LPAR-12

The 12th International Conference on

Logic for Programming, Artificial Intelligence, and Reasoning

Short Paper

Proceedings

Edited by:

Geoff Sutcliffe

Andrei Voronkov

Preface

This volume contains the short papers presented at the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-12), held 2-6 December 2006, in Montego Bay, Jamaica.

November 2006

Geoff Sutcliffe
Andrei Voronkov

Table of Contents

Extensional Higher-Order Datalog[⋆]

Vassilis Kountouriotis[1], Panos Rondogiannis[1], and William W. Wadge[2]

[1] Department of Informatics & Telecommunications
University of Athens, Athens, Greece
{grad0771,prondo}@di.uoa.gr
[2] Department of Computer Science
University of Victoria, Victoria, BC, Canada
wwadge@csr.uvic.ca

Abstract. We define a higher-order extension of Datalog based on the Horn fragment of higher-order logic introduced in [Wad91]. Programs of Higher-Order Datalog can be understood declaratively as formulas in extensional higher-order logic, in which (for example) a unary predicate of unary predicates is a set of sets of data objects. The language retains all the basic principles of first-order logic programming. In particular, programs in this extended Datalog always have a minimum Herbrand model which can be computed in a bottom-up way. We present the syntax and semantics of our extended Datalog, state the main result cited above, and describe an implementation of this new language.

1 Introduction

We define a higher-order extension of Datalog based on the Horn fragment of higher-order logic introduced in [Wad91]. Our system is extensional in the traditional sense, in which a predicate denotes its extension (ie., the set of all arguments for which the predicate is true). Ordinary logic programming, based as it is on first-order logic, has always been extensional: unary (say) predicates denote sets of data objects. Our system extends this concept to higher orders. For example, a unary predicate of unary predicates denotes a set of sets of data objects. Extensionality is a desirable feature since it gives an advantage when one wants to reason about programs (for example, functional programming is traditionally extensional).

In this paper we define the syntax and the semantics of Higher-Order Datalog. Surprisingly, the minimum model property still holds and it can be proved using the same basic tools as in classical (first-order) logic programming (extended to apply to the higher-order setting). Moreover, our construction of the minimum model immediately leads to a bottom-up proof procedure for Higher-Order Datalog (based on an immediate consequence operator, as in ordinary Datalog).

⋆ This research is supported by ΕΠΕΑΕΚ II under the task "ΠΥΘΑΓΟΡΑΣ-II: ΕΝΙΣΧΥΣΗ ΕΡΕΥΝΗΤΙΚΩΝ ΟΜΑΔΩΝ ΣΤΑ ΠΑΝΕΠΙΣΤΗΜΙΑ", Project title: *Applications of Computational Logic to the Semantic Web*, funded by the European Social Fund (75%) and the Greek Ministry of Education (25%).

We conclude the paper by describing an actual implementation of Higher-Order Datalog based on the ideas that are developed in the paper.

2 Higher-Order Logic: Syntax and Semantics

We start with the syntax and the semantics of the higher-order predicate logic (HOPL) underlying Higher-Order Datalog. We first specify the allowable types:

Definition 1. *The set of* expression types*,* tuple types*, and* predicate types *are ranged over by σ, τ and π respectively and are defined as follows:*

$$\begin{aligned}\sigma &::= \iota \mid \pi\\ \tau &::= (\sigma_0, \dots, \sigma_{n-1})\\ \pi &::= o \mid \tau \to o\end{aligned}$$

The vocabulary of HOPL allows for: bound predicate variables of every predicate type (denoted by $\mathsf{Q}, \mathsf{R}, \mathsf{S}, \dots$); free predicate variables of every predicate type (denoted by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \dots$); bound individual variables of ground type ι (denoted by $\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \dots$); constants of type ι and of every predicate type (denoted by $\mathsf{a}, \mathsf{b}, \mathsf{c}, \dots$). We will use V to denote arbitrary bound variables (either predicate or individual ones). We now specify the set of well-typed expressions of HOPL:

Definition 2. *The set of well-typed expressions of HOPL is defined as follows:*

1. *A constant of type σ is an expression of type σ; an individual variable is an expression of type ι; a predicate variable of type π is an expression of type π.*
2. *Let E be an expression of type $(\sigma_0, \dots, \sigma_{n-1}) \to o$ and let $\mathsf{E}_0, \dots, \mathsf{E}_{n-1}$ be expressions of types $\sigma_0, \dots, \sigma_{n-1}$ respectively. Then, $\mathsf{E}(\mathsf{E}_0, \dots, \mathsf{E}_{n-1})$ is an expression of type o.*
3. *If E, E_0 and E_1 are expressions of type o and V is a bound (predicate or individual) variable then $(\neg\mathsf{E})$, $(\mathsf{E}_0 \wedge \mathsf{E}_1)$, $(\mathsf{E}_0 \vee \mathsf{E}_1)$, $(\mathsf{E}_0 \leftarrow \mathsf{E}_1)$, $(\forall\mathsf{V}\,\mathsf{E})$ and $(\exists\mathsf{V}\,\mathsf{E})$ are expressions of type o.*

In the following we write $\mathit{type}(\mathsf{E})$ for the type of E. The semantics of HOPL is similar to that of functional programming: the denotations of (non-ground type) variables and constant symbols of HOPL are *monotonic* relations; moreover, the existential and universal quantifiers always quantify over monotonic relations. Of course, the denotation of negation is not monotonic; however, as we will see, the syntax of the fragment of HOPL that we will consider is guaranteed to have a unique minimum Herbrand model that is monotonic. In the following, it will often be convenient to view relations as functions whose range is the set $\{0, 1\}$. In this way, we can adopt for relations the usual definition of monotonicity:

Definition 3. *Let $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ be partial orders. Then, $f : A \to B$ is termed* monotonic *if for all $x, y \in A$ such that $x \sqsubseteq_A y$, it is $f(x) \sqsubseteq_B f(y)$.*

We write $[A \to B]$ for the set of all monotonic functions from A to B. We now specify the meaning of the types of the *variables* and the (non-logical) *constants* of HOPL:

Definition 4. *The denotation of a type with respect to a given nonempty set U is recursively defined by the function $[\![\cdot]\!]_U$ as follows:*

- $[\![\iota]\!]_U = U$
- $[\![o]\!]_U = \{0, 1\}$
- $[\![(\sigma_0, \ldots, \sigma_{n-1})]\!]_U = [\![\sigma_0]\!]_U \times \cdots \times [\![\sigma_{n-1}]\!]_U$
- $[\![\tau \to o]\!]_U = [[\![\tau]\!]_U \to \{0, 1\}]$

Now, to each set $[\![\sigma]\!]_U$ (respectively $[\![\tau]\!]_U$) of the above definition, we associate a partial order $\sqsubseteq_\sigma$ (respectively $\sqsubseteq_\tau$) as follows:

- If $\sigma = \iota$ then $\sqsubseteq_\iota$ relates every element of U to itself.
- If $\sigma = o$ then $\sqsubseteq_o$ is the numerical ordering on $\{0, 1\}$.
- If $\sigma = \tau \to o$ and f, g in $[\![\sigma]\!]_U$, then $f \sqsubseteq_\sigma g$ if $f(r) \sqsubseteq_o g(r)$ for all r in $[\![\tau]\!]_U$.
- If $\tau = (\sigma_0, \ldots, \sigma_{n-1})$ and $r = (r_0, \ldots, r_{n-1})$ and $r' = (r'_0, \ldots, r'_{n-1})$ with $r, r' \in [\![\tau]\!]_U$, then $r \sqsubseteq_\tau r'$ if $r_i \sqsubseteq_{\sigma_i} r'_i$, for all $0 \leq i \leq n - 1$.

We will often write just $\sqsubseteq$ when the subscript is obvious from context. We can now define the notions of interpretation and state as follows:

Definition 5. *An* interpretation *I of HOPL consists of: a nonempty set U, called the* domain *of I; an assignment to each constant c of type σ of an element c_I of $[\![\sigma]\!]_U$; and, an assignment to each free predicate variable of type π of an element of $[\![\pi]\!]_U$. A* state *over I is a function that assigns to each bound (predicate or individual) variable an element from the appropriate domain (with respect to the type of the variable).*

We can now define the semantics of expressions of HOPL (the cases for $\neg$, $\wedge$, $\vee$ and $\leftarrow$ are straightforward and omitted):

Definition 6. *Let I be an interpretation and s be a state over I. Then, the semantics of expressions of HOPL in a state s over I is defined as follows:*

1. $[\![\mathsf{c}]\!]_s(I) = c_I$
2. $[\![\mathsf{p}]\!]_s(I) = I(\mathsf{p})$, *where* p *is a free predicate variable*
3. $[\![\mathsf{V}]\!]_s(I) = s(\mathsf{V})$, *where* V *is a bound (predicate or individual) variable*
4. $[\![\mathsf{E}(\mathsf{E}_0, \ldots, \mathsf{E}_{n-1})]\!]_s(I) = [\![\mathsf{E}]\!]_s(I)([\![\mathsf{E}_0]\!]_s(I), \ldots, [\![\mathsf{E}_{n-1}]\!]_s(I))$
5. $[\![(\forall \mathsf{V}\, \mathsf{E})]\!]_s(I) = 1$, *if for all* $v \in [\![type(\mathsf{V})]\!]_U$ *it is* $[\![\mathsf{E}]\!]_{(s[\mathsf{V}/v])}(I) = 1$
6. $[\![(\exists \mathsf{V}\, \mathsf{E})]\!]_s(I) = 1$, *if there exists* $v \in [\![type(\mathsf{V})]\!]_U$ *such that* $[\![\mathsf{E}]\!]_{(s[\mathsf{V}/v])}(I) = 1$

An interpretation I is a *model* of a formula F if for all states s over I, $[\![\mathsf{F}]\!]_s(I) = 1$.

3 Higher-Order Datalog

The syntax of Higher-Order Datalog allows higher-order variables to be passed as parameters to predicates. For example, the following is a legitimate program:

```
closure(R,X,Y):-R(X,Y).
closure(R,X,Y):-R(X,Z),closure(R,Z,Y).
```

We now formally define the syntax of the language:

Definition 7. *A* term *is either a variable or a constant. An* atom *is either an expression of the form* $\mathsf{p}(\mathsf{t}_0,\dots,\mathsf{t}_{n-1})$*, or* $\mathsf{P}(\mathsf{t}_0,\dots,\mathsf{t}_{n-1})$*, or* $\mathsf{c}(\mathsf{t}_0,\dots,\mathsf{t}_{n-1})$*, where* $\mathsf{t}_0,\dots,\mathsf{t}_{n-1}$ *are terms. An atom is a* free-predicate *one if it of the first form above. A* literal *is an atom or the negation of an atom. A* positive literal *is an atom. A* negative literal *is the negation of an atom. A* clause C *is a formula:*

$$\forall \mathsf{V}_0 \cdots \forall \mathsf{V}_{n-1}(\mathsf{L}_0 \vee \cdots \vee \mathsf{L}_{k-1})$$

where each L_i *is a literal and* $\mathsf{V}_0,\dots,\mathsf{V}_{n-1}$ *are all the bound (predicate or individual) variables occurring in* $\mathsf{L}_0 \vee \cdots \vee \mathsf{L}_{k-1}$*.*

Definition 8. *A* definite program clause *is a clause that contains precisely one positive free-predicate atom (called the* head *of the clause). A* definite program *(or simply* program*) is a set of definite program clauses.*

As usual, definite program clauses are represented in the form $\mathsf{A} \leftarrow \mathsf{B}_0,\dots,\mathsf{B}_{k-1}$. In order for our language to preserve the properties of classical Datalog, we follow the proposal of [Wad91] and restrict our clauses so as to be *definitional*:

Definition 9. *A clause* C *is called* definitional *iff a) all arguments of predicate type in the head of* C *are bound predicate variables, and b) all these predicate variables are distinct. A program is called definitional iff all its clauses are definitional.*

The semantics of Higher-Order Datalog is based on Herbrand interpretations. The Herbrand universe U_{P} of a program P is the set of all constant symbols of type ι that appear in P. Herbrand interpretations are interpretations that use U_{P} as their underlying universe. The relation $\sqsubseteq$ generalizes in the obvious way to apply to Herbrand interpretations. We assume that HOPL has enough higher-order constant symbols to represent all monotonic relations. Given a higher-order constant c we will write c for the corresponding monotonic relation. Moreover, we assume that all Herbrand interpretations assign the same denotations to the higher-order constant symbols of the language. We can now define the operator T_{P} associated with a given program P:

Definition 10. *Let* P *be a Higher-Order Datalog program and let* I *be a Herbrand interpretation of* P*. Then, we define the operator* T_{P} *as follows:*

$$\begin{aligned} T_{\mathsf{P}}(I)(\mathsf{p}) = \{(c_0,\dots,c_{n-1}) : {} & \mathsf{p}(\mathsf{c}_0,\dots,\mathsf{c}_{n-1}) \leftarrow \mathsf{A}_0,\dots,\mathsf{A}_{k-1} \textit{ is a ground} \\ & \textit{instance of a clause in } \mathsf{P} \textit{ and for every state } s \textit{ and} \\ & \textit{for all } 0 \le i \le k-1,\ [\![\mathsf{A}_i]\!]_s(I) = 1\} \end{aligned}$$

T_{P} *is called the* immediate consequence operator *for* P*.*

Then, it is not hard to demonstrate the following theorem (which generalizes the corresponding theorem for first-order Datalog):

Theorem 1. *For every Higher-Order Datalog program* P*,* T_{P} *has a least (with respect to* $\sqsubseteq$*) fixpoint* $M_{\mathsf{P}} = \mathit{lfp}(T_{\mathsf{P}}) = T_{\mathsf{P}} \uparrow \omega$*. Moreover,* M_{P} *is a Herbrand model of* P*.*

4 Implementation

The ideas that are presented in this paper have been used in order to undertake an implementation of Higher-Order Datalog. The implemented system is actually based on an implementation of the T_{P} operator. The implementation is written in C (about 6500 lines of code), it has been tested with various small programs and is relatively stable. The current version of the system is available from `http://www.di.uoa.gr/∼grad0771`.

There exists however a problem that the bottom-up implementation of Higher-order Datalog has to face. When the order of the program is high and the number of constants is non-negligible, the bottom-up computation becomes impractical. This problem is also apparent in ordinary Datalog if the bottom-up computation is performed in a naive way. However, in our case the problem is more serious since at each step the implementation may have to generate the set of *all* monotonic relations of a given type, and this set may be very big. It remains to be seen if one can develop techniques that can prune the search space.

5 Related and Future Work

The most popular higher-order logic programming systems are λ-Prolog [NM98] and Hilog [CKW93]. Both of these systems are powerful and have found interesting applications. These languages differ from our system in that they are both *intensional* (ie., two predicates are not considered equal unless their names are the same). Both intensional and extensional systems have their corresponding areas of applications and it would be interesting to compare their relative merits.

Work on extensional higher-order logic programming is very limited. The only work that has come to our attention is that of M. Bezem [Bez99], which considers higher-order logic programming with syntax similar to that of [Wad91]. However, the notion of extensionality that is adopted in [Bez99] is different than the classical one and the main ideas are more proof-theory oriented.

There are certain aspects of this work that we would like to further investigate. The next step is to extend the semantics to apply to programs that use "currying" and that allow function symbols. Finally, a more long-term goal is the investigation of the interplay between higher-order constructs and negation.

References

[Bez99] M. Bezem. Extensionality of Simply Typed Logic Programs. In *International Conference on Logic Programming (ICLP)*, pages 395–410, 1999.

[CKW93] W.C. Chen, M. Kifer, and D.S. Warren. HILOG: A Foundation for Higher-Order Logic Programming. *J. of Logic Programming*, 15(3):187–230, 1993.

[NM98] G. Nadathur and D. Miller. Higher-Order Logic Programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logics for Artificial Intelligence and Logic Programming*, pages 499–590, 1998.

[Wad91] W.W. Wadge. Higher-Order Horn Logic Programming. In *Proceedings of the International Symposium on Logic Programming*, pages 289–303, 1991.

Deciding Weak Monadic Second-order Logics using Complex-value Datalog

Gulay Unel and David Toman
{gunel,david}@cs.uwaterloo.ca

Abstract. We use techniques developed for query evaluation of Complex-value Datalog queries for determining satisfiability of WS1S formulæ. We show that the use of these techniques—in particular the *Magic Set* rewriting of Datalog queries and the top-down resolution-based evaluation with memoing—can, in many cases, considerably improve the performance of decision procedures based on the connection between logics and automata, such as the MONA system. While in this paper we focus on the WS1S logic, a similar approach can be used for logics whose decidability can be shown using automata-theoretic techniques.

1 Introduction

Monadic second-order logics provide means to specify regular properties of systems in a succinct way. In addition, these logics are decidable by the virtue of the connection to automata theory [4, 10]. However, only recently tools based on these ideas—in particular the MONA system [8]—have been developed and shown to be efficient enough for many applications [7].

This paper introduces a technique for implementing an automata-based decision procedure for WS1S based on techniques developed for query evaluation in deductive databases, in particular on *Magic Set transformation* [2] and *SLG resolution*, a top-down resolution-based approach augmented with memoing [5, 6]. The main contribution of the paper is showing the connection between the automata-based decision procedures for WS1S and efficient techniques for query evaluation in Complex-value Datalog (Datalog$^{\mathrm{cv}}$).

2 Definitions

First, we define the syntax and semantics of the weak second-order logic of one successor:

Definition 1 (WS1S). *Formulas of WS1S are constructed from atomic formulas $s(x,y)$ and $x \subseteq y$, for x and y second-order variables, with the help of Boolean connectives $\wedge$ and $\neg$ and second-order existential quantifier $\exists x$.*

The semantics of WS1S is defined w.r.t. the set of natural numbers (successors of 0); second-order variables are interpreted as finite sets of natural numbers. The interpretation of the atomic formula $s(x,y)$ is fixed to relating singleton sets $\{n+1\}$ and $\{n\}$, $n \in \mathbf{N}$.[1] Truth and satisfiability of formulas are defined with the help of valuations mapping variables to finite sets of natural numbers in a standard way.

[1] The atomic formula $s(x,y)$ is often written as $x = s(y)$ in literature, emphasizing its nature as a *successor* function.

Definition 2. *A finite automaton is a 5-tuple $A = (N, X, S, T, F)$, where N is the set of states (nodes), X is the alphabet, S is the initial (starting) state, $T \subseteq N \times N \times X$ is the transition relation, and F is the set of final states.*

Given a WS1S formula φ, there is a finite deterministic automaton A_φ that accepts all models of φ. For weak second-order logics, such as the WS1S, finite automata are sufficient [7, 8]. A_φ can be effectively constructed from φ starting from automata for atomic formulæ and using automata-theoretic operations.

Complex Data Model. The complex-value data model is an extension of the standard relational model that allows tuples and finite sets to serve as values in the place of atomic values [1]. The extended data model induces extensions to relational query languages and leads to the definition of *complex-value relational calculus* (calc$^{\text{cv}}$) and a deductive language for complex values, Datalog$^{\text{cv}}$[2, 11].

Definition 3 (Datalog$^{\text{cv}}$). *A* Datalog$^{\text{cv}}$ atom *is a predicate symbol with variables or complex-value constants as arguments.*

A Datalog$^{\text{cv}}$ database (program) *is a finite collection of Horn clauses of the form $h \leftarrow g_1, \ldots, g_k$, where h (called head) is an atom with an optional* grouping specification *and $g_1, \ldots, g_k$ (called goals) are literals (atoms or their negations).*

The grouping is syntactically indicated by enclosing the grouped *argument in a $\langle\cdot\rangle$ constructor; the values then range over the set type of the original argument.*

We require that in every occurrence of an atom the corresponding arguments have the same finite type and that the clauses are stratified with respect to negation.

A Datalog$^{\text{cv}}$ *query is a clause of the form $\leftarrow g_1, \ldots, g_k$. Evaluation of a* Datalog$^{\text{cv}}$ *query (with respect to a* Datalog$^{\text{cv}}$ *database P) determines whether $P \models g_1, \ldots, g_k$.*

Query Evaluation. The naive technique for query evaluation in Datalog$^{\text{cv}}$ is commonly based on a fixed-point construction of the minimal Herbrand model (for Datalog$^{\text{cv}}$ programs with *stratified negation* the model is constructed w.r.t. the stratification) and then testing whether a ground (instance of the) query is contained in the model. The type restrictions guarantee that the fixpoint iteration terminates after finitely many steps. However, whenever a query is known as part of the input, techniques that allow constructing only the relevant parts of the minimal Herbrand model have been developed. Among these, the most prominent are the magic set rewriting (followed by subsequent fixed-point evaluation) [3, 9] and the top-down resolution with memoing—the SLG resolution [12].

3 Automata and Datalog for Complex Values

In this section we present the main contribution of our approach: given a WS1S formula φ we create a Datalog$^{\text{cv}}$ program P_φ representing A_φ such that the answer to a reachability/transitive closure query w.r.t. this program proves satisfiability of φ.

3.1 Representation of Automata

First, we fix the representation for automata that capture models of WS1S formulæ.

Definition 4. *Given a WS1S formula φ with free variables $x_1, \ldots, x_k$ we define a* Datalog$^{\mathrm{cv}}$ *program P_φ which represents the deterministic automaton A_φ:*

1. $\mathsf{Node}_\varphi(n)$ *representing the nodes of A_φ,*
2. $\mathsf{Start}_\varphi(n)$ *representing the starting state,*
3. $\mathsf{Final}_\varphi(n)$ *representing the set of final states, and*
4. $\mathsf{Trans}_\varphi(nf_1, nt_1, \overline{x})$ *representing the transition relation.*

where $\overline{x} = \{x_1, x_2, \ldots, x_k\}$ is the set of free variables of φ; concatenation of their binary valuations represents a letter of A_φ's alphabet X_φ.

3.2 Automata-theoretic Operations

Automata $A_{s(x,y)}$ and $A_{x\subseteq y}$ for atomic formulæ are directly represented by trivial programs $P_{s(x,y)}$ and $P_{x\subseteq y}$ consisting of ground facts. Automata-theoretic operations [8] are then represented as the following programs:

$$\begin{aligned}
P_{\neg\alpha} = P_\alpha \cup \{\ & \mathsf{Node}_{\neg\alpha}(n) \leftarrow \mathsf{Node}_\alpha(n)\\
& \mathsf{Start}_{\neg\alpha}(n) \leftarrow \mathsf{Start}_\alpha(n)\\
& \mathsf{Final}_{\neg\alpha}(n) \leftarrow \mathsf{Node}_\alpha(n), \neg\mathsf{Final}_\alpha(n)\\
& \mathsf{Trans}_{\neg\alpha}(nf_1, nt_1, \overline{x}) \leftarrow \mathsf{Trans}_\alpha(nf_1, nt_1, \overline{x})\ \}
\end{aligned}$$

$$\begin{aligned}
P_{\alpha_1\wedge\alpha_2} = P_{\alpha_1} \cup P_{\alpha_1} \cup \{\ & \mathsf{Node}_{\alpha_1\wedge\alpha_2}([n_1, n_2]) \leftarrow \mathsf{Node}_{\alpha_1}(n_1), \mathsf{Node}_{\alpha_2}(n_2)\\
& \mathsf{Start}_{\alpha_1\wedge\alpha_2}([n_1, n_2]) \leftarrow \mathsf{Start}_{\alpha_1}(n_1), \mathsf{Start}_{\alpha_2}(n_2)\\
& \mathsf{Final}_{\alpha_1\wedge\alpha_2}([n_1, n_2]) \leftarrow \mathsf{Final}_{\alpha_1}(n_1), \mathsf{Final}_{\alpha_2}(n_2)\\
& \mathsf{Trans}_{\alpha_1\wedge\alpha_2}([nf_1, nf_2], [nt_1, nt_2], \overline{x}, \overline{y}, \overline{z}) \leftarrow\\
& \quad \mathsf{Trans}_{\alpha_1}(nf_1, nt_1, \overline{x}, \overline{y}),\ \mathsf{Trans}_{\alpha_2}(nf_2, nt_2, \overline{y}, \overline{z})\ \}
\end{aligned}$$

The sets of variables $\overline{x}, \overline{y}$ represent the free variables of the formula A_{α_1} and $\overline{y}, \overline{z}$ of the formula A_{α_2}.

The projection automaton which represents the existential quantification of a given formula is defined as follows.

$$\begin{aligned}
P^u_{\exists\overline{x}:\alpha} = P_\alpha \cup \{\ & \mathsf{Node}^u_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Node}_\alpha(n)\\
& \mathsf{Start}^u_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Start}_\alpha(n)\\
& \mathsf{Final}^u_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Final}_\alpha(n)\\
& \mathsf{Final}^u_{\exists\overline{x}:\alpha}(n_0) \leftarrow \mathsf{Trans}_\alpha(n_0, n_1, \overline{x}, \overline{o}), \mathsf{Final}^u_{\exists\overline{x}:\alpha}(n_1)\\
& \mathsf{Trans}^u_{\exists\overline{x}:\alpha}(nf_1, nt_1, \overline{y}) \leftarrow \mathsf{Trans}_\alpha(nf_1, nt_1, \overline{x}, \overline{y})\ \}
\end{aligned}$$

The sets of variables $\overline{y}$ and $\overline{x}$ represent the free variables of the formula α, and $\overline{o} = \{0, 0, \ldots, 0\}$ where $|\overline{o}| = |\overline{y}|$.

Note that we need to add the states which have a path to a final state with a string of the form $(\overline{x}, \overline{o})^*$ in the automaton for α as final states. The automaton we get after the projection operation is nondeterministic. The following program creates an equivalent deterministic automaton.

$$\begin{aligned} P_{\exists\overline{x}:\alpha} = P^u_{\exists\overline{x}:\alpha} \cup \{\ & \mathsf{Node}_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Start}_{\exists\overline{x}:\alpha}(n) \\ & \mathsf{Node}_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Node}_{\exists\overline{x}:\alpha}(n_1), \mathsf{Trans}_{\exists\overline{x}:\alpha}(n_1, n, \overline{y}) \\ & \mathsf{Start}_{\exists\overline{x}:\alpha}(\{n\}) \leftarrow \mathsf{Start}^u_{\exists\overline{x}:\alpha}(n) \\ & \mathsf{Final}_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Node}_{\exists\overline{x}:\alpha}(n), \mathsf{Final}^u_{\exists\overline{x}:\alpha}(n_1), n_1 \in n \\ & \mathsf{Trans}_{\exists\overline{x}:\alpha}(n_1, \langle n\rangle, \overline{y}) \leftarrow \mathsf{Node}_{\exists\overline{x}:\alpha}(n_1), \mathsf{Next}_{\exists\overline{x}:\alpha}(n_1, n, \overline{y}) \\ & \mathsf{Next}_{\exists\overline{x}:\alpha}(n_1, n, \overline{y}) \leftarrow n_2 \in n_1, \mathsf{Trans}^u_{\exists\overline{x}:\alpha}(n_2, n, \overline{y})\ \} \end{aligned}$$

The determinization representation presented here is the Datalog$^{\mathrm{cv}}$ version of a well known subset construction algorithm. The nodes of the DFA are created beginning with the starting node and then constructing all nodes reachable from the starting node through the transition relation. To this end, the *grouping* construct of Datalog$^{\mathrm{cv}}$ is used to create all the needed elements of the powerset of the original set of nodes.

Last, the test for emptiness of an automaton has to be defined: To find out whether the language accepted by A_α is non-empty and thus whether α is satisfiable, a *reachability (transitive closure)* query is used.

$$\begin{aligned} TC_\alpha : \{\ & \mathsf{TransClos}_\alpha(n, n) \leftarrow \\ & \mathsf{TransClos}_\alpha(nf_1, nt_1) \leftarrow \mathsf{Trans}_\alpha(nf_1, nt_2, \overline{x}), \mathsf{TransClos}_\alpha(nt_2, nt_1)\ \} \end{aligned}$$

Note that the use of magic sets and/or SLG resolution automatically transforms the transitive closure query into a reachability query. By induction on the structure of φ we have:

Theorem 1. *Let φ be a WS1S formula. Then φ is satisfiable if and only if $P_\varphi, TC_\varphi \models \mathsf{Start}_\varphi(x), \mathsf{Final}_\varphi(y), \mathsf{TransClos}_\varphi(x, y)$.*

4 Experiments

We have compared our technique with the MONA system [7, 8], one of the most efficient tools for reasoning in weak second-order logics (WS1S and WS2S). In contrast to MONA, which constructs the whole automaton for a given formula, our method only constructs the nodes we need to answer the non-emptiness query. For the experiments we use XSB [12], a deductive system that supports top-down resolution with memoing. We also compare our results with CORAL [11], a deductive system that supports Datalog$^{\mathrm{cv}}$ and Magic sets.

The performance results for a set of formulas similar to the ones in T98 satisfiability test suite are given in Figures 1 and 2. The inputs we give to XSB and CORAL are the Datalog$^{\mathrm{cv}}$ programs representing the formulas, and the time required to compute the Datalog$^{\mathrm{cv}}$ representation of a formula is negligible. The response times are measured in seconds; N/A means "Not Answered".

5 Conclusions and Future Work

We have presented a technique that maps satisfiability questions for formulas in WS1S to query answering in Datalog$^{\mathrm{cv}}$ and demonstrated how advanced query evaluation techniques can provide an efficient decision procedure for WS1S.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
MONA	4.37	N/A	13.33	N/A	N/A	0.00	0.00	0.00	0.24	4.06
CORAL	2.42	4.15	4.75	12.09	14.17	N/A	N/A	N/A	N/A	N/A
XSB	0.71	0.00	0.00	0.02	0.01	0.01	0.07	N/A	8.41	N/A

Fig. 1. Performance (secs) w.r.t. increasing number of quantifiers

	#6	#7	#8	#10	#9	#1	#3	#2	#4	#5
MONA	0.00	0.00	0.00	4.06	0.24	4.37	13.33	N/A	N/A	N/A
CORAL	N/A	N/A	N/A	N/A	N/A	2.42	4.75	4.15	12.09	14.17
XSB	0.01	0.07	N/A	N/A	8.41	0.71	0.00	0.00	0.02	0.01

Fig. 2. Performance (secs) w.r.t. increasing number of variables

Extensions of the proposed approach include extending the translation to other types of automata on infinite objects, e.g., to Rabin [10] and Alternating Automata [13], and on improving the upper complexity bounds by restricting the form of Datalog^{cv} programs generated by the translation. In addition we plan to study the impact of goal reordering and other query optimization techniques on the performance of the decision procedure and to develop heuristics for this purpose. We also consider the integration of the technique proposed in this paper with more standard techniques such as BDDs.

References

1. S. Abiteoul and C. Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4(4):727–794, 1995.
2. C. Beeri, S. Naqvi, O. Shmueli, , and S. Tsur. Set construction in a logic database language. *JLP*, 10(3&4):181–232, 1991.
3. C. Beeri and R. Ramakrishnan. On the power of Magic. *JLP*, 10(1/2/3&4):255–299, 1991.
4. J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.
5. W. Chen, T. Swift, and D.S Warren. Efficient top-down computation of queries under the well-founded semantics. *JLP*, 24(3):161–199, 1995.
6. W. Chen and D.S Warren. Query evaluation under the well-founded semantics. In *PODS*, pages 168–179, 1993.
7. J. G. Henriksen, J. L. Jensen, M. E. Jörgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. MONA: Monadic second-order logic in practice. In *Proc. TACAS*, pages 89–110, 1995.
8. N. Klarlund. MONA & FIDO: The logic-automaton connection in practice. In *Computer Science Logic*, pages 311–326, 1997.
9. I. S. Mumick. *Query Optimization in Deductive and Relational Databases.* PhD thesis, Department of Computer Science, Stanford University, 1991.
10. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.
11. R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL deductive system. *VLDB Journal*, 3(2):161–210, 1994.
12. K. F. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. SIGMOD*, pages 442–453, 1994.
13. M. Y. Vardi. Reasoning about the past with two-way automata. In *Proc. ICALP*, pages 628–641, 1998.

A SAT-based Sudoku Solver[⋆]

Tjark Weber

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
webertj@in.tum.de

Abstract. This paper presents a SAT-based *Sudoku* solver. A *Sudoku* is translated into a propositional formula that is satisfiable if and only if the *Sudoku* has a solution. A standard SAT solver can then be applied, and a solution for the *Sudoku* can be read off from the satisfying assignment returned by the SAT solver. No coding was necessary to implement this solver: The translation into propositional logic is provided by a framework for finite model generation available in the Isabelle/HOL theorem prover. Only the constraints on a *Sudoku* solution had to be specified in the prover's logic.

1 Introduction

Sudoku, also known as *Number Place* in the United States, is a placement puzzle. Given a grid – most frequently a 9×9 grid made up of 3×3 subgrids called "regions" – with various digits given in some cells (the "givens"), the aim is to enter a digit from 1 through 9 in each cell of the grid so that each row, column and region contains only one instance of each digit. Fig. 1 shows a *Sudoku* on the left, along with its unique solution on the right [12]. Note that other symbols (e.g. letters, icons) could be used instead of digits, as their arithmetic properties are irrelevant in the context of *Sudoku*. This is currently a rather popular puzzle that is featured in a number of newspapers and puzzle magazines [1, 3, 9].

Several *Sudoku* solvers are available already [6, 10]. Since there are more than $6 \cdot 10^{21}$ possible *Sudoku* grids [5], a naïve backtracking algorithm would be infeasible. *Sudoku* solvers therefore combine backtracking with – sometimes complicated – methods for constraint propagation. In this paper we propose a SAT-based approach: A *Sudoku* is translated into a propositional formula that is satisfiable if and only if the *Sudoku* has a solution. The propositional formula is then presented to a standard SAT solver, and if the SAT solver finds a satisfying assignment, this assignment can readily be transformed into a solution for the original *Sudoku*. The presented translation into SAT is simple, and requires minimal implementation effort since we can make use of an existing framework for finite model generation [11] available in the Isabelle/HOL [8] theorem prover.

⋆ This work was supported by the PhD program Logic in Computer Science of the German Research Foundation.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Fig. 1. *Sudoku* example and solution

2 Implementation in Isabelle/HOL

An implementation of the *Sudoku* rules in the interactive theorem prover Isabelle/HOL is straightforward. Digits are modelled by a datatype with nine elements $1, \ldots, 9$. We say that nine grid cells $x_1, \ldots, x_9$ are *valid* iff they contain every digit.

Definition 1 (valid).

$$\text{valid}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) \equiv \bigwedge_{d=1}^{9} \bigvee_{i=1}^{9} x_i = d.$$

Labeling the 81 cells of a 9×9 grid as shown in Fig. 2, we can now define what it means for them to be a *Sudoku* solution: each row, column and region must be valid.

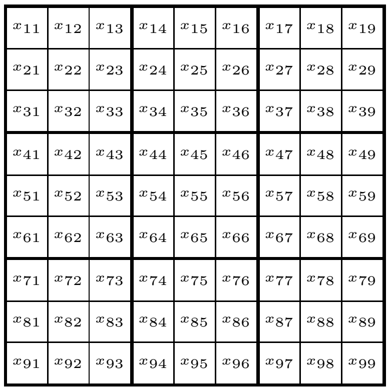
Definition 2 (sudoku).

$$\begin{aligned}\text{sudoku}(\{x_{ij}\}_{i,j \in \{1,\ldots,9\}}) \equiv{} & \bigwedge_{i=1}^{9} \text{valid}(x_{i1}, x_{i2}, x_{i3}, x_{i4}, x_{i5}, x_{i6}, x_{i7}, x_{i8}, x_{i9}) \\ \wedge & \bigwedge_{j=1}^{9} \text{valid}(x_{1j}, x_{2j}, x_{3j}, x_{4j}, x_{5j}, x_{6j}, x_{7j}, x_{8j}, x_{9j}) \\ \wedge & \bigwedge_{i,j \in \{1,4,7\}} \text{valid}(x_{ij}, x_{i(j+1)}, x_{i(j+2)}, x_{(i+1)j}, x_{(i+1)(j+1)}, x_{(i+1)(j+2)}, \\ & \qquad x_{(i+2)j}, x_{(i+2)(j+1)}, x_{(i+2)(j+2)}).\end{aligned}$$

The next section describes the translation of these definitions into propositional logic.

3 Translation to SAT

We encode a *Sudoku* by introducing 9 Boolean variables for each cell of the 9×9 grid, i.e. $9^3 = 729$ variables in total. Each Boolean variable p_{ij}^{d} (with



x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}	x_{29}
x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}	x_{39}
x_{41}	x_{42}	x_{43}	x_{44}	x_{45}	x_{46}	x_{47}	x_{48}	x_{49}
x_{51}	x_{52}	x_{53}	x_{54}	x_{55}	x_{56}	x_{57}	x_{58}	x_{59}
x_{61}	x_{62}	x_{63}	x_{64}	x_{65}	x_{66}	x_{67}	x_{68}	x_{69}
x_{71}	x_{72}	x_{73}	x_{74}	x_{75}	x_{76}	x_{77}	x_{78}	x_{79}
x_{81}	x_{82}	x_{83}	x_{84}	x_{85}	x_{86}	x_{87}	x_{88}	x_{89}
x_{91}	x_{92}	x_{93}	x_{94}	x_{95}	x_{96}	x_{97}	x_{98}	x_{99}

Fig. 2. *Sudoku* grid

$1 \leq i, j, d \leq 9$) represents the truth value of the equation $x_{ij} = d$. A clause

$$\bigvee_{d=1}^{9} p_{ij}^{d}$$

ensures that the cell x_{ij} denotes one of the nine digits, and 36 clauses

$$\bigwedge_{1 \leq d < d' \leq 9} \neg p_{ij}^{d} \vee \neg p_{ij}^{d'}$$

make sure that the cell does not denote two different digits at the same time.

Since there are just as many digits as cells in each row, column, and region, Def. 1 is equivalent to the following characterization of validity, stating that the nine grid cells $x_1, \ldots, x_9$ contain distinct values.

Lemma 1 (Equivalent characterization of validity).

$$\text{valid}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) \iff \bigwedge_{1 \leq i < j \leq 9} x_i \neq x_j$$

$$\iff \bigwedge_{1 \leq i < j \leq 9} \bigwedge_{d=1}^{9} x_i \neq d \vee x_j \neq d.$$

The latter characterization turns out to be much more efficient when translated to SAT. While Def. 1, when translated directly, produces 9 clauses with 9 literals each (one literal for each equation), the formula given in Lemma 1 is translated into 324 clauses (9 clauses for each of the 36 inequations $x_i \neq x_j$), but each clause of length 2 only. This allows for more unit propagation [14] at the Boolean level, which – in terms of the original *Sudoku* – corresponds to cross-hatching [12] of digits, a technique that is essential to reduce the search space.

The 9 clauses obtained from a direct translation of Def. 1 could still be used as well; unit propagation on these clauses would correspond to counting the digits 1 – 9 in regions, rows, and columns to identify missing numbers. However, in our experiments we did not experience any speedup by including these clauses.

This gives us a total of 11745 clauses: 81 definedness clauses of length 9, $81 \cdot 36$ uniqueness clauses of length 2, and $27 \cdot 324$ validity clauses,[1] again of length 2. However, we do not need to introduce Boolean variables for cells whose value is given in the original *Sudoku*, and we can omit definedness and uniqueness clauses for these cells as well as some of the validity clauses – therefore the total number of variables and clauses used in the encoding of a *Sudoku* with givens will be less than 729 and 11745, respectively.

Note that our encoding already yields a propositional formula in conjunctive normal form (CNF). Therefore conversion into DIMACS CNF format [4] – the standard input format used by most SAT solvers – is trivial. Isabelle can search for a satisfying assignment using either an internal DPLL-based [2] SAT solver, or write the formula to a file in DIMACS format and execute an external solver. We have employed zChaff [7] to find the solution to various *Sudoku* classified as "hard" by their respective authors (see Fig. 3 for an example), and in every case the runtime was only a few milliseconds.

	2							
			6					3
	7	4		8				
					3			2
	8			4			1	
6			5					
				1		7	8	
5					9			
							4	

1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

Fig. 3. hard *Sudoku* example and solution

4 Concluding Remarks

We have presented a straightforward translation of a *Sudoku* into a propositional formula. The translation can easily be generalized from 9×9 grids to grids of arbitrary dimension. It is polynomial in the size of the grid, and since *Sudoku* is NP-complete [13], no algorithm with better complexity is known. The translation, combined with a state-of-the-art SAT solver, is also practically successful: 9×9 *Sudoku* puzzles are solved within milliseconds.

Traditionally the givens in a *Sudoku* are chosen so that the puzzle's solution is unique; nevertheless our algorithm can be extended to enumerate all possi-

[1] This number includes some duplicates, caused by the overlap between rows/columns and regions: certain cells that must be distinct because they belong to the same row (or column) must also be distinct because they belong to the same region.

ble solutions (by explicitly disallowing all solutions found so far, and perhaps using an incremental SAT solver that allows adding clauses on-the-fly to avoid searching through the same search space multiple times).

Particularly remarkable is the fact that our solver, while it can certainly compete with hand-crafted *Sudoku* solvers, some of which use rather complex patterns and search heuristics, required very little implementation effort. Aside from Lemma 1, no domain-specific knowledge was used. The impressive performance is largely due to the SAT solver. Even the translation into propositional logic was not written by hand, but is an instance of a framework for finite model generation that is readily available in the Isabelle/HOL theorem prover. Only the *Sudoku* rules had to be defined in the prover, and this was a trouble-free task.

References

[1] Col Allan, editor. *New York Post.* News Corporation, New York City, NY, USA, 2005.

[2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[3] Giovanni di Lorenzo, editor. *Die Zeit.* Zeitverlag Gerd Bucerius GmbH & Co. KG, Hamburg, Germany, 2005.

[4] DIMACS satisfiability suggested format, 1993. Available online at `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc`.

[5] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible Sudoku grids, June 2005. Available online at `http://www.shef.ac.uk/~pm1afj/sudoku/`.

[6] DeadMan's Handle Ltd. Sudoku solver, September 2005. Available online at `http://www.sudoku-solver.com/`.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, Las Vegas, June 2001.

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[9] Robert James Thomson, editor. *The Times.* Times Newspapers Ltd., London, UK, 2005.

[10] Pete Wake. Sudoku solver by logic, September 2005. Available online at `http://www.sudokusolver.co.uk/`.

[11] Tjark Weber. Bounded model generation for Isabelle/HOL. In Wolfgang Ahrendt, Peter Baumgartner, Hans de Nivelle, Silvio Ranise, and Cesare Tinelli, editors, *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 2004)*, volume 125(3) of *Electronic Notes in Theoretical Computer Science*, pages 103–116. Elsevier, July 2005.

[12] Wikipedia. Sudoku – Wikipedia, the free encyclopedia, September 2005. Available online at `http://en.wikipedia.org/wiki/Sudoku`.

[13] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. In *IPSJ SIG Notes 2002-AL-87-2*. IPSJ, 2002.

[14] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *Proceedings of the 8th International Conference on Computer Aided Deduction (CADE 2002)*, volume 2392 of *Lecture Notes in Computer Science*. Springer, 2002.

Matchmaking and Personalization on Mobile Devices

Thomas Kleemann and Alex Sinner

University of Koblenz-Landau, Department of Computer Science
{tomkl|sinner}@uni-koblenz.de

Abstract. We describe in this paper how to perform description logic reasoning on mobile devices for personalization in a semantic mobile environment. In such an environment, semantic messages are sent to nearby mobile users. The mobile devices manage a semantic user profile, which is used to filter out unwanted messages. The semantics of the messages and profile is formalized using description logic. A matchmaking process is performed on the mobile device by a first order reasoner, Pocket KRHyper. The suitable definition of profiles and annotations and the translation from description logic to first order logic including general TBoxes is also described.[1]

1 Introduction

During the past few years mobile phones have become a ubiquitous companion. They were up to now mainly used for messaging and phone calls, but they have become powerful devices with much more potential. Our goal is to show how to use this potential for performing description logic (DL) based matchmaking for personalization on mobile devices in what we call a 'Semantic Mobile Environment'. In such an environment, so-called service nodes are installed at chosen points of interest. These nodes broadcast semantically annotated messages to nearby mobile users using, for example, bluetooth wireless technology.

On the user's mobile device, we manage a *Semantic User Profile* describing the users interests and disinterests in a description logic formalization (see Sec. 2.2). This user profile is used to filter out unwanted messages by performing matchmaking between the messages' annotation and the user profile.

In the following, we describe the formalization of the annotations of messages and the user profiles. Then we propose a matchmaking algorithm that requires, independently of the profile size, only two DL satisfiability tests. Finally, we present how this can be performed efficiently on mobile devices using the Pocket KRHyper mobile reasoner.

2 Semantic Personalization

We are using a description logic of expressiveness $\mathcal{SHI}$ to describe semantic annotations and user profiles. In the following, U is the universal role, $\top$ the top concept, and $\bot$ the bottom concept. The syntax and semantics of $\mathcal{SHI}$ are amply described in [1].

[1] This work is part of the IASON project and is funded by the "Stiftung Rheinland-Pfalz für Innovation".

2.1 Message Annotation

Every message in our environment consists of some human-readable content and an annotation describing the semantics of the respective message. We distinguish between an ontology $\mathcal{T}_W$ for describing the concepts in the world and a service ontology $\mathcal{T}_S$ (see Fig. 1) that makes it possible to distinguish between producers and consumers of goods and services. Additionally, social interests lacking a producer-consumer separation are possible. $\mathcal{T}_S$ is intentionally kept simple for clarity's sake, but could easily be extended.

$$\begin{aligned} \mathit{share} &\sqsubseteq U \\ \mathit{offer} &\sqsubseteq \mathit{share} \\ \mathit{request} &\sqsubseteq \mathit{share} \end{aligned}$$

Fig. 1. The Service Ontology $\mathcal{T}_S$

In general, annotations are of the form

$$\mathit{annotation} \equiv \exists R.C$$

where $R \sqsubseteq \mathit{share}$, $\mathcal{T}_W \models C$, and $\mathcal{T}_W \cup \mathcal{T}_S \models \mathit{annotation}$. To describe, for example, an offer of a movie of genre SciFi on a DVD, we would use the following annotation:

$$\exists \mathit{offer}.(\mathit{Movie} \sqcap \exists \mathit{hasGenre}.\mathit{SciFi} \sqcap \exists \mathit{hasMedium}.\mathit{DVD})$$

2.2 User Profile

On the receiving end for semantic messages, we have a mobile device which manages a semantic user profile. This user profile stores the positively marked interests and the disinterests determined by rejected topics. The interests and disinterests in the profile are defined as follows:

$$\mathit{profile} \equiv \mathit{interests} \sqcap \overline{\mathit{disinterests}} \tag{1}$$

$$\mathit{interests} \equiv \bigsqcup_{i=1}^{n} \mathit{positive}_i \tag{2}$$

$$\mathit{positive}_i \equiv \exists R_i.C_i \tag{3}$$

$$\mathit{disinterests} \equiv \bigsqcup_{i=1}^{m} \mathit{negative}_i \tag{4}$$

$$\mathit{negative}_i \equiv \exists R_i.C_i \tag{5}$$

$$R_i \sqsubseteq \mathit{share} \tag{6}$$

From the definition, we see that the interests and disinterests are a collection of DL concepts that follow the definition of the annotations.

The user's interests and disinterests are updated from his/her responses to messages, and allow for the use of ontological knowledge to generalize the annotation. The procedure of these updates is beyond the scope of this paper.

2.3 Matchmaking

The main motivation behind the DL formalization was to be able to compare the semantic annotation of incoming messages with the interests and disinterests stored in the semantic user profile. This is what we call matchmaking.

The decision whether a message matches a users interests is based on concept satisfiability and subsumption of the DL in use. Because the mobile clients provide limited computing power, the decision of a match is performed with only two queries.

$$\mathit{profile} \sqcap \mathit{annotation} \not\equiv \bot \tag{7}$$

$$\mathit{annotation} \sqsubseteq \mathit{profile} \tag{8}$$

The first test (7) checks whether the annotation is *compatible* with the profile. If the test fails, the annotated message is considered a *mismatch*. Otherwise, we perform the second test (8) to get a better *match degree*. If it is satisfied, the annotated message is called a *match*.

3 Mobile Reasoning

To perform the reasoning necessary for the matchmaking task, we are using the first order logic (FOL) theorem prover library Pocket KRHyper [9] for Java 2 Mobile Edition (J2ME[2]) enabled mobile devices. The reasoning engine is based on the hyper tableau calculus [3] and can be considered as a resource optimized version of the KRHyper [11] system. Pocket KRHyper is currently the only FOL automated theorem prover running on mobile devices that can tackle useful problems.

Since our matchmaking algorithm requires DL TBox reasoning, we use a transformation from description logic to FOL. The transformation into sets of clauses introduces subconcepts to reduce the complexity of the concept expression or axiom. An axiom $\exists R.C \sqsubseteq \forall S.\forall T.D$ is split into $\exists R.C \sqsubseteq sub_i$ and $sub_i \sqsubseteq \forall S.sub_j$ and $sub_j \sqsubseteq \forall T.D$ to comply with the transformation primitives. Table 1 gives the transformation primitives in abstract DL syntax, a corresponding first order formula and the generated clauses.

The clauses marked with * are not range restricted, which may cause the reasoning algorithm to consider all ground instances of the disjunctive head. Since this is very time-consuming, such constructs are better avoided in the modeling. Decidability commonly requires the tableau procedure to engage a blocking technique to avoid the infinite generation of role successors in clause **. This is neccessary for cyclical terminologies where $D \sqsubseteq C$ or D transitively has role successors in C. The blocking techniques found in [6] may be adapted to the transformation as shown in [2].

The effective test for satisfiability of a concept C inserts a single fact $C(a)$ into the knowledge base. C is satisfiable if the Pocket KRHyper finds a model for the knowledge base. A refutation indicates the unsatisfiability of the concept. Subsumption of concepts is reduced to satisfiability.

4 Resource Management and Evaluation

In order to reduce the workload introduced by the translation of DL axioms into clausal form, the knowledge base is split into three parts. The ontology is assumed to be stable

[2] `http://java.sun.com/j2me`

description logic	first order formula	clauses
$C \sqcap D \sqsubseteq E$	$\forall x.C(x) \wedge D(x) \rightarrow E(x)$	e(x) :- c(x), d(x).
$C \sqcup D \sqsubseteq E$	$\forall x.C(x) \vee D(x) \rightarrow E(x)$	e(x) :- c(x).
		e(x) :- d(x).
$C \sqsubseteq \neg D$	$\forall x.C(x) \rightarrow \neg D(x)$	false :- c(x), d(x).
$\exists R.C \sqsubseteq D$	$\forall x \forall y.R(x,y) \wedge C(y) \rightarrow D(x)$	d(x) :- c(y), r(x,y).
$\forall R.C \sqsubseteq D$	$\forall x.(\forall y.R(x,y) \rightarrow C(y)) \rightarrow D(x)$	d(x); r(x, f_{R-C}(x)). *
		d(x) :- c(f_{R-C}(x)).
$C \sqsubseteq D \sqcap E$	$\forall x.C(x) \rightarrow D(x) \wedge E(x)$	e(x) :- c(x).
		d(x) :- c(x).
$C \sqsubseteq D \sqcup E$	$\forall x.C(x) \rightarrow D(x) \vee E(x)$	e(x); d(x) :- c(x).
$\neg C \sqsubseteq D$	$\forall x.\neg C(x) \rightarrow D(x)$	c(x); d(x). *
$C \sqsubseteq \exists R.D$	$\forall x.C(x) \rightarrow (\exists y.R(x,y) \wedge D(y))$	d($f_{R-D}(x)$) :- c(x).
		r(x, $f_{R-D}(x)$) :- c(x). **
$C \sqsubseteq \forall R.D$	$\forall x.C(x) \rightarrow (\forall y.R(x,y) \rightarrow D(y))$	d(y) :- c(x), r(x,y).
$R \sqsubseteq S$	$\forall x \forall y.R(x,y) \rightarrow S(x,y)$	s(x,y) :- r(x,y)
$R^{-} \equiv S$	$\forall x \forall y.R(x,y) \leftrightarrow S(y,x)$	s(y,x) :- r(x,y).
		r(x,y) :- s(y,x).
R^{+}	$\forall x \forall y \forall z.R(x,y) \wedge R(y,z) \rightarrow R(x,z)$	r(x,z) :- r(x,y), r(y,z).

Table 1. Translation Primitives

throughout the runtime of the application and is therefore transformed only once. The profile is transformed only when it is updated. Solely the annotations of messages and the associated queries are transformed as they emerge. The knowledge base keeps track of these parts by setting a mark at each such step, so it is possible to revert it to the state it had before an update or query.

With respect to our matchmaking algorithm, the simplifications in Table 2 yield the same results, but reduce the number of clauses. The effect is a reduced memory consumption of the knowledge base itself and the hyper tableau generated for the tests. To test the satisfiability of *annotation*$\sqcap$*profile*, it is sufficient to consider only the *negative*$_i$ part of the profile, if translated according to Table 2. An annotation will fail this test if it is subsumed by the *disinterests*.

$$positive_i \sqsubseteq profile \qquad i \in \{1,...,n\} \tag{9}$$
$$negative_i \sqsubseteq \bot \qquad i \in \{1,...,m\} \tag{10}$$
$$annotation \sqsubseteq \exists R.C \tag{11}$$

Table 2. Resource Optimized Transformation of Profile and Annotation

A performance evaluation of Pocket KRHyper with respect to a selection of TPTP [10] problems can be found in [9]. First empirical tests of our matchmaking approach with ontologies for three different real world scenarios (public transport, conference program, and cinema program) performed well on contemporary phones like e.g. Nokia 6681. Matchmaking decisions took less than one second per incoming message.

5 Conclusions and Related Work

In this paper, we have described how to perform matchmaking on mobile devices for personalization in a semantic mobile environment using the Pocket KRHyper mobile theorem prover and a corresponding DL transformation. To our knowledge, the use of automated reasoning on mobile devices for personalization in a semantic mobile environment is new. A propositional tableaux-based theorem prover for JavaCard, CardTAP [4], has been implemented before. But unlike Pocket KRHyper, this was considered a toy system and was too slow for real applications.

Unlike other matchmaking approaches [7,8], we use at most two queries to calculate the match degree for arbitrary profiles. A large profile containing many interests and disinterests does not impose more tests. Thus our approach scales well even in the context of limited resources.

The capabilities of the presented DL transformation extend those of the Description Logic Programs (DLP) [5] subset of DL and Logic Programs. Despite these extensions, reasoning is still empirically successful, even on mobile devices.

References

1. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003.
2. P. Baumgartner, U. Furbach, M. Gross-Hardt, and T. Kleemann. Model based deduction for database schema reasoning. In *KI 2004: Advances in Artificial Intelligence*, volume 3238, pages 168–182. Springer Verlag, 2004.
3. P. Baumgartner, U. Furbach, and I. Niemelä. Hyper Tableaux. Technical Report 8–96, Universität Koblenz-Landau, 1996.
4. R. Goré, J. Posegga, A. Slater, and H. Vogt. System description: cardtap: The first theorem prover on a smart card. In *CADE-15: Proc. of the 15th International Conference on Automated Deduction*, pages 47–50. Springer-Verlag, 1998.
5. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proc. of the 12th International World Wide Web Conference (WWW 2003)*, pages 48–57. ACM, 2003.
6. V. Haarslev and R. Möller. Expressive ABox Reasoning with Number Restrictions, Role Hierarchies, and Transitively Closed Roles. In *KR2000: Principles of Knowledge Representation and Reasoning*, pages 273–284. Morgan Kaufmann, 2000.
7. L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proc. of the 12th International World Wide Web Conference (WWW'2003)*, pages 331–339. ACM, 2003.
8. T. D. Noia, E. D. Sciascio, F. M. Donini, and M. Mongiello. Abductive matchmaking using description logics. In *Proc. of IJCAI'03*, pages 337–342, 2003.
9. A. Sinner and T. Kleemann. Krhyper - in your pocket, system description. In *Proc. of Conference on Automated Deduction, CADE-20*, 2005.
10. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
11. C. Wernhard. System Description: KRHyper. Fachberichte Informatik 14–2003, Universität Koblenz-Landau, 2003.

A Sequent Calculus for a First-order Dynamic Logic with Trace Modalities for Promela$^+$

Florian Rabe[1], Steffen Schlager[2], and Peter H. Schmitt[2]

[1] International University Bremen
[2] Universität Karlsruhe

Abstract. In this paper we introduce the first-order dynamic logic *DLP* for Promela$^+$, a language subsuming the modelling language Promela of the Spin model checker. In *DLP* trace modalities can be used to reason about the temporal properties of programs. The definition of *DLP* includes a formal semantics of the Promela$^+$ language. A sound and relatively complete sequent calculus is given, which allows deductive theorem proving for Promela$^+$. In contrast to the Spin model checker for Promela, this calculus allows to verify infinite state models. To demonstrate the usefulness of our approach we present two examples that cannot be handled with Spin but that can be derived in our calculus.

1 *DLP*—Dynamic Logic for Promela$^+$

Dynamic Logic (DL) [4, 5] is an extension of first-order predicate logic with modalities $[\pi]F$ and $\langle\pi\rangle F$ for each program π of some programming language and DL formula F. DL allows to reason about the input/output behaviour of a program. However, sometimes it is desirable to reason about intermediate states of a program as well. This becomes possible if DL is extended with additional, so-called trace modalities, as shown in [1].

The programming language we consider in this paper is Promela$^+$ whose syntax is essentially the same as of Promela [7], the modelling language of the model checker Spin [6]. Besides the usual constructs like assignments, loops, etc. Promela offers dynamic process creation, synchronous and asynchronous communication through channels, and non-deterministic choice.

In contrast to Promela, Promela$^+$ is not restricted to finite models. E.g., it is possible to create an unbounded number of processes, integer variables are not range restricted, and, most important, the initial state of a system may be (partially) unknown. For an informal syntax and semantics of Promela programs we refer to [7, 6]. A detailed formal syntax and semantics Promela$^+$ and therefore of Promela and can be found in [8].

2 Promela$^+$ and Limited Programs

In this section we give a very brief overview over the semantics of Promela$^+$ as described in [8]. We also introduce limited programs, which are an extension of [8].

Syntax. Promela$^+$ is based on a many-sorted first-order logic over a fixed signature containing all symbols to define syntax and semantics of Promela. It is extended with six temporal operators for each program π: $[\pi]$, $\langle\pi\rangle$, $[[\pi]]$, $[\langle\pi\rangle]$, $\langle[\pi]\rangle$ and $\langle\langle\pi\rangle\rangle$. Promela variables are modelled as function symbols and are different from the logical variables.

The normal programs, which we call unlimited, are essentially the Promela programs. In addition we introduce *limited programs* which are defined by the following clause: If t is a term of sort integer and π is an unlimited program, then π^t is a limited program. Limited or unlimited programs may occur inside temporal operators, e.g., for a formula F both $[\pi]F$ and $[\pi^t]F$ are well-formed formulas.

Semantics. The semantic domain of DL are extended Kripke structures. The states are many-sorted first-order structures such that

- all states share the same universe,
- some function and predicate symbols are *rigid*, i.e. have the same interpretation in all states (e.g. arithmetics and list operation), and
- the interpretation of *non-rigid* function symbols is depending on the state encoding the values of the Promela variables and channels.

For each elementary command there is one transition relation reflecting the semantics of the programming language. A trace of a program π for an initial state s is the (possibly infinite) sequence of states of a possible run of π starting in s. If a trace is finite its last state is labelled with *timeout* or *termination.* Promela$^+$ is non-deterministic, i.e., the semantics of a program is a set of traces for each initial state. Where applicable our semantics of Promela$^+$ agrees with the semantics of Promela used by Spin.

For a trace t and a natural number n let t^n denote the initial segment of length n of t; and for a set of traces T let T^n denote the set of all such segments of traces in T. For a given state s and an assignment α to the free logical variables let val_s^α denote the interpretation function. Then the semantics of the limited program π^t is given by:

$$\mathrm{val}_s^\alpha(\pi^t) = \begin{cases} \mathrm{val}_s^\alpha(\pi)^n & \text{if } \mathrm{val}_s^\alpha(t) = n, n > 0 \\ \emptyset & \text{if } \mathrm{val}_s^\alpha(t) \leq 0 \end{cases}$$

That means that for a limited program only the first $n - 1$ computation steps are relevant.

The formula $[\pi]F$ ($\langle\pi\rangle F$) holds in state s iff *for all* (there *exists* a) possible end state(s) of π labelled with *termination* when started in s the formula F holds. The formula $[[\pi]]F$ ($[\langle\pi\rangle]F$) holds iff *for all* (there *exists* a) state(s) on *all* traces of π the formula F holds. And the formula $\langle[\pi]\rangle F$ ($\langle\langle\pi\rangle\rangle F$) holds iff there exists a trace of π on which *for all* (there *exists* a) state(s) F holds. If the program in the modal operator is a limited program the same semantics applies with respect to the traces in the interpretation of π^t. Formulas not containing modalities are interpreted as usual in first-order logic.

E.g., suppose $\mathrm{val}^{\alpha}_{s_1}(\pi) = \{(s_1, s_2, \mathit{termination}\ s_3),\ (s_1, s_2, s'_3, \mathit{timeout}\ s'_4)\}$ and F holds precisely in the states s_1, s_2, s_3 and s'_3. Then $\mathrm{val}^{\alpha}_{s_1}(\langle[\pi]\rangle F) = 1$ because F holds in all states of the first trace; $\mathrm{val}^{\alpha}_{s_1}([[\pi]]F) = 0$ because F does not hold in s'_4; $\mathrm{val}^{\alpha}_{s_1}([[\pi^3]]F) = 1$ because now only the first three states of each trace are considered; and $\mathrm{val}^{\alpha}_{s_1}([\pi^2]F) = 1$ because there is no final state within the first two states of a trace, and the universal quantification over the empty set is trivially true.

3 A Sequent Calculus for *DLP*

We use a sequent calculus to axiomatise *DLP*. As usual, the semantics of a rule is that the validity of the premisses above the line implies the validity of the conclusion. A full account (except for limited programs) is given in [8]. The core of the calculus are the rules for symbolic execution of Promela$^+$ code.

The idea of the rules for symbolic execution is that first scheduling rules introduce tags. The semantics of a tagged program $i : \pi$ is the same as of π with the restriction that the next command to be executed is from the i-th process of π. Thus scheduling rules can be seen as case distinctions that have one case for every possible scheduling decision.

Secondly, unwinding rules transform the program $i : \pi$ by replacing composed commands (selections, loops and sequences marked as atomic) with elementary ones. This is necessary since only elementary commands have a well-defined effect on the global state (due to the non-deterministic interleaving of processes). Both scheduling and unwinding lead to branches or alternatives in the proof tree reflecting the indeterminism of Promela.

Thirdly, an elementary command is executed. A typical example rule is:

$$\frac{\mathit{ex}(c) \vdash F \qquad \mathit{ex}(c)\ \vdash\ \mathit{eff}(c)[[\mathit{remProg}(\pi, i)]]F}{\vdash\ [[i : \pi]]F}$$

The purpose of this rule is to execute the first command c of the i-th process of π.

While tags are syntactic entities of *DLP*, *ex*, *eff*, and *remProg* are meta level abbreviations that allow to state several rules in one compact rule scheme: $\mathit{ex}(c)$ is a first-order formula that expresses the executability of c, $\mathit{eff}(c)$ expresses the state transitions caused by the execution of c and modifies the state in which the following formula is to be evaluated, and $\mathit{remProg}(\pi, i)$ is the program that remains to be executed after c has been executed. These abbreviations must be defined for all elementary commands c. E.g., if c spawns a new process, $\mathit{remProg}(\pi, i)$ removes c from π and adds the new process instance to π.

Having the intuitive meaning of these functions in mind it is easy to understand the rule: If c is executable, the formula $[[i : \pi]]F$, which states that F holds in all states on all traces, is reduced to F, which states that F holds in the current state, and $\mathit{eff}(c)[[\mathit{remProg}(\pi, i)]]F$, which states that F holds in all states on all traces that arise if the remaining program is executed in the next state (characterised by the state updates *eff*(c)). If c is not executable, the proof goal is closed immediately.

Note that the execution rule eliminates the tag and the scheduling rules are applicable again. This cycle is repeated until the program is completely executed and can be discharged.

Rules for Limited Programs. In order to derive formulas with non-terminating programs we apply induction rules, usually on the length of the traces. However, this is only possible if the formula contains a term that encodes this length. Therefore, limited programs are introduced by the rules given in Fig. 1. Here the rules in the right column are not necessary for completeness and are only given for symmetry. After introduction of the limited programs induction on t can be applied.

$$\begin{array}{c|c}
M \in \{[], [[]], [\langle\rangle]\} & M \in \{\langle\rangle, \langle[]\rangle, \langle\langle\rangle\rangle\} \\
\hline
\dfrac{\vdash\ \forall t : int.\ t \geq 1 \rightarrow M(\pi^t)F}{\vdash\ M(\pi)F} & \dfrac{\vdash\ \exists t : int.\ t \geq 1 \wedge M(\pi^t)F}{\vdash\ M(\pi)F}
\end{array}$$

Fig. 1. Introducing limited programs

Formulas that contain limited programs are treated by similar symbolic execution rules as those with unlimited programs. In particular the scheduling and unwinding rules are the same. For command execution one detail is changed: The limit is decremented because one program step has been executed. For example the above execution rule corresponds to

$$\frac{t \geq 2\ ,\ ex(c)\ \vdash\ F \qquad t \geq 2\ ,\ ex(c)\ \vdash\ \mathit{eff}(c)[[\mathit{remProg}(\pi, i)^{t-1}]]F}{t \geq 2\ \vdash\ [[i : \pi^t]]F}$$

The rules for discharging limited programs when the limit is reached, i.e., $t = 1$, and for the degenerate cases where $t \leq 0$ are given in Fig. 2. If $t = 1$ a special case must be distinguished: If the program is non-empty, i.e. $\pi \neq \epsilon$, it has not terminated yet and no final state exists, which means that $[\pi^t]F$ is always true and $\langle\pi^t\rangle F$ is always false. If $t \leq 0$, formulas that contain operators that quantify universally over the traces are always true.

3.1 Soundness and Relative Completeness

Soundness has to be shown separately for each rule. Most of the proofs are technical, but not difficult. Except for the rules for limited programs they can be found in [8].

The soundness of the rules for limited programs either follows directly from the definition of the semantics or from the soundness of the corresponding rule for unlimited programs—except for the introduction rules for limited programs for the operators $[\langle\rangle]$ and $\langle[]\rangle$. For the soundness of these rules the following theorem.

$\pi \neq \epsilon,\, M = []$	$\pi \neq \epsilon,\, M = \langle\rangle$	otherwise
$\dfrac{}{t = 1 \,\vdash\, M(\pi^t)F}$	$\dfrac{t = 1 \,\vdash}{t = 1 \,\vdash\, M(\pi^t)F}$	$\dfrac{t = 1 \,\vdash\, F}{t = 1 \,\vdash\, M(\pi^t)F}$

$M \in \{[], [[]], [\langle\rangle]\}$	$M \in \{\langle\rangle, \langle[]\rangle, \langle\langle\rangle\rangle\}$
$\dfrac{}{t \leq 0 \,\vdash\, M(\pi^t)F}$	$\dfrac{t \leq 0 \,\vdash}{t \leq 0 \,\vdash\, M(\pi^t)F}$

Fig. 2. Discharging limited programs

Theorem 1. *For all unlimited programs π, all formulas F, all states s, and all assignments α:*

$$\mathrm{val}_s^\alpha(M(\pi)F) = \left\{ \begin{array}{c} \inf\limits_{n\in\mathbb{N}^*} \\ \sup\limits_{n\in\mathbb{N}^*} \end{array} \right\} \mathrm{val}_s^\alpha(M(\pi^n)F) \left\{ \begin{array}{l} \text{if } M = \langle[]\rangle \\ \text{if } M = [\langle\rangle] \end{array} \right\}$$

This theorem states that it is sufficient for the evaluation of formulas containing the modalities $\langle[]\rangle$ and $[\langle\rangle]$ to consider only finite prefixes of the traces. The proof is based on quite involved an induction and can be found in [8]. It is essential for this theorem that the number of indeterministic choices in Promela is always finite. That guarantees that the sets $\mathrm{val}_s^\alpha(\pi^n)$ are always finite whereas the set $\mathrm{val}_s^\alpha(\pi)$ may even be uncountable.

The idea of the proof of relative completeness is to discharge all programs by symbolic execution if possible and to introduce limited programs and use induction for non-terminating programs. Then the remaining first-order formulas can be handled by standard methods. Relative completeness means that all valid formulas could be derived in the calculus if an oracle for arithmetic was available, i.e., a rule scheme providing all valid arithmetic formulas as axioms. Of course, in reality such a rule cannot exist but this is not harmful to "practical completeness". Rule sets for arithmetic are available, which—as experience shows—allow to derive all valid first-order formulas that occur during the verification of actual programs. Moreover, many arithmetic formulas can be automatically discharged by external decision procedures like CVC [9] and the Simplify tool, which is part of ESC/Java [3].

It is possible to give further operators, e.g., $\mathit{Inf}(\pi)F$ stating that F holds infinitely many times along every trace of π, for which the above theorem does not hold. Therefore the described technique cannot be extended to give a sound and relatively complete axiomatisation for these operators.

3.2 Examples

We now present two examples. Although they are extremely simple they cannot be verified using the model checker Spin, whereas their deductive verification can be done in a standard way. This shows the fundamental advantages of the deductive approach.

For these examples, note that in Promela **do**...**od** denotes a guarded non-deterministic choice, that is repeated until a break is encountered. Consider the program π defined as

```
do
  :: skip
  :: x=0; break
od
```

We now want to verify that $x \doteq 0$ holds in all possible final states of π. In *DLP* this property is expressed as $[\pi]x \doteq 0$. Spin cannot handle this because infinitely many and arbitrarily long runs exist for this model. However, using limited programs and induction on t in π^t this model can be easily verified deductively. The induction hypothesis is $\forall t : int.\ t \geq 1 \rightarrow [\pi^t]x \doteq 0$.

For the second example let π be defined as

```
do
  :: x != 0; x=x-1
  :: else; break
od
```

where x is decreased as long as it is non-zero, and if x is zero, the loop terminates. We want to prove validity of the formula $\forall x : int.x \geq 0 \rightarrow [\pi]x \doteq 0$ expressing that for every initial value of x greater than 0 and all terminating runs $x \doteq 0$ holds in the final state. This property cannot be verified with Spin since it does not allow arbitrary initial states. However, Spin can easily verify a similar property with a fixed value for x.

While arbitrary initial states are not provided for in Promela they naturally occur in many realistic scenarios, e.g., if a program is started in the final state of another program or if it depends on some external input. The initial state of a *DLP* verification is always arbitrary. If only certain initial states are to be considered restrictions must be included in the property to be verified.

The formal proof of this example is done by induction on x and can be found in [8]. Note that the induction hypothesis has to be specified interactively which can be very hard to find in practice.

4 Related Work and Conclusions

There are some approaches to define the semantics of Promela formally, most notably [2]. But the semantics that is induced by our semantics for Promela$^+$ is by far the most comprehensive one. Relative to this semantics we introduced a calculus that allows deductive theorem proving for Promela. No previous work in this direction exists for Promela or other non-deterministic multi-process languages. We showed that it is—in principle—possible to approach such programming languages with methods from deductive theorem proving, thus preparing the ground for implementations in automatic provers. The methods we used are very general and can be easily applied to other languages.

The given examples show that *DLP* increases the set of verifiable Promela models significantly. As a minor drawback *DLP* has only six temporal operators to make statements about traces whereas Spin allows to specify arbitrary LTL formulas. E.g., in order to express the property "*A* holds for a while, and then *B* holds forever" *DLP* must be extended by a specific modality whereas this can be easily expressed in LTL. We intend to extend our definitions to allow for arbitrary modal operators, which would render our language as expressive as LTL. First approaches have shown that it is indeed possible to give rules for the case where the modal operator is arbitrary.

References

1. B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In *International Joint Conference on Automated Reasoning*, volume 2083 of *LNCS*, pages 626–641, 2001.
2. M. del Mar Gallardo, P. Merino, and E. Pimentel. A generalized semantics of Promela for abstract model checking. *Formal Aspects of Computing*, 16(3):166–193, 2004.
3. ESC/Java (Extended Static Checking for Java). `http://research.compaq.com/SRC/esc/`.
4. D. Harel. *First-order Dynamic Logic*, volume 68 of *LNCS*. Springer, 1979.
5. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
6. G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
7. Promela Language Reference. Available at `http://spinroot.com/spin/Man/promela.html`.
8. F. Rabe. A dynamic logic with temporal operators for Promela. Master's thesis, Universität Karlsruhe, 2004. Available online at `http://i12www.ira.uka.de/~frabe/DLTP.pdf`.
9. A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In E. Brinksma and K. G. Larsen, editors, *14th International Conference on Computer Aided Verification (CA V)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002.

Designing Efficient Procedures for #2SAT

Guillermo De Ita[1*], Mireya Tovar[2], Erica Vera[2] and Carlos Guillén[3]

[1] Universidad Politécnica de Puebla, `deita@inaoep.mx`
[2] Universidad Autónoma de Puebla, `mtovar@cs.buap.mx`, `ee_vc@cs.buap.mx`
[3] Instituto Nacional de Astrofísica Optica y Electrónica, `cguillen@inaoep.mx`

Abstract. We present some advances on the design of efficient procedures for counting models of a Boolean formula Σ (#SAT(Σ) Problem). We show that if G_Σ, the constraint graph of Σ, is acyclic or contains disjointed cycles then $\#SAT(\Sigma)$ is computed in polynomial time, establishing new polynomial classes of Conjunctive Forms for #SAT.
Keywords: #SAT Problem, Counting models, Efficient Algorithms.

1. Introduction

The problem of counting models for a Boolean formula (#SAT problem) can be reduced to several different problems in approximate reasoning. For example, for estimating the degree of reliability in a communication network, computing degree of belief in propositional theories, for the generation of explanations to propositional queries, in Bayesian inference, in a truth maintenance systems, for repairing inconsistent databases [1, 3, 6, 8].

#SAT is at least as hard as the decision problem SAT(determine if a Boolean formula is satisfiable), but in many cases, even when SAT is solved in polynomial time, no computationally efficient method is known for #SAT. For example, the 2-SAT problem, SAT restricted to consider a conjunction of ($\leq$ 2)-clauses, it can be solved in linear time. However, the corresponding "counting" problem #2-SAT is a #P-complete problem.

2. Notation and Preliminaries

Let $X = \{x_1, \ldots, x_n\}$ be a set of n *Boolean variables*. A *literal* is a variable x or a negated variable $\overline{x}$. We will consider Boolean formulas in Conjunctive Forms (CF) which are conjunctions of clauses, and where each clause is a disjunction of different literals. For $k \in I\!N$, a *k-clause* is a clause consisting of exactly k literals and, a ($\leq k$)-*clause* is a clause with k literals at most. We use $v(Y)$ to indicate the variable involved in the object Y(Y can be a literal, a clause or a CF). As is usual, for each $x \in X$, $x^0 = \overline{x}$ and $x^1 = x$. We say that F is a monotonic CF if all of its variables appear in unnegated form. A k-CF is a CF containing only

[*] This research is also supported by the Autonomous University of Puebla, via academic scholarship license.

k-clauses. A $(k, s\mu)$-CF is a k-CF such that each variable appears no more than s times. An assignment s for F is a function $s : \upsilon(F) \rightarrow \{0, 1\}$. A clause c is *satisfied* by the assignment s if and only if $c \cap s \neq \emptyset$. A CF F is *satisfied* by an assignment s if each clause in F is satisfied by s. A model of F is an assigment over $\upsilon(F)$ that satisfies F. We will denote $[\![n]\!] = \{1, 2, ..., n\}$. Let $SAT(F)$ be the set of models that F has over $\upsilon(F)$. Let $\mu_{\upsilon(F)}(F) = \#SAT(F) = |SAT(F)|$ be the cardinality of $SAT(F)$.

The Graph Representation of a 2-CF. Let Σ be a 2-CF, the *constraint graph* of Σ is the undirected graph $G_\Sigma = (V, E)$, with $V = \upsilon(\Sigma)$ and $E = \{(\upsilon(x), \upsilon(y)) | (x, y) \in \Sigma\}$. Given a 2-CF Σ, a *connected component* of G_Σ is a maximal subgraph of G_Σ such that for every pair of vertices $x, y \in V$, there is a path in G_Σ from x to y. The set of *connected components* of Σ are the subformulas corresponding to the connected components of G_Σ. Let Σ be a 2-CF, $\mathcal{F} = \{G_1, \ldots, G_r\}$ is a *partition in connected components* of Σ if $\mathcal{V} = \{\upsilon(G_1), \ldots, \upsilon(G_r)\}$ is a partition of $\upsilon(\Sigma)$.

Remark 1 *If $\{G_1, \ldots, G_r\}$ is a partition in connected components of Σ, then:*

$$\mu_{\upsilon(\Sigma)}(\Sigma) = \left[\mu_{\upsilon(G_1)}(G_1)\right] \cdot \ldots \cdot \left[\mu_{\upsilon(G_r)}(G_r)\right] \tag{1}$$

In order to compute $\mu(\Sigma)$, we should first determine the set of connected components of Σ, and this procedure can be done in linear time [7]. From now on, we suppose that Σ is a connected component. We say that a 2-CF Σ is a *cycle*, a *chain* or a *free tree* if G_Σ is it too.

3. Linear Procedures for $\#2SAT$ for subclasses of 2-CF

If G_Σ is a linear chain with m edges: let us write down its associated formula Σ, without a loss of generality (ordering the clauses and its literals, if it were necessary), as: $\Sigma = \{c_1, c_2, ..., c_m\}$, where $|\upsilon(c_i) \cap \upsilon(c_{i+1})| = 1$, $i \in [\![m-1]\!]$, then $\Sigma = \left\{\{y_0^{\epsilon_1}, y_1^{\delta_1}\}, \{y_1^{\epsilon_2}, y_2^{\delta_2}\}, \ldots, \{y_{m-1}^{\epsilon_m}, y_m^{\delta_m}\}\right\}$, $\delta_i, \epsilon_i \in \{0, 1\}$, $i = 1, ..., m$.

As Σ has m clauses then $|\upsilon(\Sigma)| = n = m + 1$. The frequency of y_0 and y_m is 1, while the other variables: $y_i, i = 2, ..., m - 1$ appear twice in Σ.

Let f_i a family of clauses of Σ built as follows: $f_i = \{c_j\}_{j \leq i}$, $i \in [\![m]\!]$. Note that $f_i \subset f_{i+1}$, $i \in [\![m-1]\!]$. Let $SAT(f_i) = \{s : \upsilon(f_i) \rightarrow \{0, 1\} | s \text{ satisifies } f_i\}$, $A_i = \{s \in SAT(f_i) | y_i \in s\}$, $B_i = \{s \in SAT(f_i) | \overline{y}_i \in s\}$. Let $\alpha_i = |A_i|$; $\beta_i = |B_i|$ and $\mu_i = |SAT(f_i)| = \alpha_i + \beta_i$. In general, we can calculate the values for the pairs: (α_i, β_i) associated to each node x_i, for $i = 1, .., m$, according to the signs (ϵ_i, δ_i) of the literals in the clause c_i, by the next recurrence:

$$(\alpha_i, \beta_i) = \begin{cases} (\beta_{i-1} \quad ,\alpha_{i-1} + \beta_{i-1}) \text{ if } (\epsilon_i, \delta_i) = (0, 0) \\ (\alpha_{i-1} + \beta_{i-1}, \beta_{i-1} \quad) \text{ if } (\epsilon_i, \delta_i) = (0, 1) \\ (\alpha_{i-1} \quad ,\alpha_{i-1} + \beta_{i-1}) \text{ if } (\epsilon_i, \delta_i) = (1, 0) \\ (\alpha_{i-1} + \beta_{i-1}, \alpha_{i-1} \quad) \text{ if } (\epsilon_i, \delta_i) = (1, 1) \end{cases} \tag{2}$$

As $|SAT(f_i)| = \mu_i = \alpha_i + \beta_i$, then μ_i determines the number of models for $f_i = \{c_j\}_{j \leq i}$, $i \in [\![m]\!]$, and then μ_m =#SAT(Σ) is computed in time $O(m)$.

Let Σ a 2-CF with n variables and m clauses and where $G_\Sigma = (V, E)$ is a free tree. We compute $\mu(\Sigma)$ while we are traversing G_Σ in depth-first. The initial node $v \in V$ for the search is any vertex with degree 1, and that node v will be the root node of the tree A_Σ defined by the depth-first search. As G_Σ is a free tree then there are not *back edges* in A_Σ. The associated pair to a node v ($v \in A_\Sigma$) is denoted by (α_v, β_v). We denote with $' \rightarrow '$ the application of one of the four rules of the recurrence (2).
Algorithm Count_Models_for_free_trees(A_Σ)
Input: A_Σ the tree defined by the depth-search over G_Σ
Output: The number of models of Σ
Procedure:
Traversing A_Σ in depth-first and, while a node $v \in A_\Sigma$ is visited, assign:

1. $(\alpha_v, \beta_v) = (1, 1)$ for any leaf node $v \in A_\Sigma$.
2. If v is a father node with a unique child node u, we apply the recurrence (2) considering that $(\alpha_{i-1}, \beta_{i-1}) = (\alpha_u, \beta_u)$ and then $(\alpha_{i-1}, \beta_{i-1}) \rightarrow (\alpha_i, \beta_i) = (\alpha_v, \beta_v)$.
3. If v is a father node with a list of child nodes associated, i.e., $u_1, u_2, ..., u_k$ are the child nodes of v, then as we have already visited all the child nodes, then each pair $(\alpha_{u_j}, \beta_{u_j})$ $j = 1, ..., k$ has been defined based on (2). $(\alpha_{v_i}, \beta_{v_i})$ is obtained by apply (2) over $(\alpha_{i-1}, \beta_{i-1}) = (\alpha_{u_j}, \beta_{u_j})$. This step is iterated until computes all the values $(\alpha_{v_j}, \beta_{v_j})$, $j = 1, ..., k$. And finally, let $\alpha_v = \prod_{j=1}^{k} \alpha_{v_j}$ and $\beta_v = \prod_{j=1}^{k} \beta_{v_j}$.
4. If v is the root node of A_Σ then return$(\alpha_v + \beta_v)$.

This procedure return the number of models for Σ in time $O(n + m)$ which is the necessary time for traversing G_Σ in depth-first.

3.1. The constraint Graph contains cycles

Let Σ a $(2, 2\mu)$-CF such that $G_\Sigma = (V, E)$ is a simple cycle with m nodes, then $|V| = m = n = |E|$. Ordering the clauses in Σ: $| \, \upsilon(c_i) \cap \upsilon(c_{i+1}) \, | = 1$, and $c_{i_1} = c_{i_2}$ whenever $i_1 \equiv i_2 \mod m$, hence $y_0 = y_m$, then $\Sigma = \left\{ c_i = \{ y_{i-1}^{\epsilon_i}, y_i^{\delta_i} \} \right\}_{i=1}^{m}$, where $\delta_i, \epsilon_i \in \{0, 1\}$. Decomposing Σ as $\Sigma = \Sigma' \cup c_m$, where $\Sigma' = \{c_1, ..., c_{m-1}\}$ is a chain and $c_m = (y_{m-1}^{\epsilon_m}, y_0^{\delta_m})$ is the edge which conforms with $G_{\Sigma'}$ the simple cycle: $y_0, y_1, ..., y_{m-1}, y_0$. We can apply the linear procedure for chains in order to compute $\mu(\Sigma')$ since Σ' is a chain.

Every model of Σ' had determined logical values for the variables: y_{m-1} and y_0 since those variables appear in $\upsilon(\Sigma')$. Any model s of Σ' satisfies c_m if and only if ($y_{m-1}^{1-\epsilon_m} \notin s$ and $y_m^{1-\delta_m} \notin s$), this is, $SAT(\Sigma' \cup c_m) \subseteq SAT(\Sigma')$, and $SAT(\Sigma' \cup c_m) = SAT(\Sigma') - \{s \in SAT(\Sigma') : s \text{ falsifies } c_m\}$. Let $Y = \Sigma' \cup \{(y_{m-1}^{1-\epsilon_m}) \wedge (y_m^{1-\delta_m})\}$, and then:

$$\#SAT(\Sigma) = \mu(\Sigma) = \mu(\Sigma') - \mu(Y) = \mu(\Sigma') - \mu(\Sigma' \wedge (y_{m-1}^{1-\epsilon_m}) \wedge (y_m^{1-\delta_m}))$$

Note that this last equation can be generalized for computing $\#SAT(\Sigma)$ if G_Σ can be decomposed in a chain and in a set of cycles $S = \{C_1, C_2, ..., C_k\}$ where there are not common edges between any two different cycles of S.

***G_Σ* contains disjointed cycles:** Let $\Sigma' \subseteq \Sigma$ be the set of clauses where $G_{\Sigma'}$ is a chain (with all the variables of Σ considered, i.e. $\upsilon(\Sigma') = \upsilon(\Sigma)$). $\Sigma' = \{c_1, ..., c_m\}$ determines an order over its clauses and its literals given for $i = 1, ..., m-1, |\upsilon(c_i) \cap \upsilon(c_{i+1})| = 1$.

Let $\Gamma = \Sigma - \Sigma'$ be the set of clauses such that for each clause $d \in \Gamma$, it conforms a cycle in $G_{\Sigma'}$. We order the clauses in $\Gamma = \{d_1, ..., d_k\}$ where if $i < j$ then the variables in d_i appear in clauses of Σ' before of the clauses where appear the variables of d_j, and also order the literals for each $d_i = (l_{i_1}, l_{i_2}) \in \Gamma, i = 1, \ldots, k$ in such a way that $i_1 < i_2$, according of the induced order given by the clauses in Σ'. Note that for each $d_i = (l_{i_1}, l_{i_2})$, the node $\upsilon(l_{i_1})$ points out the beginning of one cycle. While the node $\upsilon(l_{i_2})$ points out the end of the same cycle. Let $D_i = \{c_{i_1}, c_{i_1+1}, ..., c_{i_2-1}, c_{i_2}\}$ be the set of clauses of Σ' involved by the cycle signaled by d_i, each $C_i = D_i \cup d_i, i = 1, \ldots, k$ is a cycle formula.

For each $d_i \in \Gamma, i = 1, \ldots, k$ we conform the set of clauses $Y_i = \{(\bar{l}_{i_1}), (\bar{l}_{i_2})\} \cup D_i$ which is a chain where the two extreme nodes of the chain have associated the unitary clauses $(\bar{l}_{i_1})$ and $(\bar{l}_{i_2})$. Then $\mu(\Sigma)$, where Σ contains the linear chain Σ' and the k cycles: $C_i, i = 1, \ldots, k$, and such that there are not common edges between any two different cycles, is:

$$\#SAT(\Sigma) = \mu(\Sigma') - (\sum_{i=1}^{k} \mu(Y_i)) \tag{3}$$

Example 1 *Let $\Sigma = \{(y_1, y_2), (y_2, y_3), (\overline{y}_3, y_1), (y_3, y_4), (y_4, y_5), (y_5, y_6), (\overline{y}_4, y_6), (y_6, y_7), (y_7, y_8), (y_9, y_8), (\overline{y}_9, \overline{y}_6)\}$ be a 2-CF which contains the monotonic chain; $\Sigma' = \{(y_i, y_{i+1})\}, i = 1, ..., 8$, and $\Gamma = \{d_1, d_2, d_3\} = \{(y_1, \overline{y}_3), (\overline{y}_4, y_6), (\overline{y}_6, \overline{y}_9)\}$. There are three cycles conformed by $C_1 = \{(y_1, y_2), (y_2, y_3), (\overline{y}_3, y_1)\}$, $C_2 = \{(y_4, y_5), (y_5, y_6), (\overline{y}_4, y_6)\}$ and $C_3 = \{(y_6, y_7), (y_7, y_8), (y_8, y_9), (\overline{y}_6, \overline{y}_9)\}$. And the sets of clauses $Y_i, i = 1, 2, 3$, where $Y_1 = \{(\overline{y}_1), (y_1, y_2), (y_2, y_3), (y_3)\}$, $Y_2 = \{(y_4), (y_4, y_5), (y_5, y_6), (\overline{y}_6)\}$ and $Y_3 = \{(y_6), (y_6, y_7), (y_7, y_8), (y_8, y_9), (y_9)\}$.*

We denote with $(\alpha_i, \beta_i)_{/G}$ the corresponding pair (α_i, β_i) associated to the node y_i from the graph of the formula G. Note that as Σ' is a monotonic chain we have to apply (2) when we are traversing each node of the chain, and we apply (3) in order to compute $\mu(\Sigma)$. Showing the computing of $(\alpha_i, \beta_i)_{/\Sigma}$ in parallel way with each series $(\alpha_i, \beta_i)_{/Y_j}$, for Y_1, Y_2, Y_3, we have:

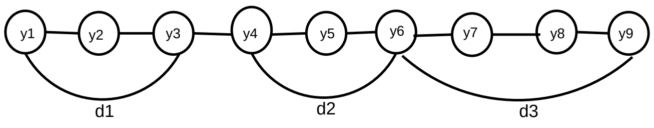


Fig. 1. The constraint graph for the formula of the example 1

$$\begin{array}{llllll} & y_1 & & y_2 & & y_3 & & y_3 \\ \mu(\Sigma') : & (1,1) & \rightarrow & (2,1) & \rightarrow & (3,2) & & \\ \mu(Y_1) : & (0,1) & \rightarrow & (1,0) & \rightarrow & (1,1) & \rightarrow & (1,0) \end{array}$$

since $(\overline{y}_1) \in Y_1$ *and* $(y_3) \in Y_1$. *Thus,* $(\alpha_3, \beta_3)_{/\Sigma} = (\alpha_3, \beta_3)_{/\Sigma'} - (\alpha_3, \beta_3)_{/Y_1} = (3,2) - (1,0) = (2,2)$*, then:*

$$\begin{array}{lccccccccc}
 & y_3 & & y_4 & & y_5 & & y_6 & & y_6 \\
\mu(\Sigma'): & (2,2) & \rightarrow & (4,2) & \rightarrow & (6,4) & \rightarrow & (10,6) & & \\
\mu(Y_2): & & & (4,0) & \rightarrow & (4,4) & \rightarrow & (8,4) & \rightarrow & (0,4)
\end{array}$$

since $(y_4) \in Y_2$ *and* $(\overline{y}_6) \in Y_2$*.Then,* $(\alpha_6, \beta_6)_{/\Sigma} = (\alpha_6, \beta_6)_{/\Sigma'} - (\alpha_6, \beta_6)_{/Y_2} = (10,6) - (0,4) = (10,2)$. *And,*

$$\begin{array}{lccccccccc}
 & y_6 & & y_7 & & y_8 & & y_9 & & y_9 \\
\mu(\Sigma'): & (10,2) & \rightarrow & (12,10) & \rightarrow & (22,12) & \rightarrow & (34,22) & & \\
\mu(Y_3): & (10,0) & \rightarrow & (10,10) & \rightarrow & (20,10) & \rightarrow & (30,20) & \rightarrow & (30,0)
\end{array}$$

since $(y_6) \in Y_3$ *and* $(y_9) \in Y_3$. $(\alpha_9, \beta_9)_{/\Sigma} = (\alpha_9, \beta_9)_{/\Sigma'} - (\alpha_9, \beta_9)_{/Y_3} = (34,22) - (30,0) = (4,22)$. $\#SAT(\Sigma) = \mu(\Sigma') - (\mu(Y_1) + \mu(Y_2) + \mu(Y_3)) = 4 + 22 = 26.$

Then, in base on the equation (3), we obtain that:

Theorem 1 *If* G_Σ *can be decomposed in a linear chain* $G_{\Sigma'}$ *(where* $v(\Sigma') = v(\Sigma)$*) and a set of cycles such that there are not common edges between any two different cycles, then* $\#SAT(\Sigma)$ *is computed in polynomial time.*

4. Conclusions

Let Σ be a 2-CF with n variables and m clause and such that G_Σ (the constraint undirected graph of Σ) is acyclic, then we show an efficient procedure to compute $\#SAT(\Sigma)$ of time complexity $O(n+m)$, which is the neccesary time for traversing G_Σ in depth-first.

We also present an efficient procedure to compute $\#SAT(\Sigma)$ being G_Σ a chain containing several cycles and such that there are not common edges between any two different cycles of G_Σ, which establish new polynomial classes for #2SAT.

References

1. Angelsmark O., Jonsson P., Improved Algorithms for Counting Solutions in Constraint Satisfaction Problems, *Int. Conf. on Constraint Programming*, 2003.
2. Creignou N., Hermann M., *Complexity of Generalized Satisfiability Counting Problems*, Information and Computation 125, (1996), 1-12.
3. Dahllöf V., Jonsonn P., Wahlström M., Counting models for 2SAT and 3SAT formulae., Theoretical Computer Sciences 332,332(1-3): 265-291, 2005.
4. De Ita G., Polynomial Classes of Boolean Formulas for Computing the Degree of Belief, *Advances in Artificial Intelligence LNAI 3315*, 2004, 430-440.
5. Greenhill Catherine , The complexity of counting colourings and independent sets in sparse graphs and hypergraphs", *Computational Complexity*, 1999.
6. Roth D., On the hardness of approximate reasoning, *Artificial Intelligence 82*, (1996), 273-302.
7. Russ B., *Randomized Algorithms: Approximation, Generation, and Counting*, Distingished dissertations Springer, 2001.
8. Vadhan Salil P., The complexity of Counting in Sparse, Regular, and Planar Graphs, *SIAM Journal on Computing*, Vol. 31, No.2, (2001), 398-427.

Exploring hybrid algorithms for SAT

Olivier Fourdrinoy, Éric Grégoire, Bertrand Mazure and Lakdhar Saïs

CRIL CNRS & IRCICA
Rue Jean Souvraz SP 18 F-62307 Lens Cedex France
{fourdrinoy,gregoire,mazure,sais}@cril.univ-artois.fr

Abstract. In this paper, several possible improvements of the combination scheme of systematic DPLL-like and local search techniques for SAT proposed by Mazure et al. are explored. Three important parameters that need to be tuned are investigated. A new weighting heuristic is described. Then, it is investigated how local search strategies that cover more diversified parts of the search space can prove useful in this combination scheme. Finally, it is studied when local search is best called within this hybrid DPLL-like algorithm.

1. Introduction

Various SAT solvers have been proposed these last years, leading to a dramatic breakthrough in the practical handling of large and hard instances. Most of them are based on one of the following two search paradigms: systematic (DPLL-like) and local search (GSAT-like). Each of these search techniques outperforms the other one with respect to some classes of instances. The combination of these search techniques can thus be a promising way to increase the general performance of these solvers. Indeed, hybridising these search methods is currently a hot topic of research within the SAT community. In [7], local search (LS) is considered as a branching heuristic for a DPLL-like algorithm. More precisely, at each node of the DPLL decision tree, a LS is performed on the currently remaining SAT sub-problem. During this LS step, a weight is computed for each variable. If LS fails to prove consistency within a pre-set amount of time, then DPLL selects the variable with the highest weight as the branching one. This basic hybrid approach proved to be efficient with respect to various sets of large and hard SAT instances [7]. However, it could still be improved in various ways. In this paper, three possible improvements are investigated. First, a new weighting heuristic is proposed. Then, it is experimented how LS strategies that cover more diversified parts of the search space can prove useful in this combination scheme. Finally, it is studied when local search is best called within this hybrid DPLL-like algorithm.

2. Weighting heuristics for falsified clauses

Although LS is by itself unable to prove inconsistency, it can provide us with some useful information that can help inconsistency to be detected. Let us elaborate on this.

Any inconsistent SAT instance contains one or several inconsistent cores [2,4,8]. This means that only one (or several) subset(s) of variables (and/or clauses) is (are) causing the unsatisfiability of the instance. LS can often help us to identify these subsets, at least to some extent [7]. Before introducing heuristics to that end, let us recall some useful definitions about inconsistent cores.

Definition 1
A SAT instance Σ is *globally inconsistent* iff Σ is inconsistent and for every set of clauses Π such that $\Pi \subset \Sigma$, Π is consistent.

When an inconsistent SAT instance is not globally inconsistent, it is called *locally inconsistent*. It is possible to define several forms of *degree of locality*. Indeed, a concept of degree of locality can be defined based on some form of ratio between the size (in terms of the number of involved clauses) of the smallest inconsistent sub-instances and the size of the initial instance. Unsatisfiable sub-instances are called *inconsistent cores*.

Definition 2
Let Σ be an inconsistent SAT instance.
Π is an *inconsistent core* of Σ iff $\Pi \subseteq \Sigma$ and Π is inconsistent.
Π is an *overall inconsistent core* of Σ iff $\Pi \subseteq \Sigma$ and Π is globally inconsistent.
An overall inconsistent core Π of Σ is *minimally inconsistent* iff for all overall inconsistent cores Ω of Σ, $|\Pi| \leq |\Omega|$.

Property 1
Let Σ be an inconsistent SAT instance. For all interpretations I of Σ, at least one clause of each inconsistent core of Σ is falsified.

Based on Property1, for each clause, the number of times that it is falsified during a failed LS can be counted as an attempt to locate or approximate inconsistent cores [7]. Indeed, intuitively, the most often falsified clauses should have a higher chance of belonging to an overall inconsistent core. As described in the introduction, this heuristic was used in [7]; a call to LS at each branching node of the DPLL search tree delivers the candidate branching variable. We believe that this approach can be refined in several ways. First, let us note that LS often encounters a sharp decrease with respect to the number of falsified clauses during the first flips. Indeed, the number of falsified clauses after the random selection of an initial truth assignment can be high. In general, the number of falsified clauses decreases quickly during the first flips. It seems natural to think that information collected during this sharp descent is not relevant to locate inconsistent cores. This can be compared to an initial noise phenomenon. Let us investigate two different strategies to address such an issue.

3. A local minima-based strategy

A first candidate strategy would rely on a threshold that would define a maximal number of falsified clauses. The counting heuristic would be inhibited each time the number of falsified clauses is greater than this threshold. Obviously enough such a strategy can only be useful when inconsistent cores are "small". Hopefully, many non-random SAT instances do exhibit quite small inconsistent cores [7]. However, the main remaining open problem with the strategy is how the threshold should be determined. Moreover, the optimal value of the threshold should depend on the nature of the instance. Let us note that if the instance contains *n* mutually independent overall inconsistent cores, leading its Max-sat value to be *c-n* where *c* is the number of clauses of the instance, then the threshold must be at least equal to *n*. Otherwise, no clause is weighted. Unfortunately, we do not have any reliable oracle informing us about the number of overall inconsistent cores. A local minimum is a non-solution state where no flip of variable can lead the number of falsified clauses to be decreased. Another strategy [5] would consist in counting falsified clauses in local minima, only. To some extent, the numbers of falsified clauses in local minima can appear as dynamic thresholds. Our preliminary experimental validation of this new heuristic is very promising. It outperforms the threshold-based strategy. Although it yields weights that are similar to the ones obtained thanks to the initial heuristic of [8], it is less time-consuming. As an illustration of the experimental tests that were conducted, Table 1 shows the number of clauses belonging to a minimally inconsistent core among the ten highest weighted clauses. The threshold strategy is investigated using different thresholds (*c*/2, *c*/5, *c*/10, *c*/100 and 1). It is compared with this local minima-based strategy and with the initial one where weights are computed in a systematic way. Such a test has been conducted on many instances. Only one instance (aim-200-2_0-no-1.cnf) is presented, because results for other instances are similar and it is known that this specific instance exhibits only one minimally inconsistent core.

Threshold	Global	c/2	c/5	c/10	c/100	1	Local minima
Clauses from core	9	9	9	8	9	5	9
Weighting calls	2000	2000	2000	1980	1200	80	1250

Table 1. Threshold comparison

4. Towards a more diversified exploration of the search space

The second possible improvement of the hybrid method of [7] concerns the initial interpretation that is selected by each LS run. The goal is to perform a more diversified exploration of the search space. Indeed, the more diversified exploration of the search space is, the better the weighting strategy could be. In [7], the initial interpretation of all called LS is chosen randomly. On the contrary, we define the initial interpretation for the current LS, based on the last failure (i.e. the last backtrack in DPLL or last interpretation of the previous LS). As a case study, we investigate a technique requiring us to reverse the value of all variables (not already assigned by DPLL). Such a "global flip" is also called a mirror. In this way, LS focuses on the last encountered problems while considering the opposite configuration.

5. Strategies for calling LS

LS, as it is grafted in DPLL in [7], is time consuming since it is called at *each* node of the DPLL search tree. Such a systematic call to LS could be relaxed as far as the current variable weights remain relevant for successive nodes in the DPLL search trees. In this respect, three strategies for calling LS are investigated.

1. LS is run systematically, i.e. for all nodes in the DPLL search tree [7]
2. LS is run as a pre-processing step, i.e. it is called once, only [3]
3. LS is run until a pre-set depth has been reached. Then, a traditional DPLL branching rule heuristic is used.

5.1 Methodology

We have implemented and compared these three strategies. As a preliminary experimental validation procedure, a simple LS algorithm, namely WSAT [9], and the basic branching MOMS heuristic [6] were considered. Basic versions of DPLL were implemented, not including the above local-minima and mirror techniques. Two families of instances have been tested:

1 AIM instances resulting from DIMACS [1], as they exhibit interesting properties: satisfiable AIM instances admit only one model whereas the unsatisfiable ones exhibit one minimally inconsistent core.

2 random instances: obtained from the traditional generation model [2]. Each group of instances contains 300 problems (6 groups of 50 problems with 50, 100, 150, 200, 250 and 300 variables, respectively). These random instances are divided into four subsets. The first one (Rand@3.25) contains easy instances that are satisfiable and that are located before the phase transition threshold. Then, Rand@4.25s and Rand@4.25u contain hard satisfiable and unsatisfiable, respectively. These instances were generated at the 4.25 threshold. Finally, Rand@5.25 contains some easy unsatisfiable instances located after the threshold. They are thus intended to represent instances with a lot of models (Rand@3.25), instances where models are grouped inside clusters with a small number of clusters (Rand@4.25s), instances almost globally inconsistent with large inconsistent cores (Rand@4.25u) and instances with a lot of small inconsistent cores (Rand@5.25).

These tests have been conducted on a Pentium 3 2.4Ghz under Linux Fedora core 2.

Instances	DP-LS-ALL	DP-LS-PRE	DP-LS-DEPTH	WSAT
Aim_yes	710.07	611.55	502.29	530.18
Aim_no	1750.21	1529.30	557.69	-
Rand@3.25	0.2	0.10	0.34	0.15
Rand@4.25s	329.04	302.32	231.43	262.32
Rand@4.25u	425.43	367.61	331.17	-
Rand@5.25	325.4	291.08	61.33	-

Table 2. AIM and random instances results

5.2 Results and comments

Table 2 summarises the results, reporting the average times in seconds to solve the instances. DP-LS-ALL, DP-LS-PRE and DP-LS-DEPTH represent the DPLL-solvers:

- with systematic calls to LS [7];
- with a single call to LS as a pre-processing step;
- calling LS until a pre-set depth (set to 5, as a case study) has been reached, before a standard DPLL branching rule heuristic is applied.

As we hoped it, a limited number of calls to LS appeared to be more efficient to solve unsatisfiable instances. Indeed, the single LS run approach suffers from the fact that LS explores limited parts of the search space, only. The branching rule of DPLL is thus based on findings about a specific subset of the search space, only. In this respect, it seems that DP-LS-PRE is probably limited by the lack of reactivity of such a LS-based branching heuristics. DP-LS-ALL is handicapped by the time consumed by the numerous calls to LS.

6. Conclusions and perspectives

In this paper, several improvements of the combination scheme proposed in [7] have been proposed. Three specific points have been addressed. First, a new constraint weighting heuristic has been described. Then, a new LS strategy to diversify the part of the explored search sub-space has been proposed. Finally, several strategies for calling LS have been experimentally studied. Our preliminary experimental results appear to validate our expectations and they encourage us to continue in this way. We are currently working on a very extensive validation of the approach and are currently working on an implementation of such hybrid techniques in competitive current solvers. We are also investigating to which extent the DP-LS-DEPTH heuristic limits the search to a small number of clusters of models.

Acknowledgements

We thank the anonymous reviewers for their numerous comments that helped us to improve the paper. This work has been supported by the *Région Nord/Pas-de-Calais*.

References

[1] Y. Asahiro, K. Iwama, and E. Miyano Random Generation of Test Instances with Controlled Attributes, In *Cliques, Coloring, and Satisfiability: The Second DIMACS Implementation Challenge.* Vol. 26 of DIMACS, 377-394, 1996.

[2] V. Chvtal and E. Szemeredi, Many hard examples for resolution, *Journal of the ACM*, 35(4):759–768, 1988.

[3] J. M. Crawford & L.D. Auton, Experimental results on the crossover point in satisfiability problems, *Proceedings of AAAI'93,* 21-27, 1993.

[4] C. P. Gomes, B. Selman, N. Crato and H. Kautz. Heavy-tail phenomena in satisfiability and constraint satisfaction, *Journal of Automated Reasoning*, 24(1-2):67-100 2000.

[5] F. Hutter, D. A. D. Tompkins, H. H. Hoos. Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT. *Proc. of the Eighth Int. Conf. on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science vol. 2470, 233-248, 2002.

[6] R. J. Jeroslow and J. Wang, Solving propositional satisfiability problems, *Annals of Mathematics and Artificial Intelligence,* 1:167-188, 1990.

[7] B. Mazure, L. Saïs, and É. Grégoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22:319-331, 1998.

[8] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman and L. Troyansky. Determining computational complexity from characteristic 'phase transitions', *Nature*, 133-137, 1999.

[9] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. *Proceedings of AAAI'94*, 337-343, 1994.

A Logical Language for Dominoes

Fernando Raymundo Velázquez-Quesada[1] and Francisco Hernández-Quiroz[2]

[1] Universidad Nacional Autónoma de México, Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas, Circuito escolar, Ciudad Universitaria. C.P. 04510, México D.F.
frvq@uxmcc2.iimas.unam.mx

[2] Universidad Nacional Autónoma de México, Facultad de Ciencias, Circuito exterior, Ciudad Universitaria. C.P. 04510, México D.F.
fhq@fciencias.unam.mx

Abstract. The mathematical theory of games does not allow explicit reasoning about the knowledge of the agents that interact in these competitive situations. In recent years there has been much work oriented towards analyzing the flow of information in games using logical tools. The main point is to express explicitly the way the information (and therefore, the knowledge) flows through the actions in a game. In this paper we present a logical language that allows to express knowledge of players and how actions modify this knowledge in the game of dominoes. We define a language based on propositional epistemic logic for expressing knowledge at a given stage of a dominoes match. Then we add actions to it to describe how knowledge is modified by each action that occurs during the game. We describe the semantics of the language in an informal way, and finally we present some examples of how the language can express situations, knowledge of the players and how this knowledge changes during a match.

1 Introduction

The mathematical theory of games, presented for the first time by J. von Neumann and O. Morgenstern [1], does not allow explicit reasoning about the knowledge of the agents that interact in these these competitive situations. We would like to express explicitly the way the information (and therefore, the knowledge) flows through the actions in a game.

In recent years there has been much work oriented towards analyzing the flow of information in games using logical tools (see [2], [3] or [4]). A persistent theme through these years has been "the pursuit of a broad conception of logic, oriented toward information structures and actions transforming these" [5]. In our work, we use a logical language to express knowledge of players in a game, and how actions modify it. We focus in one particular game: dominoes.

The set of tiles for dominoes consists of all the possible pairs (28) taken from numbers 0 through 6. Four players sit around a table, partners sitting opposite. Each player takes 7 of the randomly shuffled closed tiles on the table. Any player of the winning couple of the last match starts the next one playing any of her tiles (in the first round, tile $\boxed{6 \vert 6}$ starts) and the flow of the play moves to the right. Each player must draw a tile in her turn, except when she cannot (in which case she *passes*: losses her turn). To draw a tile, it must match any of the two free-ends on the table. This tile is placed with its matching side next to the matching end on the table, leaving the other side of the tile as the new free-end. The winner is the couple to which belong the player that finishes playing all her tiles first. If no player can draw a tile and all still have tiles, then it is said that the match is *closed* and the winner is the couple whose tiles sums less points.

The idea of using logic to study dominoes comes from Eric Schwarz, who analyzed a partnership match using first order logic [6]. Our work is based in propositional epistemic logic ($\mathcal{EL}$) instead of first order logic. By using $\mathcal{EL}$ we avoid using a set of sentences to describe the properties of the knowledge we are modeling (as this description is implicit in the relations of the model). As a modal (multimodal) logic, $\mathcal{EL}$'s formulae are clean, transparent and compact. Most importantly, they are computationally solvable whereas first-order logic is undecidable. The goal of expressing the way knowledge changes is achieved by adding actions to the language (becoming propositional dynamic epistemic logic). This also allows us to avoid sentences related to time.

2 Language $\mathfrak{D}\mathcal{EL}_{\mathfrak{D}}$

Our language is based on the language of epistemic logic. The classical source for epistemic logic is [7], in which J. Hintikka showed that a modal approach to single-agent epistemic attitudes could be informative and rewarding. This language comprises a set of atomic propositions, a set of logical connectives and a set of modal operators. Then it is possible to express the knowledge of a group of agents about different situations, and even more, the knowledge they can have about the knowledge of other agents.

We define the set of players as $\mathcal{J} = \{a, b, c, d\}$ and the set of tiles as $\mathcal{F} = \{\boxed{0 \vert 0}, \boxed{0 \vert 1}, \boxed{0 \vert 2}, \ldots, \boxed{5 \vert 5}, \boxed{5 \vert 6}, \boxed{6 \vert 6}\}$. Our language is based on a set of atomic propositions that expresses the basic facts about the game.

Definition 1 (Atomic propositions for dominoes $\Phi_{\mathfrak{D}}$). *The set of* atomic propositions for dominoes *($\Phi_{\mathfrak{D}}$) is defined as*

$$\Phi_{\mathfrak{D}} = \Big\{ \ \boxed{x \vert y}^{\,i} \ , \ \boxed{x \vert y}^{\,t} \ , \ \mathit{tiles}^i_n \ , \ \mathit{tiles}^t_n \ , \ \mathit{pts}^i_n \ , \ \mathit{pts}^t_n \ , \\ \mathit{lesspoints}^{i,j} \ , \ \mathit{turn}^i \ , \ \overline{u \ldots v} \ , \ \overline{u \ldots v}\boxed{x \vert y}^{\,i} \ , \ \boxed{x \vert y}\overline{u \ldots v}^{\,i} \ \Big\}$$

where $i, j \in \mathcal{J}$, $\boxed{x \vert y} \in \mathcal{F}$ and u, v are possible free-ends on the table. Each proposition has the intuitive meaning shown in table 1.

For describing how actions modify the knowledge of the players, we need a language of actions in order to express that a formula holds after an action has been executed. This language is built from a set of a basic actions of the game using sequential composition and non-deterministic choice.

Table 1. Intuitive meaning for atomic propositions

Proposition	**Intuitive meaning**
$\boxed{x\vert y}^{\,i}$	Player i has the tile $\boxed{x\vert y}$ in her hand.
$\boxed{x\vert y}^{\,t}$	The tile $\boxed{x\vert y}$ is on the table (has been played).
tiles^i_n	Player i has n tiles in her hand.
tiles^t_n	There are n tiles on the table.
pts^i_n	The tiles player i has in her hand sum n points.
pts^t_n	The tiles on the table sum n points.
$\mathit{lesspoints}^{i,j}$	The tiles of player i and player j sum less points than those of the other players.
turn^i	It is player i's turn.
$\overline{u\ldots v}$	The free-ends on the table are u and v.
$\overline{u\ldots v\boxed{x\vert y}}^{\,i}$	The free-ends on the table were u and v before player i threw $\boxed{x\vert y}$.
$\overline{\boxed{x\vert y}^{\,i}_{u\ldots v}}$	The free-ends on the table were u and v after player i threw $\boxed{x\vert y}$.

Definition 2 (Action language for dominoes $\mathcal{AL}_{\mathfrak{D}}$). *The actions α of the* action language for dominoes *($\mathcal{AL}_{\mathfrak{D}}$) are given by the following rule:*

$$\alpha ::= \varepsilon \mid \boldsymbol{\alpha}^i_{\boxed{x|y}} \mid \boldsymbol{\alpha}^i_{\epsilon} \mid (\alpha;\beta) \mid (\alpha \cup \beta)$$

where $\alpha, \beta \in \mathcal{AL}_{\mathfrak{D}}$.

The action ε is the "do nothing action", $\boldsymbol{\alpha}^i_{\boxed{x|y}}$ has the intuitive meaning "player i draws tile $\boxed{x|y}$ on a free-end x on the table", $\boldsymbol{\alpha}^i_{\epsilon}$ means "player i passes", $(\alpha;\beta)$ represents the sequential composition of the actions α and β, and $(\alpha \cup \beta)$ represents the non-deterministic choice between α and β.

Based on the atomic propositions on $\Phi_{\mathfrak{D}}$ and the language of actions $\mathcal{AL}_{\mathfrak{D}}$, we define the dynamic epistemic logical language for dominoes. With this language we are able to express the situation at certain stage of a match, the knowledge of the players and how actions modifies the situation and their knowledge.

Definition 3 (Language $\mathcal{DEL}_{\mathfrak{D}}$). *The formulae φ of the* dynamic epistemic logical language for dominoes *($\mathcal{DEL}_{\mathfrak{D}}$) are given by the following rule:*

$$\varphi ::= \top \mid p \mid \neg\varphi \mid (\varphi \vee \psi) \mid K_i\varphi \mid C_{\mathcal{B}}\varphi \mid [\alpha]\varphi$$

where $p \in \Phi_{\mathfrak{D}}$, $i \in \mathcal{J}$, $\mathcal{B} \subseteq \mathcal{J}$, $\varphi, \psi \in \mathcal{DEL}_{\mathfrak{D}}$ and $\alpha \in \mathcal{AL}_{\mathfrak{D}}$.

The logical connectives $\neg$ and $\vee$ have the usual meaning ($\top$ is the proposition that is always true). The formula $K_i\varphi$ has the intended meaning "the agent i knows that φ", the formula $C_{\mathcal{B}}\varphi$ means "it is common knowledge between agents in $\mathcal{B}$ that φ (everybody in $\mathcal{B}$ knows φ, everybody in $\mathcal{B}$ knows that everybody in $\mathcal{B}$ knows φ and so on.)" and $[\alpha]\varphi$ means "after the action α is executed, φ holds".

3 Informal Semantics of $\mathcal{DEL}_{\mathfrak{D}}$

Epistemic logic is a multimodal logic, and therefore the language has meaning in structures called *Kripke models* after Saul Kripke's classical presentation [8]. For epistemic logic, the intuitive idea behind Kripke models (also known as *possible world models*), as presented by Hintikka [7], is that there are a number of other possible worlds besides the real one. Given her current information, an agent may not be able to distinguish between this real world and the other possible (but not real) worlds. In the game of dominoes, these possible worlds have a concrete and direct interpretation: all the possible distribution of the tiles among the players.

Usually, given a set of agents $\mathcal{A}$ and a set of atomic propositions Φ, a possible world model M is defined as a tuple $M = (W, R_i, V)$ where

- $W \neq \emptyset$ is the set of possible worlds.
- R_i is a binary relation over W for each agent $i \in \mathcal{A}$ ($R_i \subseteq (W \times W)$).
- $V : \Phi \to 2^W$ assigns to every atomic proposition a subset of W.

For our purposes, we need to make modifications to the possible world model:

1. In a possible world given, each atomic proposition has a truth value independently of the truth value of the others atomic propositions. In dominoes, the truth values of some atomic propositions are related (like $\boxed{0|2}^{\,a}$ and $\boxed{0|2}^{\,c}$, it is not possible that two players have the same tile). We need to define truth values for an atomic proposition in a way that the truth values of its related propositions are related.
2. Some of the atomic propositions (turn^i, $\overline{u \ldots v}$, $\underline{u \ldots v}\boxed{x|y}^{\,i}$ and $\boxed{x|y}^{\,i}_{\overline{u \ldots v}}$) can not be given a truth value by just knowing distribution of the tiles at a certain stage. To give truth value to these propositions we need to keep a record of the actions that have been executed during the match. A formal description of the possible world model for dominoes and the history of a match (a sequence whose entries record an action and the free-ends that result from that action) will appear in a future paper.
3. To give a truth value to each formulae of the form $[\alpha]\varphi$, we need also to specify how each action modify the possible world-history of a match pair. Following ideas from [9], each action is then consider as a map that takes the possible world model and the history that describes the situation of the game *before* the action, and returns the possible world model and the history that describes the situation of the game *after* the action. Again, a formal definition of these actions as a maps will appear in a future paper.

4 Examples of the Use of the Language

We give some examples of the use of the language on a dominoes match:

- $\boxed{x|y}^{\,\mathcal{J}} \equiv \boxed{x|y}^{\,a} \vee \boxed{x|y}^{\,b} \vee \boxed{x|y}^{\,c} \vee \boxed{x|y}^{\,d}$: Tile $\boxed{x|y}$ is in some player's hand.
- $\boxed{x|-}^{\,i} \equiv \boxed{x|0}^{\,i} \vee \boxed{x|1}^{\,i} \vee \cdots \vee \boxed{x|6}^{\,i}$: Player i has at least one tile with x.
- $K_c \neg K_b \boxed{1|6}^{\,c}$: Player c knows that b does not know that she has $\boxed{1|6}$.
- $\mathit{tiles}^a_7 \to [\boldsymbol{\alpha}^a_{\boxed{x|y}}]\mathit{tiles}^a_6$: If a has 7 tiles, then after she plays any tile, she will have six of them.
- $(\mathit{turn}^a \wedge \overline{4 \ldots 6}) \to [\boldsymbol{\alpha}^a_{\boxed{6|5}}]\overline{5 \ldots 4}$: If it is a's turn with 4 and 6 as free-ends, and a plays $\boxed{6|5}$, then the free-ends will be 5 and 4.

- $(\overline{\mathit{0}\ldots\mathit{6}} \wedge \mathit{turn}^d) \rightarrow [\boldsymbol{\alpha}^d_\epsilon]C_\mathcal{J}\ (\neg\boxed{\mathit{0}\bullet-}^{\,d} \wedge \neg\boxed{\mathit{6}\bullet-}^{\,d})$: If it is d's turn with 6 and 0 as free-ends of the table, and d pass, then it will be common knowledge among all the players that neither d has 0 nor 6.
- $\mathit{closed} \equiv \overline{x\ldots x} \wedge \neg\boxed{x\bullet-}^{\,\mathcal{J}}$: The match is closed (no tile can be played).
- $\mathit{tiles}_n^{\mathcal{J}^{-a}} \equiv \bigvee_{i\in\mathcal{J}^{-a}} (\mathit{tiles}_n^i)$: Player b, c or d has n tiles ($\mathcal{J}^{-i} = \mathcal{J} - \{i\}$).
- $\mathit{won}^{i,j} \equiv (\mathit{tiles}_0^i \wedge \neg\mathit{tiles}_0^{\mathcal{J}^{-i}}) \vee (\mathit{tiles}_0^j \wedge \neg\mathit{tiles}_0^{\mathcal{J}^{-j}}) \vee (\mathit{closed} \wedge \mathit{lesspoints}^{i,j})$: The couple i and j has won the match.

5 Conclusions and Further Work

We have defined a logical language $\mathcal{DEL}_\mathcal{D}$ to describe situations during a game of dominoes. Due to space constraints, we do not define here a formal semantics for the language. In a further paper a formal semantics will be offered.

As pointed by S. Druiven in [3], we can distinguish four kinds of knowledge in a game environment: *game knowledge* (knowledge players have about the rules of the game they are playing), *definite knowledge* (knowledge about the situation of the game that is developed as a consequence of the actions in the game), *strategic knowledge* (knowledge about strategies over the game) and *historical knowledge* (knowledge about previous matches). For the time being, the language $\mathcal{DEL}_\mathcal{D}$ can only express definite knowledge. We expect to extend the language and the semantic model to express not only definite knowledge but strategic and historical knowledge too. Most importantly, we are interested in the way strategic and historical knowledge influence the way definite knowledge of a single player evolves during a match.

References

1. von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior. Princenton University Press (1944)
2. van Ditmarsch, H.P.: Knowledge games. PhD thesis, Institute for Logic, Language and Computation (University of Amsterdam) (2000)
3. Druiven, S.: Knowledge development in games of imperfect information. Master's thesis, Institute for Knowledge and Agent Technology (University Maastricht) & Artificial Intelligence (University of Groningen) (2002)
4. Kooi, B.P.: Knowledge, chance, and change. PhD thesis, Institute for Logic, Language and Computation (University of Amsterdam) (2003)
5. van Benthem, J., Dekker, P., van Eijk, J., de Rijke, M., Venema, Y.: Logic in action. Institute for Logic, Language and Computation (University of Amsterdam) (2001)
6. Schwarz, E.: An instance of a complete communication cycle within co-operative games: the case of domino. Unpublished (2001)
7. Hintikka, J.: Knowledge and Belief. Ithaca, N.Y. Cornell University Press (1962)
8. Kripke, S.: Semantical analysis of modal logic i. normal modal propositional calculi. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik (1963) 67–96
9. Baltag, A., Moss, L.S., Solecki, S.: The logic of public announcements, common knowledge and private suspicious. Technical Report SEN-R9922, CWI, Amsterdam (1999)

Fregean Algebraic Tableaux:
Automating Inferences in Fuzzy Propositional Logic

Christopher Lepock[1] and Francis Jeffry Pelletier[2]

[1] University of Alberta clepock@ualberta.ca
[2] Simon Fraser University jeffpell@sfu.ca

Abstract. We develop a tableau procedure for finding theorems and consequence relations of RPL$_\triangle$ (i.e., Ł$_\aleph$ extended with constants and a determinacy operator). RPL$_\triangle$ includes a large number of proposed truth-functions for fuzzy logic. Our procedure simplifies tableaux for infinite-valued systems by incorporating an insight of Frege's. We take formulas of the language to be names for their truth-values, which permits them to be manipulated in the tableaux as if they were algebraic variables.

1 Fuzzy logic

The central idea of fuzzy logic is that the truth-values are the [0..1] interval, with 0 corresponding to classical falsity and 1, to classical truth. There is little consensus on how best to define infinite-valued connectives. A wide variety of truth-functions have been proposed for different purposes in the literature. Our research has focused on the system RPL, which is particularly interesting in three ways. First, it contains the system Ł, arguably the most plausible fuzzy extension of the classical conditional (see [1], and the arguments of [2, pp. 114-8] and [3, pp. 366-7]). Second, it can talk about the truth-values of its formulas. Thus, it allows one to model approximately valid arguments and approximately true sentences by allowing one to say just how truth-preserving an inference is or how true a sentence is. Third, in it, we can define a wide variety of the connectives that have been proposed in the literature. It is particularly important for theorem-proving systems to have this breadth; the capacity to determine easily what the tautologies of these new truth-functions are will make it much easier to determine their merits and demerits for the applications for which they were proposed. Using $[\![\varphi]\!]$to be the truth-value of φ, RPL$_\triangle$ is defined as:

$[\![\varphi \supset \psi]\!]$= min(1, 1-$[\![\varphi]\!]$+$[\![\psi]\!]$)
$[\![\neg\varphi]\!]$= 1 -$[\![\varphi]\!]$
$[\![C_i]\!]$= i, for each rational $i \in [0..1]$
$[\![J_i\varphi]\!]$= 1 if $[\![\varphi]\!]$= i, 0 otherwise.

In the interests of space, we will only discuss a two other automated procedures in detail; but there are a number of others worth mentioning. There are several procedures that, like those of [4] or [5] apply just to the fragment of Ł having only the truth-functions $\wedge, \vee, \neg$. [6] gives a procedure that can determine theoremhood in any decidable propositional fuzzy logic. As the run time of this algorithm is at least doubly exponential, its use is not practically feasible. [7] outlines a procedure that finds a finite number of subintervals of [0..1] such that if the formula in question takes the value 1 on an arbitrarily chosen value from each subinterval, the formula takes the value 1 on the entire range of truth-values. This procedure also appears to be quite complex in operation.

[8] and [9–12] describe methods which are closely related to one another and to the method we will outline below. Although these systems were designed for $L_\aleph$, they can be extended to cover the full $RPL_\triangle$. Both are, in effect, tableau systems. Hähnle's constraint tableaux are extensions of signed tableaux to infinite-valued logics; Beavers's procedure consists of decomposing a formula into a set of equations to be checked for satisfiability, although he does not present these equations in tableau format. The central idea of Beavers's procedure is that each formula of $L_\aleph$ corresponds to a set of linear polynomial functions, determined by the truth-functions corresponding to the connectives found in the formula. Each formula will exhaustively divide the [0..1] interval into a bunch of disjoint subintervals, where each of these subintervals is describable by a linear function. The formula as a whole is merely the piecewise combination of these functions. Therefore, to find whether a formula is a logical truth (always takes the value 1), the task is to determine whether this combination of functions ever takes a value less than 1; and this amounts to determining whether any of the functions describing the formula over a subinterval takes a value less than 1. Beavers reports various implementations of an algorithm that makes use of a linear programming package to evaluate the polynomials generated by the formulas.

Hähnle's constraint tableaux are a variant on signed tableaux for finite-valued logics, which consist of formulas prefixed with signs indicating their truth values. In constraint tableaux, rather than representing individual truth values, the signs place constraints on the truth values of their formulas. The decomposition rules of constraint tableaux result in new constraints applied to the formulas of which the original was composed, and inequations (equations with inequality relations) representing the relations between the variables used in the signs. There are no branching rules; instead, Hähnle uses binary variables (variables that can take only 0 and 1 as values), which represent the same information as new branches of the tableau would contain. For instance, the decomposition rules for $\supset$ are:

$$\begin{array}{c} \boxed{\leq i}\ (A \supset B) \\ \Downarrow \\ \boxed{\geq (1-i+jy)}\ A\ \ (y \leq i) \\ \boxed{\leq (j+y)}\ B \end{array} \qquad\qquad \begin{array}{c} \boxed{\geq i}\ (A \supset B) \\ \Downarrow \\ \boxed{\leq (1-i+j)}\ A \\ \boxed{\geq j}\ B \end{array}$$

In the rule on the left, y is a binary variable. The information contained in the signs can also be entered into the series of inequations; for instance, from $\boxed{\leq k_2}\ B$ we can infer $[\![B]\!] \leq k_2$. To show that a formula A is a theorem of fuzzy logic, we start a tableau with $\boxed{\leq i}\ A$ and decompose A fully. We then evaluate the resulting inequations, and say that A is a theorem if and only if the least i that satisfies every inequation is 1.

Applying the decomposition rules for these tableaux is simple, particularly since there are no branching rules. The difficult part is determining whether a branch is open or closed. To do this, Hähnle treats the series of inequations associated with a tableau as a problem in linear programming, and in his implementation he passes these inequations off to a linear programming package.

2 Fregean Algebraic Tableaux

The use of signs in tableaux for infinite-valued logics is unnecessarily complicated. Each application of a conditional decomposition rule in constraint tableaux forces the introduction of a new variable (the variable j in the rule listed above), in order to relate the constraints found in the signs for antecedent and consequent to each other. This problem arises because, like classical tableaux, constraint tableaux permit only syntactic entities to occur in the tableau itself. Frege argued that propositions designate their truth-valuesany true proposition designates an object called 'the true', and any false proposition designates 'the false'. (Different propositions have different "senses"; they designate these objects in different ways, like "the author of *Über Sinn und Bedeutung*" and "Gottlob Frege" name the person in different ways.)

Fregean algebraic tableaux, or FAT, simplify constraint tableaux by taking formulas to designate their truth-values. On this approach, we can intelligibly mix logical formulas and arithmetical signs in the same expressions. This makes expressions like $p < k$ meaningful; this expression says that the truth-value named by p is less than k. A formula containing a connective, however, can be thought of as naming a truth-value that is a function of those named by the subformulas it contains. Truth-functional constants can be thought of as propositions that wear their truth-values on their sleeves; the constant C_1 refers to the same thing as the number '1', and so we need make no distinction between them. This approach permits us to express all the information found in the signs of constraints in the lines of the tableau itself. The result is a system that can combine the advantages of a tableau procedure with the efficiency of Beavers's purely semantic procedure.

Part of our motivation in developing Fregean tableaux was to facilitate the teaching of fuzzy logic. The only way to really understand a logical system, especially one that understands truth in such an unusual way, is to work within it, and thus learn not just what is tautologous, but why. Tableaux are familiar and intuitive, and the part of the task assigned to the linear programmer in an automated procedure requires only high-school algebra to perform by hand (for fairly simple formulas, at any rate).

In the interests of space, we will present the procedure here for the basic connectives of RPL$_\triangle$:$\supset, \neg$, the constants, and the J-operators. As noted above, a plethora of further connectives can be defined in terms of these. The complete system implemented in our automated prover. FLAT, uses separate rules for many of these connectives (which eliminates a number of duplicated branches). The procedure has been proven sound and complete.

We begin with a conclusion A, which we wish to prove, and a possibly empty set of premises Γ. The procedure has three steps.

Step 1. If one wants to prove that the conclusion takes the value 1 whenever the premises do, the first line of the tableau should be A <1. Then, for any formula $\phi \in \Gamma$, a line of the form $\phi \geq 1$ should be entered in the tableau.

One may want to prove restrictions on the truth-values of one's conclusion, or put restrictions on the truth-values of the premises, other than that they are absolutely true. There are two ways to do this. One can state the restrictions using the language of RPL$_\triangle$, and then use the rules applying to those connectives to decompose the formulas in the tableau. It is easier, however, to enter such restrictions directly into the tableau as inequalities. So, if one wants to prove that $[\![\mathrm{A}]\!] \geq n$ for some n other than 1, the first line of the tableau would be A $< n$. To say that for some premise B, $[\![\mathrm{B}]\!] \geq k$, one would enter the line B $\geq k$. Thus we could evaluate the correctness of the claim: "whenever each of the premises takes a value greater than or equal to k, the conclusion must take a value $\geq n$. (In the examples of this paper, we always use 1 as the "designated value.")

Step 2. These two categories of rules may be applied anywhere in the derivation.
(a) *Replacement rules.*
Rule C: A constant truth function may be replaced by the truth-value it signifies.
Rule N: In any line of a tableau, any expression of the form $\neg\phi$ may be replaced with $1 - \phi$. (E.g., from $\neg p \geq C_{0.5}$, infer the line $\neg p \geq 0.5$ by rule C, and then $1 - p \geq 0.5$ by rule N.)
The justification of these rules (as should be obvious) is that, in keeping with our Fregean outlook, we take $\neg\phi$ and C_i to be names for $1-[\![\phi]\!]$ and i, respectively.
(b) *Decomposition rules.* These can be applied at any time to any formulas of the forms specified. Names are given above the rules for ease of reference.

⊃GE

$$\begin{tikzcd} \phi \supset \psi \geq \chi \arrow[d] \\ \begin{array}{l} \phi \leq \psi - \chi + 1 \\ \chi \leq 1 \end{array} \end{tikzcd}$$

⊃LE

$$\begin{tikzcd} & \phi \supset \psi \leq \chi \arrow[dl] \arrow[dr] & \\ \phi \geq \psi - \chi + 1 & & \begin{array}{l} \phi \leq \psi \\ \chi \geq 1 \end{array} \end{tikzcd}$$

⊃SG

$$\begin{tikzcd} \phi \supset \psi > \chi \arrow[d] \\ \begin{array}{c} \phi < \psi - \chi + 1 \\ \chi < 1 \end{array} \end{tikzcd}$$

⊃SL

$$\begin{tikzcd} & \phi \supset \psi < \chi \arrow[dl] \arrow[dr] & \\ \phi > \psi - \chi + 1 & & \begin{array}{l} \phi \leq \psi \\ \chi > 1 \end{array} \end{tikzcd}$$

JGE

$$\begin{tikzcd} & \mathrm{J}_i\phi \geq \chi \arrow[dl] \arrow[d] \arrow[dr] & \\ \begin{array}{l} \phi \geq i \\ \phi \leq i \\ \chi \leq 1 \end{array} & \begin{array}{l} \phi > i \\ \chi \leq 0 \end{array} & \begin{array}{l} \phi < i \\ \chi \leq 0 \end{array} \end{tikzcd}$$

JLE

$$\begin{tikzcd} & \mathrm{J}_i\phi \leq \chi \arrow[dl] \arrow[d] \arrow[dr] & \\ \begin{array}{l} \phi > i \\ \chi \geq 0 \end{array} & \begin{array}{l} \phi < i \\ \chi \geq 0 \end{array} & \begin{array}{l} \phi \geq i \\ \phi \leq i \\ \chi \geq 1 \end{array} \end{tikzcd}$$

$$\begin{tikzcd}[row sep=small]
& \text{JSG} & \\
& \mathrm{J}_i\phi > \chi \arrow[dl] \arrow[d] \arrow[dr] & \\
\phi \geq i & \phi > i & \phi < i \\
\phi \leq i & \chi < 0 & \chi < 0 \\
\chi < 1 & &
\end{tikzcd}
\qquad
\begin{tikzcd}[row sep=small]
& \text{JSL} & \\
& \mathrm{J}_i\phi < \chi \arrow[dl] \arrow[d] \arrow[dr] & \\
\phi > i & \phi < i & \phi \geq i \\
\chi > 0 & \chi > 0 & \phi \leq i \\
& & \chi > 1
\end{tikzcd}$$

Step 3. When no further decomposition or replacement rules can be applied, linear programming can determine whether the resulting set of inequations is feasible. When calculating by hand, this question can be resolved by simple algebra. (In a hand tableau, a branch closes if one can derive impossible inequalities or equations indicating that some formula must take a truth-values outside [0..1]. The lines of any complete open branch containing no logical symbols except propositional variables describe a set of valuations where all the premises take a value greater than the designated value and the conclusion takes a lower value – i.e., a set of counterexamples to the inference or formula being investigated.)

3 A comparison of FAT and constraint tableaux

The chief advantage of FAT over constraint tableaux is that the former never introduce new variables to be calculated over; all information is represented using subformulas of the premises and conclusion at the head of the tableau. The rules are also simpler (compare Hähnle's $\leq i$ $(A \supset B)$ rule with our $\supset$LE), which can lead in some cases to drastically simpler tableaux. For example, consider

$$p \supset (p \supset (\ldots . \neg p) \ldots),$$

with p repeated n times excluding the negated instance. Formulas of this form take the value 1 for $[\![p]\!] \leq \frac{n-1}{n}$, and the value $\frac{1-[p]}{n}$ otherwise. Thus, none is a tautology of $\text{Ł}_{\aleph}$. When comparing the two systems, assume for simplicity that the procedure applies all possible decomposition rules and then checks the results for feasibility, without checking for closed branches before all formulas have been decomposed. This is obviously inefficient, but so is using the linear programming module to check if branches that have not been fully decomposed are closed; since there are many ways of deciding which branches to continue to decompose and which not to, we will not assume any particular routine for doing so.

Under that scenario, the reader can easily verify the following statistics. A constraint tableau generated in that manner for a formula of the above form has $2n$ branches (from the possible configurations of the n binary variables generated); the FAT tableau has $n + 1$. Each branch of the constraint tableau has $2n$ lines containing no logical operators (which are the lines that the linear programmer must work with); the number of lines of that type in the FAT vary from 1 to n on different branches. Each branch of the constraint tableau has $n + 1$ distinct variables; each branch of the FAT has one (namely, p). The Fregean tableau allows the linear programming module to calculate the result from far less data and with far fewer possibilities to consider.

The moral of the story is: do less work; get FAT.

4 Conclusion

The FAT method has been implemented in Java and runs on the usual platforms (Linux/Unix, Mac, Windows). The user is allowed to vary the designated value, so that we can test whether, if all premises take at least the value r then so does the conclusion. There is also an option, for use with invalid arguments, to find the lowest value that the conclusion could take when the premises are all designated. (A test for what the 'most invalid' is for that the argument). The method of determining the validity of arguments in fuzzy logic is, we believe, superior to the methods in the literature in various ways. To begin with, our procedure covers a wider range of operators than other theorem-proving procedures (excepting [6]) Although Hähnle's and Beavers's methods could be extended to the class of connectives we describe, this has not been done so far. A more important improvement is that our rules are much more natural extensions of finitely-many-valued tableau rules. Each tableau rule is a statement about what truth-values the subformulas must name, which makes them easier to understand than Beavers's "general polynomial formulas" or Hähnle's signs. The Fregean insight our method uses allows linear programming to be used for determining validity in an ideal way both for pedagogy and for ease of understanding in novel applications.

References

1. Hájek, P.: The Metamathematics of Fuzzy Logic. Kluwer, Dordrecht (1998)
2. Williamson, T.: Vagueness. Routledge, London (1994)
3. Paoli, F.: A really fuzzy approach to the sorites paradox. Synthèse **134** (2001) 363–387
4. Kenevan, J., Neapolitan, R.: A model theoretic approach to propositional fuzzy logic using beth tableaux. In Zadeh, L., Kacprzyk, J., eds.: Fuzzy Logic for the Management of Uncertainty. Wiley, NY (1992)
5. Lee, R., Chang, C.L.: Some properties of fuzzy logic. Info. and Contr. **19** (1971) 417–431
6. Gehrke, M., Kreinovich, V., Buochon-Meunier, F.: Propositional fuzzy logics: Decidable for some (algebraic) operators; undecidable for more complicated ones. Inter. Jour. Intel. Systems **14** (1999) 935–947
7. Morgan, C., Pelletier, F.: Some notes on fuzzy logic. Ling. and Phil. **1** (1977) 79–97
8. Beavers, G.: Automated theorem proving for łukasiewicz logics. Studia Logica **52** (1993) 183–196
9. Hähnle, R.: Automated Theorem Proving in Multiple-Valued Logics. Oxford UP, Oxford (1993)
10. Hähnle, R.: Many-valued logic and mixed integer programming. Annals of Mathematics and Artificial Intelligence **12** (1994) 231–264
11. Hähnle, R.: Proof theory of many-valued logic – linear optimization – logic design: Connections and interactions. Soft Computing **1** (1997) 107–119
12. Hähnle, R.: Tableaux for many-valued logics. In D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J., eds.: Handbook of Tableaux Methods. Kluwer, Dordrecht (1999) 529–580

Reasoning on Multimodal logic with the Calculus of Inductive Constructions

Houda Anoun

LaBRI-Bordeaux 1
anoun@labri.fr

Abstract. Multimodal Categorial grammars are a formalism well suited for dealing with the syntax-semantics interface in Computational Linguistics. Unfortunately their expressive power is impaired by a loss of intuition and readability. We introduce Icharate, a collection of tools for proving syntactic and semantic properties of entire classes of grammars, which aims at facilitating the study of these classes, and explaining different linguistic phenomena. Icharate is built upon a formalization of multimodal grammars in the Calculus of Inductive Constructions using Coq proof assistant.

1 Introduction & Motivations

Human language is an exciting puzzle. During the last decades, several formalisms have been defined to model human cognition and to explain people's faculty to learn, analyze and understand easily different natural languages.
Amongst these proposals, we find M. Moortgat's multimodal categorial grammars [Moortgat97]. This model is based on logical type theory and provides an easy syntax/semantics interface, owing to the Curry-Howard correspondence. Moreover, this formalism is powerful enough to deal with complex natural language phenomena. Unfortunately, the strength of multimodal grammars is counterbalanced by an obvious loss of simplicity and readability. Thereby, there is an imperious need for tools which facilitate the study and use of such complicated linguistic formalisms. The ambitious project aiming at producing an automatic tool for multimodal logics is not realizable as the proof-search problem is undecidable. We believe that the use of a proof assistant such as **Coq** [CoqA04] will lead to a satisfactory compromise.
Therefore, we propose a meta-linguistic toolkit, $\mathcal{ICHARATE}$, built upon **Coq** proof assistant and dedicated to the study of syntactic and semantic properties of multimodal logics. Our toolkit is a framework intended for both neophytes and senior researchers in computational linguistics. On the one hand, the combination between automatic proof steps and manual steps is beneficial as it will help the beginners improve their comprehension and intuition about these logics. On the other hand, $\mathcal{ICHARATE}$ is an easily extendable framework: it is well suited for researchers who want to test their new ideas or formally check their conjectures. The various proofs are then automatically verified by the proof-checker and recorded in different libraries where they can be consulted at any time.

2 Preliminaries

Multimodal logics are composed of two distinct parts: an invariant part namely the core logic (inference rules), and a structural part which allows a controlled management of resources (e.g., local associativity, local permutation) [Moortgat97]. Deduction rules do not depend upon the words (i.e., grammar terminals) of the chosen natural language; they rather express the way in which such words can combine by using their syntactic types. Each type can be either primitive or composite. The first class contains atomic types which represent complete expressions, such as the type **np** (noun phrases), **n** (common nouns) and **s** (well-formed sentences). Composite categories are built from primitive types by using families of binary type constructors ($/_i$, $\backslash_i$, $\bullet_i$) and unary ones ($\Diamond_j$, $\Box_j$) where indexes (modes of compositions) are employed to encode features or to restrict the application of structural rules [Moortgat97]. For instance, the improved type $((\mathbf{np}\backslash_a\mathbf{s})/_a(\Diamond_{to}\mathbf{pp}))/_a\mathbf{np}$ can be assigned to ditransitive verbs (e.g., '*give*') whose indirect objects are prepositional phrases starting only with *'to'* (e.g., '*I give a present to Sue*').
Semantics and syntax are interdependent in multimodal categorial grammars. Following **Montague** semantics [Gamut91], the meaning of words is represented by simply typed λ-terms. The semantics of an expression is built in parallel with its syntactic derivation in a compositional fashion.
The proof terms that the logical system operates on are sequents of the form ($\Gamma \vdash \mathbf{A} \triangleright \mathbf{a}$) where Γ is a context (i.e., a structured binary tree that take into account both linear ordering and hierarchical grouping of resources) of typed syntactic variables, **A** is a type and **a** is a simply typed λ-term which encapsulates the derivational semantics. We show below some examples of multimodal deduction rules.

$$\frac{}{x : A \vdash A \triangleright x}\,Ax \qquad \frac{\Gamma \vdash A/_iB \triangleright f \quad \Delta \vdash B \triangleright b}{(\Gamma, \Delta)^i \vdash A \triangleright (f\ b)}\,/_iE \qquad \frac{(\Gamma, x : B)^i \vdash A \triangleright f}{\Gamma \vdash A/_iB \triangleright \lambda x.f}\,/_iI \qquad \frac{\Gamma \vdash A \triangleright a \quad \Delta \vdash B \triangleright b}{(\Gamma, \Delta)^i \vdash A \bullet_i B \triangleright (a, b)}\,\bullet_iI$$

The core logic has very limited expressive power. Strength of multimodal grammars stems from the possibility to add different packages of refined and constrained structural rules. For example, we can assume that commutativity is locally accessible when the resources are combined using a particular mode **c**. Therefore, we are able to account for the freedom of word order required by some linguistic constituents such as adjuncts (e.g., '*yesterday*' whose type $s/_cs$ allows the derivation of both sentences 'Yesterday Houda slept peacefully' and 'Houda slept peacefully yesterday').
More powerful structural rules can be defined to allow the communication between different modes. The rule **MC(i, j)** (rule of controlled contraction) is an example of such **interaction principles**.

$$\frac{\Gamma[((\Delta_1, \Delta_3)^j, (\Delta_2, \Delta_3)^j)^i] \vdash C \triangleright x}{\Gamma[((\Delta_1, \Delta_2)^i, \Delta_3)^j] \vdash C \triangleright x}\,MC(i, j)$$

3 The Icharate toolkit

In [Esslli04], we presented a first version of the meta-linguistic $\mathcal{ICHARATE}$ toolkit. This first formalization was restricted to the syntactic level of multimodal grammars; moreover, it suffered from several shortcomings (loss of readability, absence of a user friendly interface ...). In this paper, we present three main improvements of our toolkit[1] over the previous version.

3.1 Towards a two-level approach to semantics

We use **Coq** as a meta-language in order to deeply embed our object language, namely multimodal logic. This deep embedding is obtained by formalizing the data structures handled by multimodal logic (e.g., syntactic types, contexts, structural rules, deduction rules, λ-terms) in terms of inductive types.
We adopt a two-level approach to semantics: the deep-embedding allows us to reason about the structure of semantics, while we rely on the shallow embedding to reason about the semantic contents of expressions.
We use a deep-embedded λ-calculus to compute the derivational semantics of a sentence given its syntactic derivation. This semantics is computed in a compositional manner by means of a recursive function which maps each deduction step into a computational step within the simply typed λ-calculus. For example, the introduction of both connectives $/_i$ and $\backslash_i$ is semantically interpreted as the abstraction of a hypothetical resource. Deep embedding of λ-terms is worthwhile as it allows us to reason about the syntactic structure of the derivational semantics. For instance, we are able to prove in a formal way that the derivational semantics of a sentence is linear (each binder binds exactly one variable) if its syntactic derivation is established inside a grammar with linear structural rules.
The shallow embedding corresponding to a deeply embedded term is computed by a recursive translation function which translates this term into **Coq** inherent logic. This translation can act as a pretty printer as it replaces all occurrences of De Bruijn indices with named variables. It is also used to reason about natural languages semantics thanks to the existing tools of the meta-language (tactics, induction schemes ...). For example, we are able to establish in a formal way the validity of meta-mathematical sentences such as '*Every positive integer has a predecessor*' whose translation into **Coq**'s logic yields the provable formula
'$\forall$ *(x:nat), x >0* $\Rightarrow \exists$ *y:nat| x=y+1*'.

3.2 Reasoning on Classes of Multimodal Grammars

Coq is based on higher order logic, thus offering an adequate environment for the specification and proof of generic syntactic and semantic properties of multimodal grammars. Moreover, it provides a framework which harmoniously combines two complementary paradigms namely reasoning and computation. Thanks to reflection techniques [Alvarado02], we are able to benefit from the complete

[1] The current version of $\mathcal{ICHARATE}$ is available at *www.labri.fr/˜anoun/Icharate*

automation of the latter tool.
In order to understand the behavior of different interaction principles, we need to enhance the restricted set of inference rules by a range of derived rules which can be applied to specific classes of grammars (e.g., grammars containing a specific structural rule, grammar whose structural rules verify a given predicate). These derived rules are valuable in that they can help us study the expressive power of the different interaction principles, together with their ability to explain various linguistic phenomena. $\mathcal{ICHARATE}$'s library contains a wide range of derived rules dedicated to the processing of various linguistic phenomena such as unbounded dependencies, crossed dependencies, ellipsis and parasitic gaps ... [Moortgat97]. For instance, we can account for parasitic gaps within a setting that contains the structural rule **MC(i, j)** using the following derived rule:

$$\frac{MC(i,j) \in R \ \ (a : A\ ,\ c : C)^i \overset{R}{\vdash} D \rhd f}{(f_1 : A/_j B\ ,\ f_2 : C/_j B)^i \overset{R}{\vdash} D/_j B \rhd \lambda b.f[a := (f_1\ b), c := (f_2\ b)]}\ PG$$

This generic rule simplifies the derivation of the noun phrase *'a book which Mary did review _ without reading _ '* which illustrates the parasitic gaps phenomenon (both extracted elements are semantically dependent). The main steps of the relative clause's derivation are summarized below:

$$\frac{\dfrac{\dfrac{}{(a : np\backslash_i s, c : (np\backslash_i s)\backslash_i(np\backslash_i s))^i \vdash np\backslash_i s}}{(did\ review : (np\backslash_i s)/_j np, without\ reading : ((np\backslash_i s)\backslash_i(np\backslash_i s))/_j np)^i \vdash (np\backslash_i s)/_j np}\ PG}{\dfrac{\vdots}{(Mary : np\ ,\ (did\ review : (np\backslash_i s)/_j np\ ,\ without\ reading : ((np\backslash_i s)\backslash_i(np\backslash_i s))/_j np)^i)^i \vdash s/_j np}}$$

The whole derivational semantics encapsulated by this derivation is nothing else but the term 'λx. **without_reading**(x, **did_review**(x))(**Mary**)'.

$\mathcal{ICHARATE}$'s library also contains a number of refutation lemmas which aim at proving the ill-formedness of some sentences within a given multimodal setting. For example, using a simple polarity computation [Esslli04] we are able to prove that the expression *(*which Mary did review the book*) is ungrammatical.

3.3 User-friendly Interface

In order to have a handy toolkit, we provided $\mathcal{ICHARATE}$ with a user friendly interface which guarantees both readable notations and ease of interactive proof construction. These requirements are achieved by means of **PCoq** interface [Bertot01]. This system enables us to define various pretty-printers that can generate mathematical readable notations ($\vdash$, $\bullet_i$, $/_i$, $\diamond_j$,...). Moreover, it allows the interactive construction of proofs through mouse clicks on some relevant places (*proof_by_pointing* technique). Such clicks run a collection of tactics which simplify the current goal by generating a range of sub-goals potentially easier to solve. Figure 1 illustrates the use of some proof-by-pointing rules involved in the proof of a simple derived rule within the multimodal setting.

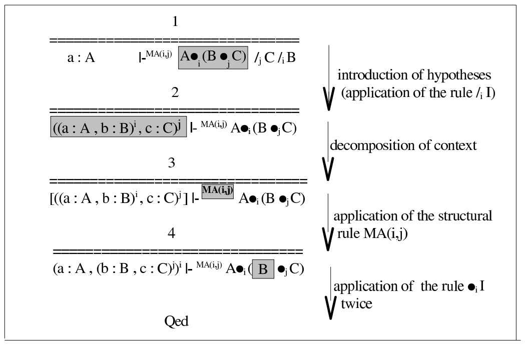

1
a : A ⊢^MA(i,j) A●_i(B ●_j C) /_j C /_i B
introduction of hypotheses (application of the rule /_i I)
2
((a : A , b : B)^i , c : C)^j ⊢^MA(i,j) A●_i(B ●_j C)
decomposition of context
3
[((a : A , b : B)^i , c : C)^j] ⊢^MA(i,j) A●_i(B ●_j C)
application of the structural rule MA(i,j)
4
(a : A , (b : B , c : C)^j)^i ⊢^MA(i,j) A●_i(B ●_j C)
application of the rule ●_i I twice
Qed

Fig. 1. Example of derivation using proof-by-pointing

4 Conclusions & Future work

$\mathcal{ICHARATE}$ is a meta-linguistic toolkit dedicated to the study of multimodal logics. The current version allows users to interactively prove syntactic and semantic properties of different classes of grammars. The adopted approach provides the user with a convenient means of reasoning about both the structure of semantics and its contents. Moreover, $\mathcal{ICHARATE}$ comes with an important range of tactics which allow the combination of manual steps and automatic steps, thus freeing users from the burden of carrying out the proof of some technical goals.
Our toolkit can be extended by formalizing other linguistic models and trying to build bridges between them. We also intend to define more powerful refutation tactics, decision procedures and automation tools to ease the interaction with the toolkit

References

[Alvarado02] Alvarado, C.: Réflexion pour la réécriture dans le calcul des constructions inductives PhD thesis, Paris Sud University, Orsay, (2002)
[Esslli04] Anoun, H., Castéran, P., Moot, R.: Proof Automation for Type-Logical Grammars Technical report, ESSLLI, Nancy, http://esslli2004.loria.fr, (2004)
[Bertot01] Amerkad, A., Bertot, Y., Pottier, L., Rideau, L.: Mathematics and Proof Presentation in Pcoq INRIA Reseach report, (2001)
[CoqA04] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The CIC, Springer Verlag, (2004)
[Gamut91] GAMUT, L.T.F.: Language and Meaning, Vol 2: Intentional Logic and Logical Grammar University of Chicago Press, (1991)
[Moortgat97] Moortgat, M.: Categorial Type Logic Chapter 2 in Van Benthem & ter Meulen (eds) Handbook of Logic and Language, Elsevier (1997)

A Hierarchical Logic for Network Configuration

Roger Villemaire, Sylvain Hallé, Omar Cherkaoui, and Rudy Deca [⋆]

Université du Québec à Montréal
C. P. 8888, Succ. Centre-Ville, Montréal, Canada H3C 3P8
{villemaire.roger,cherkaoui.omar}@uqam.ca

Abstract. We describe ongoing work on a logic on trees aimed at describing properties of configurations of network equipments. This logic generalises first-order logic to a setting where variables form a forest instead of a set. We describe the motivation and the formal approach taken by this logic. Finally be briefly present the kind of sentence for which a generalised Herbrand universes construction leads to a finite structure.

1 Introduction

The configuration of a computer network naturally forms a forest of trees whose nodes contain parameter-value pairs, such as the one shown in Figure 1. Note that a parameter's name, such as `router` is fixed, since it depends on the structure of the software running on the network equipment. On the other hand, the values, such as `router1` (the name of the router), are configurable.

The main network configuration problem is to assure appropriate values for the network equipments' parameters. This hierarchical structure follows the mode/sub-mode hierarchy of the command line interface used to configure devices.

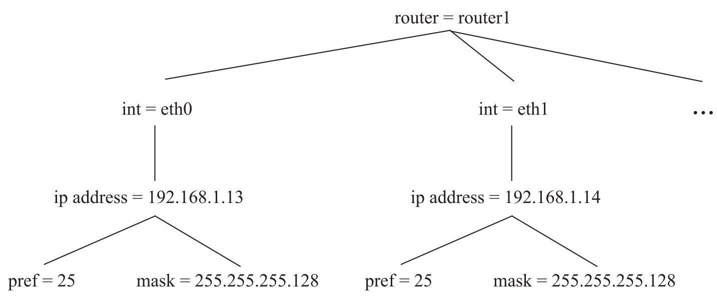


Fig. 1. A portion of the configuration tree for a router

[⋆] We gratefully acknowledge the financial support of the National Science and Engineering Council of Canada.

1.1 Two Simple Examples

Example 1: IP addresses A version-4 IP address is a string of four bytes that uniquely identifies components in a network. According to the Classless Inter-Domain Routing (CIDR) scheme [7, 18], each IP address is described by two attributes, its mask and prefix, that are linked by a relation. For instance, an address like `206.13.01.48/25`, having a network prefix of 25 bits, must carry a mask of at least `255.255.255.128`, while the same address with a network prefix of 27 bits must not have a subnet mask under `255.255.255.224`. This relation must be verified for all IP addresses in all routers of a network such as the one depicted in Figure 1.

Example 2: Virtual Private Networks While the first example showed a property between nodes of a single router, we now consider a property between nodes of *different* routers.

A *Virtual Private Network* (VPN) service [17, 19, 20] is a private network constructed within a public network such as a service provider's network. Usually, the VPN is used to link together several geographically dispersed sites of a customer by a protected communication throughout the provider's network.

Most of the configuration of a VPN is realised in routers placed at the border between the client's and the provider's networks. On the client side, these routers are called *customer edge* (CE) routers, and on the provider side, they are called *provider edges* (PE).

An important issue is to ensure the transmission of routing information between the sites forming a VPN without making this information accessible from the outside. One frequently used method consists in using the Border Gateway Protocol (BGP). This method involves the configuration of each PE to make it a "BGP neighbour" of the other PEs [17]; this entails that one interface in each PE router must have its IP address declared as a BGP neighbour in each other PE router. A typical VPN service involves tens of routers in which an average of 10 dependencies must be checked at multiple locations in the tree.

1.2 Aim of a Configuration Logic

The above examples are representative of configuration situations that arise in network management. The usefulness of logic in this setting is to have a formalism not only to describe properties that must be fulfilled by a correct configuration but also to provide an algorithm to help automate the configuration task.

Such a logic should first have decidable model checking in order to check that a configuration actually fulfills some necessary properties. Furthermore, in order to help automate the configuration task, one should able to build a finite configuration satisfying some property. This last requirement includes an algorithm to find (if possible) acceptable values for new parameters of a configuration in such a way that some conditions, assuring proper function of the network, are satisfied. In logical terminology, this entails the logic should be decidable and have the small model property.

A logic for configuration could be based on first-order logic, with a binary relation to encode the tree structure. But in order to get a decidable logic, one has to restrict oneself to some fragment of first order logic.

Decidable fragments of first-order logic obtained by limiting the types and alternations of quantifiers have been thoroughly studied [6, 14]. But since [8] showed that the class of $\forall^2\exists^*$ (two $\forall$ followed by any number of $\exists$ in prenex form) with one binary predicate is undecidable, this considerably limits the expressiveness of such fragments.

Another approach would be to consider the tree structure to be a part of the logic. In the context of network configuration, quantifiers are needed to express things like, "for all routers r, there exists an interface i of r"; however, general quantification on *all* nodes (as in "for all parameters") is never necessary. The situation is therefore similar to multi-modal logics like LTL and CTL [5] where the operators $\langle a \rangle$ and $[a]$ respectively denote "there exists an action a" and "for all actions a". But for configurations, a would rather be a parameter (with a value) and contrary to multi-modal logics, relations on tuples of nodes are needed.

Guarded logic is, as a matter of fact, a fragment of first-order logic allowing n-ary relations, which generalise multi-modal logic. Guarded logic and a further generalisation called weakly-guarded logic have been showed to have the small model property [1, 11]. Unfortunately, it has been shown in [21] that neither guarded nor weakly guarded logics are sufficient for describing properties of configurations.

Since a configuration is a set of trees, another possible approach is to use query logics for tree-structured documents like the Tree Query Logic (TQL) [2, 3]. This logic was used in [9, 10] to verify configuration properties of network services, so the logic is appropriate, even if natural network properties tend to be somewhat cumbersome to express. Nevertheless, TQL is a very powerful query logic, which can express recursion (which is of no need in our setting) and therefore has undecidable model checking [4]. Since in [9, 10] only a very small fragment of TQL was used, it is interesting to look at a fragment of TQL which would be sufficient for describing properties of configurations. This was done in [21] where a configuration logic named CL was introduced.

There are many more applications of logic to configurations which have been proposed in the literature, for instance [15], [22], [13], [16] or [12]. But neither [15], [16] nor [22] considers hierarchical parameter-value pairs, while [13] considers a logic on trees, but its quantifiers are general first-order quantifiers. Moreover, none of these works (except for [16]) considers model construction.

2 Recasting Configuration logic

Even if [21] shows that CL doesn't have the small model property, one could ask for a reasonable fragment of it, expressive enough for usual network properties, but allowing a small model construction. While the approach of [21] was sufficient to do model-checking, it was not clear how one could build models in that setting.

One possible way to construct a model is to generalise Skolem functions and Herbrand universes. While Herbrand universes do not have to be in general finite, one could expect a natural syntactic restriction on formulas which would be sufficient for finiteness.

In order to generalise Skolem functions to CL one has to introduce a proper notion of function symbol in CL. The most natural way to do this is to consider CL as a logical language in which the set of variables forms a forest (set of trees). A variable x represents a couple formed of a parameter name (which if fixed) and a value (which has to be given by a valuation).

Consider a fixed set $Names$ of possible *parameter names* (for short just names). The variables in CL must be a structure of the following kind.

Definition 1. *A* named forest *is a set of trees whose nodes are labelled by names. If N is a node of a forest, we write $label(N)$ for its label.*

A valuation sending variables to nodes in a configuration has to preserve the hierarchical structure, so it must fulfill the following definition.

Definition 2. *A* named forest morphism *(n.f.m. for short) $\alpha : F_1 \rightarrow F_2$ from a named forest F_1 to a named forest F_2 is a partial function from the nodes of F_1 to the nodes of F_2 (with domain $dom(\alpha)$) such that:*

- *$dom(\alpha)$ is a sub-forest (i.e. the parent of an element of $dom(\alpha)$ is also in $dom(\alpha)$).*
- *if N is a root, then $\alpha(N)$ is a root*
- *$label(\alpha(N)) = label(N)$*
- *if N_2 is a child of N_1, then $\alpha(N_2)$ is a child of $\alpha(N_1)$.*

The appropriate generalisation of function symbols to CL is to consider a symbol of the form $v.f(\bar{w})$, where v is a variable and $\bar{w}$ a finite sequence of variables. A valuation ρ will interpret $v.f(\bar{w})$ to be a descendant of $\rho(v)$.

We have succeeded in generalising to this setting a formal definition of semantics and a sound notion of terms and substitutions. This opens the way to a generalisation of Skolem functions and Herbrand universes constructions to CL.

The remaining question is: under which circumstances is such a Herbrand universe finite? A complete characterisation is still the topic of ongoing work, but we succeeded in obtaining a sufficient condition which is true whenever an existentially quantified variable is not in the scope of a universal variable of the same type. For instance in example 2, the neighbour entry depends on the existence of an interface on another router and not on the existence of another neighbour entry in the same router.

References

1. Andréka, H., van Benthem, J., Németi, I.: *Modal Languages and Bounded Fragment of Predicate Logic.* ILLC Research Report ML-96-03, 59 pages, 1996.

2. Cardelli, L.: *Describing semistructured data.* SIGMOD Record, vol. 30, no. 4, pp. 80-85, 2001.
3. Cardelli, L., Ghelli, G.: *TQL: A query language for semistructured data based on the ambient logic.* Mathematical Structures in Computer Science, vol. 14 , no. 3, pp. 285-327, 2004.
4. Charatonik, W., Talbot, J-M.: *The Decidability of Model Checking Mobile Ambients.* Proceedings of the 15th International Workshop on Computer Science Logic, pp. 339-354, September 10-13, 2001.
5. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking.* MIT Press, Cambridge, MA, 2000.
6. Dreben, B., Goldfarb W. D.: *The Decision Problem: Solvable Classes of Quanticational Formulas.* Addison-Wesley, Reading, MA, 1979.
7. Fuller, V., Li, T., Yu, J., Varadhan, K.: *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy.* RFC 1519, 1993.
8. Goldfarb, W. D.: *The Unsolvability of the Gödel Class.* Journal of Symbolic Logic, vol. 49, pp. 1237-1252, 1984.
9. Hallé, S., Deca, R., Cherkaoui, O., Villemaire, R.: *Automated Validation of Service Configuration on Network Devices.* Proceedings of Management of Multimedia Networks and Services: 7th IFIP/IEEE International Conference, (MMNS 2004), San Diego, CA, USA, October 2004, LNCS 3271, pp. 176-188, 2004.
10. Hallé, S., Deca, R., Cherkaoui, O., Villemaire, R., Puche, D.: *A Formal Validation Model for the Netconf Protocol.* Proceedings of Utility Computing: 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, (DSOM 2004), Davis, CA, USA, November 15-17, LNCS 3278, pp. 147-158, 2004.
11. Hodkinson, I. M.: *Loosely Guarded Fragment of First-Order Logic has the Finite Model Property.* Studia Logica, vol. 70, no. 2, pp. 205-240, 2002.
12. Jackson, D.: *Automating First-Order Relational Logic.* Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering, San Diego, California, ACM Press, New York, NY, pp. 130-139, 2000.
13. Klarlund, N., Koistinen, J., Schwartzbach M. I.,: *Formal Design Constraints*, SIGPLAN Notices, vol. 31, no. 10, pp. 370-383, ACM Press, New York, NY, 1996.
14. Lewis, H. R.: *Unsolvable Classes of Quanticational Formulas.* Addison-Wesley, Reading, MA, 1979.
15. Lowe, H.: *Extending the Proof Plan Methodology to Computer Configuration Problems.* Applied Artificial Intelligence, vol. 5, no. 3, pp. 227-252, 1991.
16. Narain, S.: *Network Configuration Management via Model Finding.* ACM Workshop On Self-Managed Systems, October 31-November 1, 2004, Newport Beach, CA.
17. Pepelnjak, I., Guichard, J.: *MPLS VPN Architectures.*, Cisco Press, 2001.
18. Rekhter, Y., Li, T.: *An Architecture for IP Address Allocation with CIDR.* RFC 1518, 1993.
19. Rosen, E., Rekhter, Y.: *BGP/MPLS VPNs.* RFC 2547, 1999.
20. Scott, C., Wolfe, P. Erwin, M.: *Virtual Private Networks.* O'Reilly, 1998.
21. Villemaire, R., Hallé, S., Cherkaoui, O.: *Configuration Logic: A Multi-Site Modal Logic.* Proceedings of the 12th International Symposium on Temporal Representation and Reasoning, TIME 2005, June 23-25, Burlington, Vermont, USA, 2005.
22. Zeller, A., Snelting, G.: *Unified Versioning through Feature Logic*, ACM Transactions on Software Engineering and Methodology, vol. 6, no. 4, pp. 398-441, ACM Press, New York, NY, 1997.

LPAR

On a mission, to boldly
go where no reasonable
conference has gone before