The development of interactive algorithms for the Mathematical Environment

A. A. Letichevsky [a] J. V. Kapitonova [a] V. A. Volkov [a]
A. Chugayenko [a] V. Khomenko [a] D. R. Gilbert [b]

[a] *Glushkov Institute of Cybernetics, National Academy of Sciences of Ukraine*
252022, 40 Glushkov ave., Kiev, Ukraine
E-mail: {let,kap,vlad,avch,vh}@d105.icyb.kiev.ua

[b] *City University*
Northampton Square,
London EC1V 0HB, UK
E-mail: drg@soi.city.ac.uk

Abstract

The Mathematical Environment which is under development at the Glushkov Institute of Cybernetics is a system of tools supporting the interactive manipulation of knowledge represented in the form of (formalized) mathematical texts. The system is implemented using a simulator for the Action Language, which has itself been developed using the algebraic programming system APS. The theoretical background of this project is the theory of interaction of agents and environments, constraint solving and the Evidence Algorithm. The main concepts underlying the project and the methodology of its development are explained in this paper in terms of the theory of interaction. The Evidence Algorithm is considered as an example of an interactive algorithm for the Mathematical Environment.

1 Introduction

The Mathematical Information Environment project which is being carried out at the Glushkov Institute of Cybernetics is based on previous investigations into automatic theorem proving which started in 1965 when the first prover for (elementary) group theory was developed [1]. Later, following the ideas of Glushkov [5], the SAD system was developed in the 1980's, based on a language of practical mathematical logic, which is a formalized mathematical language similar to natural language, and the Evidence Algorithm. A brief historical sketch of this activity has been presented in [3].

Our current project renews and re-engineers the main ideas of the SAD system, combining it with contemporary results in the theory of interaction, concurrent constraint programming, algebraic programming and computer algebra. The main ideas of the project have been presented in [13]. They include the following:

1. The development of a formalized mathematical language (FML) which is a subset of a natural mathematical language.

2. The development of a formal semantics of FML based on transformations of the behaviour of a mathematical knowledge base.

3. The development of a structure of the mathematical knowledge base and the representation of the transformational semantics of FML using this implementation.

4. An experimental implementation of FML on top of APS which can be integrated with systems of computer algebra and systems which support distributed computations.

5. The development of tools and their application to real texts in computer science.

In this paper we present a short sketch of the theoretical background of the project, instrumental tools for its development and fragments of specifications of an evidence algorithm. This includes the elements of a theory of interaction of agents and environments, the Action Language and its implementation, a calculus for the Evidence Algorithm and discussion of its extensions.

The experimental part of the project is being developed using the algebraic programming system APS [9] and the Algebraic Programming LANguage APLAN (the source language of the system). The language is based on rewriting logic, and provides for the development of computational algorithms as deductive tools and a semantics which is able to combine different programming paradigms. In many cases APLAN programs can be considered as executable algebraic specifications of algorithms and methods.

2 Agents and environments

The Mathematical Information Environment is a multiagent system comprising agents cooperatively working over a distributed knowledge base. In the development of this environment we use the model of interaction presented in [11,12,10]. In contrast to the main traditional theories of interaction including CCS [15], CSP [7], ACP [2], which are based on an implicit and hence not formalized notion of environment, we study agents and environments as objects of different types. Mathematically agents are represented by means of labelled transition systems with divergence and termination, considered up to bisimilarity. Environments are agents supplied with an insertion function which describes the change of behaviour of an environment after inserting an agent into this environment.

2.1 Agents

Agents are objects which can be recognised as separate from the rest of a world or an environment. They exist in time and space, change their internal state, and can interact with other agents and environments, performing observable actions. Agents can be objects in real life or models of components of an information environment in a computerised world. The notion of *agent* formalizes such objects as software components, programs, users, clients, servers, active components of distributed knowledge bases and so on.

Agents with the same behaviour are considered as equivalent and are identified when reasoning mathematically. The equivalence of agents is characterized in terms of an algebra of behaviors $F(A)$ which is a free two sorted (actions $a \in A$ and behaviors $u \in F(A)$) continuous algebra with approximation and two operations. These are *nondeterministic choice* which is an associative, commutative and idempotent binary operation $u+v,\ u, v \in F(A)$, and *prefixing* $a.u,\ a \in A,\ u \in F(A)$ (like basic ACP). The algebra $F(A)$ is closed relative to the limits (least upper bounds) of the ordered sets of finite behaviors. Finite elements are generated by three termination constants Δ (successful termination), $\perp$ (the minimal element of approximation relation) and the deadlock element 0.

The notion of an abstract agent is introduced as a transition closed set of behaviors. All known compositions in various kinds of process algebras (including parallel and sequential composition) can be then defined by means of continuous functions over the behaviors of agents.

Each behavior $u \in F(A)$ over an action set A can be represented in the form

$$u = \sum_{i \in I} a_i.u_i + \varepsilon \tag{1}$$

where a_i are actions, u_i are behaviors, I is a finite (for finite elements) or infinite set of indices, $\varepsilon = \Delta, \perp, \Delta + \perp, 0$. If all summands in the representation (1) are different then this representation is unique up to the associativity and commutativity of nondeterministic choice.

2.2 Environments

An *environment* E is an agent over an *environment algebra of actions* C with an *insertion function.* The insertion function `Ins` of an environment is a function of two arguments: $\texttt{Ins}(e, u) = e[u]$. The first argument e is a behavior of an environment, the second is a behavior of an agent over an action algebra A in a given state u (the action algebra of agents can be a parameter of the environment). An insertion function is an arbitrary function continuous in both of its arguments. The result is a new behavior of the same environment.

The notion of an environment gives the possibility of defining a new type of agent equivalence which is in general weaker than bisimilarity. This is *insertion equivalence* which depends on an environment and its insertion function. Two agents (in given states) or behaviors u and v are *insertion equivalent* with respect to an environment E, written $u \sim_E v$ if for all $e \in E$ $e[u] = e[v]$. Each agent u defines the transformation $\mathtt{Tr}_u^E : E \to E$ of its environment: $\mathtt{Tr}_u^E(e) = e[u]$ and $u \sim_E v$ iff $\mathtt{Tr}_u^E = \mathtt{Tr}_v^E$. We shall also use the notation $[u]$ for $\mathtt{Tr}_u^E$.

After inserting an agent into an environment, the new environment can accept new agents to be inserted, and the insertion of several agents is something that we will often wish to describe. We shall use the notation

$$e[u_1, \ldots, u_n] = e[u_1] \ldots [u_n]$$

for the insertion of several agents.

3 Constraint machine

The constraint machine considered in this section is the first prototype of the Mathematical Information Environment. It describes the environment in a very abstract way and can be used as its specification. The model is based on the ideas of concurrent constraint programming [17] and describes distributed computations over a constraint store. We restrict ourselves to a simple case which is sufficient to explain the main ideas.

3.1 Constraint agents

Constraint agents are agents over a set of actions interpreted as transformations of a constraint store. The notion of a constraint store considered in this example generalises the notion of memory in imperative programming and is a special case of the constraint store used in concurrent constraint programming (the use of some special version of a general notion of a constraint system).

Constraint store. The state of a constraint store is a finite set or a conjunction of elementary constraints which in our example are logical formulas with the following syntax:

- $x = a,\ x \in V,\ a \in D$;
- $p(t_1, \ldots, t_n)$, p – predicate symbol, $t_1, \ldots, t_n$ – terms with variables from V;
- $\exists x\ P$, $x \in V$, P – conjunction of elementary constraints.

In this definition V is a set of variables, D is a data domain. All algebraic operations and predicates are interpreted over the domain D, so that the values of terms and predicates can be computed as soon as the values of all variables which these terms and predicates depend on are known. The constraints of type $x = a$ are called assignments (to a variable x).

As an example a set of integers as D, arithmetic predicates such as $\leq$, $=$, $\neq$ and linear combinations of variables as terms (linear constraints) can be considered. A more complex example is a field (or a ring) F as a domain for variables and nonlinear constraints (polynomials or rational functions) with the same basic predicates.

Conjunctions of elementary constraints are called constraints. A mapping $\sigma : V \to D$ is called a valuation of variables. A constraint c is called consistent if there exists a valuation σ such that $c\sigma$ is true where $c\sigma$ denotes the result of substituting $\sigma(v)$ instead of all free occurrences of $v \in V$ to c. The set of all consistent constraints will be denoted as `Consc`. The set V is partitioned into the set V_0 of local (internal) variables and the set V_1 of global (external) variables. A constraint c is called internal (external) if it does not contain free occurrences of external (internal) variables. The set of internal (external) constraints will be denoted `Intc` (`Extc`).

Let us define the entailment relation $\vdash$ on the set of constraints so that $c \vdash c'$ iff for each valuation σ from $c\sigma$ is true it follows that $c'\sigma$ is also true. States of a constraint store will be identified with conjunctions of constraints which are in this store and will be considered up to the equivalence of constraints defined in the following way. Constraints c and c' are called equivalent ($c \sim c'$) if for any constraint d, $c \vdash d$ iff $c' \vdash d$. Note that a consistent constraint can contain no more than one assignment to the same variable.

Let us consider the following types of actions over a constraint store: `tell` c, `ask` c, $x := t$ where c is a constraint, $x \in V$, t – term. The algebra A of actions generated by these actions is an algebra with combination (associative and commutative operation over the actions) defined by the following equations:

$$\texttt{tell } c \times \texttt{tell } c' = \texttt{tell } c \wedge c'$$

$$\texttt{ask } c \times \texttt{ask } c' = \texttt{ask } c \wedge c'$$

$$\texttt{tell } c \times \texttt{ask } c' = \texttt{tell } c \times (x := t) = \texttt{ask } c \times (x := t) = \emptyset$$

We also add to A the neutral action δ with equations:

$$\delta \times a = a \times \delta = a$$

Agents over A are called constraint agents.

Constraint machine. The constraint machine is an environment for constraint agents. The state of the machine is an expression $s[u]$ where u is a constraint agent, s is the state of a constraint store which is considered as the local store of agent u inserted to it. Locality means that internal constraints cannot be observed from outside. The existential closure $E(c)$ of a constraint c is introduced in order to hide internal variables. It is defined as $E(c) = \exists(x_1, \ldots, x_n)c$ where $x_1, \ldots, x_n$ are all internal variables which occur free in c. Note that $E(c)$ is an external constraint, and $E(c) \sim$**true** if $c \in \texttt{Intc} \cap \texttt{Consc}$. We shall also extend this closure to agents assuming that $E(u + v) = E(u) + E(v)$, $E(a.u) = E(a).E(u)$, $E(\texttt{tell } c) = \texttt{tell } E(c)$, $E(\texttt{ask } c) = \texttt{ask } E(c)$. The transitions of the constraint machine are defined (in SOS style) in the Figure 1.

The insertion function of the constraint machine implements parallel insertion defined by the equation $s[u][v] = s[u \| v]$. To prove the correctness of this definition note that the bisimilarity of states $s[\Delta]$ and $s'[\Delta]$ is the same as the equivalence of constraints s and s'.

$$\frac{u \overset{\texttt{tell}\ c}{\rightarrow} u',\ (s \wedge c) \in \texttt{Consc}}{s[u] \overset{\texttt{tell}\ E(s \wedge c)}{\rightarrow} (s \wedge c)[u']}$$

$$\frac{u \overset{\texttt{ask}\ c}{\rightarrow} u',\ s \vdash c}{s[u] \overset{\delta}{\rightarrow} s[u']}$$

$$\frac{u \overset{\texttt{ask}\ c}{\rightarrow} u',\ s \not\vdash c,\ s \wedge c \in \texttt{Consc}}{s[u] \overset{\texttt{ask}\ E(c)}{\rightarrow} s[u']}$$

$$\frac{u \overset{x:=t}{\rightarrow} u',\ s \wedge t = d \in \texttt{Consc},\ s \wedge x = d \in \texttt{Consc}}{s[u] \overset{\texttt{tell}\ E(x=t=d)}{\rightarrow} (s \wedge x = d)[u']}$$

$$\frac{u \overset{x:=t}{\rightarrow} u',\ s \wedge x = d \wedge t = d' \in \texttt{Consc},\ s \wedge x = d' \in \texttt{Consc}}{(s \wedge x = d)[u] \overset{\texttt{tell}\ E(x=t=d')}{\rightarrow} (s \wedge x = d')[u']}$$

$$\frac{s \vdash c}{s[\Delta] \overset{\texttt{ask}\ c}{\rightarrow} s[\Delta]}$$

$$s[u + \Delta] = s[u + \Delta] + s[\Delta],\ s[u + \perp] = s[u + \perp] + \perp$$

Fig. 1. Transitions of the constraint machine

Distributed constraint machine. We can forget that $s[u]$ is an environment and consider it as an agent $E(s[u])$ with closed internal constraints. This agent generates only external actions and is called a constraint agent with a local store. Several agents each with their own local store can be inserted to a constraint machine and the state $e = s[u][E(s_1[u_1]), \ldots, E(s_n[u_n])] = s[u \| E(s_1[u_1]) \| \ldots \| E(s_n[u_n])]$ can be considered as a state of a constraint machine with distributed memory $(s_1, \ldots, s_n)$ and shared memory s. Each state, reachable from $e = e[\Delta]$ can be represented in this form. The construction can be iterated and we can obtain a multilevel constraint machine with distributed memory. The top level environment can include the external observer (user of a machine) who can control the process of computation and restrict the nondeterminism of a machine behavior. More complex models can change the insertion function for higher levels of the machine and thus formalize the inclusion of interpreters and control systems for distributed knowledge bases.

An ordinary memory state can be considered as a special type of a constraint store state. It is a state of a type $(x_1 = d_1, \ldots x_n = d_n)$. If conditions are considered as elementary constraints, then imperative agents (programs) can be considered as constraint agents without tell-actions.

The extension of a constraint machine for more a complete description of the Mathematical Environment must include a type system for functions and predicates and algorithms for computing entailment and consistency. These algorithms will use the structure of the knowledge base which has been described by examples in [13].

4 Action Language

The Action Language [12] is used for the syntactical representation of agents (especially constraint ones) as programs. It has a simple abstract syntax parametrised by the syntax of actions and procedure calls and semantics parameterised by the insertion function of an environment.

$$\begin{aligned}\texttt{Prog} ::= \ & \texttt{Act} \ | \ \texttt{TermConst} \ | \ \texttt{ProcCall} \ | \ (\texttt{Prog} + \texttt{Prog}) \ | \ (\texttt{Prog} \| \texttt{Prog}) \ | \\ & (\texttt{Prog}; \texttt{Prog}) \ | \texttt{Loc}(\texttt{SetVar}, \ \texttt{Prog})\end{aligned}$$

`TermConst` are some termination constants, defined in Section 2.1, (at least Stop for Δ should be used). Three main compositions are nondeterministic choice, parallel and sequential composition. The construction `Loc` is used for the description of local variables and components of a distributed environment. The intensional semantics of a program without locality is an agent which can be obtained by means of unfolding procedure calls and defining transitions on a set of program states. The interaction semantics of simple programs can be defined for program agents by means of an insertion function. Local components are considered as environments for programs within them and as agents for environments of higher levels.

The Action Language has been implemented by means of a simulator [4,18], a program which generates all histories of agent activity which are possible as a result of interaction with a given environment. The simulator is implemented as an interactive program using the algebraic programming system APS [9]. The functionalities of the simulator permit forward and backward moves along histories and automatic search for program states with given properties (successful termination, deadlocks and so on).

5 Symbolic computation using algebraic programming

The APS is a programming system based on rewriting logic. The main equivalence relation (basic congruence) on a set of (graph) terms is achieved by means of a set of interpreters for operations which define a basic canonical form. The main strategy of rewriting is one-step syntactic rewriting with postcanonisation by means of reducing the rewritten node to a basic canonical form. All other strategies are the combinations of the main strategy with different ways of navigating over the tree representing a term. Strategies of rewriting can be chosen from a library of strategies or written as procedures or functions in APLAN. The system also contains a built-in unification algorithm which can be used for the extension of the algebraic paradigm to logic programming.

APS supports a number of basic computer algebra tools. Among them are computation with integer and rational numbers of arbitrary accuracy, operations over polynomials in various representations (natural, vector and recursive), differentiation and simple integration, transformations over transcendent functions, algorithms for non-canonical simplification in various algebras.

The APS has been applied to the development of some special computer algebra systems. Among these are

- the applied computer algebra system AIST [19] for teaching mathematics in secondary level schools where the APS was used for the implementation of the mathematical kernel of the system (algebraic and trigonometric simplifications, proofs of identities and solutions of equations),
- the package of tools supporting numeric-analytical computations with functions which are solutions of ordinary differential equations with polynomial coefficients [20],
- tools for constraint logic programming [21], and
- a work-bench for programming and experimenting with critical pairs and completion like algorithms.

The expressivity and flexibility of APLAN facilitates the parameterisation of the AL. All the parameters of the AL are implemented by means of rewriting rules and canonical forms. These parameters include the syntax and semantics of operations in the action algebra, the unfolding function for procedure calls, the intensional semantics, which includes the definition of sequential and parallel compositions for agents, and the definition of insertion function. These definitions are very close to the original mathematical definitions of those parameters. For instance, in order to obtain the implementation of the insertion function for the constraint machine it is sufficient to represent the definitions in Figure 1 in the form of conditional rewriting rules, and to implement the entailment relation of corresponding constraint system.

For special types of constraint systems there exist powerful methods for solving the satisfaction problem and finding concrete solutions for systems of constraints. For example, the solution of linear constraints over integers, especially finite domain constraints (the case when the domains of all variables have lower and upper bounds) is such an area. Computer algebra methods should be used for nonlinear constraints, especially the use of Grobner bases for solving the satisfaction problem for systems of algebraic equations. In simple cases the algorithms are already directly implemented in APS. However the integration of the Mathematical Information Environment with powerful solvers and computer algebra systems is very desirable for the solution of more complex problems, and this is included in our future plans.

6 Evidence algorithm

In this section a specification of the Evidence Algorithm [5,6] is presented in the form of a logical calculus. This specification is a reconstruction of the evidence algorithm as implemented in SAD in the form proposed by A.Degtyarev and A.Lyaletsky. For the simplicity it is formulated only for propositional calculus and first order predicate calculus. The algorithm is a kind of sequent algorithm and makes use of the construction of an auxiliary goal as the main inference step. This makes the algorithm understandable and can easily be used in cooperation with a mathematician who can correct and control the direction of the search for proofs.

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

6.1 Propositional calculus

The elementary objects of the calculus are propositional formulas and sequents. Propositional formulas are considered up to equivalence which is defined by means of all boolean equations except that of distributivity, which is the source of exponential explosion. The function `Can` defined in the Appendix by means of a system of rewriting rules defines the reduction of propositional formulas to a canonical form. The associativity, commutativity and idempotence of conjunction and disjunction as well as the laws of contradiction, exclusive third and the laws for propositional constants are used implicitly in these equations.

(Ordinary) sequents have the syntax $x \Rightarrow y$ where x and y are propositional formulas.

Calculus of auxiliary goals.

Auxiliary goal: $(v, u \Rightarrow z, P)$, where z is a literal, u, v are propositional formulas, P is a conjunction of ordinary sequents (the empty conjunction is 1).

Rules:

$$(v, x \wedge y \Rightarrow z,\ P) \vdash (v \wedge x, y \Rightarrow z,\ P)$$
$$(v, x \vee y \Rightarrow z,\ P) \vdash (v, x \Rightarrow z,\ (v \Rightarrow \neg y) \wedge P)$$

In these laws conjunctions and disjunctions are considered up to commutativity, so it is not necessary to introduce alternatives which exchange x and y.

Lemma 1 . *$(v, u \Rightarrow z,\ P) \vdash (v', u' \Rightarrow z,\ P')$, where u' is literal or $u' = z \wedge u''$.*

Calculus of conditional sequents.

Conditional sequent: $(w, u \Rightarrow P)$, where w is a conjunction of literals, u, P are propositional formulas. Formulas of the calculus are expressions of type (w, Q) where Q is a conjunction of ordinary sequents.

Axioms:

$$(w, u \Rightarrow 1)$$
$$(w, 0 \Rightarrow P)$$
$$(0, Q)$$

Q is a conjunction of ordinary sequents.

Rules:

$$
\begin{aligned}
(w, u \Rightarrow 0) &\vdash (w, 1 \Rightarrow \neg u)\\
(w, u \Rightarrow x \wedge y) &\vdash (w, u \Rightarrow x \wedge u \Rightarrow y)\\
(w, u \Rightarrow x \vee y) &\vdash (w, \neg x \wedge u \Rightarrow y)\\
(w, x \wedge y \Rightarrow z) &\vdash (w \wedge x, y \Rightarrow z)
\end{aligned}
$$

x is a literal.

$$\frac{(1, w \wedge u \Rightarrow z, 1) \vdash (v, z \wedge y \Rightarrow z, P)}{(w, u \Rightarrow z) \vdash (w \wedge \neg z, P)}$$

z is a literal, P is a conjunction of ordinary sequents.

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

(w', F') is an axiom.

Theorem 6.1 . *P is tautology $\Leftrightarrow (1, 1 \Rightarrow P) \vdash Q$, Q is an axiom.*

6.2 Predicate calculus

.

Formulas are also considered up to the renaming of bound variables and equations $\neg\exists xp = \forall x \neg p$, $\neg\forall xp = \exists x \neg p$.

Calculus of auxiliary goals.

Auxiliary goal: $(s, v, u \Rightarrow z, P)$, z is a literal, u, v are predicate formulas, P is a conjunction of sequents, s is a sequence of variables. Two types of variables are distinguished in the sequence s. They are arbitrary constants which appear after removing the universal quantifiers and unknowns which appear after removing the existential quantifiers. The order of variables corresponds to the order of removing quantifiers and is used for the definition of correct (consistent with a given sequence) substitutions for unification: the values of unknowns can depend on those constants which appear before them only.

Rules:

$$
\begin{aligned}
(s, v, x \wedge y \Rightarrow z,\ P) &\vdash (s, v \wedge y,\ x \Rightarrow z,\ P)\\
(s, v, x \vee y \Rightarrow z,\ P) &\vdash (s, v, x \Rightarrow z,\ v \Rightarrow \neg y \wedge P)\\
(s, v, \exists xp \Rightarrow z, P) &\vdash ((s, a), v, \texttt{lsub}(p, x := a) \Rightarrow z, P)\\
(s, v, \forall xp \Rightarrow z, P) &\vdash ((s, u), v \wedge \forall xp, \texttt{lsub}(p, x := u) \Rightarrow z,\ P)
\end{aligned}
$$

a is a new constant, u is a new unknown, `lsub` is a substitution with corresponding renaming of bound variables.

Calculus of conditional sequents.

Conditional sequent: $(X, s, w, u \Rightarrow P)$, X is a valuation (binding) for unknowns, s is a sequence of variables, w is a conjunction of literals, u and P are predicate formulas. Formulas of the calculus are expressions of type (X, s, w, Q) where Q is a conjunction of sequents.

Axioms:

$$(X, s, w, u \Rightarrow 1)$$
$$(X, s, w, 0 \Rightarrow P)$$
$$(X, s, 0, Q)$$

Q is a conjunction of sequents.

Rules:

$$(X, s, w, u \Rightarrow 0) \vdash (X, s, w, 1 \Rightarrow \neg u)$$
$$(X, s, w, u \Rightarrow x \wedge y) \vdash (X, s, w,\ (u \Rightarrow x) \wedge (u \Rightarrow y))$$
$$(X, s, w, u \Rightarrow x \vee y) \vdash (X, s, w, \neg x \wedge u \Rightarrow y);$$
$$(X, s, w, u \Rightarrow \exists xp) \vdash (X, (s, y), w, \neg(\exists xp) \wedge u \Rightarrow \texttt{lsub}(p, x := y))$$
$$(X, s, w, u \Rightarrow \forall xp) \vdash (X, (s, a), w, u \Rightarrow \texttt{lsub}(p, x := a))$$
$$\frac{(s, 1, w \wedge u \Rightarrow z, 1) \vdash (t, v, x \wedge y \Rightarrow z, P)}{(X, s, w, u \Rightarrow z) \vdash (Y, t, w \wedge \neg z, P)}$$

z is literal, P is a conjunction of ordinary sequents, Y is mgu of $x = z$ w.r.t. X, Y is compatible with t.

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

Y is mgu of px and $\neg(py)$ in w (contrary pairs).

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

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

Theorem 6.2 . *P is tautology $\Leftrightarrow (1, 1 \Rightarrow P) \vdash Q$, Q is an axiom.*

6.3 Implementation

In order to implement the algorithms based on the calculi defined above, a simple constraint system is introduced for the constraint store of a constraint machine. In the case of the propositional calculus the elementary constraints are boolean variables or their negations. In the case of the predicate calculus elementary constraints are elementary predicate formulas (literals).

Programs in the AL (constraint agents) use parts of formulas of the calculi above to represent their states. They are considered as procedure calls of the AL. There are three procedures in the AL-program for propositional calculus. Two of them, namely procedures `prove` and `aux` represent the calculus of conditional sequents and auxiliary goals correspondingly. The third procedure `prove_aux` is used to combine the two calculi. The corresponding fragments of the rewriting system for the unfolding operator of the AL-program for the propositional calculus are represented in the Appendix (this is a simplified version, the actual program contains some optimisations based on the recognition of patterns of formulas). This Appendix also represents the rewriting rules for the canonical reduction of propositional formulas. In addition to ask and tell statements, the action algebra of the AL also contains actions of type `Mesg` used to inform the user about the proofs of formulas. One of such proofs for the formula

```
(x & y |/ ~(x) & ~(y) <=> ~(x & ~(y) |/ ~(x) & y))
```

is also represented in this Appendix.

The predicate calculus utilises a more complicated structure for the constraint store. The state of this store includes not only elementary formulas introduced as premises by tell statements, but also an ordering sequence for free variables.

6.4 Constraint extensions

The form of the Evidence Algorithm admits easy extension by the introduction of interpreted operations and predicates, especially set theoretical ones for which specialized solvers (which may be not complete but are sufficiently efficient) can be developed instead of unification. This is in some way similar to the introduction of constraint solvers to logic programming systems. For example the implementation of Gauss algorithm in APS for symbolic solving of linear equations, or the solving of satisfaction problem for algebraic equations using Groebner bases technique.

6.5 FML-extension

Formalized mathematical texts written in the Formal Mathematical Language (FML) which is used for the representation of mathematical texts contains special linguistic constructions such as attributes or typed expressions, making the formal language closer to ordinary natural mathematical language. These constructions can be used as an extension of the first order predicate language and also as extensions of the calculus of the Evidence Algorithm. A special point is the use of definitions and the calls to them in the Evidence Algorithm.

7 Initial results and conclusions

Initial implementations of the constraint machine and the Evidence Algorithm have been made using a simulator for the Action Language [4], itself constructed using APS. Experiments have been performed using fragments of mathematical texts as inputs to the Evidence Algorithm program and they have demonstrated that this system is an initial, usable prototype of the Mathematical Information Environment. We plan to extend the system with interfaces to definitions and distributed knowledge bases, to integrate it with existing systems of computer algebras and solvers, and to improve its efficiency.

References

[1] F. V. Anufriev, V. V. Fedurko, A. A. Letichevsky, Z. M. Aselderov, and I. Diduch. On an algorithm for proving theorems in group theory. *Cybernetics*, (1):23–29, 1966.

[2] J.A.Bergstra, J.W.Klop, Process algebra for synchronous communication, Information and Control, 1984, vol. 60, 1/3, 109-137.

[3] Degtyarev A. I., Kapitonova J. V, Letichevsky A. A.,Lyaletsky A. V., Morohovetc M. K., A brief historical sketch on Kiev school on automated theorem proving, Second Int. Theorema workshop, RISC-Linz, Castle of Hagenberg, Austria, 29-30 June, 1998

[4] D.R.Gilbert, A.A.Letichevsky, A universal interpreter for nondeterministic concurrent programming languages, in Maurizio Gabbrielli (ed) Fifth Compulog network area meeting on language design and semantic analysis methods, 1996, Compulog Net.

[5] V.M.Glushkov. On problems of automata theory and artificial intelligence. *Cybernetics*, (2), 1970.

[6] 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.

[7] C.A.R.Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[8] J. V. Kapitonova and A. A. Letichevsky. *On constructive mathematical descriptions of subject domains*, *Cybernetics*, (4):17–25, 1988.

[9] J. V. Kapitonova, A. A. Letichevsky, and S. V. Konozenko. Computations in APS. *Theoretical Computer Science*, 119:145–171, 1993.

[10] A.A.Letichevsky, D.R.Gilbert, Agents and environments, 1st International scientific and practical conference on programming, Glushkov Institute of Cybernetics, National Academy of Sciences of Ukraine, Proceedings 2-4 September, 1998.

[11] A.A.Letichevsky, D.R.Gilbert Toward an implementation theory of nondeterministic concurrent languages, Technical Report, Department of Computer Science, City University, London, ISSN 1364-4009, 1996.

[12] , A.A.Letichevsky, D.R.Gilbert, A general theory of action languages, Cybernetics and System Analysis, 1, 1998, p.16-36.

[13] Letichevsky A.A. Kapitonova J.V Mathematical Information Environment, Second Int. Theorema workshop, RISC-Linz, Castle of Hagenberg, Austria, 29-30 June, 1998

[14] J.Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[15] R.Milner, Communication and Concurrency, Prentice Hall, 1989.

[16] D.M.R.Park. Concurrency and automata on infinite sequences. In *Proc. 5th GI Conf*, volume 104 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[17] V. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[18] T.Valkevych, D.R.Gilbert, A.A.Letichevsky, A generic workbench for modelling the behaviour of concurrent and probabilistic systems, in Rudolf Berghammer and Yassine Lakhnech (eds.), Workshop on Tool Support for System Specification, Development and Verification, at TOOLS98, Malente, Germany, 1998.

[19] M.S. L'vov , A.B. Kuprienko and V.A. Volkov. AIST:Applied Computer Algebra System for Mathematical Training, In Proc. of *The Rhine Workshop on Computer Algebra Application*, (March 22-24, 1994, Karlsruhe, Germany), pp.67–78

[20] A.A.Letichevsky, Denisenko P.N., Bilenko V.I., Volkov V.A. Implementation of numerical-analytical approximation methods of functions which are defined by ordinary differential equations, *Cybernetics and System Analysis*, 1997, (1):108–112.

[21] Y.V.Kapitonova, A.A.Letichevsky, V.A.Volkov, M.S.Lvov, Tools for solving problems in the scope of algebraic programming, in J.Calmet and J.A.Campbell (eds) Integrating Symbolic Mathematical Computation and Artificial Intelligence LNCS v.958 Springer 1995, pp. 30-47.

Appendix

Definition of procedure prove

```
proc_def_rs:=rs(x,y,z,u,v,w,m,P,Q,F,H)(

..................................

/* Calculus of conditional sequents */

    /* (w,u=>1) is an axiom */
    prove(u=>1) = Mesg("prove 1\nevident"),

    /* (w,0=>P) is an axiom */
    prove(0=>P) = Mesg(P" is evident"),

    /* (u=>0) |- (w,1=> ~(u) */
    prove(u=>0) = prove(1=> ~(u)),

    /* (w,u=>x&y) |- (w,(u=>x)&(u=>y)) */
    prove(u=>x& y) = (
        prove(u=>x);
        prove(u=>y)
    ),

    /* (w,u=>x|/y |- (w, ~(y)&u=>x   */
    prove(u=>x|/y) = mrg(
        Mesg("prove "(x|/y)"\nlet " ~(x)).(
            prove(Can(~(x)&u)=>y);
            Mesg((x|/y)" proved")
        )
        +
        Mesg("prove "(x|/y)"\nlet " ~(y)).(
            prove(Can(~(y)&u)=>x);
            Mesg((x|/y)" proved")
        )
    ),

/*
               (1,w&u=>z,1)|-(v,z&y=>z,Q)
             ------------------------------
                   (w,u=>z)|-(w& ~(z),Q)
*/

    prove(u=>z) =
        ask z.Mesg("z is evident")
        +
        aux(1,u=>z,1)
    ),
    prove_aux(z,1) = (
        Mesg(z" is evident")
    ),
    prove_aux(z,Q) = (
        Mesg("prove "z"\nauxiliary goal is " ~(z)=>Q);
        tell ~(z);
        prove Q;
        Mesg(z" proved")
    ),

/*
                      (w,F) |- axiom
                    ------------------
                     (w,F&H)|- (w,H)
*/

    prove(F&H)  = (
        prove(F);
        prove(H)
    ),

/* P is tautology <=> (1,1=>P)|- axiom */
    prove 0 = 0,
    prove P = (
        prove(1=>Can P);
        Mesg("\ntheorem proved")
    )
);
```

Definition of procedure aux

```
proc_def_rs:=rs(x,y,z,u,v,w,m,P,Q,F,H)(

...................................

/* Calculus of auxiliary goals */
    aux(v,z&y=>z,P) = (
        prove_aux(z,Can_aux P)
    ),
    aux(v,z=>z,P) = (
        prove_aux(z,Can_aux P)
    ),
    /* (v,x&y=>z,P) |- (v&y,x=>z,P) */
    aux(v,x&y=>z,P) = (
       aux(x&v,y=>z,P)
       +
       aux(y&v,x=>z,P)
    ),

    /* (v,x|/y=>z,P) |- (v,x=>z,(v => ~(y))&P) */
    aux(v,x|/y=>z,P) = (
        aux(v,x=>z,(v=>Can(~(y)))&P)
        +
        aux(v,y=>z,(v=>Can(~(x)))&P)
    ),
    aux(v,x=>z,Q) = 0,

......................................

);
```

Canonical reduction rules

```
Can:=rs(A,B)(
     (A<=>B) = Can((A->B)&(B->A)),
     (A ->A) = 1,
     (A ->B) = Can(~(A)|/B),
    ~(A<=>B) = Can(~((A->B)&(B->A))),
    ~(A ->B) = Can(A & ~(B)),
    ~(~(A) ) = Can A,
    ~(A&  B) = Can(~(A)|/ ~(B)),
    ~(A|/ B) = Can(~(A)&  ~(B)),
     (A&  B) = mrg(Can A & Can B),
     (A|/ B) = mrg(Can A |/Can B)
);
```

Proof in propositional calculus

```
prove (x & y |/ ~(x) & ~(y) <=> ~(x & ~(y) |/ ~(x) & y))
prove (~(y) |/ ~(x)) & (y |/ x) |/ (~(y) |/ x) & (y |/ ~(x))
    let ~((~(y) |/ ~(x)) & (y |/ x))
    prove ~(y) |/ x
        let ~(x)
        ~(y) is evident
    ~(y) |/ x proved

    prove y |/ ~(x)
        let ~(~(x))
        y is evident
    y |/ ~(x) proved
(~(y) |/ ~(x)) & (y |/ x) |/ (~(y) |/ x) & (y |/ ~(x)) proved

prove ~(y) & ~(x) |/ ~(y) & x |/ y & ~(x) |/ y & x
    let ~(~(y) & x |/ y & ~(x) |/ y & x)
    prove ~(y)
        auxiliary goal is (y |/ ~(x)) & (~(y) |/ ~(x)) => ~(x)
        ~(x) is evident
    ~(y) proved
    prove ~(x)
        auxiliary goal is (~(y) |/ x) & (~(y) |/ ~(x)) => ~(y)
        ~(y) is evident
    ~(x) proved
~(y) & ~(x) |/ ~(y) & x |/ y & ~(x) |/ y & x proved
theorem proved
```