If o is a variable, then $I(o) = w_I(o)$.
2. If o is a term of the form $o = f(t_1, \ldots, f_{\alpha(f)})$ for terms $t_1, \ldots, t_{\alpha(f)}$, then $I(o) = J(f)(J(t_1), \ldots, J(t_{\alpha(f)}))$.
3. If $o = p(t_1, \ldots, t_{\alpha(p)})$ is an atom, then $I(o) = J(p)(J(t_1), \ldots, J(t_{\alpha(p)}))$.
4. If $o = \varphi$ for a $\varphi \in \mathfrak{F}(sig)$, then $I(o) = J(o)$.

In analogy to the case of first order logic we call a temporal state I a model of a formula φ if $I(\varphi) = 1$ and write $I \models \varphi$. If $I(\varphi) = 0$ we write $I \not\models \varphi$. φ is called *satisfiable* if $I \models \varphi$ for a temporal state I, *valid* of a *tautology* if $I \models \varphi$ for *every* temporal state I and *unsatisfiable* of a *contradiction* if $I \not\models \varphi$ for every temporal state I. Two formulas φ_1 and φ_2 are said to be *semantically equivalent* (written $\varphi_1 \equiv \varphi_2$) if for every temporal state I we have that $I \models \varphi_1$ if and only if $I \models \varphi_2$.
The interpretation of formulas which are built using the temporal operator $\circ$ is not possible with only one temporal state since $\circ$ is intended to model the change of time. So we will define the concept of a temporal interpretation as an infinite sequence of states. There the fact that the sequences will be infinite models the assumption that *time does not end*, that is at every possible point t of time, there is always a next point $t + 1$ of time. Note that this requires discrete time, that is the set of indices of a temporal interpretation is countable infinite.

Definition 2.22 (Temporal Interpretation) *A temporal interpretation is an infinite sequence $\mathcal{I} = (I_0, I_1, I_2, \ldots)$ of temporal states $I_j = (U_I, F_I, P_I, W_{I,j})$.*

Note that each state in a temporal interpretation is defined over the same universe and the interpretation of the function symbols and the predicate symbols is identical in every state. The only difference lies in the value of the variables and in the special temporal constructs as we will see now: For this let k be any natural number and let $\mathcal{I} = (I_0, I_1, I_2 \ldots)$ be an interpretation[5]. If we are given a formula $\psi = \circ\varphi$, then we define $I_k(\psi) = I_k(\circ\varphi) = I_{k+1}(\varphi)$. If $\psi \in \mathcal{L}_{\text{temp}}(\text{sig})$ is any temporal formula, then we define $\mathcal{I} \models \psi$ if and only if $I_0 \models \psi$. If Ψ is a set of temporal formulas, then we define $\mathcal{I} \models \Psi$ if and only if $\mathcal{I} \models \psi$ for every $\psi \in \Psi$.
An important special case of the above definition of temporal states and temporal interpretations is the case that *sig* is a signature with $\alpha(p) = 0$ for every $p \in Pr$. So there are no predicate symbols with a nonzero arity. This is the case of *propositional temporal logic*. We will see that this case can be handled much easier than the first order temporal logic case (this is a consequence of the undecidability of first order logic and therefore of first order temporal logic).
In analogy to the case of first order predicate logic we define the concept of a *logical consequence*: A temporal formula is a logical consequence of a set of temporal formulas if every model of this set is also a model of the formula.

[5] We will omit the term *temporal* for states and interpretations from now on if it is clear from the context that we are talking about temporal states and temporal interpretations.

Definition 2.23 *Let $\Phi \subseteq \mathcal{L}_{temp}(sig)$ be a set of temporal formulas and let $\varphi \in \mathcal{L}_{temp}(sig)$ be a formula. The φ is a logical consequence of Φ (written $\Phi \models \varphi$) if for every interpretation $\mathcal{I}$ we have: If $\mathcal{I} \models \Phi$, then $\mathcal{I} \models \varphi$.*

Again in analogy to first order logic we write $\psi \models \varphi$ if $\{\psi\} \models \varphi$ and define $\Phi \models \Psi$ for sets $\Psi \subseteq \mathcal{L}_{\text{temp}}(\text{sig})$ if $\Phi \models \psi$ for every $\psi \in \Psi$.

3 A Simple Linear Programming Language and its Semantics

In this section we will develop a primitive model for the induction of temporal logic programs. Therefore we will now define a simple linear temporal logic programming language which is similar to the language introduced in [Mal97], [Mal98] and TEMPLOG (see [AM88]). The programming language will be called *linear* since every statement will have a form which is similar to a horn clause. So every program statement S from our programming language can be written as

$$S = A \leftarrow B_1, \ldots, B_n$$

We will consider S as a description of *relations* between points of time. If $B_1, \ldots, B_n$ model the present point of time and the head of S has the form $\circ A$ then A will be assumed to model the *next* point of time. This is of course only possible if a suitable discretization of time has been carried out. But this is rather a topic of modeling than a topic of logical semantics so we only assume that the underlying discretization fits our needs and don't bother with this topic from now on.
So now assume that the signature $sig = (F, Pr, V, \alpha)$ is given. Recall the definition of $\mathcal{L}_{\text{temp}}(\text{sig})$ from the last section. We will define some subsets of $\mathcal{L}_{\text{temp}}(\text{sig})$ which we will need for our formalism.

Definition 3.1 *1. For a signature sig the set $\mathcal{L}^{atom}_{temp}(sig)$ of atoms is defined by*

$$\mathcal{L}^{atom}_{temp}(sig) := \{p(t_1, \ldots, t_{\alpha(p)}) | t_1, \ldots, t_{\alpha(p)} \in \mathfrak{T}(sig), p \in Pr\}$$

2. For a signature sig the set $\mathcal{L}^{lit}_{temp}(sig)$ of temporal literals is defined by

$$\mathcal{L}^{lit}_{temp}(sig) := \mathcal{L}^{atom}_{temp}(sig) \cup \{\neg\varphi | \varphi \in \mathcal{L}^{atom}_{temp}(sig)\}$$

3. For a signature sig the sets $\mathcal{L}^{tempatom}_{temp}(sig)$ and $\mathcal{L}^{templit}_{temp}(sig)$ of temporal atoms and temporal literals are defined by

$$\begin{aligned} \mathcal{L}^{tempatom}_{temp}(sig) &:= \{\circ^n\varphi | \varphi \in \mathcal{L}^{atom}_{temp}(sig), n \geq 0\} \\ \mathcal{L}^{templit}_{temp}(sig) &:= \mathcal{L}^{tempatom}_{temp}(sig) \cup \{\neg\varphi | \varphi \in \mathcal{L}^{tempatom}_{temp}(sig)\} \end{aligned}$$

So we have:

$$\mathcal{L}^{\text{atom}}_{\text{temp}}(\text{sig}) \subset \mathcal{L}^{\text{lit}}_{\text{temp}}(\text{sig}) \subset \mathcal{L}^{\text{tempatom}}_{\text{temp}}(\text{sig}) \subset \mathcal{L}^{\text{templit}}_{\text{temp}}(\text{sig})$$

In full analogy to the case of first order logic we define clauses, horn-clauses and temporal clauses. A temporal clause is a finite set of temporal literals. So a temporal clause C has the form $C = \{\psi_1, \ldots, \psi_n\}$ for some n and $\psi_i \in \mathcal{L}^{\text{templit}}_{\text{temp}}(\text{sig})$ for every i. The literal ψ_i is said to occur

- *positive* in C if $\psi_i \in \mathcal{L}^{\text{tempatom}}_{\text{temp}}(\text{sig})$ and

- *negative* in C if $\psi_i \in \mathcal{L}_{\text{temp}}^{\text{templit}}(\text{sig}) \setminus \mathcal{L}_{\text{temp}}^{\text{tempatom}}(\text{sig})$.

For C we define the set POSITIVES as

$$\text{POSITIVES}(C) := \{\psi \in C | \psi \in \mathcal{L}_{\text{temp}}^{\text{tempatom}}(\text{sig})\}$$

So C is a temporal horn clause iff $|\text{POSITIVES}(C)| \leq 1$. In these terms the definitions of definite clauses, unit clauses and goals can be reformulated as

1. $C = \{\psi_1, \ldots, \psi_n\}$ is a *definite clause* iff $|\text{POSITIVES}(C)| = 1$ and $n \geq 2$.
2. $C = \{\psi_1, \ldots, \psi_n\}$ is a *unit clause* iff $|\text{POSITIVES}(C)| = 1$ and $n = 1$.
3. $C = \{\psi_1, \ldots, \psi_n\}$ is a *goal* iff $|\text{POSITIVES}(C)| = 0$ and $n \geq 1$.

Definition 3.2 *A temporal definite clause is a set $C = \{\psi_1, \ldots, \psi_n\}$ with the following properties:*

1. $\psi_1 \in \mathcal{L}_{temp}^{tempatom}(sig)$ *and*
2. $\text{POSITIVES}(C) = \{\psi_2, \ldots, \psi_n\}$.

Definite temporal clauses will be written as $\circ\varphi \leftarrow \psi_2, \ldots, \psi_n$ with $\psi_1 = \circ\varphi$. Now *temporal unit clauses* form a special case of general definite temporal clauses. Assume that START is a symbol which does not occur in *sig*. A temporal unit clause is then given as a clause of the form $\psi \leftarrow \text{START}$ for some $\psi \in \mathcal{L}_{\text{temp}}^{\text{atom}}(\text{sig})$.
Given a definite temporal clause $C = \varphi \leftarrow \varphi_1, \ldots, \varphi_n$ we define two sets $\text{HEAD}(C)$ and $\text{BODY}(C)$ as follows:

$$\begin{aligned} \text{HEAD}(C) &= \{\varphi\} = \text{POSITIVES}(C) \\ \text{BODY}(C) &= \{\varphi_1, \ldots, \varphi_n\} \end{aligned}$$

The temporal programming language $\mathcal{L}_t$ is defined as follows.

Definition 3.3 *A temporal program P is any finite nonempty set consisting of*

- *definite clauses,*
- *unit clauses,*
- *definite temporal and*
- *temporal unit clauses.*

The language of all such programs is denoted by $\mathcal{L}_t$.

Now let $P \in \mathcal{L}_t$ and let $\mathcal{J}$ be a temporal interpretation. That is, $\mathcal{J} = (I_0, \ldots, I_j, \ldots)$ is an infinite sequence of temporal states as defined in Definition 2.21. Again we have:

Lemma 3.1 *Let $P \in \mathcal{L}_t$ be a program and let G be a goal. Then $P \models G$ iff $P \cup \{\leftarrow G\} \models \Box$.*

Proof. Let $P \in \mathcal{L}_t$ be a program and let $\mathcal{J} = (I_0, I_1, \ldots, I_j, \ldots)$ be an arbitrary temporal interpretation. We have:

$$\begin{array}{lll} P \models \alpha & \text{iff} & \mathcal{J} \models P \succ \mathcal{J} \models_t G \\ & \text{iff} & \text{for every } j : I_j \models P \succ J_j \models G \\ & \text{iff} & \text{for no } j : I_j \models P \succ I_j \not\models G \\ & \text{iff} & \text{for no } j : I_j \models P \succ I_j \models \neg G \\ & \text{iff} & \text{for no } j : I_j \models P \text{ and } I_j \models \neg G \\ & \text{iff} & \text{for every } \mathcal{J} : \mathcal{J} \not\models P \cup \{\leftarrow G\} \\ & \text{iff} & P \cup \{\leftarrow G\} \models \Box. \end{array}$$

So the claim is proved. □

In the sequel we will have to introduce a proof procedure which is capable of handling temporal clauses. This procedure was introduced by Abadi and Manna under the name *temporal resolution* in [AM88] as a special case of a more general proof method introduced in [AM86]. We will see that SLD-resolution has to be only slightly modified in order to reach this.

3.1 A Temporal Proof Procedure

We assume that the reader is familiar with the concept of SLD-resolution. A resolvent of a goal G with a program clause C using the unifier σ is again a goal. In our temporal logic programming language this will be exactly the same. But we will have to change the SLD-resolution rule such that it takes the operator $\circ$ into account.

Definition 3.4 (Temporal SLD-Resolution) *Let $P \in \mathcal{L}_t$ be a program, let $C \in P$ be a clause and let $G = \phi_1, \ldots, \phi_n$ with $\phi_i = \circ^{j_i}\varphi_i$ ($j_i \geq 0$) a goal. Then a temporal SLD-resolvent of C and G is a goal which is built using the following rules:*

(FO) *If $C = \varphi \leftarrow \circ^{j'_1}\psi_1, \ldots, \circ^{j'_m}\psi_m$ and if there is an i such that $j_i = 0$ and φ_i and φ are unifiable with $\sigma = mgu(\varphi, \varphi_i)$ then the resolvent is*

$$\left(\phi_1, \ldots, \phi_{i-1}, \circ^{j'_1}\psi_1, \ldots, \circ^{j'_m}\psi_m, \phi_{i+1}, \ldots, \phi_n\right)\sigma$$

(T) *If $C = \circ^k\varphi \leftarrow \circ^{j'_1}\psi_1, \ldots, \circ^{j'_m}\psi_m$ and if there is an i such that $j_i = k$ and φ_i and φ are unifiable with $\sigma = mgu(\varphi, \varphi_i)$ then the resolvent is*

$$\left(\phi_1, \ldots, \phi_{i-1}, \circ^{j'_1}\psi_1, \ldots, \circ^{j'_m}\psi_m, \phi_{i+1}, \ldots, \phi_n\right)\sigma$$

So this proof procedure is just a slight modification to the procedure of SLD-resolution. If G' is a resolvent of a clause $P_i \in P$ and G we write $G' = \text{Res}(P_i, G, \sigma)$ with $\sigma = \text{mgu}(G_j, P_i)$ for some j. So we have $P \cup \{\leftarrow G\} \vdash G'$ if there is an i and a j such that $G' = \text{Res}(P_i, G, \text{mgu}(G_j, P_i)$ while $P \cup \{\leftarrow G\} \vdash^* G'$ iff there is a sequence $G_0, \ldots, G_i, \ldots, G_n$ of goals with $G_0 = G, G_n = G'$ and for all $i < n$ we have $P \cup \{\leftarrow G\} \vdash^* G_i$ and $P \cup \{\leftarrow G_i\} \vdash G_{i+1}$.

Without proving this we will now state the result which is important for us, namely that temporal SLD-resolution is sound and refutation complete. The concepts of *soundness* and *refutation completeness* need some further explanation. Therefore let $\vdash$ be *any* proof procedure, let Φ be a set of formulas and let φ be any formula. $\vdash$ is called

- *sound* if $\Phi \vdash^* \varphi$ implies $\Phi \models \varphi$ and
- *refutation complete* if $\Phi \models \varphi$ implies $\Phi \cup \{\neg\varphi \vdash^* \square$.

Intuitively, these concepts mean that

- if $\vdash$ is sound, then only logical consequences of the premises can be derived using this procedure and
- if $\vdash$ is refutation complete, then one can detect a contradiction by assuming that the negation of the formula to be proved is true.

The following theorem states that temporal SLD-resolution has both of the above properties.

Theorem 3.1 *Let $P \in \mathcal{L}_{temp}(sig)$ be a program and let G be a goal. Then it holds that*

$$P \models G \quad \textit{iff} \quad P \cup \{\neg G\} \vdash^* \square$$

3.1.1 Reasoning About Properties of Temporal Programs

In this subsection we will concentrate on temporal logic programs over propositional signatures and especially on procedures for proving logical consequence in this case. We will propose several results which rely on the representation of a program $P \in \mathcal{L}_{\text{temp}}(\text{sig})$ by an automaton called the *representing automaton for P*.
The approach which we will describe relies on techniques from the field of *model checking* (see e.g. [BCM+90] and especially [RPVW95]). The idea of this approach will be the following: To prove that $P \models \varphi$ holds, one constructs an automaton for the program P and another automaton for the negation of φ, i.e. for $\neg\varphi$. The construction will yield that if there is no accepting sequence of transitions in the automaton for P which is also possible in the automaton for $\neg\varphi$, then $P \cup \{\neg\varphi\} \models \square$, i.e. $P \models \varphi$. Since we only allow propositional formulas and programs at this point of the discussion, the problem is decidable.
The approach from [RPVW95] presents an algorithm for constructing an automaton for a single formula. So we first have to build such a formula from P. Let P be the program which consists of the set $\{P_1, \ldots, P_n\}$. Since every statement P_i is a temporal horn clause, every P_i has the form

$$P_i = \varphi^{(i)} \leftarrow \varphi_1^{(i)}, \ldots, \varphi_{n_i}^{(i)}$$

or

$$P_i = \varphi^{(i)} \leftarrow \text{START}.$$

In the latter case we will identify P_i with $\varphi^{(i)}$ from now on. So a formula representing the conjunction of all program statements from P is clearly given as

$$\begin{aligned}
\hat{P} &= \bigwedge_{i=1}^{n} P_i \\
&= \bigwedge_{i=1}^{n} P_i = \varphi^{(i)} \leftarrow \varphi_1^{(i)}, \ldots, \varphi_{n_i}^{(i)} \\
&\equiv \bigwedge_{i=1}^{n} \left(\varphi^{(i)} \vee \neg \left(\varphi_1^{(i)} \wedge \cdots \wedge \varphi_{n_i}^{(i)} \right) \right) \\
&\equiv \bigwedge_{i=1}^{n} \left(\varphi^{(i)} \vee \bigvee_{j=1}^{n_i} \neg \varphi_j^{(i)} \right)
\end{aligned}$$

One easily sees that for any interpretation $\mathcal{J}$ we have $\mathcal{J} \models P$ if and only if $\mathcal{J} \models \hat{P}$. So to prove that $P \models \varphi$, we can restrict on proving that the formula $\hat{P} \wedge \neg\varphi$ is unsatisfiable.
To construct the automaton we need some more concepts from the field of automata theory. Finite automata (see section 2.6) are only capable of accepting finite words. But we will soon see that we need a way to handle infinite sequences of symbols, so called *infinite words* or *ω-words*. These words are accepted by so called *$\omega-$automata*, especially *Büchi-automata*. We will use an extended concept of Büchi-automata which we will define below.
Recall that a *word* over an alphabet Σ was defined to be a finite sequence of symbols from Σ. The concept of words and languages will now be expanded to infinite sequences.

Definition 3.5 ($\omega-$word, $\omega-$language) *Let Σ be a finite alphabet. A total function $\alpha : \mathbb{N} \to \Sigma$ is called an infinite word, or an omega$-$word. An $\omega-$language is any set of $\omega-$words. The set of all $\omega-$words over the alphabet Σ is denoted with Σ^ω.*

So an $\omega-$word can be characterized by the sequence of function values of α. If w is an $\omega-$word, we also write $w = \alpha(0)\alpha(1)\ldots$.
For every $\omega-$word α we identify two sets $\mathrm{Occ}(\alpha)$ and $\mathrm{Inf}(\alpha)$ of symbols which occur (infinitely often) in α :

$$\begin{aligned}
\mathrm{Occ}(\alpha) &= \{\sigma \in \Sigma | \text{ there is an } i \text{ such that } \alpha(i) = \sigma\} \\
\mathrm{Inf}(\alpha) &= \{\sigma \in \Sigma | \text{ for every } i \text{ there is a } j > i \text{ such that } \alpha(j) = \sigma\}
\end{aligned}$$

An $\omega-$automaton is an automaton which accepts infinite words.

Definition 3.6 ($\omega-$automaton) *An $\omega-$automaton is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, q_0, Acc)$ with a finite set Q of states, a finite alphabet Σ, an initial state $q_0 \in Q$, a partial transition relation $\Delta : Q \times \Sigma \to 2^Q$ and an acceptance component Acc.*

An infinite sequence of states which are visited processing an infinite word α is called a *run*.

Definition 3.7 *Let $\mathcal{A} = (Q, \Sigma, \Delta, q_0, Acc)$ be an $\omega-$automaton. A run of $\mathcal{A}$ on an infinite word α is an infinite sequence $\gamma = \gamma(0), \gamma(1) \ldots$ of states satisfying*

1. $\gamma(0) = q_0$
2. *for every $i > 0$ we have $\gamma(i) \in \Delta(\gamma(i-1), \alpha(i))$.*

Different kinds of $\omega-$automata only differ in the restrictions which are put on the acceptance component Acc. In the case that Acc is a set of states from G, the corresponding automaton is called a *Büchi-automaton.*

Definition 3.8 (Büchi-automaton) *Let $\mathcal{A} = (Q, \Sigma, \Delta, q_0, Acc)$ be an $\omega-$automaton. $\mathcal{A}$ is called a Büchi-automaton if $Acc = F \subseteq Q$. A word $\alpha \in \Sigma^\omega$ is accepted by $\mathcal{A}$ if there is a run γ of $\mathcal{A}$ on α with $Inf(\gamma) \cap F \neq \emptyset$.*

So an infinite word α is accepted by a Büchi-automaton $\mathcal{A}$ if there is a run γ of $\mathcal{A}$ on α such that γ infinitely often leads to an accepting state of $\mathcal{A}$. Since the set of accepting states is a subset of the set Q of all states and Q is finite, at least one accepting state must be visited infinitely often.
In analogy to the theory of finite automata, one defines the language accepted by a Büchi-automaton as

$$\begin{aligned} L(\mathcal{A}) &= \{\alpha \in \Sigma^\omega | \alpha \text{ is accepted by } \mathcal{A}\} \\ &= \{\alpha \in \Sigma^\omega | \text{there is a run } \gamma \text{ of } \mathcal{A} \text{ on } \alpha \text{ with } Inf(\gamma) \cap F \neq \emptyset\} \end{aligned}$$

Example 3.1 *Consider the automaton $\mathcal{A} = (\{q_0, q_1, q_2\}, \{a, b\}, \Delta, q_0, \{q_1, q_2\})$ from figure 4 every $\omega-$word can either start with a or b. Since q_1 and q_2 are final states, the set of $\omega-$words accepted by this automaton is exactly the set*

$$L(\mathcal{A}) = \{w \in \{a, b\}^\omega | w = xa^\omega, x \in \{a, b\}\}$$

that is the set of all words which start with a or b followed by infinitely many occurrences of a.

For the construction of representing automata, we will extend the concept of Büchi-automata a little bit. Therefore we will now define a *labeled* Büchi-automaton.

Definition 3.9 (Labeled Büchi-Automaton) *A labeled Büchi-automaton is a tuple $\mathfrak{A} = (\mathcal{A}, \mathcal{D}, \mathcal{L})$ with*

1. *$\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ is a Büchi-automaton,*
2. *$\mathcal{D} \neq \emptyset$ is some finite set (the domain) and*
3. *$\mathcal{L} : Q \to \mathcal{D}$ is a function (the labeling function).*

$\mathcal{L}$ will assign labels to every state $q \in Q$. The other concepts for Büchi-automata carry over to this extension.

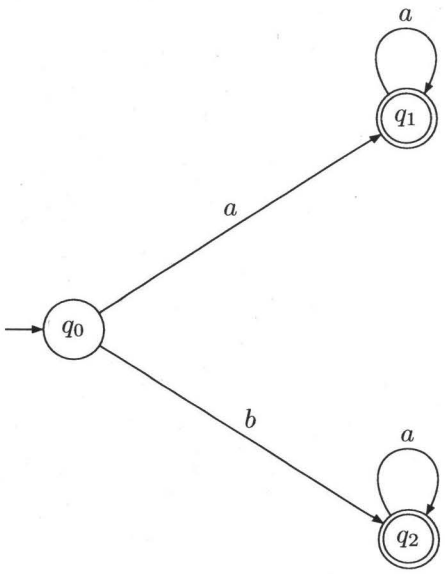

Figure 4: Büchi-automaton

3.1.2 Construction of the Representing Automaton

The construction of the representing automaton for a program P was originally designed to proceed in three steps: First, the formula representing the program P was transformed to an equivalent formula in negation-normalform, i.e. to a formula in which negation symbols only occur in front of atoms. Due to the representation of P by

$$\hat{P} = \bigwedge_{i=1}^{n} \left(\varphi^{(i)} \vee \bigvee_{j=1}^{n_i} \neg \varphi_j^{(i)} \right)$$

this step can be skipped since $\hat{P}$ is already in negation-normalform. This second step consists of a transformation of this formula into a graph while the third step consists of transferring this graph into a Büchi automaton. The three steps were proposed in [RPVW95] and extended in [vW99]. We will concentrate on the transformation into a set of nodes and the transformation of the set of nodes into a Büchi-automaton.

Transformation of the formula into a set of nodes The nodes in the graph will have a format as depicted in figure 5. The semantics of this node format will be the

Name	*Incoming*	*New*	*Old*	*Next*	*Father*

Figure 5: Node Format

following:

1. *Name* is a string which denotes the name of the formula which is represented by the node.
2. *Incoming* represents the set of nodes already generated which have the current node as a successor.
3. *New* is a set of formulas which have not been processed in any other node before and must hold in the current node.
4. *Old* is the set of formulas which must hold in the current node and which have already been processed.
5. *Next* is the set of all temporal formulas which have to hold in a **direct** successor of the current node.
6. *Father* is the node which has been split in order to reach the current node (we will soon make this precise).

The set of all nodes which have been completely generated will be denoted by $\mathcal{N}$.

Given input $\hat{P}$, an initial node n is created with $New(n) = \{\hat{P}\}, Old(n) = \emptyset, Next(n) = \emptyset$ together with an incoming edge which is labeled with INIT. The next steps are processed in depth-first-search:

1. Take the next node n which is not fully processed.
2. If $New(n) = \emptyset$, then set $\mathcal{N} = \mathcal{N} \cup \{n\}$ and call an update procedure (see below).
3. If $New(n) \neq \emptyset$, then take a formula φ from $New(q)$, set $New(q) = New(q) \setminus \{\varphi\}$ and continue as follows:
   a) If φ is a literal, then
      i. if $\neg\varphi \in Old(q)$, then abort, since a contradiction is found.
      ii. if $\neg\varphi \notin Old(q)$, then set $Old(q) = Old(q) \cup \{\varphi\}$.
   b) If φ is not a literal, then
      i. if $\varphi = \varphi_1 \wedge \varphi_2$, then set $New(q) = New(q) \cup \{\varphi_1, \varphi_2\}$.
      ii. if $\varphi = \varphi_1 \vee \varphi_2$, then call a splitting operation (see below)
      iii. if $\varphi = \circ\psi$, then we add an arc from n to a node representing the formula ψ which is labeled with $\circ\psi$.

Two operations are used in the above algorithm which have not yet been explained. These are the *update* and *splitting* operations. We will present them here:

The Update-operation This operation is called whenever a node n is fully processed, that is it has no more formulas in $New(n)$ which have to be processed. Then it has to be checked if there is a node n' in $\mathcal{N}$ which is labeled with the same formulas as n, i.e. $Old(n) = Old(n')$ and $New(n) = New(n')$.

1. If such a node n' exists, we have to process as follows:
   a) $\mathcal{N} = \mathcal{N} \setminus \{n'\}$
   b) $Incoming(n) = Incoming(n) \cup Incoming(n')$
   c) $\mathcal{N} = \mathcal{N} \cup \{n\}$
2. If no such node n' exists, we proceed as follows:
   a) $\mathcal{N} = \mathcal{N} \cup \{n\}$
   b) create a node n' and an edge (n, n')
   c) set $New(n') = Next(n)$
   d) set $Old(n') = Next(n') = \emptyset$

The Splitting-operation This operation is only called when a node is processed which represents a formula φ of the form $\varphi = \varphi_1 \vee \varphi_2$. So φ can be fulfilled in three ways: either φ_1 is fulfilled or φ_2 is fulfilled or both φ_1 and φ_2 are fulfilled. So in order to show that φ can **not** be fulfilled one has to show that both restrictions of φ ($\varphi = 0$ and $\varphi = 1$) cannot be fulfilled. In order to reach this, the splitting-operation creates two new nodes n_1 and n_2 with $New(n_1) = \{\varphi_1\}$ and $New(\varphi_2) = \{n_2\}$. These two nodes are added to the list of unprocessed nodes and the algorithm continues.

After processing every node a set $\mathcal{N}$ of fully processed nodes is created. This set will then be transformed into a Büchi-automaton in order to reason about the (non-)emptiness of the accepted language described above.

Transformation of the set of nodes into a Büchi-automaton The transformation of the set computed graph into a Büchi-automaton is now straightforward: for every node $n_i \in \mathcal{N}$ a state q_i is created and the alphabet Σ consists of a single symbol c (this is only needed to ensure that Σ is not empty). The set of initial states is the set of states which are constructed from nodes which are labeled with *Init*. Finally there is a transition from q_i to q_j if and only if n_i occurs in $Incoming(n_j)$.

The labeling of the states is now constructed as follows: If $sig = (F, Pr, V, \alpha)$ is the signature of interest, then $\mathcal{D} = 2^{Pr}$. Clearly $\mathcal{D}$ is finite since Pr is. We define two Pos and Neg in the following way:

$$Pos(q) = Old(q) \cap Pr$$
$$Neg(q) = \{\varphi | \neg\varphi \in Old(q) \& \varphi \in Pr\}$$

The labeling of a state q is now defined as the set of variables from Pr which are compatible with $Old(q)$, i.e. every set of variables which have to be true in order to reach the state:

$$\begin{aligned}\mathcal{L}(q) &= \{X \in \mathcal{D} | X \text{ is compatible with } Old(q)\} \\ &= \{X \in \mathcal{D} | X \subseteq Pr \ \& \ Pos(X) \subseteq X \ \& \ X \cap Neg(X) = \emptyset\}\end{aligned}$$

3.1.3 Proving Logical Consequences Using the Representing Automaton

Recall the fact that for any set Φ of temporal formulas and any temporal formula φ one has

$$\Phi \models \varphi \text{ if and only if } \Phi \cup \{\neg\varphi\} \models \Box$$

In our case we have $\Phi = P$. The condition can be checked in the case of propositional signatures by constructing the representing automaton A_P for P and the representing automaton $A_{\neg\varphi}$ for φ and checking if the language accepted by the product automaton $\mathcal{A} = A_P \times A_{\varphi}$ is empty. This procedure is justified by the following theorem:

Theorem 3.2 *Let $P \in \mathcal{L}_t$ be a program, let $\varphi \in \mathcal{L}_{temp}^{tempatom}(sig)$ be a goal and let $\mathcal{A} = A_P \times A_{\neg\varphi}$. Then $P \models \varphi$ if and only if $L(\mathcal{A}) = \emptyset$.*

The proof of the above theorem is given in [RPVW95] and we will skip it here. But we mention that the structure of $\mathcal{A}$ can be exploited to refine programs which we will see in later sections.

We conclude this subsection by formulating algorithms which implement the proof strategy described above. These formulations are restrictions of a more general algorithm described in [RPVW95] for the full linear temporal logic LTL. Again we assume that t is a global table containing nodes as described above and $\mathcal{N}$ is the set of fully processed nodes. The first algorithm (algorithm 1) implements the update-operation. The second algorithm (algorithm 2) is a simple implementation of the splitting procedure described above.

The original formulation of the general LTL verification algorithm in [RPVW95] uses a recursive procedure called EXPAND in order to process a node. This procedure has

Algorithm 1 Operator UPDATE

Input: node $n = [Name, Incoming, New, Old, Next, Father]$

find $n' \in \mathcal{N}$ with $Old(n) = Old(n')$ and $New(n) = New(n')$
if such n' exists **then**
  $\mathcal{N} \leftarrow \mathcal{N} \setminus \{n'\}$
  $Incoming(n) \leftarrow Incoming(n) \cup Incoming(n')$
  $\mathcal{N} \leftarrow \mathcal{N} \cup \{n\}$
else
  $\mathcal{N} \leftarrow \mathcal{N} \cup \{n\}$
  create new node n' and edge $e = (n, n')$
  $New(n') \leftarrow Next(n)$
  $Old(n') \leftarrow Next(n)$
end if

Algorithm 2 Operator SPLIT

Input: node $n = [Name, Incoming, New, Old, Next, Father]$ with $Name(n) = \varphi_1 \vee \varphi_2 =: \varphi$

create new node n_1
create new node n_2
$New(n_1) \leftarrow \{\varphi_1\}$
$New(n_2) \leftarrow \{\varphi_2\}$
add n_1 and n_2 to the set of nodes to be processed

two inputs: the actual node to be processed and the set $\mathcal{N}$ of completely processed nodes. At least $\mathcal{N}$ has to be passed by reference since it is possible that it is changed in this algorithm. Furthermore three operations *new-name*, New_1 and New_2 are used which have the following semantics:

- *new-name* creates a string which has not been created before. This string will be the entry of the *Name* field of a node to be created.
- The semantics of the other two operations is defined as follows (note that the semantics are simpler than those presented in [RPVW95][6] since our logic is a strict subset of LTL): If $\varphi = \varphi_1 \vee \varphi_2$ is a formula then

$$New_1(\varphi) = \{\varphi_1\}$$
$$New_2(\varphi) = \{\varphi_2\}$$

The complete operator is stated in algorithm 3.
Since we have constructed a single formula $\hat{P}$ from the program P we can now construct the whole graph for P by constructing an initial node in the way mentioned above and expand this node starting with $\mathcal{N} = \emptyset$. This is summarized in algorithm 4.
All which is necessary to prove logical implication is now to give an algorithm which is able to transform a graph G_P into a representing Büchi-automaton A_P. Such an algorithm is given in algorithm 5
Note that the automaton A_P generated from G_P has an empty set of final states. This enables us to inherit the set of final states from the automaton $A_{\neg\varphi}$ for the negation of the formula φ to be proved which we will construct now. So assume that any temporal formula φ is given. The construction of $A_{\neg\varphi}$ can be done using algorithm 6. But $\neg\varphi$ must have final states in order to accept any words from a language. But since we work on automata constructed from formulas $\neg\varphi$, we can directly exploit the structure of this formula. So we define a state $q \in Q$ to be accepting, i.e. $q \in Q_f$ if every path from an initial state q_0 to q has the property that the union of all $Old(q')$ for every q' on this path satisfies the formula $\neg\varphi$. We will call such a path *accepting*.
Now we have defined every operation which is necessary. The task of testing whether $P \models \varphi$ holds can now be solved as follows:

1. Build G_P from P using algorithm 4.
2. Build A_P from G_P using algorithm 5.
3. Build $A_{\neg\varphi}$ from φ using algorithm 6.
4. Build $\mathcal{A} = A_P \times A_{\neg\varphi}$.
5. If $L(\mathcal{A}) = \emptyset$, then $P \models \varphi$.

An implementation of this strategy can be done very efficiently using symbolic representations of finite automata. For example, one can use Binary Decision Diagrams ([Bry86]) or Integer Decision Diagrams ([Gun97]).

[6] there a further operation $Next_1$ is proposed which is not needed here since we only operate on formulas φ which would yield $Next_1(\varphi) = \emptyset$

Algorithm 3 Operator EXPAND

Input: node n, set $\mathcal{N}$ of nodes

Output: manipulated set $\mathcal{N}$

if $New(n) = \emptyset$ **then**
  if there is $n' \in \mathcal{N}$ with $Old(n) = Old(n')$ and $Next(n) = Next(n')$ **then**
    $Incoming(n') \leftarrow Incoming(n') \cup Incoming(n)$ {$\mathcal{N}$ is manipulated directly}
    return $\mathcal{N}$
  else
    $s \leftarrow$ *new-name*()
    return EXPAND($[s, New(n), Next(n), \emptyset, \emptyset, s]$)
  end if
else
  $\varphi \leftarrow$ next formula from $New(n)$
  $New(n) \leftarrow New(n) \setminus \{\varphi\}$
  if $\varphi = 0$ or φ_1 or $\varphi = p$ or $\varphi = \neg p$ for some variable p **then**
    if $\varphi = 0$ or $\neg\varphi \in Old(n)$ **then**
      return $\mathcal{N}$
    else
      $Old(n) \leftarrow Old(n) \cup \{\varphi\}$
      return EXPAND($n, \mathcal{N}$)
    end if
  else if $\varphi = \varphi_1 \vee \varphi_2$ **then**

$$n_1 \leftarrow \begin{bmatrix} \textit{new-name}() \\ Incoming(n) \\ New(n) \cup (\{New_1(\varphi), \}) \setminus Old(n) \\ Old(n) \cup \{\varphi\} \\ Next(n) \\ Name(n) \end{bmatrix}^T$$

$$n_2 \leftarrow \begin{bmatrix} \textit{new-name}() \\ Incoming(n) \\ New(n) \cup (\{New_2(\varphi), \}) \setminus Old(n) \\ Old(n) \cup \{\varphi\} \\ Next(n) \\ Name(n) \end{bmatrix}^T$$

    return EXPAND(n_2, EXPAND($n_1, \mathcal{N}$))
  else if $\varphi = \varphi_1 \wedge \varphi_2$ **then**

$$n' \leftarrow \begin{bmatrix} Name(n) \\ Incoming(n) \\ New(n) \cup (\{\varphi_1, \varphi_2\}) \setminus Old(n) \\ Old(n) \cup \{\varphi\} \\ Name(n) \end{bmatrix}^T$$

    return EXPAND($n', \mathcal{N}$)
  else
    $\{\varphi = \circ\psi\}$

$$n' \leftarrow \begin{bmatrix} Name(n) \\ Incoming(n) \\ New(n) \\ Old(n) \cup \{\varphi\} \\ Next(n) \cup \{\psi\} \\ Name(n) \end{bmatrix}^T$$

    return EXPAND($n', \mathcal{N}$)
  end if
end if

Algorithm 4 Operator MAKEGRAPH

Input: formula $\hat{P}$ created from a program P
Output: graph $G_P = (\mathcal{N}, E)$ for P

1: $s \leftarrow$ *new-name*()
2: $n \leftarrow \begin{bmatrix} s \\ \{\text{INIT}\} \\ \{\hat{P}\} \\ \emptyset \\ \emptyset \\ s \end{bmatrix}^T$
3: **return** EXPAND$(n, \emptyset)$

Algorithm 5 Operator TRANSFORM

Input: graph $G_P = (\mathcal{N} = \{n_1, \ldots, n_{|\mathcal{N}|}\}, E)$ for P
Output: representing Büchi-automaton $A_P = (Q, \Sigma, \Delta, Q_0, Q_f)$ for P

1: $Q \leftarrow \emptyset$
2: $Q_f \leftarrow \emptyset$
3: $\Sigma \leftarrow \{c\}$
4: **for** $i = 1, \ldots, |\mathcal{N}$ **do**
5: $\quad Q \leftarrow Q \cup \{q_i\}$
6: $\quad$ **if** INIT $\in$ *Incoming*(n_i) **then**
7: $\quad\quad Q_0 \leftarrow Q_0 \cup \{q_i\}$
8: $\quad$ **end if**
9: **end for**
10: **for** $i = 1, \ldots, |Q|$ **do**
11: $\quad \Delta(q_i, c) \leftarrow \emptyset$
12: $\quad$ **for** $j = 1, \ldots, |Q|$ **do**
13: $\quad\quad$ **if** $n_i \in$ *Incoming*(n_j) **then**
14: $\quad\quad\quad \Delta(q_i, c) \leftarrow \Delta(q_i, c) \cup \{q_j\}$
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: **end for**
18: **return** $(Q, \Sigma, \Delta, Q_0, Q_f)$

Algorithm 6 Operator TRANSFORMFORMULA

Input: formula φ

Output: representing Büchi-automaton $A_{\neg\varphi} = (Q, \Sigma, \Delta, Q_0, Q_f)$ for $\neg\varphi$

```
1: G ← MAKEGRAPH(¬φ)
2: A ← TRANSFORM(G)
3: for i = 1, ..., |A.Q| do
4:    if every path from a q ∈ A.Q_0 to A.q_i is accepting then
5:       A.Q_f ← A.Q_f ∪ {A.q_i}
6:    end if
7: end for
8: return A
```

3.2 The First Order Case

We will now relax the restrictions on the signature *sig* in the way that we now allow arbitrary sets of variables, function symbols and predicate symbols. The only restriction will be that every symbol has a fixed arity. Without loss of generality we can assume that we operate on the set $V = \{x_1, \ldots, x_i, \ldots\}$ or at least on a suitable subset of it. Since the problem of deciding logical consequence in first order logic is undecidable (see [SH61], even when we restrict ourselves to sets Φ of clauses as axioms, the problem remains undecidable, see [SS88]). So we can't hope for a decision procedure in full generality.

3.2.1 A Primitive Algorithm

For now assume that $sig = (F, Pr, V, \alpha)$ is given and P is a linear temporal logic program over this signature. Since the temporal SLD-resolution principle is sound and refutation complete, we can give a goal α to P and check if $P \models \alpha$ holds. Therefore we have to construct an SLD-tree rooted with $\leftarrow \alpha$ and check if it is possible that we derive the empty clause $\square$. An algorithm which implements this principle has to implement a *fair* search rule, i.e. every possible resolution step which is possible has to be carried out after a finite amount of time. We will present an algorithm which performs a breadth-first-search in order to ensure fairness. The principle is as follows:

- The SLD-tree will be seen as a directed acyclic labeled graph rooted with a node which is labeled with $\leftarrow \alpha$.
- Every node in the tree will have a parameter, called its *level*.
- A path from the root node to any other node visits nodes in ascending order wrt. their levels.
- An expansion is carried out in order to construct the nodes on the $n+1-$st level from the nodes on the $n-$th level of the graph.

- $P \models \alpha$ holds if and only if there is a level l and a node n on this level such that n is labeled with $\square$.

The formal construction will be given below. To formulate it we need some concepts for nodes, arcs etc.

Definition 3.10 *An SLD-node is a tuple*

$$(\alpha, \text{SUCC}, n, l)$$

where α is a temporal goal, $\text{SUCC} \in \mathbb{N}^*$ *is a sequence of integers and n and l are integers. If a node has no successors, this will be denoted by* $\text{SUCC} = (-1)$.

The interpretation of this concept of an SLD-node is the following: *alpha* is the actual goal which still has to be processed, n is the index of the node, l is its level and SUCC is the sequence of indices of the successors in the graph, i.e. if $G = (V, E), n_i \in V$ is a node and $j \in \text{SUCC}$, then $(n_i, n_j) \in E$. Furthermore we need two projections functions LABEL and LEVEL defined as follows:

$$\text{LABEL}\,((\alpha, \text{SUCC}, n, l)) = \alpha$$
$$\text{LEVEL}\,((\alpha, \text{SUCC}, n, l)) = l$$

The algorithm which we will present will use a global table which contains all nodes which have been generated so far. This table will be denoted by t. To formulate this algorithm we will define two operators EXPANDNODE and EXPANDLEVEL which have the following semantics:

- EXPANDNODE is given a program P and a node n and input and it constructs every possible successor in an SLD-tree which is rooted with the goal $\text{LABEL}(n)$
- EXPANDLEVEL is given a graph $G = (V, E)$, a program P and an integer l as input and expands every $n \in V$ with $\text{LEVEL}(n) = l$.

Initially, the global algorithm is given a program P and a goal α as input. To check whether $P \models \alpha$ holds, we first create a node n_0 with

$$n_0 = (\alpha, (-1), 0, 0)$$

and then call EXPANDLEVEL for level 0. If no positive result is found, we call EXPANDLEVEL for level 1 and so forth. The operators and the global algorithm are formulated in the following algorithms.
A formalization of the outline from above is now given in algorithm 9
The soundness of algorithm 9 is immediate: one only has to notice that

- every resolution step on the current level is indeed carried out and
- the search strategy which is implemented is fair, i.e. no possible expansion of a node has to wait infinitely long.

Algorithm 7 Operator EXPANDNODE

Input: temporal program $P = \{P_1, \ldots, P_n\}$ and node $k = (\alpha, \text{SUCC}, r, l)$

1: Let α have the form $\alpha_1, \ldots, \alpha_m$
2: max $\leftarrow$maximum index of a node in t
3: **for** $i = 1, \ldots, m$ **do**
4: **for** $j = 1, \ldots, n$ **do**
5: Let P_j have the form $P_j = \text{HEAD}(P_j) \leftarrow B_1, \ldots, B_r$
6: **if** $\sigma = \text{mgu}\,(\alpha_i, \text{HEAD}(P_j))$ exists **then**
7: $k' \leftarrow \begin{bmatrix} (\alpha_1, \ldots, \alpha_{i-1}, B_1, \ldots, B_r, \alpha_{i+1}, \ldots, \alpha_m)\,\sigma, \\ (-1), \\ \max + 1, \\ \text{LEVEL}(k) + 1 \end{bmatrix}^T$
8: $t \leftarrow t \cup \{k'\}$
9: max $\leftarrow$ max $+1$
10: k.SUCC $\leftarrow$ SUCC $\circ$ (max)
11: **end if**
12: **end for**
13: **end for**

Algorithm 8 Operator EXPANDLEVEL

Input: Graph $G = (V, E) = (\{v_1, \ldots, v_n\})$, temporal program P, integer l

1: **for** $i = 1, \ldots, n$ **do**
2: **if** $\text{LEVEL}(v_i) = l$ **then**
3: EXPANDNODE(P, v_i)
4: **end if**
5: **end for**

Algorithm 9 Checking algorithm for $P \models \alpha$ in first order temporal logic

Input: temporal program P, temporal goal α
Output: **true** if $P \models \alpha$

```
1: t ← ∅
2: V ← ∅
3: E ← ∅
4: i ← 0
5: n_0 ← (α, (−1), 0, 0)
6: t ← t ∪ {n_0}
7: while true do
8:    ExpandLevel((V, E), P, i)
9:    for j = 1, ..., |V| do
10:      if Level(v_j) = i then
11:         if Label(v_j) = □ then
12:            return true
13:         end if
14:      end if
15:   end for
16: end while
```

So the following theorem holds.

Theorem 3.3 *Let P be a temporal program and let α be a temporal goal. Then $P \models \alpha$ iff algorithm 9 returns* true *given input (P, α).*

4 Generalisation and Specialisation

This section will give an introduction into the topic of generalizing and specializing temporal programs. The first two subsections will be rather informal and are written in order to give a feeling about when a program is more general or more special than another. Generalization and Specialization always has to be seen relative to some ordering $\geq$, a so called *generality ordering*. Such orderings are quite easy to find in the case of nontemporal programs. In section 5 we will present concrete orderings for temporal clauses and programs. In this section we will only assume that a suitable ordering $\geq$ is chosen. The results will therefore be very general.

4.1 Generalization

Generalization of a program is a task which can be described as follows: Given a program P, a set $\mathcal{B}$ of background knowledge and sets $\mathcal{E}^+, \mathcal{E}^-$ of positive and negative examples such that there is an $e \in \mathcal{E}^+$ with $P \cup \mathcal{B} \not\models e$, find a modification of P such that e is implied after modifying P. The set $\mathcal{B}$ will remain unchanged by this modification. If Θ is an operator which performs the generalization-operation, then we expect $P \cup \mathcal{B} \geq \Theta(P \cup \mathcal{B}) = \Theta(P) \cup \mathcal{B}$. Some approaches to generalizing programs are the following:

1. For two clauses $C_1, C_2 \in P$, find a generalization C, i.e. a clause C which $C_1 \geq C$ and $C_2 \geq C$. The result might then be either $(P \setminus \{C_1, C_2\}) \cup \{C\}$ or $P \cup \{C\}$.
2. Find a clause $C_1 \in P$ which has to be only slightly modified to C_2 in order to result in a program which satisfies the constraints posed by $\mathcal{E}^+$. Then the result might be either $P \cup \{C_2\}$ or $(P \setminus \{C_1\}) \cup \{C_2\}$.

Both approaches can be applied. Their properties and how they can be exploited will be the topic of section 6.

4.2 Specialization

Specialization of a program means the process of modifying the program because it is in some sense *too general*. This refers to the concept of being *too strong*. If a program P, a set $\mathcal{B}$ of clauses and sets $\mathcal{E}^+, \mathcal{E}^-$ of positive and negative examples are given, then $P \cup \mathcal{B}$ is too strong wrt. $\mathcal{E}^-$ if $P \cup \mathcal{B}$ is not consistent wrt. $\mathcal{E}^-$ which means that $P \cup \mathcal{B} \cup \{\neg e | e \in \mathcal{E}^-\}$ is not satisfiable. This again implies that there is at least one $e \in \mathcal{E}^-$ such that $P \cup \mathcal{B} \models e$. So P is not a correct solution of the model inference problem induced by $\mathcal{E}^+$ and $\mathcal{E}^-$. The task which has to be performed in this case is to specialize P in such a way that still every $e \in \mathcal{E}^+$ is implied by $P \cup \mathcal{B}$ but none of the examples from $\mathcal{E}^-$ is implied after the specialization operation. Again, several approaches are possible:

1. Find a pair C_1, C_2 of clauses from P such that C_1 and C_2 can be specialized to a clause C in order to yield a correct program. The result of the specialization operation might either be $(P \setminus \{C_1, C_2\}) \cup \{C\}$ or $P \cup \{C\}$.

2. Find a single C_1 clause which has to be only slightly modified to C_2 in order to yield a correct program. The result might be $(P \setminus \{C_1\}) \cup \{C_2\}$ or $P \cup \{C_2\}$.

Again, one can choose among these and several other approaches in order to specialize a program P. A systematic study of specialization operation with respect to concrete orderings will be performed in section 6.

4.3 A Modification of the Backtracing Algorithm for Temporal Programs

In this subsection we will concentrate on a refinement the well known Backtracing algorithm (see [Sha81]) to include temporal operations. The basic task which was performed by this algorithm is: if a negative example e is implied by a program P, then for every refutation $P \cup \{\neg G\} \vdash^* \square$ there is a *false* clause C which is used in this derivation. Using an oracle which can be posed questions about truth or falsity of ground objects, this clause can be identified yielding an information how to refine P in order not to imply e anymore. The original Backtracing algorithm was only defined to work with first order clauses. But a generalization to temporal clauses consisting of literals from $\mathcal{L}_{\text{temp}}^{\text{templit}}$ (sig) is easy: One only has to ensure that the oracle which is used can handle queries about temporal ground atoms.

Algorithm 10 Temporal Backtracing Algorithm

Input: Program P, atom $e \in B_{sig}$ such that $P \models e$ and SLD-tree S for $P \cup \{\neg e\} \vdash^* \square$.
Output: $P_i \in P$ which is false wrt. P_{cor}

1: $n \leftarrow$ height(S) =number of resolution steps in the derivation of $\square$
2: $C \leftarrow \square$
3: $k \leftarrow 0$
4: $C_1 \leftarrow$left predecessor of C
5: $C_2 \leftarrow$right predecessor of C
6: **if** C_2 is not ground **then**
7: $\quad \sigma \leftarrow$arbitrary ground substitution for C_2
8: $\quad C_2 \leftarrow C_2\sigma$ set $C_2 = C_2\sigma$.
9: **end if**
10: **if** O answers *no* for input C_2 **then**
11: $\quad$ **return** C_2
12: **end if**
13: $k \leftarrow k + 1$
14: **if** $k < n$ **then**
15: $\quad C \leftarrow C_1$
16: $\quad$ **goto** 4
17: **end if**
18: **return** C_1

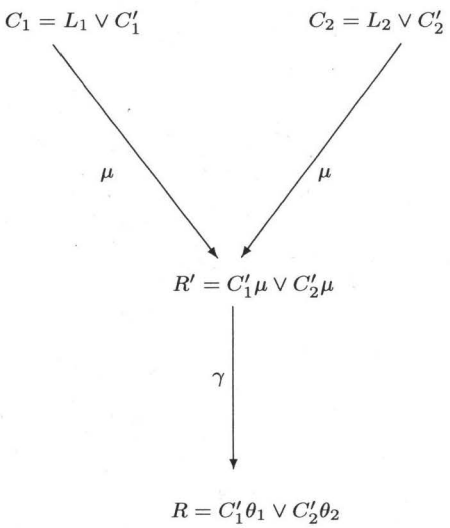

Figure 6: The V-Operator

4.4 Temporal V- and W-Operators

In first order Inductive Logic Programming, two operations have gained high profile: the so called V- and W-operators. Both operations use clauses to construct other clauses. We will consider both operations separately.

4.4.1 The Temporal V-Operator

The V-operator is an operation to construct from a clause $C_1 = L_1 \vee C_1'$ and a clause R another clause C_2, such that R is a resolvent of C_1 and C_2. The task is now to find substitutions θ_1, θ_2 such that we have the situation in figure 6. Here R is a factor of another resolvent R', i.e. an instance which is generated by applying θ_1 and θ_2. The classical V-operator which was introduced in [MB88] can also be applied in a temporal logic context since the only difference is that the operator $\circ$ has to be taken into account. But this does not depend on the algorithm implementing the V-operator, so we can just state the classical definition and use it in temporal logic. This algorithm is stated as algorithm 11 below. We will present an example for the application of the V-operator which is a slight modification of an example from first order logic which was presented in [NCdW97].

Algorithm 11 V-Operator

Input: Clauses $C_1 = L_1 \vee C_1'$ and R

Output: Clause C_2 such that R is an instance of a resolvent of C_1 and C_2

1: Choose θ_1 such that $C_1'\theta_1 \subseteq R$
2: Choose L_2 and C_2' such that there is a substitution θ_2 with $L_1\theta_1 = \neg L_2\theta_2$ and $C_2\theta_2 = R \setminus C_1'\theta_1$
3: **return** $C_2 = L_2 \vee C_2'$

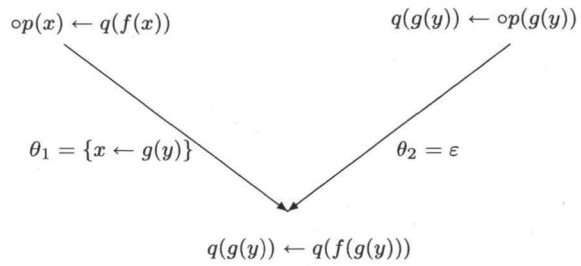

Figure 7: Clauses derived by the V-operator

Example 4.1 *Let C_1 and R be given as*

$$\begin{aligned} C_1 &= \circ p(x) \leftarrow q(f(x)) \\ &\equiv \circ p(x) \vee \neg q(f(x)) \\ R &= q(g(y)) \leftarrow q(f(g(y))) \\ &\equiv q(g(y)) \vee \neg q(f(g(y))) \end{aligned}$$

with $L_1 = \circ p(x)$). Algorithm 11 proceeds as follows:

1. *We have $L_1\theta_1 \subseteq R$ iff $\circ p(x)\theta_1 \subseteq \{q(g(y)), \neg q(f(g(y)))\}$ iff $\theta_1 = \{x \leftarrow g(y)\}$.*
2. *We have*

$$\begin{aligned} R \setminus C_1\theta_1 &= \{q(g(y)), \neg q(f(g(y)))\} \setminus \{\neg q(f(g(y)))\} \\ &= \{q(g(y))\} \\ &= C_2'\theta_2 \end{aligned}$$

   together with $L_1\theta_1 = \circ p(x)\theta_1 = \neg L_2\theta_2$, so we choose $\theta_2 = \varepsilon$.

3. *The algorithm now returns $C_2 = \neg \circ p(g(y)) \vee q(g(y)) \equiv q(g(y)) \leftarrow \neg p(g(y))$.*

So the algorithm has produced a clause C_2 such that R is an instance of a resolvent of C_1 and C_2. The resolution step is depicted in figure 7.

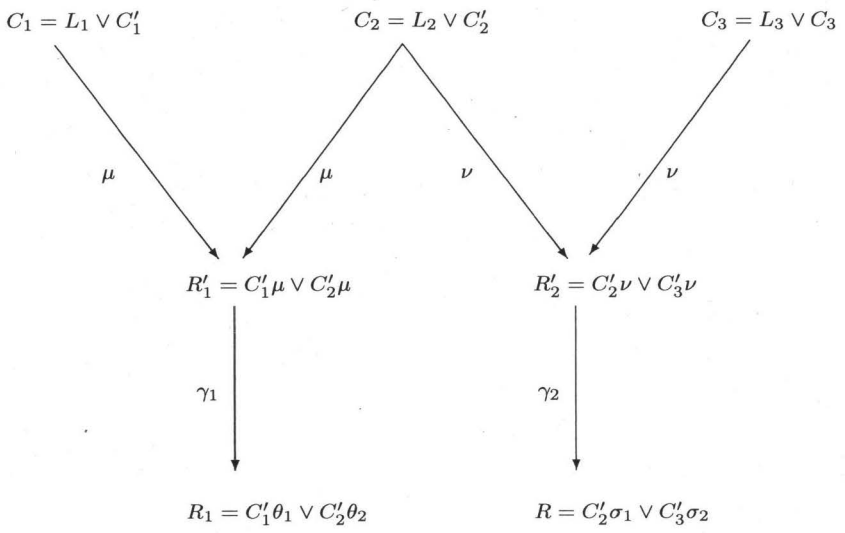

Figure 8: The W-Operator

4.4.2 The Temporal W-Operator

The W-operator can be seen as the combination of two V-operators which enables an ILP system to introduce (*invent*) new predicate symbols to the program if this is necessary. The basic setting is as depicted in figure 8: given R_1 and R_2 the operator will construct clauses C_1, C_2 and C_3 such that R_1 in an instance of a resolvent R_1' of C_1 and C_2 and R_2 is an instance of a resolvent R_2' of C_2 and C_3. Again, we can directly use the classical W-operator algorithm which is known from first order logic programming. This algorithm is presented in algorithm 12. The basic idea is the following: it is only necessary to construct C_2. The remaining clauses C_1 and C_3 can then be computed using algorithm 11. The literal L_2 can then be chosen freely and needs not have a topsymbol which already occurs in the language which is used in the actual program. Therefore an execution of the W-operator can introduce new predicates into the language. If it does so, depends on a concrete implementation which is used.

Example 4.2 *We will present a nontemporal example. An extension to temporal clauses is obvious. Assume that the following clauses R_1 and R_2 are given:*

$$R_1 = p(x_1, y_1) \leftarrow q(x_1, z_1), q(z_1, y_1)$$
$$R_2 = p(x_2, y_2) \leftarrow q(x_2, c), r(c, b)$$

If we set $L_2 = \neg s(z_4, y_4)$ we can find $C_2' = \neg q(x_4, z_4) \vee p(x_4, y_4)$ and $\sigma_2 = \{y_4 \leftarrow b, x_4 \leftarrow x_2, z_4 \leftarrow c\}$. Furthermore we get $\sigma_1 = \{x_4 \leftarrow x_2, y_4 \leftarrow y_2, z_4 \leftarrow z_2\}$. Two runs

Algorithm 12 W-Operator

Input: Clauses R_1, R_2

Output: Clauses C_1, C_2, C_3 such that R_1 is an instance of a resolvent of C_1 and C_2 and R_2 is an instance of a resolvent of C_2 and C_3

1: find C_2' and θ_2, σ_1 such that $C_2'\theta_2 \subseteq R$ and $C_2'\sigma_1 \subseteq R$. If no such C_2' exists, set $C_2' = \emptyset$
2: $L_2 \leftarrow$ arbitrary literal
3: $C_2 \leftarrow L_2 \vee C_2'$
4: $C_1 \leftarrow V(R_1, C_2)$
5: $C_3 \leftarrow V(R_2, C_2)$
6: **return** $\{C_1, C_2, C_3\}$

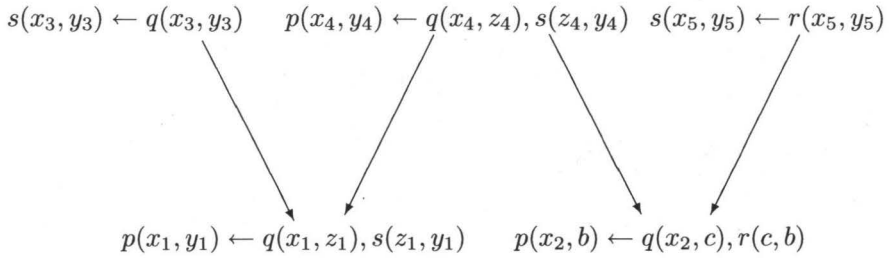

Figure 9: Clauses derived by the W-operator

of the V-operator yield the clauses $C_1 = s(x_3, y_3) \leftarrow q(x_3, y_3)$ *and* $C_3 = s(x_5, y_5) \leftarrow r(x_5, y_5)$ *so we have the situation as depicted in figure 9.*

If $V(C_1, R)$ and $W(C_1, C_2, R)$ denotes the result of the application of the V- resp. W-operator, then the following theorem is immediately.

Theorem 4.1 *1.* $V(C_1, R) \models C_1$ *and*

2. $W(C_1, C_2, R) \models C_1, W(C_1, C_2, R) \models C_2$.

5 A Subsumption based Generality Order

In this chapter we will expand the concept of a Generality Ordering to the class of all temporal atoms and clauses. We can in principle find at least two kinds of orderings: among those there are orderings which are based on some concept of subsumption and orderings based on implication. Furthermore one can define purely syntactical orderings, which are based on precedences on the set of predicate symbols.

5.1 Motivation and Definitions

The first ordering which we will define is an extension of the subsumption ordering $\succcurlyeq_c$ from first order logic. We will first define an ordering on literals and then extend this ordering to the set of temporal clauses. The ordering $\succcurlyeq_{ta}$ on single literals will intuitively be defined in the following way: Given formulas $\psi_1, \psi_2 \in \mathcal{L}_{\text{temp}}^{\text{templit}}(\text{sig})$.

- If both ψ_1 and ψ_2 are nontemporal literals, then we use the ordering $\succcurlyeq_a$,
- if ψ_1 is temporal and ψ_2 is nontemporal, then ψ_1 is assumed to be greater than ψ_2 and
- if both ψ_1 and ψ_2 are temporal, then the literal which contains more $\circ$'s is assumed to be greater.

This concept is formalized in the following definition.

Definition 5.1 *Let sig be a signature. The ordering* $\succcurlyeq_{\text{ta}} \subseteq \mathcal{L}_{temp}^{templit}(sig) \times \mathcal{L}_{temp}^{templit}(sig)$ *is defined as follows: if* $\psi_1, \psi_2 \in \mathcal{L}_{temp}^{templit}(sig)$, *then*

1. *if* $\psi_1, \psi_2 \in \mathcal{L}_{temp}^{tempatom}(sig)$, *then* $\psi_1 \succcurlyeq_{\text{ta}} \psi_2$, *if and only if* $\psi_1 \succcurlyeq_a \psi_2$,
2. *if* $\psi_1 \in \mathcal{L}_{temp}^{tempatom}(sig)$ *and* $\psi_2 \in \mathcal{L}_{temp}^{templit}(sig) \setminus \mathcal{L}_{temp}^{tempatom}(sig)$, *then* $\psi_2 \succcurlyeq_{\text{ta}} \psi_1$,
3. *if* $\psi_1 \in \mathcal{L}_{temp}^{templit}(sig) \setminus \mathcal{L}_{temp}^{tempatom}(sig)$ *and* $\psi_2 \in \mathcal{L}_{temp}^{tempatom}(sig)$, *then* $\psi_1 \succcurlyeq_{\text{ta}} \psi_2$ *and*
4. *if* $\psi_1, \psi_2 \in \mathcal{L}_{temp}^{templit}(sig) \setminus \mathcal{L}_{temp}^{tempatom}(sig)$ *for* $\psi_1 = \circ\psi_1', \psi_2 = \circ\psi_2'$, *then* $\psi_1 \succcurlyeq_{\text{ta}} \psi_2$ *if and only if* $\psi_1' \succcurlyeq_{\text{ta}} \psi_2'$.

We write $\varphi \approx \psi$ if $\varphi \succcurlyeq_{\text{ta}} \psi$ and $\psi \succcurlyeq_{\text{ta}} \varphi$ as well as $\varphi \succ_{\text{ta}} \psi$ if $\varphi \succcurlyeq_{\text{ta}} \psi$ and not $\varphi \approx \psi$. Similarly we write $\varphi \preccurlyeq_{\text{ta}} \psi$ and $\varphi \prec_{\text{ta}} \psi$ for $\psi \succcurlyeq_{\text{ta}} \varphi$ respectively $\psi \succ_{\text{ta}} \varphi$.
This definition can be easily generalized to sets of temporal literals, i.e. to temporal clauses. The definition of the subsumption ordering for temporal clauses is exactly the same as the definition for first order clauses.

Definition 5.2 *Let sig be a signature. The subsumption ordering* $\succcurlyeq_{\text{tc}}$ *on the set of all temporal clauses over sig is defined as:* $C \succcurlyeq_{\text{tc}} D$ *if there is a substitution* θ *such that* $C\theta \subseteq D$.

As above, we write $C \sim D$ if $C \succcurlyeq_{tc} D$ and $D \succcurlyeq_{tc} C$ as well as $C \succ_{tc} D$ if $C \succcurlyeq_{tc} D$ and not $C \sim D$. Similarly we write $C \preccurlyeq_{tc} D$ and $C \prec_{tc} D$ for $D \succcurlyeq_{tc} C$ respectively $D \succ_{tc} C$.

5.2 The Concept of Reducedness in Temporal Logic

In first order ILP one often has to deal with the case that a clause C is given and one looks for the *smallest* clause D with the property that $C \approx D$. This leads to the concepts of reducedness and reduced clauses. We will define this concept for temporal clauses and show how reduced clauses can be computed. In contrast we will also give algorithms for computing the inverse of a reduced clause. We will see that the concept of reducedness is formally exactly the same in temporal logic as in first order logic. The only difference is the use of the temporal ordering $\succcurlyeq_{tc}$ instead of the first order ordering $\succcurlyeq_c$. Therefore note that the relation $\sim$ as well as the relation $\approx$ is an equivalence relation. This a simple consequence of the properties of the properties of $\subseteq$. Then we define the reduced form of a clause C as follows.

Definition 5.3 *Let C be a temporal clause. C is reduced if there is no $D \subset C$ such that $D \sim C$.*

So if C is reduced it makes sense to call C the *smallest* member of its equivalence class. In [NCdW97] an algorithm for the computation of the reduced form of any first order clause is presented. We will now present an algorithm for doing this in the case of temporal clauses. Therefore we have to give several properties of reduced temporal clauses. We don't have to prove them since their proofs in the case of first order logic does not refer to the structure of the involved clauses. So the proofs also hold in the case of temporal logic. The two relevant results are stated in the following lemma.

Lemma 5.1

1. *Let C and D be reduced clauses. If $C \sim D$, then C and D are variants.*
2. *Let C be a clause. If there is a substitution θ, such that $C\theta \subseteq C$, then there exists a reduced clause D, such that $C \sim D$.*

Algorithm 13 is just a formulation of the properties of reduced clauses which were stated in lemma 5.1. So the reduction of a clause can also be computed if the clause contains temporal literals. What is left to develop is an efficient method for finding the substitutions θ which are used in the algorithm.

Example 5.1 *Assume that $C = \{p(x_1, x_2), \circ q(y_1, y_2), p(x_2, x_1), \circ q(y_3, y_3)\}$. Then algorithm 13 works as follows:*

1. *First we have $D = C = \{p(x_1, x_2), \circ q(y_1, y_2), p(x_2, x_1), \circ q(y_3, y_3)\} = \{\varphi_0, \varphi_1, \varphi_2, \varphi_3\}$, so $m_0 = 4$. The $j-$loop starts searching the possible candidates for substitutions.*

   a) $j = 0$: there is no substitution θ such that $D\theta \subseteq D \setminus \{\varphi_0\}$.

Algorithm 13 Reduction Algorithm for Temporal Clauses

Input: temporal clause C
Output: reduced temporal clause D with $D \sim C$

```
1: D ← C
2: i ← 0
3: Let D = {φ_0, ..., φ_{m_i}}
4: for j = 0, ..., m_i do
5:    if there exists θ such that Dθ ⊆ D \ {φ_j} then
6:       D ← Dθ
7:       i ← i + 1
8:       goto 3
9:    else
10:      return D
11:   end if
12: end for
```

b) $j = 1$: *The algorithm finds* $\theta = \{y_1 \leftarrow y_3, y_2 \leftarrow y_3\}$ *because* $D\theta = \{p(x_1,x_2), p(x_2,x_1), \circ q(y_3,y_3)\} = \{\varphi_0, \varphi_1, \varphi_2, \varphi_3\} \subset D$. *So we set* $D = \{p(x_1,x_2), p(x_2,x_1), \circ q(y_3,y_3)\} = D\theta$ *and quit the* $j-$*loop.*

2. *Now* $D = \{p(x_1,x_2), p(x_2,x_1), \circ q(y_3,y_3)\} = \{\varphi_0, \varphi_1, \varphi_2\}$, *so* $m_1 = 3$. *Again we start the* $j-$*loop.*

   a) $j = 0$: *there is no substitution* θ *such that* $D\theta \subseteq D \setminus \{\varphi_0\}$.

   b) $j = 1$: *there is no substitution* θ *such that* $D\theta \subseteq D \setminus \{\varphi_1\}$.

   c) $j = 2$: *there is no substitution* θ *such that* $D\theta \subseteq D \setminus \{\varphi_2\}$.

   So we find that $D = \{p(x_1,x_2), p(x_2,x_1), \circ q(y_3,y_3)\}$ *is reduced.*

The inversion of the operation or reducing a clause is not that easy. For example assume that $D = \{p(x)\}$. Then D is a reduced version of $C_1 = \{p(x), p(y)\}$. But D is also a reduced version of $C_2 = \{p(x), p(y), p(z)\}$. Moreover, D is a reduced version of $C_n = \bigcup_{i=1}^{n} p(x_i)$ for every value of n. Which of these several clauses should be returned by an algorithm? One might argue that it should return the *smallest clause* $C \sim D$ *with the property that* C *is not reduced.* But this criterion is not well-defined since if C_1 satisfies this criterion so does every variant C_2 of C_1. An approach from [NCdW97] uses the assumption that one gives an upper bound m for the number of literals to occur in the non reduced clause. The algorithm presented there is given a reduced clause D as input and computes variants of every C which is subsume equivalent to D, i.e. $C \sim D$ and which satisfies the constraint of $|C| \leq m$. To keep the formulation of the algorithm short and readable we define the criterion $\mathcal{C}$ as follows: a clause D, a subset $\{\psi_1, \ldots, \psi_n\} \subseteq D$ and a set $M = \{\varphi_1, \ldots, \varphi_n\}$ satisfy criterion $\mathcal{C}$ if

1. every φ_i contains a variable which does not occur in D and

2. if $x_1, \ldots, x_m$ is the sequence of all these new variables, then there is a substitution $\theta = \{x_1 \leftarrow t_1, \ldots, x_m \leftarrow t_m\}$ with $\varphi_i\theta = \psi_i$.

An algorithm for computing a non reduced clause from a reduced one is now stated in algorithm 14.

Algorithm 14 Inverse Reduction Algorithm for Temporal Clauses

Input: reduced temporal clause D, integer m
Output: variant of all (non)reduced temporal clauses C with $D \sim C$ and $|C| \leq m$.

```
1: l ← 0
2: if |D| ≤ m then
3:   output D
4:   while l < (m − |D|) do
5:     l ← l + 1
6:     for every set M = {φ_1, ..., φ_l} with φ ∈ D for every i do
7:       find every E = {ψ_1, ..., ψ_l} such that D, M and E (up to variants) satisfy
         criterion C.
8:       output D ∪ E
9:     end for
10:  end while
11: end if
```

The next example completes this section and shows how the original clause from example 5.1 can be reconstructed by algorithm 14.

Example 5.2 *Consider the temporal clause* $D = \{p(x_1, x_2), p(x_2, x_1) \circ q(y_3, y_3)\}$ *which has been shown to be a reduced clause. If* $m = 4$, *then algorithm 14 proceeds as follows.*

1. *We have* $|D| = 3 < m = 4$, *so the algorithm outputs* D *and sets* l *to* 0. *We have* $C_1 = \{p(x_1, x_2), p(x_2, x_1) \circ q(y_3, y_3)\} = D$.

2. *Now* $l = 0 < 1 = m - |D|$, *so* $l = 1$ *and we receive the following three sets* M_i *which are candidates for being added to* D :

   a) $M_1 = \{p(x_1, x_2)\}$
   b) $M_2 = \{p(x_2, x_1)\}$
   c) $M_3 = \{\circ q(y_3, y_3)\}$

   These three sets are processed in order to find literals which can be added to D.

   a) *There's no substitution* θ *which fits to* M_1.
   b) *There's no substitution* θ *which fits to* M_2.
   c) *For* M_3 *we receive*
      i. $M_3^{(1)} = \{\circ q(y_1, y_3)\}$ *and* $\theta_1 = \{y_1 \leftarrow y_3\}$

ii. $M_3^{(2)} = \{\circ q(y_3, y_1)\}$ *and* $\theta_1 = \{y_1 \leftarrow y_3\}$

iii. $M_3^{(3)} = \{\circ q(y_1, y_1)\}$ *and* $\theta_1 = \{y_1 \leftarrow y_3\}$

iv. $M_3^{(4)} = \{\circ q(y_1, y_2)\}$ *and* $\theta_1 = \{y_1 \leftarrow y_3, y_2 \leftarrow y_3\}$

v. $M_3^{(5)} = \{\circ q(y_2, y_1)\}$ *and* $\theta_1 = \{y_1 \leftarrow y_3, y_2 \leftarrow y_3\}$

d) This yields the sets

i. $C_2 = \{p(x_1, x_2), p(x_2, x_1), \circ q(y_1, y_3), \circ q(y_3, y_1)\}$

ii. $C_3 = \{p(x_1, x_2), p(x_2, x_1), \circ q(y_3, y_3), \circ q(y_3, y_1)\}$

iii. $C_4 = \{p(x_1, x_2), p(x_2, x_1), \circ q(y_3, y_3), \circ q(y_1, y_1)\}$

iv. $C_5 = \{p(x_1, x_2), p(x_2, x_1), \circ q(y_3, y_3), \circ q(y_1, y_2)\}$

v. $C_6 = \{p(x_1, x_2), p(x_2, x_1), \circ q(y_3, y_3), \circ q(y_2, y_1)\}$

3. Now $l = 1 \not< 1$, *so the algorithm stops.*

The sequence of outputs of algorithm 14 is now given as follows:

i	C_i
1	$\{p(x_1, x_2), p(x_2, x_1) \circ q(y_3, y_3)\}$
2	$\{p(x_1, x_2), p(x_2, x_1), \circ q(y_1, y_3), \circ q(y_3, y_1)\}$
3	$\{p(x_1, x_2), p(x_2, x_1), \circ q(y_3, y_3), \circ q(y_3, y_1)\}$
4	$\{p(x_1, x_2), p(x_2, x_1), \circ q(y_3, y_3), \circ q(y_1, y_1)\}$
5	$\{p(x_1, x_2), p(x_2, x_1), \circ q(y_3, y_3), \circ q(y_1, y_2)\}$
6	$\{p(x_1, x_2), p(x_2, x_1), \circ q(y_3, y_3), \circ q(y_2, y_1)\}$

One sees that C_5 *is the original clause from example 5.1.*

5.3 Least Generalizations and Greatest Specializations

In this section we will turn our attention to the concepts of least generalizations and greatest specializations of clauses. Since clauses are sets of literals, we can restrict ourselves to the simpler case of pairwise computation of generalizations and specializations. Recall that a generalization of two literals ψ_1 and ψ_2 with respect to an ordering $\geq$ is a literal ψ such that $\psi \geq \psi_1$ and $\psi \geq \psi_2$. The basic task for computing such generalizations will be the detection of generalizations for atoms. This can be done using an algorithm which is only a slight generalization of the Anti-unification-algorithm (the situation is depicted in figure 10. If ψ_1 and ψ_2 are two temporal literals, then we will compute a generalization ϕ_1 by applying an Anti-Unification-algorithm and a specialization ϕ_2 by applying a unifier which we will receive from a unification algorithm). Therefore note that we defined the ordering $\succcurlyeq_{ta}$ in the way that an atom which contains more temporal operators is greater than one which contains less. This has to be kept in mind in a refinement of the well known first order Anti-Unification-algorithm. But before we will have to give an algorithm for the computation of a unifier of two temporal atoms. Therefore we extend the definition of a unifier as follows.

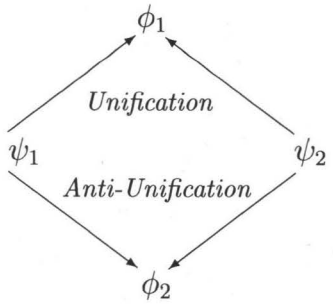

Figure 10: Generalization and Specialisation

Definition 5.4 *Let $\psi_1, \psi_2 \in \mathcal{L}_{temp}^{templit}(sig)$ be temporal literals. ψ_1 and ψ_2 are called unifiable if there is a substitution θ such that $\psi_1\theta = \psi_2\theta$. θ is then called a unifier.*

All properties of unifiers from first order logic carry over to temporal logic. In particular the concept of a most general unifier carries over to temporal logic. In order to compute unifiers we can rely on well known algorithms from first order logic and slightly modify them to take the operator $\circ$ into account. We will use the Martelli-Montanari-algorithm from [MM82] since it is easier to understand than the linear algorithm from [PW78] although the runtime can still be exponential in the length of the literals. The algorithm works similar to an equation solver. Literals are considered to be equations: in order to unify ψ_1 and ψ_2, we solve the equation $\psi_1 = \psi_2$.
In order to formalize the intuitive idea from the last paragraph, we define the function $\textsc{Next} : \mathcal{L}_{\text{temp}}^{\text{templit}}(\text{sig}) \to \mathbb{N}$ which returns the number of $\circ'$s in a temporal literal. That is if $\varphi = \circ^k\psi$ for some $\psi \in \mathcal{L}_{\text{temp}}^{\text{lit}}(\text{sig})$, then $\textsc{Next}(\varphi) = k$. Furthermore we assume that $\textsc{Matrix} : \mathcal{L}_{\text{temp}}^{\text{templit}}(\text{sig}) \to \mathcal{L}_{\text{temp}}^{\text{templit}}(\text{sig})$ is a function which given input $\varphi = \circ^k\psi$ returns $\textsc{Matrix}(\varphi) = \psi$. Since atoms with a different number of $\circ'$s can never be unifiable[7] we can restrict to pairs $(\psi_1, \psi_2) \in \mathcal{L}_{\text{temp}}^{\text{templit}}(\text{sig})^2$ with $\textsc{Next}(\psi_1) = \textsc{Next}(\psi_2)$. To state the modified unification algorithm we need some more definitions. Formally we have only defined unifiers for sets of literals. A *unifier for a set of equations* $E = \{s_i = t_i | i = 1, \ldots, n\}$ where all s_i and t_i are terms is a substitution θ with $s_i\theta = t_i\theta$ for every i. Two sets E_1 and E_2 of equations are said to be *equivalent* if they have the same set of unifiers, i.e. every unifier for E_1 is also a unifier of E_2 and vice versa. Finally a set E of equations is called *in solved form* if $E = \{x_1 = t_1, \ldots, x_n = t_n\}$, $x_i \neq x_j$ for $i \neq j$ and no x_i occurs in any of the terms t_j. The aim of algorithm 16 is now to produce a most general unifier for to temporal literals ψ_1 and ψ_2 by constructing a set of equations from ψ_1 and ψ_2 and bringing this set of equations into solved form. It is then straightforward to construct a substitution from the solved set of equations: such a substitution is given as the composition of all substitutions which can be obtained from the single equations. We formalize this in algorithm 15.

[7] This is a consequence of the fact, that variables can only be substituted with terms

Algorithm 15 Construction of a Unifier from a Set of Equations

Input: set E of equations in solved form
Output: substitution θ obtained from E

1: Let $E = \{x_1 = t_1, \ldots, x_n = t_n\}$
2: $\theta \leftarrow \varepsilon$
3: **for** $i = 1, \ldots, n$ **do**
4: $\quad \theta \leftarrow \theta \circ \{x_i \leftarrow t_i\}$
5: **end for**
6: **return** θ

It is now straightforward to show that algorithm 16 indeed produces a most general unifier of two literals ψ_1 and ψ_2 if such a unifier exists. Therefore if suffices to check the proof of correctness of the algorithm when it is restricted to nontemporal literals. We refer the reader to [Apt97] for this proof.
The inverse operation of unification will be carried out by an algorithm which is similar to the Anti-Unification-algorithm. In analogy to algorithm 16 we only have to modify this algorithm in such a way that it takes the temporal operator $\circ$ into account. Therefore it suffices to check if $\text{NEXT}(\psi_1) = \text{NEXT}(\psi_2)$ holds. The modified algorithm is stated in algorithm 17.
Now we are able to compute generalizations and specializations of single temporal literals. In the next section the algorithms for these computations will be used for the computations of least generalizations and greatest specializations of temporal clauses.

5.4 Algorithms for the Computation of Generalizations and Specializations

In this section we will exploit the algorithms from the last section for building algorithms for computing least generalizations and greatest specialization of clauses with respect to the subsumption order for temporal clauses. Therefore we will first state results about the existence of such generalizations and specializations.

5.5 Specialization

The first case which we will consider is the case of computing a greatest specialization of two clauses (we will consider this case first, since it is simpler than the case of computing a generalization). We will first examine the case of general temporal clauses and then concentrate on the special case of temporal horn clauses.
Recall that a specialization of two clauses C_1 and C_2 wrt. an ordering $\geq$ is any clause D such that $D \geq C_1$ and $D \geq C_2$. A specialization D which has the property that $D \geq C$ for every specialization C of C_1 and C_2 is called a *greatest specialization*. In out case $\geq = \succcurlyeq_{tc}$ and a greatest specialization of C_1 and C_2 wrt. $\succcurlyeq_{tc}$ will again be denoted by $\text{GSS}(C_1, C_2)$. We will restrict our attention to the situation of two clauses since if we can prove the existence of $\text{GSS}(C_1, C_2)$, then we can expand this result to

Algorithm 16 Unification-Algorithm for Temporal Literals

Input: $\psi_1, \psi_2 \in \mathcal{L}_{\text{temp}}^{\text{templit}}(\text{sig})$

Output: Unifier θ for ψ_1 and ψ_2 if ψ_1 and ψ_2 are unifiable.

1: $k_1 \leftarrow \text{NEXT}(\psi_1)$
2: $k_2 \leftarrow \text{NEXT}(\psi_2)$
3: **if** $k_1 \neq k_2$ **then**
4: **output** *not unifiable*
5: **stop**
6: **else**
7: $E \leftarrow \emptyset$
8: $\varphi_1 \leftarrow \text{MATRIX}(\psi_1)$
9: $\varphi_2 \leftarrow \text{MATRIX}(\psi_2)$
10: Let $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$ be the ordered sequences of terms occurring in φ_1 and φ_2.
11: **if** $n \neq m$ **then**
12: **output** *not unifiable*
13: **stop**
14: **else**
15: $E \leftarrow E \cup \{s_1 = t_1, \ldots, s_n = t_n\}$ /* since $n = m$ */
16: **while** E is not in solved form **do**
17: $e \leftarrow s_1 = t_1$
18: $E \leftarrow E \setminus \{e\}$
19: **end while**
20: **if** e has the form $f(t_1^{(1)}, \ldots, t_m^{(1)}) = f(t_1^{(2)}, \ldots, t_m^{(2)})$ **then**
21: $E \leftarrow E \cup \{t_i^{(1)} = t_i^{(2)} | i = 1, \ldots, m\}$
22: **if** E is solved **then**
23: **goto** 49
24: **else**
25: **goto** 17
26: **end if**
27: **end if**
28: **if** e has the form $f(t_1^{(1)}, \ldots, t_{m_1}^{(1)}) = g(t_1^{(2)}, \ldots, t_{m_2}^{(2)})$ **then**
29: **output** *not unifiable*
30: **stop**
31: **end if**
32: **if** e has the form $x = x$ **then**
33: **if** E is solved **then**
34: **goto** 49
35: **else**
36: **goto** 17
37: **end if**
38: **end if**
39: **if** e has the form $t = x$ and t is not a variable **then**
40: $E \leftarrow \cup\{x = t\}$
41: **end if**
42: **if** e has the form $x = t$, x does not occur in t but occurs elsewhere **then**
43: apply $\{x \leftarrow t\}$ to all equations
44: **end if**
45: **if** e has the form $x = t$, x occurs in t and $x \neq t$ **then**
46: **output** *not unifiable*
47: **stop**
48: **end if**
49: obtain θ from E with algorithm 15
50: **end if**
51: **end if**

Algorithm 17 Anti-Unification-Algorithm for Temporal Atoms

Input: $\psi_1, \psi_2 \in \mathcal{L}_{\text{temp}}^{\text{templit}}(\text{sig})$
Output: greatest lower bound ψ of ψ_1 and ψ_2

1: **if** $\text{NEXT}(\psi_1) \neq \text{NEXT}(\psi_2)$ **then**
2: **return** $\top$
3: **else**
4: obtain ψ by applying the Anti-Unification-algorithm to $\text{MATRIX}(\psi_1)$ and $\text{MATRIX}(\psi_2)$
5: **return** $\circ^{\text{NEXT}(\psi_1)}\psi$
6: **end if**

finite sets $\{C_1, \ldots, C_n\}$ of clauses by using

$$\text{GSS}(C_1, \ldots, C_n) = \text{GSS}(C_1, \text{GSS}(C_2, \ldots, C_n)). \tag{1}$$

So assume that C_1 and C_2 are temporal clauses. Without loss of generality we can assume that C_1 and C_2 are standardized apart. We define $D = C_1 \cup C_2$. First note that D is again a clause. Second, $C_1 \succcurlyeq_{tc} D$ and $C_2 \succcurlyeq_{tc} D$ since $C_i\varepsilon = C_i \subseteq D$ for $i = 1, 2$. Now assume that C is a specialization of C_1 and C_2. So $C_1 \succcurlyeq_{tc} C$ and $C_2 \succcurlyeq_{tc} C$. So there exist substitutions θ_1 and θ_2 with $C_1\theta_1 \subseteq C$ and $C_2\theta_2 \subseteq C$. Since C_1 and C_2 are standardized apart, θ_i only acts on variables which occur in C_i. Defining $\theta = \theta_1 \circ \theta_2$, we have

$$\begin{aligned} D\theta &= (C_1 \cup C_2)\theta \\ &= C_1\theta \cup C_2\theta \\ &= C_1\theta_1 \cup C_2\theta_2 \\ &\subseteq C \end{aligned}$$

So $D \succcurlyeq_{tc} C$ and D is indeed a GSS of C_1 and C_2. Now assume that $\{C_1, \ldots, C_n\}$ is any finite set of temporal clauses. Then we use equation 1 to define $\text{GSS}(C_1, \ldots, C_n)$. Now note that this construction again yields a GSS. This is proved by induction on n. For $n = 2$ the claim has already been proved. Now assume that the claim holds for a fixed but arbitrary value of n and consider $\{C_1, \ldots, C_n, C_{n+1}\}$. Since the induction hypothesis holds, we have that $C := C_1 \cup \cdots \cup C_n = \text{GSS}(C_1, \ldots, C_n)$. Furthermore we have that $C_1 \cup C = \text{GSS}(C_1, C)$, so $\text{GSS}(C_1, \ldots, C_n, C_{n+1}) = C_1 \cup \cdots \cup C_n \cup C_{n+1}$. This proves the following theorem.

Theorem 5.1 *Let sig be any signature. Then every finite set $\{C_1, \ldots, C_n\}$ of temporal clauses over sig has a greatest specialization wrt. $\succcurlyeq_{tc}$ GSS which is given by*

$$GSS(C_1, \ldots, C_n) = \bigcup_{i=1}^{n} C_i$$

The result from the above theorem is quite useful. But since we work with temporal horn clauses we have to establish the existence of a greatest specialization of a finite set of temporal horn clauses which is again a temporal horn clause. Therefore assume that the set of all temporal horn clauses over *sig* contains an additional element $\bot$ which is assumed to be smaller than every temporal horn clause C i.e. $C \succcurlyeq_{tc} \bot$. This element will prove to be useful due to the special form of (temporal) horn clauses. So now assume that $\{C_1, \ldots, C_n\}$ is a finite set of temporal horn clauses over *sig*. Again we assume without loss of generality that all clauses are standardized apart. Then we define a partition

$$\{C_1, \ldots, C_n\} = \{C_1, \ldots, C_k\} \cup \{C_{k+1}, \ldots, C_n\}$$

such that $\{C_1, \ldots, C_k\}$ contains only definite temporal horn clauses and $\{C_{k+1}, \ldots, C_n\}$ contains only temporal goals. We distinguish the following cases:

Case 1 $k = 0$. Then $\mathrm{GSS}(C_1, \ldots, C_n) = C_1 \cup \cdots \cup C_n$.

Case 2 $k \geq 1$. Let $\{D_1, \ldots, D_k\}$ be the set of heads of clauses in $\{C_1, \ldots, C_k\}$. Two cases can be identified:

Case 2.1 $\{D_1, \ldots, D_k\}$ is not unifiable. Then we set $\mathrm{GSS}(C_1, \ldots, C_n) = \bot$.

Case 2.2 $\{D_1, \ldots, D_k\}$ is unifiable. Then let σ be a most general unifier. We set $C := \mathrm{GSS}(C_1, \ldots, C_n) := C_1\sigma \cup \cdots \cup C_n\sigma$.

The cases 1 and 2.1 are trivially correct. We need to establish the correctness of case 2.2 in order to show the existence of a greatest specialization of $\{C_1, \ldots, C_n\}$. Therefore first note again that $C_1\sigma \cup \ldots C_n\sigma$ is again a horn clause. This is due to the fact that the set of heads of clauses in $\{C_1, \ldots, C_n\}$ is unifiable. Furthermore one sees that $C_i \succcurlyeq_{tc} C$ for every i. Assume now that D is any temporal horn clause with $C_i \succcurlyeq_{tc} D$ for every i. So there are substitutions θ_i such that $C_i\theta_i \subseteq C$ for every i and θ_i only affects variables from C_i since all C_i are assumed to be standardized apart. Set $\theta = \theta_1 \circ \cdots \circ \theta_n$. Then we have for $i = 1, \ldots, k$:

$$D_i\theta = D_i\theta_i = \textsc{Head}(C)$$

So θ is a unifier for $\{D_1, \ldots, D_k\}$. Since σ is a most general unifier there is a substitution γ such that $\theta = \sigma \circ \gamma$. So we have

$$\begin{aligned} C\gamma &= C_1\sigma \circ \gamma \cup \cdots \cup C_n\sigma \circ \gamma \\ &= C_1\theta \cup \cdots \cup C_n\theta \\ &\subseteq D. \end{aligned}$$

This yields $C \succcurlyeq_{tc} D$, so D is indeed a GSS which is a temporal horn clause. This proves the following theorem.

Theorem 5.2 *Let sig be a signature and let $\{C_1, \ldots, C_n\}$ be a finite set of temporal horn clauses over sig. Then there exists a greatest specialization $GSS(C_1, \ldots, C_n)$ which is either $\bot$ or a temporal horn clause over sig.*

To conclude this section we summarize the last arguments in algorithm 18 and present an example.

Algorithm 18 GSS-Computation for Temporal Clauses

Input: Set $M = \{C_1, \ldots, C_n\}$ of temporal horn clauses
Output: $\mathrm{GSS}(C_1, \ldots, C_n)$

1: $M_1 \leftarrow \{C_1, \ldots, C_k\} = \{C \in M | \textsc{Head}(C) \neq \emptyset\}$
2: $M_2 \leftarrow M \setminus M_1$
3: **if** $k = 0$ **then**
4: **return** $C_1 \cup \cdots \cup C_n$
5: **else**
6: **if** $\{\textsc{Head}(C_1), \ldots, \textsc{Head}(C_k)\}$ is not unifiable **then**
7: **return** $\bot$
8: **else**
9: $\sigma \leftarrow \mathrm{mgu}\,(\{\textsc{Head}(C_1), \ldots, \textsc{Head}(C_k)\})$
10: **return** $C_1\sigma \cup \cdots \cup C_n\sigma$
11: **end if**
12: **end if**

Example 5.3 *If* $sig = (\{(f,1),(g,1)\}, \{(p,1),(q,1),(r,1)\}, \{x, y, z, \ldots\})$ *and*

$$\begin{aligned} C_1 &= p(x) \leftarrow \circ q(f(x)), r(g(x)) \\ C_2 &= \leftarrow r(y), r(f(y)) \\ C_3 &= \circ r(z) \leftarrow p(z) \end{aligned}$$

then algorithm 18 yields

$$\begin{aligned} GSS(C_1, C_2) &= p(x) \leftarrow \circ q(f(x)), r(g(x)), r(y), r(f(y)) \\ GSS(C_1, C_3) &= \bot \\ GSS(C_2, C_3) &= \circ r(z) \leftarrow p(z), r(y), r(f(y)) \end{aligned}$$

5.6 Generalization

In the last section we have established algorithms for the computation of greatest specializations of sets of temporal (horn) clauses. In this section we have to attack the dual problem: find a least generalization of two or more temporal (horn) clauses. To obtain such algorithms we will exploit several results from the general theory of ILP.

Definition 5.5 *Let* $\psi_1, \psi_2 \in \mathcal{L}_{temp}^{templit}(sig)$ *be literals.* ψ_1 *and* ψ_2 *are said to be compatible if they have the same top-symbol, the same sign and* $\textsc{Next}(\psi_1) = \textsc{Next}(\psi_2)$.

Recall that for clauses $C = \{C_1, \ldots, C_n\}$ and $D = \{C_1, \ldots, D_n\}$ a selection is a pair (L, M) with compatible $L \in C, M \in D$. The least generalization $LGA(\psi_1, \psi_2)$ of two atoms ψ_1 and ψ_2 can be computed using algorithm 17. Similarly if $C = \{C_1, \ldots, C_n\}$ and $D = \{D_1, \ldots, D_n\}$ are temporal clauses and $S = \{(L_1, M_1), \ldots, (L_m, M_m)\}$ is any sequence of selections, we define

$$\begin{aligned} C_S &= L_1 \vee \cdots \vee L_m \\ D_S &= M_1 \vee \cdots \vee M_m \end{aligned}$$

and

$$LGA(C_S, D_S) = \bigvee_{i=1}^{m} LGA(L_i, M_i)$$

We can prove the following result:

Theorem 5.3 *Let sig be a signature and let $C = \{C_1, \ldots, C_n\}$ and $D = \{D_1, \ldots, D_n\}$ be temporal clauses over sig and let $S = \{(C_{i_1}, D_{i_1}), \ldots, (C_{i_m}, D_{i_m})\}$ be the sequence of all selections from C and D. The a least generalization of C and D is given by*

$$LGS(C, D) = \bigvee_{j=1}^{m} LGA(C_{i_j}, D_{i_j}).$$

To prove theorem 5.3 we establish the following lemma.

Lemma 5.2 *Let sig be a signature and let C and D be temporal clauses over sig.*

1. *If S_1 and S_2 are sequences of selections from C and D such that S_1 and S_2 contain the same selections, then*

$$LGA(C_{S_1}, D_{S_1}) \sim LGA(C_{S_2}, D_{S_2})$$

2. *If S_1 and S_2 are sequences of selections from C and D such that every selection from S_2 occurs in S_1, then*

$$LGA(C_{S_2}, D_{S_2}) \succcurlyeq_{tc} LGA(C_{S_1}, D_{S_2})$$

3. *If S is any sequence of selections from C and D, then*

$$LGA(C_S, D_S) \succcurlyeq_{tc} C \quad LGA(C_S, D_S) \succcurlyeq_{tc} D$$

Proof.

1. Let S_1 and S_2 contain the same selections. We proceed by induction on n, where n is the number of selections in S_1 and S_2. For $n = 1$ the claim is obvious. So assume that the claim holds for $n = m$. If S_1 and S_2 are sequences of selections which contain $m + 1$ identical selections, then we reorder S_2 in such a way that the $i-$th elements of S_1 and S_2 become identical. For the initial sequence S of S_1 and S_2 of length m the induction hypothesis holds. So we have that

$$\begin{aligned} LGA(C_{S_1}, D_{S_1}) &= LGA(C_S, D_S) \vee LGA(C_{m+1}, D_{m+1}) \\ &= LGA(C_{S_2}, D_{S_2}) \\ &\sim LGA(C_{S_2}, D_{S_2}) \end{aligned}$$

2. Let

$$\begin{aligned} S_2 &= \{(L_1, M_1), \ldots, (L_n, M_n)\} \\ S_1 &= S_2 \cup \{(L_{n+1}, M_{n+1}), \ldots, (L_{n+m}, M_{n+m})\}. \end{aligned}$$

Then we have

$$\begin{aligned} LGA(C_{S_2}, D_{S_2}) &= \bigcup_{i=1}^{n} LGA(L_i, M_i) \\ &= \left(\bigcup_{i=1}^{n} LGA(L_i, M_i) \right) \varepsilon \\ &\subseteq \bigcup_{i=1}^{n} LGA(L_i, M_i) \cup \bigcup_{i=n+1}^{n+m} LGA(L_i, M_i) \\ &= \bigcup_{i=1}^{n+m} LGA(L_i, M_i) \\ &= LGA(C_{S_1}, D_{S_1}) \end{aligned}$$

So $LGA(C_{S_2}, D_{S_2}) \succcurlyeq_{tc} LGA(C_{S_1}, D_{S_1})$.

3. Let i be arbitrary. If $LGA(L_i, M_i)$ is computed with algorithm 17, then there exist substitutions $\theta_{i_1}, \theta_{i_2}$ such that

$$\begin{aligned} LGA(L_i, M_i)\theta_{i_1} &= L_i \\ LGA(L_i, M_i)\theta_{i_2} &= M_i \end{aligned}$$

So we have

$$\begin{aligned} \left(\bigvee_{i=1}^{m} LGA(L_i, M_i) \right) \theta_{i_1} &\subseteq \bigvee_{i=1}^{n} L_i = C \\ \left(\bigvee_{i=1}^{m} LGA(L_i, M_i) \right) \theta_{i_2} &\subseteq \bigvee_{i=1}^{n} M_i = D \end{aligned}$$

So $LGA(C_S, D_S) \succcurlyeq_{tc} C$ and $LGA(C_S, D_S) \succcurlyeq_{tc} D$.

□

Proof of theorem 5.3 Let C_1 and C_2 be temporal clauses, let S be a sequence of all selections from C_1 and C_2 and let $C = LGA(C_{1,S}, C_{2,S})$. Then we have $C \succcurlyeq_{tc} C_1$ and $C \succcurlyeq_{tc} C_2$ with lemma 5.2.3. Now let $D = \{D_1, \ldots, D_m\}$ be a clause with $D \succcurlyeq_{tc} C_1$ and $D \succcurlyeq_{tc} C_2$. So there are substitutions θ_1 and θ_2 as well as literals $L_1, \ldots, L_m \in C_1$ and $M_1, \ldots, M_m \in C_2$ such that $D_i\theta_1 = L_i$ and $D_j\theta_2 = M_j$ for every $i, j \in \{1, \ldots, m\}$. Define the sequence of selections S' as

$$S = \{(L_1, M_1), \ldots, (L_m, M_m)\}$$

and define $G := LGA(C_{1,S'}, C_{2,S'}) := \{K_1, \ldots, K_m\}$. So there are substitutions σ_1, σ_2 with $G\sigma_1 = C_{1,S'}$ and $G\sigma_2 = C_{2,S'}$. Furthermore we have $(D_1 \vee \cdots \vee D_m)\,\theta_1 = C_{1,S'}$ and $(D_1 \vee \cdots \vee D_m)\,\theta_2 = C_{2,S'}$. So there is a substitution γ such that $(D_1 \vee \cdots \vee D_m)\gamma = K_1 \vee \cdots \vee K_m = G$. So $D \succcurlyeq_{tc} G$. Since every selection from S' also occurs in S, lemma 5.2.2 yields $G \succcurlyeq_{tc} C$ and therefore $D \succcurlyeq_{tc} C$ so C is indeed a least generalization of C_1 and C_2 □

The arguments from the proof of theorem 5.3 are summarized in algorithm 19.

Algorithm 19 LGS-Computation for Temporal Clauses

Input: temporal clauses $C_1 = \{L_1, \ldots, L_n\}, C_2 = \{M_1, \ldots, M_n\}$
Output: Least Generalization LGS of C_1 and C_2

1: $S \leftarrow \emptyset$
2: **for** $i = 1, \ldots, n$ **do**
3: **for** $j = 1, \ldots, n$ **do**
4: **if** L_i and M_j are compatible **then**
5: $S \leftarrow S \cup \{(L_i, M_j)\}$
6: **end if**
7: **end for**
8: **end for**
9: $m \leftarrow$Number of pairs in S
10: **return** $\bigvee_{i=1}^{m} LGA(L_i, M_i)$

In analogy to the computation of a greatest specialization we have to add the element $\perp$ to the possible clauses if we are working with horn clauses. The algorithm will then work exactly as algorithm 19 except that the result is $\perp$ if the heads of the two clauses are both nonempty and not compatible.
To conclude this section we illustrate algorithm 19 in an example.

Example 5.4 *Consider the two clauses*

$$\begin{aligned} C_1 &= \{p(x), \circ q(x,y), r(x, f(y))\} \\ C_2 &= \{\circ q(a, g(b)), r(f(z_1), z_2), \neg p(b)\} \end{aligned}$$

The set S of all selections of C_1 and C_2 is given as

$$S = \{(\circ q(x,y), \circ q(a, g(b))), (r(x, f(y)), r(f(z_1), z_2))\}$$

So we have

$$\begin{aligned} LGS(C_1, C_2) &= LGA(C_{1,S}, C_{2,S}) \\ &= LGA\left(\circ q(x,y), \circ q(a, g(b))\right) \vee LGA\left(r(x, f(y)), r(f(z_1, z_2))\right) \\ &= \circ q(z_3, z_4) \vee r(z_5, z_6) \end{aligned}$$

6 Theory Refinement Operators for Linear Clauses

6.1 General Concepts

In this section we will adopt the definitions of downward and upward refinement operators (see [NCdW97]) for the field of temporal logic programming and present concrete examples for both upward and downward refinement operators. We will present two principle kinds of refinement operators: operators which work on clauses and operators which work on whole programs, i.e. sets of clauses. But before we can state such operators and present algorithms to compute them we will first introduce some more formalism which will enable us to reason about properties of refinement operators more precisely.
So let M be any set and let $\geq$ be any quasi-order on M. Furthermore let ρ_u be an upward refinement operator and let ρ_d be a downward refinement operator on M. That is for every $C \in M$ it holds that

$$\rho_u(C) \subseteq \{D | D \geq C\}$$
$$\rho_d(C) \subseteq \{D | C \geq D\}$$

For $\rho \in \{\rho_u, \rho_d\}$ and $C \in M$ we define

$$\rho^1(C) = \rho(C)$$
$$\rho^{n+1}(C) = \{D | \text{ there exists } E \in \rho^n(C) \text{ with } D \in \rho(E)\}$$
$$\rho^*(C) = \bigcup_{i \geq 1} \rho^i(C)$$

In the following definition we define several properties of refinement operators.

Definition 6.1 *Let ρ_u be an upward refinement operator and let ρ_d be a downward refinement operator on a quasi ordered set $(M, \geq)$ and let $\rho \in \{\rho_u, \rho_d\}$.*

1. *ρ is called locally finite if for every $C \in M$ the set $\rho(C)$ is finite and can be computed effectively.*
2. *ρ_d is called complete if for every $C_1, C_2 \in M$ with $C_1 > C_2$ there exists en element $C_3 \in \rho_d^*(C_1)$ with $C_2 \approx C_3$. Similarly ρ_u is called complete if for every $C_1, C_2 \in M$ with $C_1 < C_2$ there is an element $C_3 \in \rho_u^*(C_1)$ with $C_2 \approx C_3$.*
3. *ρ_d is called proper if for every $C \in M$ it holds that $\rho_d(C) \subseteq \{D | C > D\}$. Similarly ρ_u is called proper if for every $C \in M$ it holds that $\rho_u(C) \subseteq \{D | D > C\}$.*
4. *ρ is called ideal if it is locally finite, complete and proper.*

One can show quite easily that there are signatures *sig* such that no ideal refinement operators for the set $(\mathfrak{C}(sig), \succcurlyeq_c)$ exist. This fact is closely related to the nonexistence of some kinds of covers. We refer the reader to [NCdW97] for more details. But we

note that this result is also important for the field of temporal logic: since the set of all first order horn clauses over *sig* can be embedded into the set of all temporal horn clauses over *sig*, this nonexiststence-result carries over to temporal logic.
In the next sections we will proceed as follows: first we will concentrate on refinement operators for clauses. We will define some such operators both for the upward and the downward case. We will also prove their properties. Similarly we will concentrate on the case of refinement operators for whole programs.

6.2 Refinement Operators for Clauses

in this section we will make the following assumptions:

- *sig* is an arbitrary but fixed signature.
- C is a temporal clause over *sig* which has the form

$$C = \{C_1, \ldots, C_n\}$$

with $C_i \in \mathcal{L}_{\text{temp}}^{\text{templit}}(\text{sig})$ for every i.

6.2.1 Upward Refinement Operators

Recall that for any pair C_1, C_2 of temporal clauses we have $C_1 \succcurlyeq_{tc} C_2$ if there is a substitution θ such that $C_1\theta \subseteq C_2$. So if ρ is an upward refinement operator for clauses and C is a temporal clause, we have

$$D \in \rho(C) \rightsquigarrow D \succcurlyeq_{tc} C \rightsquigarrow D\theta \subseteq C \text{ for some } \theta$$

This has to be ensured when defining such an operator. In the sequel we will present several upward refinement operators for temporal clauses under subsumption and prove their properties.

1. The first upward refinement operator which we will investigate is a very simple one: the initial set C of literals will be included in every step of refinement. This will cause the refined clauses to grow but it makes the analysis easy as we will see soon. Below we will see how the definition can be relaxed in order to receive *smaller* (i.e. consisting of less literals) refinement clauses which have the same properties as the clauses found by $\rho_{1,u}$. Therefore we define

$$\rho_{1,u}(C) := \left(\bigcup_{i,j=1,\ldots,n, i \neq j} \text{LGA}(C_i, C_j) \right) \cup C$$

That is: $\rho_{1,u}(C)$ contains of C and every possible least generalization of a pair C_i, C_j.

First we prove that $\rho_{1,u}$ can indeed be used for refining temporal clauses under subsumption that is $\rho_{1,u}$ defines an upward refinement operator for the subsumption ordering on temporal clauses.

Lemma 6.1 *$\rho_{1,u}$ is an upward refinement operator with respect to $\succcurlyeq_{tc}$.*

Proof. Since $C \subseteq C$ it suffices to show that there is a substitution θ such that $\left(\bigcup_{i,1=1,\ldots,n,i\neq j} \mathrm{LGA}(C_i, C_j)\right)\theta \subseteq C$. We have for every i and every j with $i \neq j$ that $\mathrm{LGA}(C_i, C_j) \succcurlyeq_{ta} C_i$ and $\mathrm{LGA}(C_i, C_j) \succcurlyeq_{ta} C_j$ so it also holds that $\mathrm{LGA}(C_i, C_j) \succcurlyeq_{tc} C_i$ and $\mathrm{LGA}(C_i, C_j) \succcurlyeq_{tc} C_j$. So there exist substitutions θ_i, θ_j such that $\mathrm{LGA}(C_i, C_j)\theta_i = C_i$ and $\mathrm{LGA}(C_i, C_j)\theta_j = C_j$. Assume that

$$\{C^{(1)}, \ldots, C^{(k)}\} = \{\mathrm{LGA}(C_i, C_j) | i, j = 1, \ldots, n, i \neq j\}$$

For $C^{(i)} = \mathrm{LGA}\,(C_{i_1}, C_{i_2})$ algorithm 17 computes two substitutions $\theta_{i_1}^{(i)}$ and $\theta_{i_2}^{(i)}$ such that $C^{(i)}\theta_{i_1} = C_{i_1}$ and $C^{(i)}\theta_{i_2} = C_{i_2}$. So if we set $\theta := \theta_1^{(1)} \circ \cdots \circ \theta_1^{(k)} \circ \theta_2^{(1)} \circ \cdots \circ \theta_2^{(k)}$ the claim follows. □

The following lemma states another property of $\rho_{1,u}$:

Lemma 6.2 *$\rho_{1,u}$ is locally finite.*

Proof. For $C = \{C_1, \ldots, C_n\}$ it holds that

$$|\{\mathrm{LGA}(C_i, C_j) | i, j = 1, \ldots, n, i \neq j\}| \leq n^2 - n.$$

So $|\rho_{1,u}(C)| \leq |C| + n^2 - n = n + n^2 - n = n^2$. So for arbitrary C we have that $\rho_{1,u}(C)$ is finite. The computability of $\rho_{1,u}$ follows from algorithm 17. So $\rho_{1,u}$ is indeed locally finite. □

Another nice property of $\rho_{1,u}$ is the completeness, that is for every pair C_1, C_2 of temporal clauses such that $C_2 \succ_{tc} C_1$ only a finite number of iterations of $\rho_{1,u}$ is necessary to compute a clause C_3 which is subsume equivalent to C_2.

Lemma 6.3 *$\rho_{1,u}$ is complete.*

Proof. This is a consequence of the fact that every application of $\rho_{1,u}$ only adds least generalizations. □

However, the following lemma is a negative result.

Lemma 6.4 *$\rho_{1,u}$ is not proper.*

Proof. trivial, since $C \subseteq C$. □

So the condition of idealness is not fulfilled.

Corollary 6.1 *$\rho_{1,u}$ is not ideal.*

2. As we have already mentioned, the operator $\rho_{1,u}$ in general constructs refinements which are *too large* in the sense that they contain too much literals. The next operator will avoid this but we will see that we have to put another restriction on a refinement operator in order to reach an operation which can be useful in practice. This will then be made precise when we define the operator $\rho_{3,u}$. But before we will define the operator $\rho_{2,u}^{(i,j)}$ which gets two additional inputs i and j together with the clause to be refined. These indices are the *numbers* of the literals which have to be refined. That is, if $C = \{C_1, \ldots, C_n\}$ and the operator is given i and j, we refer to the literals C_i and C_j as objects to be refined. This yields a very simple operator which has unfortunately bad properties. We define

$$\rho_{2,u}^{(i,j)}(C) := \mathrm{LGA}(C_i, C_j)$$

The property of being an upward refinement operator for subsumption is proved directly from the properties of the operation LGA.

Lemma 6.5 *For every choice of i and j $\rho_{2,u}^{(i,j)}$ is an upward refinement operator with respect to $\succcurlyeq_{tc}$.*

Proof. trivial, since for every pair i, j we have

$$\mathrm{LGA}(C_i, C_j) \succcurlyeq_{tc} C_i$$

and

$$\mathrm{LGA}(C_i, C_j) \succcurlyeq_{tc} C_j.$$

□

Since $\rho_{2,u}^{(i,j)}(C)$ is always a singleton, the locally finiteness is trivial.

Lemma 6.6 *For every choice of i and j $\rho_{2,u}^{(i,j)}$ is locally finite.*

However, $\rho_{2,u}^{(i,j)}$ is a strictly weaker operator than $\rho_{1,u}$.

Lemma 6.7 *There are i, j such that $\rho_{2,u}^{(i,j)}$ is not complete.*

Proof. Consider the clauses

$$C^{(1)} = \{\underbrace{p(x_1, x_2)}_{=C_1}, \underbrace{p(x_3, x_4)}_{=C_2}, \underbrace{q(x_1)}_{=C_3}\}$$
$$C^{(2)} = \{q(y_1)\}$$

Then for $i = 1$ and $j = 2$ we have

$$\begin{aligned}\rho_{2,u}^{(i,j)}(C^{(1)}) &= \rho_{2,u}^{(1,2)}(C^{(1)}) \\ &= \mathrm{LGA}\,(C_1, C_2) \\ &= \mathrm{LGA}\,(p(x_1, x_2), p(x_3, x_4)) \\ &= \{p(x_1, x_2)\}\end{aligned}$$

On the other hand we have $C^{(2)}\{y_1 \leftarrow x_1\} = \{q(x_1)\} \subseteq C^{(1)}$ and therefore $C^{(2)} \succcurlyeq_{tc} C^{(1)}$. But for every i one has $\left(\rho_{2,u}^{(1,2)}\right)^{i+1}(C^{(1)}) = \{p(x_1, x_2)\}$ and therefore no $C_3 \approx C_2$ with $C_3 \in \rho_{2,u}^{(1,2)}(C_1)$ exists. So, $\rho_{2,u}^{(1,2)}$ is not complete. □

Furthermore we have the following property.

Lemma 6.8 *There are i and j such that $\rho_{2,u}^{(i,j)}$ is not proper.*

Proof. Consider the clause $C = \{p(x)\}$. Only one pair is possible, namely $(1, 1)$ so $\rho_{2,u}^{(1,1)}(C) = C \not\succcurlyeq_{tc} C$. So the claim is proved. □

Corollary 6.2 *In general, $\rho_{2,u}^{(i,j)}$ is not ideal.*

3. In order to overcome the shortcomings of the operator $\rho_{2,u}^{(i,j)}$ we now define an operator which forms a new clause as the union of every application of $\rho_{2,u}^{(i,j)}$ which is possible. This operator is then closely related to $\rho_{1,u}$ and will therefore inherit the property of being complete.

$$\rho_{3,u}(C) := \bigcup_{i,j=1,\ldots,n} \rho_{2,u}^{(i,j)}(C)$$

Again we have immediately:

Lemma 6.9 *$\rho_{3,u}$ is an upward refinement operator with respect to $\succcurlyeq_{tc}$.*

Proof. We have that for every i and every j $\rho_{2,u}^{(i,j)}(C) \succcurlyeq_{tc} C$, that is there exists θ_{ij} such that $\rho_{2,u}^{(i,j)}(C)\theta_{ij} \subseteq C$. Since for distinct (i, j) $\rho_{2,u}^{(i,j)}(C)$ can be made variable disjoint, the composition

$$\theta = \theta_{11} \circ \cdots \circ \theta_{nn}$$

yields the claim. □

Lemma 6.10 *$\rho_{3,u}$ is locally finite.*

Proof. For $C = \{C_1, \ldots, C_n\}$ the claim follows from

$$|\rho_{3,u}(C) \leq n^2 \cdot \underbrace{\max\left\{\left|\rho_{2,u}^{(i,j)}(C)\right| \mid i,j \in \{1,\ldots,n\}\right\}}_{\leq 1}$$
$$\leq n^2$$

□

Lemma 6.11 *$\rho_{3,u}$ is complete.*

Proof. The claim follows from the equality

$$\rho_{3,u}(C) = \rho_{1,u}(C) \setminus C$$

and the completeness of $\rho_{1,u}$. □

Again, $\rho_{3,u}$ is not ideal, since the following lemma shows that the property of being proper is not fulfilled.

Lemma 6.12 *$\rho_{3,u}$ is not proper.*

Proof. The proof is exactly identical to the proof of lemma 6.8. □

Corollary 6.3 *$\rho_{3,u}$ is not ideal.*

4. The last operator which we will define is based on the set of all literals which are compatible to at least one other literal. Therefore let

$$I := \{i \leq n | C_i \text{ is compatible to at least one } C_j, i \neq j\}$$

and define

$$\rho_{4,u}(C) := \left(\bigcup_{i,j \in I} \mathrm{LGA}(C_i, C_j)\right) \cup \{C_k | k \notin I\}$$

Lemma 6.13 *$\rho_{4,u}$ is an upward refinement operator with respect to $\succcurlyeq_{tc}$.*

Proof. Let $C = \{C_1, \ldots, C_n\}$ be any temporal clause. We distinguish two cases:

Case 1 $I = \emptyset$. Then $\rho_{4,u}(C) = C \succcurlyeq_{tc} C$.

Case 2 $I \neq \emptyset$. Then let i, and j be indices in I[8]. Then $\mathrm{LGA}(C_i, C_j) \succcurlyeq_{tc} C_i$ and $\mathrm{LGA}(C_i, C_j) \succcurlyeq_{tc} C_j$. The rest of the proof is analogous to the proof of this property of $\rho_{1,u}$. □

Again, the locally finiteness of this operator is trivial.

[8]Note that i and j need not necessarily be distinct.

Lemma 6.14 *$\rho_{4,u}$ is locally finite.*

This operator is complete. The difference in the importance for practical applications is that there are often less pairs C_i, C_j which have to be checked for the existence of a nontrivial least generalization.

Lemma 6.15 *$\rho_{4,u}$ is complete.*

Proof. This property is proved similar to the completeness proof of $\rho_{1,u}$. □

Lemma 6.16 *$\rho_{4,u}$ is not proper.*

Proof. If $I = \emptyset$, then $\rho_{4,u}(C) = C \not\succ_{tc} C$. □

Corollary 6.4 *$\rho_{4,u}$ is not ideal.*

The results from this section are collected in the following theorem.

Theorem 6.1 *The upward refinement operators $\rho_{1,u}, \rho_{2,u}^{(i,j)}, \rho_{3,u}$ and $\rho_{4,u}$ have the following properties:*

1. *$\rho_{1,u}$ is locally finite and complete.*
2. *$\rho_{2,u}^{(i,j)}$ is locally finite.*
3. *$\rho_{3,u}$ is locally finite and complete.*
4. *$\rho_{4,u}$ is locally finite and complete.*

6.2.2 Downward Refinement Operators

In this section we will concentrate on the dual case of the results from section 6.2.1, namely we will introduce operators which specialize clauses in order to receive a clause which is more specific than the initial clause. This task is necessary whenever an ILP system given a program P is asked to construct a program P' which implies *less* than the initial program. In contrast to the last section we will only consider operators for the subsumption ordering since the corresponding operators for the lexicographic path ordering have the same negative properties as the upward refinement operators for the lexicographic path ordering which have been introduced in the last section. Since the only difference between the operators from this section and the upward refinement operators from last section is that we now specialize instead of to generalize, all results from last section also hold in this case. So this section will be rather short.

1. The dual operator to $\rho_{1,u}$ is given by replacing the least generalization by the greatest specialization.

$$\rho_{1,d}(C) := \left(\bigcup_{i,j=1,\ldots,n,i\neq j} \mathrm{GSA}(C_i, C_j) \right) \cup C$$

Therefore the following lemmata are dual to the lemmata stating the properties of $\rho_{1,u}$. They are proved analogously so we will omit the proofs here.

Lemma 6.17 *$\rho_{1,d}$ is a downward refinement operator with respect to $\succcurlyeq_{tc}$.*

Lemma 6.18 *$\rho_{1,d}$ is locally finite.*

Lemma 6.19 *$\rho_{1,d}$ is complete.*

Since $C \subseteq \rho_{1,d}(C)$, we have that in general $\rho_{1,u}(C) \not\succcurlyeq_{tc} C$.

Lemma 6.20 *$\rho_{1,d}$ is not proper.*

Corollary 6.5 *$\rho_{1,d}$ is not ideal.*

2. Again, we receive a downward refinement operator by replacing the LGS operation by the GSS operation.

$$\rho_{2,d}^{(i,j)}(C) := \mathrm{GSA}(C_i, C_j)$$

We will now concentrate on the properties of this operator.

Lemma 6.21 *For every choice of i and j $\rho_{2,d}^{(i,j)}$ is a downward refinement operator with respect to $\succcurlyeq_{tc}$.*

Lemma 6.22 *For every choice of i and j $\rho_{2,d}^{(i,j)}$ is locally finite.*

Proof. Again, this is trivial since $\rho_{2,d}^{(i,j)}(C)$ always consists of a singleton and is therefore finite. □

However, we will have to give a new counterexample in order to show that $\rho_{2,u}^{(ij)}$ is in general not complete.

Lemma 6.23 *There are i, j such that $\rho_{2,d}^{(i,j)}$ is not complete.*

Proof. Consider the clause $C = \{p(x), q(x)\}$ and $i = j = 1$. Then one easily verifies that $\rho_{2,d}^{(1,1)} = \{p(x)\}$ but there is no $D \in \left(\rho_{2,d}^{(1,1)}\right)^*$ with $D \approx \{q(x)\}$. □

Lemma 6.24 *There are i and j such that $\rho_{2,d}^{(i,j)}$ is not proper.*

Proof. Consider a clause which is a singleton, e.g. $C = \{p(x)\}$. Then $\rho_{2,d}^{(1,1)} = \{p(x)\} = C$ and the claim follows. □

Corollary 6.6 *In general, $\rho_{2,d}^{(i,j)}$ is not ideal.*

3. In analogy to $\rho_{3,u}$ one now has

$$\rho_{3,d}(C) := \bigcup_{i,j=1,\ldots,n} \rho_{2,d}^{(i,j)}(C)$$

The properties remain the same.

Lemma 6.25 *$\rho_{3,d}$ is a downward refinement operator with respect to $\succcurlyeq_{tc}$.*

Lemma 6.26 *$\rho_{3,d}$ is locally finite.*

Lemma 6.27 *$\rho_{3,d}$ is complete.*

Lemma 6.28 *$\rho_{3,d}$ is not proper.*

Proof. The proof is identical to the proof of lemma 6.24. □

Corollary 6.7 *$\rho_{3,d}$ is not ideal.*

4. In analogy to the operator $\rho_{4,u}$ from the last section we define the operator $\rho_{4,d}$ and state that it has the same properties.

$$\rho_{4,d}(C) := \left(\bigcup_{i,j \in I} \mathrm{GSA}(C_i, C_j) \right) \cup \{C_k | k \notin I\}$$

Lemma 6.29 *$\rho_{4,d}$ is a downward refinement operator with respect to $\succcurlyeq_{tc}$.*

Lemma 6.30 *$\rho_{4,d}$ is locally finite.*

Lemma 6.31 *$\rho_{4,d}$ is complete.*

Lemma 6.32 *$\rho_{4,d}$ is not proper.*

Corollary 6.8 *$\rho_{4,d}$ is not ideal.*

Again we conclude this section by summarizing the results in a theorem.

Theorem 6.2 *The downward refinement operators $\rho_{1,d}, \rho_{2,d}^{(i,j)}, \rho_{3,d}$ and $\rho_{4,d}$ have the following properties:*

1. *$\rho_{1,d}$ is locally finite and complete.*
2. *$\rho_{2,d}^{(i,j)}$ is locally finite.*
3. *$\rho_{3,d}$ is locally finite and complete.*
4. *$\rho_{4,d}$ is locally finite and complete.*

6.3 Refinement Operators for Sets of Clauses

In section 6.2 we have introduced several operators ρ which modify a clause C in order to receive a clause $\rho(C)$ which is either more general (in case of an upward refinement operator) or more specific (in case of a downward refinement operator) than the initial clause C. in this section we will introduce several operators which work on programs, i.e. sets of clauses. Again we can distinguish between downward and downward refinement operators.

Upward Refinement Operators In this section we will present two operators which can be used to refine a program P. Therefore assume that $P = \{P_1, \ldots, P_n\}$ is a program so each P_i is a temporal horn clause. The two upward refinement operators Θ_1^u and Θ_2^u are defined as

$$\begin{aligned} \Theta_1^u(P) &= \bigcup_{i,j=1,\ldots,n} \mathrm{LGS}(P_i, P_j) \\ \Theta_2^u(P,i,j) &= P \cup \mathrm{LGS}(P_i, P_j) \end{aligned}$$

These operators can be seen as a *global one* (operator Θ_1^u) and a *local one* (operator Θ_2^u). This is due to the fact that Θ_1^u adds *every* possible refinement of two clauses to the original program (note that $P \subseteq \Theta_1^u(P)$, since $\mathrm{LGS}(P_i, P_i) = p_i \in \Theta_1^u(P)$ for every i). In contrast, Θ_2^u Only adds these refinement which it is told (via the parameters i and j) to do. The problem using Θ_2^u is to detect which values of i and j are needed to call Θ_2^u.
However we won't bother with implementation details here and concentrate on the formal properties of the two operators. First we show that in the case of positive information the set of logically implied objects is not negatively affected by the application of any of the two introduced operators.

Lemma 6.33 *Let $P \in \mathcal{L}_t$ be a temporal logic program and let $\psi \in \mathcal{L}_{temp}^{tempatom}(sig)$ be a ground goal. Then it holds that*

1. *if $P \models \psi$, then $\Theta_1^u(P) \models \psi$*
2. *if $P \models \psi$, then $\Theta_2^u(P,i,j) \models \psi$ for every i and j.*

Proof. Both claims can be proved by exploiting $\Theta_1^u(P) = P \cup M_1$ and $\Theta_2^u(P) = P \cup M_2$ for some sets M_1, M_2 of temporal horn clauses. Since $P \models \psi$, any proof of $\Theta_1^u(P) \cup \{\neg\psi\} \vdash^* \square$ or $\Theta_2^u(P) \cup \{\neg\psi\} \vdash^* \square$ can be carried out independent of the sets M_1 and M_2. $\square$

The general situation in which one of the above operators has to be called, is the following: Assume that P is the actual temporal program which an ILP system has constructed and $\mathcal{E}^+$ and $\mathcal{E}^-$ are the given sets of positive and negative examples. If there is $e \in \mathcal{E}^+$ such that $P \not\models e$, then a call to Θ_1^u or Θ_2^u can eventually construct a program which has this property. Of course one has to be careful not to generalize too much since it could happen that after the application of Θ_1^u a negative example is implied, i.e. $\Theta_1^u(P) \models e'$ for some $e' \in \mathcal{E}^-$. This is clearly a situation in which Θ_2^u should be preferred. Nevertheless since

$$\begin{aligned}
\Theta_1^u(P) &= \Theta_1^u(\{P_1, \ldots, P_n\}) \\
&= \bigcup_{i,j=1,\ldots,n} \mathrm{GSS}(P_i, P_j) \\
&= P \cup \bigcup_{i,j=1,\ldots,n, i \neq j} \mathrm{GSS}(P_i, P_j) \\
&= P \cup \bigcup_{i,j=1,\ldots,n, i \neq j} \Theta_2^u(P, i, j)
\end{aligned}$$

we have that every call of Θ_1^u can be simulated by several single calls of Θ_2^u.

A problem which we have to attack is that after applying one of the operators Θ_1^u and Θ_2^u the new program eventually becomes *too big* in the sense that some of the clauses may contain more literals. Here we will now adopt the concept of reducedness introduced in chapter 5. We will work as follows: if $P = \{P_1, \ldots, P_n\}$ is a temporal program, we define for P_i the clause $\mathrm{RED}(P_i)$ to be a reduced clause which is subsume equivalent to P_i. Similarly for P we have $\mathrm{RED}(P) = \{\mathrm{RED}(P_1), \ldots, \mathrm{RED}(P_n)\}$. This construction enables us always to keep the *smallest* program which fits our needs. This approach will now be justified. Therefore we will first prove the following lemma.

Lemma 6.34 *Let P be a temporal program and let G be temporal ground goal. Then every Resolution step in a refutation $P \cup \{\neg G\} \vdash^* \square$ can also be carried out in* $\mathrm{RED}(P)$.

Proof. Let $P = \{P_1, \ldots, P_n\}$ be given and assume that

$$\mathrm{RED}(P) = \{\mathrm{RED}(P_1), \ldots, \mathrm{RED}(P_n)\}$$

is constructed from P. Furthermore let $G = G_1, \ldots, G_m$ be a temporal ground goal. Assume further that $P \cup \{\neg G\} \vdash^* \square$. Since every P_i is subsume equivalent to its reduced version $\mathrm{RED}(P_i)$ we have that there are substitutions θ_i such that $P_i\theta_i \subseteq \mathrm{RED}(P_i)$. We now consider a resolution step $G_1, \ldots, G_n = G^{(1)} \vdash G^{(2)}$. Then $G^{(2)} = \mathrm{Res}(P_i, G, \sigma)$ for some i and $\sigma = \mathrm{mgu}(\mathrm{HEAD}(P_i), G_j)$ for some j. If

$$P_i = A \leftarrow B_1, \ldots, B_m$$

then

$$G^{(2)} = (G_1, \ldots, G_{j-1}, B_1, \ldots, B_m, G_{j+1}, \ldots, G_n)\,\sigma$$

Since $A\theta_i^1 \in \text{RED}(P_i)$ we have that $\sigma' = \text{mgu}(A\sigma_i^1, G_j)$ exists due to $P_i \approx \text{RED}(P_i)$. So we can construct the new Resolvent

$$G^{(2)\prime} = \left(G_1, \ldots, G_{j-1}, B_1\theta_i^1, \ldots, B_m\theta_i^1, G_{j+1}, \ldots, G_n\right)\sigma'$$

and therefore the claim is proved. □

An application of lemma 6.34 is now given by the following theorem.

Theorem 6.3 *Let P be a temporal logic program and let ψ be a temporal ground goal. If $P \models \psi$, then $\text{RED}(P) \models \psi$.*

Proof. If $P \models \psi$, then there exists a temporal SLD-refutation for $P \cup \{\neg\psi\} \vdash^* \square$. Each step in this refutation uses an input clause from P. Due to lemma 6.34 each of these steps can be simulated using a clause from $\text{RED}(P)$. So we have that $\text{RED}(P) \models \psi$.□

So theorem 6.3 allows the usage of reduced clauses for theory refinement. Furthermore during inducing a program an ILP system can always keep the actual programs reduced. So no unnecessary information will be stored at any point of time. This approach is now formulated in algorithms 20 and 21 for both operators Θ_1^u and Θ_2^u.

Algorithm 20 Global Upward Theory Refinement; Θ_1^u

Input: temporal program $P = \{P_1, \ldots, P_n\}$
Output: refined temporal program P'

```
1:  P' ← ∅
2:  for i = 1,...,n do
3:     P'_i ← RED(P_i)
4:     P' ← P' ∪ {P'_i}
5:  end for
6:  for i = 1,...,n do
7:     for j = 1,...,n do
8:        if i ≠ j then
9:           P' ← P' ∪ {RED(LGS(P'_i, P'_j))}
10:       end if
11:    end for
12: end for
13: return P'
```

Although the programs which are constructed using algorithms 20 and 21 only contain clauses without redundant information, there is still the possibility that there are clauses which are *useless* in the sense that they are not needed in any refutation. Since we deal with both positive and negative examples, there are two possibilities:

positive If $e \in \mathcal{E}^+$ is a positive example, a program P is accurate if $P \models e$. So if this is the case, then there is a refutation for $P \cup \{\neg e\} \vdash^* \square$. We define the set POSDEP

Algorithm 21 Local Upward Theory Refinement; Θ_2^u

Input: temporal program $P = \{P_1, \ldots, P_n\}$, integers i, j
Output: refined temporal program P'

1: **if** $i = j$ **then**
2: **return** P
3: **else**
4: **return** $P \cup \{\text{LGS}(P_i, P_j)\}$
5: **end if**

to be the set of all clauses from P which occur in at least one such refutation. Of course it is possible that there exists more than one such SLD-refutation. But for the theory it is not important which of these refutations is used. But we should note that if $P \models e$ then there exists at least one set $\mathcal{U} \subseteq P$ with $\mathcal{U} \models e$ which has the property that every $\mathcal{V} \subseteq P$ with $|\mathcal{V}| < |\mathcal{U}|$ does not imply this example. Such sets are called *minimal contradictive wrt.* (P, e). Unfortunatelly it is undecidable if an arbitrary subset $\mathcal{U} \subseteq P$ is minimal contradictive wrt. (P, e) (see [Kol03]). Formally we define a set PosDep by

$$\begin{aligned}\text{PosDep}(P, e) \quad = \quad & \{P_i \in P | P_i \text{ is used in at least one refutation} \\ & \text{of } P \cup \{\neg e\} \vdash^* \square\}\end{aligned}$$

and the set of all clauses which are needed in at least one refutation of at least one positive example $e \in \mathcal{E}^+$ as

$$\text{Pos}\left(P, \mathcal{E}^+\right) = \bigcup_{e \in \mathcal{E}^+} \text{PosDep}(P, e)$$

So clearly the clauses $C \in P \setminus \text{Pos}(P, \mathcal{E}^+)$ are useless in the sense described above. They can in principle be removed from P.[9]

negative The case of negative information is more complicated than the above case of positive information. The main reason for this is the undecidability of logical implication. Therefore we will exploit the technique of *negation as failure* which we have introduced in chapter 3. That is for testing if $P \not\models e$ for some $e \in \mathcal{E}^-$ it suffices to show that there is a finitely failed fair SLD-tree rooted with $\neg e$. But we will not go into this further since it will not yield new information which can be used in the theory.

A philosophical note on the above remarks is that the principle which we have described corresponds with the principle of *Occam's Razor* (see [BEHW87]) which can be roughly described as: *If you have the choice between two explanations for a phenomenon, then choose the one which is simpler.* It is obvious that the approach of removing clauses

[9] Of course it might be possible that these clauses will be needed again at a later point of time. So they will have to be reconstructed by an ILP-algorithm. This is a drawback between the speed of inference and the size of the inferred programs.

which are not needed to derive the positive examples are *redundant* in this sense and that a program which contains such clauses is *not simple* in the sense of Occam's Razor. The simplest program in this sense would then be the union of all minimal contradictive sets for every $e \in \mathcal{E}^+$. Since in general, these sets are not uniquely determined, there is the possibility that two or more programs have to be considered to be the simplest solutions. In general, if there are n elements in $\mathcal{E}^+, m_i$ is the number of minimal contradictive sets wrt. (P, e_i) and $m = \sum_{i=1}^{n} m_i$ with $\{M_1, \ldots, M_m\}$ is the multiset[10] which consists of all minimal contradictive subsets for some $e \in \mathcal{E}^+$ then the total size of *minimal* solutions for the model inference problem induced by $(\mathcal{E}^+, \mathcal{E}^-)$ is given as

$$t = m - |\{(i,j) | i \neq j, M_i = M_j\}|$$

Downward Refinement Operators In the case of downward refinement of programs we can again consider a global and a local approach. The interpretation is the same as in the case of upward refinement except that we do not generalize but specialize now. The definition of the two operators which can be used is then given as

$$\begin{aligned} \Theta_1^d(P) &= \bigcup_{i,j=1,\ldots,n} \mathrm{GSS}(P_i, P_j) \\ \Theta_2^d(P,i,j) &= P \cup \mathrm{GSS}(P_i, P_j) \end{aligned}$$

We immediately have results dual to the results for the upward refinement operators Θ_1^u and Θ_2^u.

Lemma 6.35 *Let P be a program and let ψ be a ground goal. Then*

1. *if $P \not\models \psi$, then $\Theta_1^d(P) \not\models \psi$*
2. *if $P \not\models \psi$, then $\Theta_2^d(P,i,j) \not\models \psi$ for every i, j*

Proof. Every $C \in \Theta_1^d(P) \setminus P$ ($C \in \Theta_2^d(P,i,j)$) is a specialization of two clauses from P, i.e. $C = \mathrm{GSS}(P_i, P_j)$ for some $i, j \in \{1, \ldots, n\}$. So we have $P \models C$. Assuming $\Theta_1^d(P) \models \psi$ or $\Theta_2^d(P,i,j) \models \psi$ yields $P \models \psi$ which contradicts the assumption $P \not\models \psi$. □

The above lemma enables an ILP system to refine programs and to ensure that if a negative example is not implied by the original program P then is is also not implied by the refined program P'.

We can also adopt the concept of reducing the created program which we have described in the last section for upward refinement. So the programs created by the operators Θ_1^d and Θ_2^d can also be seen as *smallest* programs which are possible solutions of the model inference problem.

To conclude this section implementations of the two operators are given in algorithms 22 and 23.

[10] Roughly speaking, a multiset is a set in which elements can occur more than once.

Algorithm 22 Global Downward Theory Refinement; Θ_1^d

Input: temporal program $P = \{P_1, \ldots, P_n\}$
Output: refined temporal program P'

1: $P' \leftarrow \emptyset$
2: **for** $i = 1, \ldots, n$ **do**
3: $\quad P'_i \leftarrow \textsc{Red}(P_i)$
4: $\quad P' \leftarrow P' \cup \{P'_i\}$
5: **end for**
6: **for** $i = 1, \ldots, n$ **do**
7: $\quad$ **for** $j = 1, \ldots, n$ **do**
8: $\quad\quad$ **if** $i \neq j$ **then**
9: $\quad\quad\quad P' \leftarrow P' \cup \{\textsc{Red}\left(\text{GSS}(P'_i, P'_j)\right)\}$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: **end for**
13: **return** P'

Algorithm 23 Local Downward Theory Refinement; Θ_2^u

Input: temporal program $P = \{P_1, \ldots, P_n\}$, integers i, j
Output: refined temporal program P'

1: **if** $i = j$ **then**
2: $\quad$ **return** P
3: **else**
4: $\quad$ **return** $P \cup \{\text{GSS}(P_i, P_j)\}$
5: **end if**

7 Complexity Issues

In this section we will investigate properties of certain classes of programs in $\mathcal{L}_{\text{temp}}(\text{sig})$ with respect to identifiability in the PAC-setting (a reader which is not familiar with this model is referred to [Fis99]). If we are able to show that a class $\mathcal{C}$ of programs from $\mathcal{L}_{\text{temp}}(\text{sig})$ has $v := \text{VCDIM}(\mathcal{C}) < \infty$ then $\mathcal{C}$ is learnable in the PAC setting.

7.1 The VC-Dimension of some Classes of Linear temporal Programs

We will now analyze the complexity of some classes of $\mathcal{L}_{\text{temp}}(\text{sig})$ programs by stating their *VC-Dimension*. We will restrict on propositional programs since this case is more tractable than the general first order case where we would have to deal with many results of undecidability when handling relations defining nontotal function graphs.
We will now define four classes of $\mathcal{L}_t$ programs. For this assume that a clause C is again given in set notation that is $C = \{C_1, \ldots, C_n\}$. Recall that $\text{NEXT}(L)$ was defined on literals to be the number of occurrences of $\circ$ in L. We extend this concept to clauses by defining

$$\text{NEXT}(C) = \max\{\text{NEXT}(C_i) | i = 1, \ldots, n\}$$

Definition 7.1 *Let $\mathcal{L}_t$ be defined over a propositional signature sig. Let $n, j \in \mathbb{N}$ be arbitrary integers. Then*

$$\begin{aligned}
\text{SINGLE}_n &= \{P \in \mathcal{L}_t | P = \{P_1, \ldots, P_n\} \text{ and } \forall i : \text{NEXT}(P_i) \leq 1\} \\
\text{BOUNDED}_{n,j} &= \{P \in \mathcal{L}_t | P = \{P_1, \ldots, P_n\} \text{ and } \forall i : \text{NEXT}(P_i) \leq j\} \\
\text{UNBOUNDED}_n &= \{P \in \mathcal{L}_t | P = \{P_1, \ldots, P_n\}\} \\
\text{UNBOUNDED} &= \bigcup_{n \geq 0} \text{UNBOUNDED}_n
\end{aligned}$$

For $n \geq 2$ and j fixed we clearly have the following (in)equalities:

$$\begin{aligned}
\text{SINGLE}_n &= \text{BOUNDED}_{n,1} \\
\text{BOUNDED}_{n,j} &\subset \bigcup_{j \geq 1} \text{BOUNDED}_{n,j} \\
&= \text{UNBOUNDED}_n \\
&\subset \bigcup_{n \geq 0} \text{UNBOUNDED}_n \\
&= \text{UNBOUNDED}
\end{aligned}$$

The first class to be analyzed is SINGLE_n for a fixed value of n. This class consists of all programs with at most n program statements which only contain literals with at most one application of $\circ$. We have that for any $X \in \text{SINGLE}_n$ the size of X is bounded from above by n. In particular, X is a finite set. If we make the assumptions that

- every clause in an element $X \in \text{SINGLE}_n$ is seen as a set and
- this set is considered to be unordered,

then the following lemma is immediately:

Lemma 7.1 *Let sig be a propositional signature. Then there are* $2^{2|Pr|+1}$ *different program clauses which can be built from sig.*

Proof. Since every statement has a nonempty head we have that there are $2|Pr|$ different heads (for every predicate symbol p and $\circ p$ are possible). The number of tails is now given by the number of subsets of the set $\{1, \ldots, 2|Pr\}$. This number is equal to $2^{2|Pr|}$. So we have $2|Pr| \cdot 2^{2|Pr|} = 2^{2|Pr|+1}$ different clauses. □

Furthermore we have that SINGLE_n consists of

$$\sum_{k=0}^{n} \binom{2^{2|Pr|+1}}{k} \tag{2}$$

different programs. This number can be estimated as follows:

$$\begin{aligned} \sum_{k=0}^{2^{2|Pr|+1}} \binom{2^{2|Pr|+1}}{k} &\leq \left(\frac{e \cdot 2^{2|Pr|+1}}{n}\right)^n \\ &\leq \left\lceil \left(\frac{e \cdot 2^{2|Pr|+1}}{n}\right)^n \right\rceil < \infty \end{aligned}$$

In particular, the number of programs in SINGLE_n is finite.
Now the value of $\text{VCDIM}(\text{SINGLE}_n)$ will be constructed. Therefore we will exploit the fact that there are only finitely many programs in SINGLE_n. So there is an enumeration $\{P_1, \ldots, P_m\} = \text{SINGLE}_n$ with $P_i \neq P_j$ for $i \neq j$ and some value of m. So for every $X \in \text{SINGLE}_n$ we have

$$\begin{aligned} \Pi_{\text{SINGLE}_n}(X) &= \{X \cap P \mid P \in \text{SINGLE}_n\} \\ &= \{X \cap P_i \mid i = 1, \ldots, m\} \\ &= \bigcup \{X \cap P_i \mid i = 1, \ldots, m\} \\ &= X \end{aligned}$$

Since $|X| \leq n$, we have

$$\text{VCDIM}(\text{SINGLE}_n) = |\Pi_{\text{SINGLE}_n}| = 2^k \Leftrightarrow k = \log_2 n.$$

This proves the following theorem.

Theorem 7.1 *For every* $n \in \mathbb{N}$ *the class* SINGLE_n *is PAC-learnable. It has*

$$\text{VCDIM}(\text{SINGLE}_n) = \log_2 n.$$

The next class we will examine is the class $\text{BOUNDED}_{n,j}$ for given values of n and j. This is simple, since $\text{BOUNDED}_{n,j}$ is also finite. The only difference to SINGLE_n is the number of different programs in $\text{BOUNDED}_{n,j}$. So again for $X \in \text{BOUNDED}_{n,j}$ we have $\Pi_{\text{BOUNDED}_{n,j}} = X$ and therefore the following result is immediately:

Theorem 7.2 *For every $n, j \in \mathbb{N}$ the class $\text{BOUNDED}_{n,j}$ is PAC-learnable. It has*

$$\text{VCDIM}\,(\text{BOUNDED}_{n,j}) = \log_2 n.$$

In contrast to the two classes which we have examined so far, the class UNBOUNDED_n is infinite. But since for every $X \in \text{UNBOUNDED}_n$ it holds that $|X| \leq n$, we can again state a positive result concerning the learnability of this class. So let n be any fixed integer and let X be a program from UNBOUNDED_n. Then we again have

$$\begin{aligned}\Pi_{\text{UNBOUNDED}_n}(X) &= \{X \cap P \mid P \in \text{UNBOUNDED}_n\} \\ &= \{X \cap P \mid P = \{P_1, \ldots, P_m\} \text{ for } m \leq n\} \\ &= \{X \cap P \mid |P| \leq n\} \\ &= X\end{aligned}$$

So we again have (since $|P| \leq n < \infty$) the following theorem.

Theorem 7.3 *For every $n \in \mathbb{N}$ the class UNBOUNDED_n is PAC-learnable. It has*

$$\text{VCDIM}\,(\text{UNBOUNDED}_n) = \log_2 n.$$

In contrast to these three positive results concerning the PAC-learnability of classes of propositional $\mathcal{L}_t$–programs, we will now see that the class UNBOUNDED of **all** propositional $\mathcal{L}_t$–programs is not PAC-learnable.
Of course for $X \in \text{UNBOUNDED}$ we still have $\Pi_{\text{UNBOUNDED}}(X) = X$. But since X can contain any finite number of clauses, the value of VCDIM (UNBOUNDED) is not bounded anymore. This is a simple consequence of the following lemmata.

Lemma 7.2 *For every $X \in \text{UNBOUNDED}$ there is a minimal value k with $X \in \text{UNBOUNDED}_k$ which is uniquely determined.*

Proof. trivial. □
Now we will show that the class UNBOUNDED is not PAC-learnable.

Lemma 7.3

$$\text{VCDIM}\,(\text{UNBOUNDED}) = \infty$$

Proof. Assume that there is $k \in \mathbb{R}$ such that $k = \text{VCDIM}\,(\text{UNBOUNDED})$. Then choose the maximum value $n_o \in \mathbb{N}$ with $\log n_0 \leq k$. So we have

$$\text{VCDIM}\,(\text{UNBOUNDED}) = \text{VCDIM}\,(\text{UNBOUNDED}_{n_0}).$$

Assume now that $P \in \text{UNBOUNDED} \setminus \text{UNBOUNDED}_{n_0}$. Due to lemma 7.2 there is a uniquely determined minimal value n such that $P \in \text{UNBOUNDED}_{n_0+n}$. But since

$$\Pi_{\text{UNBOUNDED}_{n_0+n}}(P) = P \notin 2^{\text{UNBOUNDED}_{n_0}}$$

we have a contradiction. So indeed $\text{VCDIM}(\text{UNBOUNDED}) = \infty$ as claimed. □
An immediate consequence of this lemma is the following theorem which concludes this subsection.

Theorem 7.4 *The class* UNBOUNDED *is not PAC-learnable.*

7.2 Efficiency Results for some Classes of Programs

This subsection will briefly consider results regarding the number of examples which are needed for identifying a program from a class which is PAC learnable. Since the classes $\text{SINGLE}_n, \text{BOUNDED}_{n,j}$ and UNBOUNDED_n have identical values

$$\text{VCDIM}(\text{SINGLE}_n) = \text{VCDIM}(\text{BOUNDED}_{n,j}) = \text{VCDIM}(\text{UNBOUNDED}_n) = \log n$$

we will see that the number of examples needed for identifying a program from any of these three classes is always the same. Therefore we exploit a result from the theory of PAC-learning which is well known. It is stated in the following lemma.

Lemma 7.4 *Let $\mathcal{C}$ be a class which is PAC-learnable and let ε, δ be given with $0 < \varepsilon \leq \frac{1}{2}$ and $0 < \delta < 1$. Then for every consistent learning algorithm A the number $m(\varepsilon, \delta)$ can be estimated by*

$$m(\varepsilon, \delta) \leq \max\left\{\frac{4}{\varepsilon}\ln\frac{4}{\delta}, \frac{8\text{VCDIM}(\mathcal{C})}{\varepsilon}\ln\frac{13}{\varepsilon}\right\}$$

In the case of $\text{SINGLE}_n, \text{BOUNDED}_{n,j}$ and UNBOUNDED_n this number gives

$$m(\varepsilon, \delta) \leq \max\left\{\frac{4}{\varepsilon}\ln\frac{4}{\delta}, \frac{8\log n}{\varepsilon}\ln\frac{13}{\varepsilon}\right\}$$

Example 7.1 *Assume that $\mathcal{C} = \text{UNBOUNDEX}_{156}$. Then for $\varepsilon = 0.02, \delta = 0.04$ an algorithm needs at most*[11]

$$\begin{aligned} m(\varepsilon, \delta) &\leq \max\left\{\frac{4}{\varepsilon}\ln\frac{4}{\delta}, \frac{8\log_2 n}{\varepsilon}\ln\frac{13}{\varepsilon}\right\} \\ &= \max\left\{\frac{4}{0.02}\ln\frac{4}{0.04}, \frac{8\log_2 156}{0.02}\ln 130.02\right\} \\ &= \max\left\{200\ln 100, 400\frac{\ln 156}{\ln 2}\ln 650\right\} \\ &= \max\{921, 18875\} \\ &= 18875 \end{aligned}$$

examples to ensure the PAC-criterion for ε and δ.

[11] We will use (rounded) integers here.

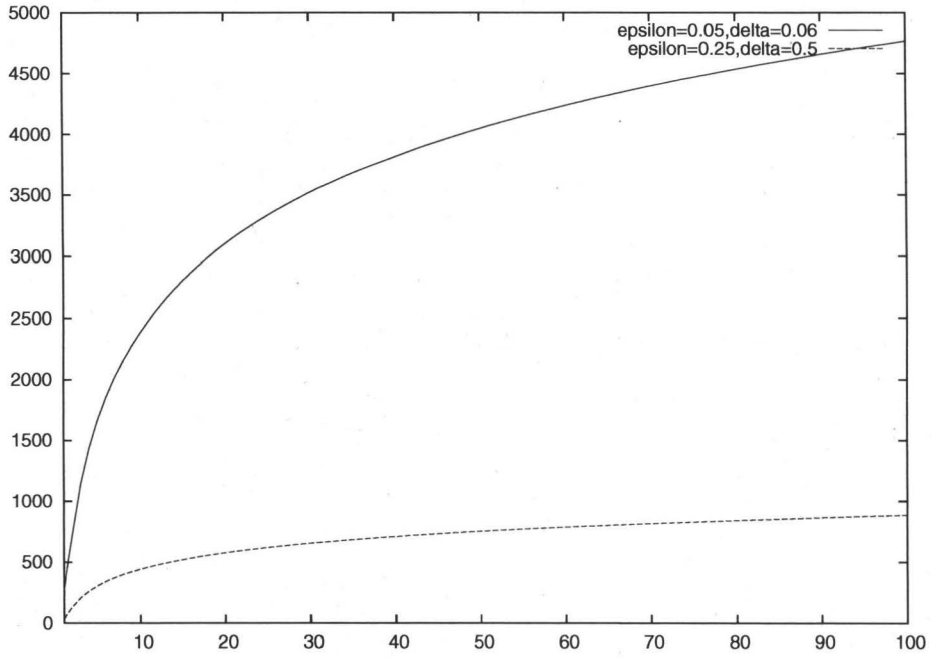

Figure 11: Maximum number of examples depending on ε and δ.

One clearly sees from this example that the number of positive and negative examples necessary for learning the classes SINGLE_n, $\text{BOUNDED}_{n,j}$ and UNBOUNDED_n can grow very fast with n. But in the example $n = 156$ is chosen very large. In many practical applications this value will be much smaller.
Figure 11 shows the growth of the function $m(\varepsilon, \delta)$ for some fixed values of ε and δ and n ranging from 1 to 100.

8 Conclusions

We have introduced a framework for the synthesis of programs written in a linear temporal programming language from positive and negative examples. Although the language which we have uses is a very simple one, many important concepts arising in temporal logic can be formulated in our language. In particular, every finite state automaton can be described by a program in our language. Therefore every temporal program which can be modeled by such a finite automaton can be described by a program from our language.
Further work will be considered with two topics:

- Induction of programs in more powerful languages, that is temporal languages containing constructs for fixpoints, and induction of programs containing constraints. Good candidates for an analysis afe full LTL which contains *Always*– and *Until*–constructs and CTL respectively CTL* containing state as well as path formulas.
- Theory Refinement of propositional temporal programs based on the structure of their representing automata. This approach will not be limited to the language presented in this report, since also more powerful propositional languages such as LTL ([MP92]), CTL ([EC82], [CE81]) and CTL* ([ES84]) can be represented by suitable ω–automata.

List of Algorithms

References

[AM86] M. Abadi and Z. Manna. A Timely Resolution. In *Symposium on Logic in Computer Science*, pages 176–186. IEEE, 1986.

[AM88] M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8:277–295, 1988.

[Apt97] K.R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[BEHW87] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Occam's Razor. *Information Processing Letters*, 24:377–380, 1987.

[Bry86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, pages 52–71, 1981.

[Die00] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics.* Springer Verlag, 2000.

[EC82] E.A. Emerson and E.M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[ES84] E.A. Emerson and A.P. Sistla. Deciding Full Branching Time Logic. *Information and Control*, 61(3):175–201, 1984.

[Fis99] P. Fischer. *Algorithmisches Lernen.* Teubner Verlag, 1999. (in German).

[Gun97] J. Gunnarsson. *Symbolic Methods and Tools for Discrete Event Dynamic Systems.* PhD thesis, Linköping University, Linköping, Sweden, 1997.

[Hop69] J.E. Hopcroft. *Formal Languages and Their Relation to Automata.* Addison-Wesley, 1969.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Kol03] R. Kolter. Implementation Issues in Inductive Logic Programming. Technical Report 325/03, Fachbereich Informatik, Universität Kaiserslautern, 2003.

[Mal97] R. Malik. *Automatische Synthese diskreter Steuerungen aus logischen Spezifikationen.* PhD thesis, Universität Kaiserslautern, Kaiserslautern, Germany, 1997. (in German).

[Mal98] R. Malik. Automated Deduction of Finite-State Control Programs for Reactive Systems. In *Proceedings of the 15th International Conference on Automated Deduction (CADE).* Springer Verlag, 1998. Lecture Notes in Computer Science 1421.

[MB88] S.H. Muggleton and W. Buntine. Machine Invention of First-order Predicates by Inverting Resolution. In *Proceedings of the 5th International Conference on Machine Learning*, pages 339–352. Morgan Kaufmann, 1988.

[MM82] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer Verlag, 1992.

[NCdW97] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming.* Springer Verlag, 1997.

[Pnu92] A. Pnueli. System Specification and Refinement in Temporal Logic. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 1–38. Springer Verlag, 1992. Lecture Notes in Computer Science 652.

[PW78] M.S. Paterson and M.N. Wegman. Linear Unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Rau02] W. Rautenberg. *Einführung in die mathematische Logik.* Vieweg Verlag, second edition, 2002. (in German).

[RPVW95] R.Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[SH61] H. Scholz and G. Hasenjaeger. *Grundzüge der mathematischen Logik.* Springer Verlag, 1961. (in German).

[Sha81] E.Y. Shapiro. An Algorithm that Infers Theories from Facts. In *Proceedings of the 7th Joint Conference on Artificial Intelligence (IJCAI-81)*, pages 446–451. Morgan Kaufmann, 1981.

[Sho67] J.R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[SS88] M. Schmidt-Schauss. Implication of Clauses is Undecidable. *Theoretical Computer Science*, 59:287–296, 1988.

[vW99] C. van Wyk. An LTL Verification System Based on Automata Theory. Master's thesis, University of Stellenbosch, 1999.

[Wes01] D.B. West. *Introduction to Graph Theory.* Prentice Hall, 2001.

Index