Liet. matem. rink., T. 43, spec. nr., 423–428

Automatization of proof-search for a fragment of the first-order linear tense logic

Romas ALONDERIS (MII)
e-mail: romui@ktl.mii.lt

1. Introduction

Various modal and tense logics have been investigated lately for the possibility to apply them in computer science: namely for program construction and verification. The first-order linear tense logic is one of them. As it is known, this logic is neither complete nor decidable. However, some fragments of the logic are complete or/and decidable, see [1, 2, 3, 4]. One such fragment (decidable and complete) is presented in [3]. The author of the present paper has written a program *prover.exe* in the Pascal programming language, implementing the decision procedure given in [3]. The aim of the present paper is to discuss the program *prover.exe*.

The paper is organized as follows: first, we present theoretical results, following from the work [3]; then, it is discussed the work of the program *prover.exe*, performing the syntactical analysis and proof-search. In the end, we give an evaluation of the complexity of the program *prover.exe*.

2. Theoretical results

In this section, we present theoretical results which follow from the work [3], We use usual denotation of the formulas, logical connectives, quantifiers and temporal operators. Operator "next", for example, is denoted by '$\circ$', and operator "always" – by '$\Box$'.

Further, we define primary sequents. The general shape of a primary sequent is: $\Sigma, \Box\Omega \rightarrow \theta(\vee_i F_i)$. Here Σ and Ω are multisets of formulas; Σ is $\emptyset$ or consists of atomic formulas; $\Box\Omega$ is $\emptyset$ or consists of the formulas (called kernel-formulas) of the following shape: $\Box\forall x(P(x) \supset \circ Q(x))$ (P and Q are arbitrary atomic formulas); θ is $\emptyset$ or '$\Box$'; F_i is $\circ^n A$ or $\exists x \circ^n A(x)$, here $n \geqslant 0$ and A is an arbitrary atomic formula.

We assume that there are no function symbols and that all predicate symbols are of arity one or zero.

Also, every primary sequent has to satisfy the periodicity condition, which is defined as follows. Assume that $S = \Sigma, \Box\Omega \rightarrow \theta(\vee_i F_i)$ is some sequent (with antecedent and succedent defined as above). Two kernel-formulas are called identical if they are the same or differ only in the bound variables. By dropping kernel-formulas, we can get $\Box\Omega'$ from $\Box\Omega$, such that there are no identical kernel-formulas in $\Box\Omega'$. The sequent S is called periodic if:

1. For every kernel-formula in $\Box\Omega'$ in the antecedent of which is a formula B, there is a kernel-formula in $\Box\Omega'$ in the consequent of which is the formula $\circ B$ (it can be the same kernel-formula, e. g., $\Box\forall x(B(x) \supset \circ B(x))$) and
2. There is no $\Box\Omega'_1 \subset \Box\Omega$, where $\Box\Omega'_1 \neq \Box\Omega'$, such that $\Box\Omega'_1$ would satisfy condition 1.

Condition 2 can be replaced by the following one:

2'. Antecedents and consequents of every two kernel-formulas in $\Box\Omega'$ are different.

It can be seen that if condition 1 holds, then conditions 2 and 2' are equivalent.

So that to determine if a primary sequent is valid, we use a procedure, let us call it proof-search, consisting of applications of the following two procedures: procedure IS (Iterated Saturation) and GIS (Generalized Iterated Saturation), see [3]. Now we describe these procedures.

IS: Suppose that $S = \Sigma, \Box\Omega \rightarrow \vee_i F_i$ is a primary sequent, and $\Box\Omega$ consists of the following kernel-formulas: $\Box\forall x(P_1(x) \supset \circ Q_1(x)),\ldots, \Box\forall x(P_m(x) \supset \circ Q_m(x))$. Let us assume, further, that

$$P_1(a_1^1), \ldots, P_1(a_{k_1}^1),$$
$$\ldots$$
$$P_n(a_1^n), \ldots, P_n(a_{k_n}^n),$$

where $0 \leqslant n \leqslant m$, are all formulas in Σ in which the predicate symbols $P_1,\ldots, P_n$ occur. Then, applying IS to the sequent S, we get the sequent $S_1 = \Sigma_1, \Box\Omega_1 \rightarrow \vee_i F_i^1$. Here $\Box\Omega_1 = \Box\Omega$. Σ_1 consists of the formulas

$$Q_1(a_1^1), \ldots, Q_1(a_{k_1}^1),$$
$$\ldots$$
$$Q_n(a_1^n), \ldots, Q_n(a_{k_n}^n).$$

Formula F_i^1 is obtained from F_i by dropping one $\circ$; if there is no $\circ$ in F_i, then the whole formula F_i is dropped, i. e., $F_i^1 = \emptyset$.

GIS: Suppose that $S = \Sigma, \Box\Omega \rightarrow \Box(\vee_i F_i)$ is a primary sequent. Applying procedure GIS to the sequent S, we get two sequents S_1 and S_2. $S_1 = \Sigma, \Box\Omega \rightarrow \vee_i F_i$, and S_2 is obtained from S as follows: the antecedent is obtained in the same way as applying IS to the sequent $\Sigma, \Box\Omega \rightarrow \vee_i F_i$, and the succedent remains unchanged.

A Primary sequent $\Sigma, \Box\Omega \rightarrow \vee_i F_i$ is called an axiom if and only if there is a formula $P(a)$ (or P) in the multiset Σ, such that, for some i, $F_i = P(a)$ ($F_i = P$) or $F_i = \exists x P(x)$.

The proof-search of a primary sequent S, i. e., determination if S is valid, is performed as follows. First, we perform contraction of the kernel-formulas: from several identical kernel-formulas only one is left. The obtained sequent is equivalent to the sequent S (see [3]), and we denote it also by S. Two main cases are possible further:

1. '$\Box$' does not occur in the succedent of the primary sequent S: $S = \Sigma, \Box\Omega \rightarrow \vee_i F_i$. If S is an axiom, then it is obvious that S is valid. If S is not an axiom and its

succedent or Ω is empty, then it is obvious that S is not valid. In other cases, we apply the procedure IS to the sequent S and obtain a sequent S_1. If there are no atomic formulas in the antecedent of S_1, then, as it follows from the work [3], S is not valid. Otherwise, if S_1 is not an axiom, then further we apply the procedure IS to this sequent; this process is repeated until we get an axiom or there is no '$\circ$' in the succedent of an obtained sequent. It is proved in the work [3] that: 1) if during this process we get an axiom, then the sequent S is valid 2) otherwise, sequent S is not valid.

2. '$\Box$' occurs in the succedent of the primary sequent S: $S = \Sigma, \Box\Omega \rightarrow \Box(\vee_i F_i)$. If $\Omega = \emptyset$, then it is obvious that S is not valid. In other cases, we apply the procedure GIS to the sequent S and obtain two sequents S_1 ('$\Box$' does not occur in the succedent) and S_2 ('$\Box$' occurs in the succedent). Further, as in the case 1., we check if S_1 is valid; if S_1 is not valid, then, as it follows from the work [3], sequent S is not valid; the proof-search is stopped; if S_1 is valid, then, we apply the procedure GIS to the sequent S_2. Suppose that there are k kernel-formulas in the sequent S. If having repeating this process k times we do not get that S is not valid, then, as it follows from the work [3], sequent S is valid.

We get that the class of the primary sequents is decidable.

3. Implementation of the proof-search in Pascal

The procedure given above was implemented in the Pascal programming language. Further, we give denotation and discuss the program itself.

3.1. *Denotation*

1. Predicate symbols are denoted by capital letters, possibly with a digit suffix, e. g., K, $N034$. The length of any predicate symbol is supposed to be no more than 5 symbols.
2. Function symbols are denoted by any of the letters f, g and h, possibly with a digit suffix. The length of any function symbol is supposed to be no more than 5 symbols.
3. Bound variables are denoted by any of the letters x, y, z and w, possibly with a digit suffix. The length of any bound variable is supposed to be no more than 5 symbols.
4. Free variables are denoted by any of the letters a, b, c and d, possibly with a digit suffix. The length of free variables is not restricted.
5. $\exists x = E : x, \forall x = A : x, \supset = =>, \wedge = /\backslash, \vee = \backslash/, \neg = !, \circ = \mathrm{o}, \Box = [], \rightarrow = ->$.

An example of a sequent, written in our denotation system:

$$E; ! [] \mathrm{o}B; []A : x!(R\{x, f\{b, c\}\} => D) -> !\mathrm{o}G\{45\}.$$

Observe that:

1. Formulas in a sequent are separated by semicolons.
2. Arguments of predicate and function symbols are between braces and separated by commas.
3. A number can be an argument of a predicate or function symbol.

It is supposed that there is no more than one formula in the succedent of any sequent.

Now we give the definition of a formula (so that it would be clear how to parenthesize formulas):

a) an atomic formula is a formula;

c) if F is a formula, then $!F$, $E : xF$, $A : xF$, oF and $[]F$ are also formulas;

b) if F and G are formulas, then $(F/\backslash G)$, $(F\backslash/G)$ and $(F => G)$ are also formulas.

The sequent, validity of which is checked, has to be in a file "data.txt". The program reads only the symbols which are between the first two @ signs in this file. Therefore, the file can contain several sequents. It is important only that the sequent investigated is between the first two @ signs. Also, in the "data.txt" file, spaces and line division can be freely used. The program will ignore them.

As far as the primary sequents are concerned, the formula in the succedent is parenthesized as follows: if there is no disjunction in the succedent, then there are no parentheses in it; otherwise, there is a couple of parentheses in the succedent: a closing parenthesis at the end and an opening parenthesis after '->', if there is no '[]' in the succedent, or after '[]', if '[]' is in the succedent.

3.2. *Work of the program*

First, the program checks if the sequent investigated is syntactically correct. Sequents with functions symbols and predicate symbols of arity greater than one are supposed to be syntactically correct because of possible extension of the program in future by including these symbols. The check of syntax is performed in several steps. Suppose that S is the sequent investigated.

1) Program successively reads two adjacent symbols c and cc of the sequent and checks if cc can succeed c. For example, if c is ';', then cc can be '(', '[', '!', '○' or a capital letter. If cc is some other symbol, e. g., '−', then the part of the sequent till cc and an error message are displayed on the screen. In the case of a bound variable, it is additionally checked if x succeeds A : (or E :), or it occurs in a predicate symbol, e. g., $A\{x\}$. These two cases are different because x can be succeeded by '○', '[', '!', a capital letter or digit in the first case, and by ',' , '}' or a digit in the second.

2) Even if no error was found in the step 1), there may be errors in S. For example, suppose that S is $=> B) \rightarrow A;$. The succession of symbols is correct here, however the sequent begins and ends in wrong symbols. Therefore, in addition to succession of symbols, the program also checks the first and last symbols of the sequent. For example, the first symbol of a sequent can be: '(', '[', '!', '○', '−' and a capital letter. If it is not so, then an error message is given.

3) Having performed steps 1) and 2), we are sure that our sequent begins and ends in right symbols and that the succession of symbols is correct. This, however, is not enough.

For example, let us take the sequent $A- > B- > C$. Steps 1) and 2) would not find an error here. Therefore, the program checks also if there is exactly one arrow in the sequent.

4) Further, the program checks braces. Because steps 1) and 2) have already been performed, it is enough to make sure that the numbers of opening and closing braces are equal in each formula. Performing this step, the program will find an error, e. g., in such a sequent: $A\{a, b\}\}- >$. One can see that such an error will not be found in steps 1)–3).

5) Further, the program checks parentheses. The fact that steps 1)–4) have already been performed makes it easier to implement step 5). Parenthesizing of formulas is checked separately in the antecedent and succedent. In the antecedent, parentheses are checked according to the definition of the formula. The formula in the succedent is supposed to be parenthesized so as we have mentioned earlier: if there is no disjunction in the succedent, then there are no parentheses in it; otherwise, there is a couple of parentheses in the succedent: a closing parenthesis at the end and an opening parenthesis after '->', if there is no '[]' in the succedent, or after '[]', if '[]' is in the succedent. Therefore, we will get that the sequent $\rightarrow (A/\backslash B => C\backslash/D)$ is syntactically correct. This, however, is not so important to us because we will get a message that the sequent is not primary and the program will be stopped. In the step 5), it is also checked if there is more than one formula in the succedent. If so, then an error message is given.

6) Also, the following checks are performed in addition. It is checked if all bound variables are bounded by a quantifier. It is checked if all occurrences of the same function and predicate symbol are of the same arity. It is checked if the length of function and predicate symbols do not exceed five.

If a sequent is syntactically correct, then it is checked if the sequent is primary. This check is also performed in several steps.

1) It is checked if the sequent contains symbols which cannot occur in a primary sequent, e. g., '!' or 'f'. Further, it is checked if there are symbols in the antecedent which can occur in the succedent, but cannot in the antecedent, e. g., 'E :'; then, on the contrary, if there are symbols in the succedent which can occur in the antecedent, but cannot in the succedent, e. g., '=>'.
2) Further, the structure of formulas is checked: separately in the antecedent and succedent. For example, the fact that the sequent $A : x((A\{x\} => B\{x\}) => C\{x\})- >$ is not a primary one, would not be found in step 1).
3) In the end, contraction of kernel-formulas is performed, and it is checked if the sequent satisfies the periodicity condition.

This ends syntactical analysis of the sequent S. It assures us that the sequent is a primary one. Next, it is checked if the sequent S is valid. A valid sequent is called derivable.

Here we have two main cases:

(i) The sequent investigated is such that there are kernel-formulas in its antecedent, and '[]' does not occur in its succedent. If it is not so, the program works as it is described in item (ii). Otherwise, the program checks if S is an axiom. If it is not, then the contraction of identical atomic formulas in the antecedent is performed. As it follows from [3], S' is equivalent to S. Further, the procedure IS is applied to the sequent S', and a sequent S_1 is obtained. If there are no atomic formulas in the

antecedent of S_1, then the message that the sequent S is not derivable is given. If S_1 is an axiom, then a message that S is derivable is given. Otherwise, IS is applied further to the sequent S_1. The process (of IS applications) is continued while it is obtained an axiom or a sequent with no'o' in the succedent. In the first case, it is given a message that S is derivable, and in the second – that S is not derivable.

(ii) The sequent investigated is such that there are kernel-formulas in the antecedent and '[]' occurs in the succedent. Then the procedure GIS is applied to the sequent S. As the result, two sequents S_1 ('[]' does not occur in the succedent) and S_2 ('[]' occurs in the succedent) are obtained; further, it is checked – in the same way as it is described in item (i) – if S_1 is derivable. If S_1 is not derivable, then a message that S is not derivable is given. If S_1 is derivable, then the procedure GIS is applied to the sequent S_2. Suppose that there are k kernel-formulas in S. If having repeating this process k times we do not get that S is not derivable, then it is given a message that S is derivable.

3.3. *Complexity evaluation*

Suppose that a sequent S has k kernel-formulas and n is the greatest number, such that o^n occurs in the succedent of S. So that to determine if a sequent S is derivable, k applications of the procedure GIS and $k \cdot n$ applications of the procedure IS is needed, in the worst case. So we get such an evaluation of the proof-search procedure complexity: $T_S(k, n) = k + k \cdot n = O(k \cdot n)$. As the procedures IS and GIS are comparatively simple, the evaluation is good. The work of the program confirms that: the proof-search is performed fast, and does not require much memory.

References

[1] J. Hodkinson, F. Wolter, M. Zakharyaschev, Decidable fragments of first-order temporal logics, *Annals of Pure and Applied Logic*, **106**, 85–134 (2000).

[2] Grädel, I. Walukievicz, Guarded fixed point logic, *Proc. of LICS*, 45–54 (1999).

[3]R. Pliuškevičius. On the completeness and decidability of the Horn-like fragment of the first-order linear temporal logic. *Lithuanian Math. J.*, **41**(4), 477–492 (2001) (in Russian).

[4] F. Wolter, M. Zakharyaschev, Axiomatizing the monodic fragment of first-order temporal logics, *Annals of Pure and Applied Logic*, **118**, 133–145 (2002).

Įrodymo paieškos automatizacija pirmos eilės, tiesinio laiko logikos fragmentui

R. Alonderis

Naudojant Pascalio programavimo kalbą buvo parašyta programa, realizuojanti įrodymo paieškos procedūrą, skirtą vienam pirmos eilės tiesinio laiko logikos išsprendžiamam fragmentui. Darbe yra aptariami kai kurie šios programos veikimo principai, įvertinamas jos sudėtingumas.