Evidence Algorithm and its extensions (extended abstract)

A.A.Letichevsky, J.V.Kapitonova, A.B.Godlevsky, V.A.Volkov

(Glushkov Institute of Cybernetics, National Academy of Sciences of Ukraine)

The use of deductive tools in software verification is one of the promising approaches to make the development of software more reliable and productive. A great importance for successful application of this approach is the possibility of flexible adjustments of deductive tools to subject domains and specific properties of a design process. Existing general theorem provers like Otter, Isabelle, or PVS are based on fixed strategies of proof search and it is not easy to strengthen their possibilities using the knowledge of a subject domain. The idea of using domain oriented strategies for deduction was suggested by A.Slissenko in the paper [11] were a general treatment of language of strategies has been described. Later a strategy description language of this type was implemented for a small domain by a PhD student of A. Slissenko [12].

In this paper an alternative approach for writing specialized strategies of proving statements of logic languages is considered. It is based on the development of multilevel proving machine (PM) for the implementation of a logic calculus used to define the Evidence Algorithm, the main deductive tool for our proof system, and adjusting this machine to subject domains at different levels. The low level of PM is the language APLAN implemented in algebraic programming system APS [9]. APS supports rewriting and solving equations in different subject domains. The next level is the Action Language (AL) supported by simulator of AL designed in APLAN. It is based on a general theory of interaction [10] and supports non-deterministic programming of interactive (transition) systems. The main part of logic calculi can be implemented already at this level (as a non-deterministic program in AL). The highest level is the level of proving machine. The language of PM is a specialization of AL. The main instructions of the language are common actions of environment of PM and mathematical (proof) agent inserted into this environment. The development of environment structure is an important part of a design of adjustable prover. The implementation of PM instructions can be designed at the level of PM or AL or APLAN.

Some details of our approach has been already described in [3, 2]. In this paper we give a refined version of the calculus of Interactive Evidence Algorithm (IEA), more detailed description of PM and a case study: temporal extension of IEA and solution as a standard bench mark the generalized Railroad Crossing problem.

The calculus of Interactive Evidence Algorithm. The main ideas of Evidence Algorithm belong to V.M.Glushkov [7, 8]. The prototypes of a version,

implemented in APS, are A.Degtyarev and A.Lyaletsky calcui [4, 1]. Similar approach is under development in the scope of THEOREMA project [6].

The IEA is a kind of a sequent calculus and makes use of the construction of an auxiliary goal as the main inference step. This makes the algorithm understandable and it can easily be used in cooperation with a mathematician who can correct and control the direction of the search for proofs at each step of nondeterministic choice or control the selection of auxiliary goals, especially when they are taken from the large knowledge base.

The calculus for IEA is represented as a combination of two calculi. The first one is the calculus of conditional sequents, the second is the calculus of auxiliary goals. The inference in the calculus of auxiliary goals is used as a one step inference in the calculus of conditional sequents.

The basis for the calculus is the set of logic formulas of the first order predicate calculus built up from terms, predicate terms, binary propositional connectives, negation and quantifiers. The tools of APLAN can be used to define the signature of terms and predicate terms. Terms are built up from primary terms, which include symbolic atoms and numeric constants. All logic formulas are considered up to some equivalence. This equivalence includes all boolean equations except that of distributivity (this equation is the source of exponential explosion), renaming of bound variables and equations $\neg\exists xp = \forall x\neg p,\ \neg\forall xp = \exists x\neg p$. Ordinary sequents have the syntax $x \Rightarrow y$ where x and y are logic formulas (an assumption and a goal). The inference rules manipulate with ordinary sequents and there are the rules eliminating quantifiers. Free variables appearing as a result are divided into two groups: unknowns and fixed. The main unit of a calculus is a conditional sequent, which has the syntax: (X, s, w, Q) where Q is a conjunction of ordinary sequents, X is a valuation (substitution) that is a partial function from unknowns to terms, s is a partial order called admissibility relation (it is defined on the set of free variables occurring in sequent formulas), w is a conjunction of literals. Semantically a conditional sequent is equivalent to the set of ordinary sequents Q to be proved with X, s, and w denoting some shared information about these sequents. In particular X can be used for substitution for unknowns, s is used to define the admissibility of solutions when unknowns get new values, w is a common part of assumptions. Conditional sequents are also considered up to some equivalence. Two conditional sequents are equivalent if after substitution of variables and adding literal assumptions w to all sequents it will be obtained the set of ordinary sequents with equivalent assumptions and goals.

The axioms of the calculus of conditional sequents are:

$$(X, s, w, u \Rightarrow 1)$$

$$(X, s, w, 0 \Rightarrow P)$$

$$(X, s, 0, Q)$$

$$(X, s, w, 1)$$

Q is a conjunction of sequents.

The rules (from the statements to be proved to axioms):

$$(X, s, w, u \Rightarrow 0) \vdash (X, s, w, 1 \Rightarrow \neg u) \quad (1)$$

$$(X, s, w, u \Rightarrow x \wedge y) \vdash (X, s, w, \ (u \Rightarrow x) \wedge (u \Rightarrow y)) \quad (2)$$

$$(X, s, w, u \Rightarrow x \vee y) \vdash (X, s, w, \neg x \wedge u \Rightarrow y); \quad (3)$$

$$(X, s, w, u \Rightarrow \exists x p) \vdash (X, \mathtt{addv}(s, y), w, \neg(\exists x p) \wedge u \Rightarrow \mathtt{lsub}(p, x := y)) \quad (4)$$

$$(X, s, w, u \Rightarrow \exists x(z)p) \vdash$$
$$(X, \mathtt{addp}(s, (z, y)), w, \quad \neg(\exists x p) \wedge u \Rightarrow \mathtt{lsub}(p, x := y)) \quad (5)$$

$$(X, s, w, u \Rightarrow \forall x p) \vdash (X, \mathtt{addv}(s, a), w, u \Rightarrow \mathtt{lsub}(p, x := a)) \quad (6)$$

$$(X, s, w, u \Rightarrow \forall x(z)p) \vdash (X, \mathtt{addp}(s, (z, a)), w, u \Rightarrow \mathtt{lsub}(p, x := a)) \quad (7)$$

Note that the disjunction and conjunction are considered as commutative operations. Therefore the rules (2) and (3) are non-deterministic and depend on which of two members of the disjunction or conjunction are selected as a goal.

To menage the admissibility relation some bound variables in the formula can be marked by free variables. Especially when a new variable z appears in a formula as a result of quantifier elimination then all outermost occurrences of quantifier formulas Qxp (Q is a quantifier) are changed to $Q(x(z))p$. A conditional sequent is transformed according to the rule (4) if a variable x in $\exists xp$ is not marked. Thus, in the right hand side of the rule there is a symbol y of a new unknown, which is added to the set of variables independently (not in the admissibility relation with any other variable). This is done by the function `addv`. The function `lsub` changes all free occurrences of x in p to y and marks all bounded variables in the outermost quantifier formulas by y as it has been explained above. The rule (5) corresponds to the case when bound variable is already marked by previously appeared free variable. The change of s is done by the function `addp`. The rules (6-7) refer to the universal quantifier in a similar way. The variable a is a new fixed variable.

The next rule is applied when the goal is a literal. This rule is conditional and the condition is the existence of a successful inference in the calculus of auxiliary goals:

$$\frac{\mathtt{aux}(s, 1, w \wedge u \Rightarrow z, 1) \vdash \mathtt{aux}(t, v, x \wedge y \Rightarrow z, P)}{(X, s, w, u \Rightarrow z) \vdash (Y, t, w \wedge \neg z, P)}$$

The formulas of the calculus of auxiliary goals have the syntax

$$\mathtt{aux}(s, v, u \Rightarrow z, P)$$

where s is an admissibility relation, v and u are logic formulas, z is a literal, P is a conjunction of ordinary sequents. If such an inference exists the last formula provides the auxiliary goal for the proof of a given sequent. In this rule z is a literal, P is a conjunction of ordinary sequents, Y is a most general unifier (mgu) of $X(x)$ and $X(z)$, Y is admissible w.r.t. t. To define the admissibility let for any valuation X the occurrence relation $\mathtt{occ}(X)$ consistutes with all pairs (x, v)

such that x occurs in $X(v)$. A valuation X is admissible w.r.t. s if the transitive closure of $s \cup \mathtt{occ}(X)$ is a partial order.

The rule

$$(X, s, w, u \Rightarrow z) \vdash (Y, s, 0, u \Rightarrow z)$$

is applicable if there exists a pair of contrary literals px and $\neg(py)$ in w such that Y is a mgu of x and y admissible w.r.t. s.

In the last rule

$$\frac{(X, s, w, F) \vdash (X', s', w', F')}{(X, s, w, F \wedge H) \vdash (X', s', w, H)}$$

(X', s', w', F') is an axiom.

The rules of the calculus of auxiliary goals are the following:

$$\mathtt{aux}(s, v, x \wedge y \Rightarrow z,\ P) \vdash \mathtt{aux}(s, v \wedge y,\ x \Rightarrow z,\ P)$$
$$\mathtt{aux}(s, v, x \vee y \Rightarrow z,\ P) \vdash \mathtt{aux}(s, v, x \Rightarrow z,\ v \Rightarrow \neg y \wedge P)$$
$$\mathtt{aux}(s, v, \exists x p \Rightarrow z, P) \vdash \mathtt{aux}(\mathtt{addv}(s, a), v, \mathtt{lsub}(p, x := a) \Rightarrow z, P)$$
$$\mathtt{aux}(s, v, \exists x(y)p \Rightarrow z, P) \vdash \mathtt{aux}(\mathtt{addp}(s, (y, a)), v, \mathtt{lsub}(p, x := a) \Rightarrow z, P)$$
$$\mathtt{aux}(s, v, \forall x p \Rightarrow z, P) \vdash \mathtt{aux}(\mathtt{addv}(s, u), v \wedge \forall x p, \mathtt{lsub}(p, x := u) \Rightarrow z,\ P)$$
$$\mathtt{aux}(s, v, \forall x(y)p \Rightarrow z, P) \vdash \mathtt{aux}(\mathtt{addp}(s, (y, u)), v \wedge \forall x(y)p, \mathtt{lsub}(p, x := u) \Rightarrow z,\ P)$$

a is a new constant, u is a new unknown, other functions are defined as previously.

IEA is sound and complete as a first order predicate calculus. Denoting $(w, u \Rightarrow P)$ a conditional sequent with empty valuation and empty admissibility relation we can formulate this result as follows.

Theorem 1. *P is tautology $\Leftrightarrow (1, 1 \Rightarrow P) \vdash Q$, Q is an axiom.*

Proving Machine. The calculus IEA can be considered as a transition system with states represented by conditional sequents. The equivalence of conditional sequents can be implemented by systematic reduction of them to some normal form using corresponding rewriting. The system is not deterministic and can be implemented as a simple AL program with trivial environment by simulator of AL which is the starting point for the development of PM.

The Action Language (implemented in APLAN) has as the main syntactic constructions prefixing $a.P$ where a is an action and P is a program, nondeterministic choice $P + Q$ of two programs P and Q and procedure calls with arbitrary syntax. The semantics of procedure calls is defined by means of unfolding function represented as a system of rewriting rules. The sequential and parallel compositions $(P; Q)$ and $P \| Q$ are defined as procedure calls recursively.

To implement IEA as an AL program one must define the unfolding for procedure call `prove` P where P is a statement or conditional sequent. If P is a statement a procedure call must be unfolded to $\mathtt{prove}(1, 1 \Rightarrow P)$ and farther according to the inference rules considered as conditional rewriting rules.

To have a convenient observation of a proof search, the transitions of a system are labelled by the information about the inference rules applied for doing

these transitions as actions. If a program stopped at a state `prove` Q where Q is an axiom, the sequence of observable actions provides the proof of a source statement. If the system is simulated in interactive mode, a user can do the non-deterministic choices instead of the prover and help it to do the search in promising directions. In the automatic mode the search of a proof is depth-first-search with the volume of the search tree bounded implicitly by the two important parameters: the maximal number of variables introduced by the elimination of a quantifiers and the number of nested auxiliary goals. There are also some simple heuristics used to order the non-deterministic choices for brunching. This approach corresponds to rigid implementation. The change of a rule means the change the entire algorithm.

More flexible implementation can be obtained by splitting the state of a transition system to the composition of agents and environments. This is done so that the current state of an environment includes valuation, admissibility relation, literal assumptions and the assumptions of the left-hand side of a conditional sequent, which is currently transformed by an inference rule. The goal of this sequent is considered as an agent inserted into this environment.

Moreover, conjunctions of goals or conjunction of ordinary sequents can be considered as independent agents as soon as the problem of information exchange between them will be solved. Now the calculus actually has been implemented in this way with sequential processing of conjuncts.

The simulator adjusted to support the proving process is actually the proving machine. The state of environment is the internal state of this machine, the set of actions is the set of machine instructions and the insertion function defines the implementation of these instructions. The agent programs written on a specialized Action Language are interpreted by the proving machine. One of such programs is a program, which implements the IEA. Now we can adjust our machine to different subject domains or develop different algorithms on the suspended machine architecture.

The following is the list of the main actions, the instruction set of the proving machine.

```
Let  <formula>
tell <literal>
ask 0
ask 1
start_aux <literal>
fresh V <prog>
fresh C <prog>
solve <equation>
block <prog>
Mesg <text>
start_lkb <literal>
```

The action `Let` P adds the formula P to the assumptions of the current state, separates literal assumptions from all others and checks for contrary pairs. If they

appeared the result can be represented as non-deterministic choice of several new states. During this action formulas are reduced to normal forms. The action `tell` adds a literal to assumptions without any control. The action `ask0` checks if the conjunction of assumptions is 0. If the check is positive the system falls into the dead-lock (failer) state and the alternative branch must be selected if it exist. The action `ask1` checks if the conjunction of assumptions is not 0.

The action `start_aux` returns to a program all current assumptions to start the process of searching for auxiliary goals. This process is defined in terms of AL and APLAN and finishes by non-deterministic choice for proving different goals.

The action `block` is used for the management of stack mechanisms for exploration branches. The action `Mesg` is the only observable action. It informs about states of a proving process.

Except of internal state the prover has the external component of its state called local knowledge base (lkb). It contains the set of assumptions, which do not change at the time of proving (axioms, definitions, theorems). To update the lkb the the named `Let` statement is used: `Let <name>: <formula>`. The action `start_lkb` is used for the search of auxiliary goals in lkb. To make the search faster lkb has a special structure.

Extensions and specialization of IEA. The simplest way of specialization is the extension of rewriting systems reducing terms and formulas to normal forms. The important extension is introducing types by means of a special predicate $(x : \tau)$. Bounded quantifiers can be reduced by means of rewriting

$$\forall(x : \tau)p = \forall x((x : \tau) \to p)$$

$$\exists(x : \tau)p = \exists x((x : \tau) \wedge p)$$

There are two main points where the support for manipulating with special predicates can be introduced. The first one is the instruction `solve <equation>`. The second one is the generation of auxiliary goals. The first examples are manipulating equalities and inequalities (including arithmetical ones, canonical rewriting, higher order functional definitions), transitive binary relations, set theoretical $\in$, type inference and type checking.

Another extension, which was investigated in the scope of IEA implementation, is the enrichment of a calculus by temporal modalities. This extension has been developed to formalize the requirements for reactive systems represented as attributed transition systems. These systems have not only actions as observable entities but also attributes representing the values changing in time. We use an operator **after** similar to the **next** operator of temporal logic to express local or static properties of a system. A formula $u \to$ `after` $a\ v$ where u and v are predicate logic formulas over attributes, and a is an action means that if at the current moment of time u is true then after performing an action a at the next moment of time the formula v will be true. We distinguish synchronous and asynchronous semantics when define the models for static requirements. In

synchronous case only one transition is allowed at the current moment of time. The asynchronous semantics allows the combination of actions and several rules can be applied to define a transition. We use also the traditional modalities **always** and **sometime** (and $\diamond$) to define dynamic properties like safety or liveness conditions. Special strategies for proving such properties of transition system defined by static transition rules were developed. The main strategy use the induction over transitions for proving **always** properties.

Railroad Crossing Problem. We use the formulation of a generalized railroad problem from [5]. Below is the input text containing the program for the extended PM. This text contain the specification of a problem and two properties of a system, safety and liveness conditions (the last is trivial because it is already assumed in the specifications).

```
(Railroad Crossing Problem)(

    Let Duration Theorem: Forall(p,q,d)(
        always(dur p > d -> q)->always(dur q >= dur p +(-1)*(d+1))
    ); is correct



    Let C1:(d_min>d_close);
    Let C2:(d_close>0);

    Let(WT=d_min+(-1)*d_close);



/* ------------  Environment spec ------------------------ */

    Let CrCm: always(InCr->Exist x (dur Cmg(x) > d_min));

    Let OpnOpnd:always(  dur   DirOp >d_open ->(gate=opened));
    Let ClsClsd:always(  dur ~(DirOp)>d_close->(gate=closed));

/* ------------  Controller spec ------------------------ */


    Let Contr1: always(Exist  x (dur Cmg(x) > WT)  -> ~(DirOp));
    Let Contr2: always(Forall x (WT >= dur Cmg(x)) ->   DirOp );




/* ------------- Safety and Liveness --------------------- */

    prove Safety:always(InCr->(gate=closed));
    prove Liveness: always(Forall x (WT >= dur Cmg x) -> DirOp)

);
```

We use linear discrete time. The temporal modality `always` is considered as a higher order predicate (predicate over predicates) with semantics expressed in terms of timed predicates, and introduce a special strategy for proving `always` statements. Another notion is a temporal functional duration. An expression `dur`P is a time which passed from the last moment when P was false till the current moment of time. This semantics is slightly different from the standard duration calculus semantics and can be expressed in terms of timed predicates. Instead of developing the axiomatic theory of duration calculus we introduce

the useful for a class of problems theorem called Duration Theorem which is used as a standard knowledge item in lkb. This theorem belongs to higher order predicate logic but can be used as a source for auxiliary goals in the slightly modified calculus of auxiliary goals. The proof of safety in the current version of PM is about 1 min.

References

1. M. Morokchovets A. Degtiarev, A. Lyaletsky. Deductive processing of mathematical texts. In *Rewriting Technique and Efficient Theorem Proving, International Workshop 29-31 May 2000*, pages 28–53. Glushkov Institute of Cybernetics, 2000.
2. J. Kapitonova A. Letichevsky and V. Volkov. Reading formalizrd mathematical texts. In *Rewriting Technique and Efficient Theorem Proving, International Workshop 29-31 May 2000*, pages 62–88. Glushkov Institute of Cybernetics, 2000.
3. A.A.Letichevsky, J.V.Kapitonova, V.A.Volkov, A.Chugajenko, V.Chomenko, and D.R.Gilbert. The development of interactive algorithms for a mathematical environment. In Alessandro Armando and Tudor Jebelean, editors, *CALCULEMUS 99, Systems for Integrated Comptuttion and Deduction*, Electronic Notes in Theoretical Computer Sciences http://www.elsevier.nl/locate/entcs, pages 33–50, 1999.
4. A.I.Degtyarev and Lyaletski. Logical inference in sad (in russian). In *Mathematical foundations of artificial intelligence systes*, pages 3–11. Institute of Cybernetics, Kiev, 1981.
5. D. Beauquier and A. Slissenko. A first order logic for specification of timed algorithms: Basic properties and desidable class. Technical Report TR-00-23, University Paris 12, 2000.
6. B. Buchberger. Theory exploration versus theorem proving. In *CALCULEMUS: Systems for Integrated Computation and Deduction, Trento, Italy, 1999, Electronic Notes in Theoretical Computer Science,*. Elsevier, 1999.
7. V. M. Glushkov. On problems of automata theory and artificial intelligence. *Kibernetika*, (2), 1970.
8. V. M. Glushkov, J. V. Kapitonova, A. A. Letichevsky, K. P.Vershinin, and N. P. Maliovany. To the development of a practical formalized language for writing mathematical theories. *Cybernetics*, (2):19–28, 1972.
9. J. V. Kapitonova, A. A. Letichevsky, and S. V. Konozenko. Computations in aps. *Theoretical Computer Science*, 119:145–171, 1993.
10. Alexander Letichevsky and David Gilbert. A model for interaction of agents and environments. In Didier Bert Christine Choppy Peter Moses, editor, *Resent Trends in Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 1999.
11. A. Slissenko. On measures of information quality of knowledge processing systems. *Information Sciences: An International Journal*, (57-58):389–402, 1991.
12. V. V. Tarasov. *Inference search control in expert systems based on explicit metarules of inference search, PhD thesis.* St. Petersburg Institute for Informatics and automation, Russian Academy of Sciences, 1996.

This article was processed using the LaTeX macro package with LLNCS style