Algebra of behavior transformations and its applications

Alexander LETICHEVSKY

Glushkov Institute of Cybernetics
National Academy of Sciences of Ukraine
Ukraine

Abstract

The model of interaction of agents and environments is considered. Both agents and environments are characterized by their behaviors represented as the elements of continuous behavior algebra, a kind of the ACP with approximation relation, but in addition each environment is supplied by an insertion function, which takes the behavior of an agent and the behavior of an environment as arguments and return a new behavior of this environment. Each agent can be considered as a transformer of environment behaviors and a new kind of equivalence of agents weaker than bisimulation is defined in terms of the algebra of behavior transformations. Arbitrary continuous functions can be used as insertion functions and rewriting logic is used to define computable ones. The theory has applications for studying distributed computations, multi agent systems and semantics of specification languages.

1 Introduction

The topic of these lectures belongs to an intensively developing area of computer science: the mathematical theory of communication and interaction. The paradigm shift from computation to interaction and the wide-spread occurrence of distributed computations attract a great deal of interest among researchers in this area.

Concurrent processes or agents are the main objects of the theory of interaction. Agents are objects, which can be recognized 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 and changing their place among other agents (mobility). Agents can be objects in real life or models of components of an information environment in a computerized world. The notion of agent formalizes such objects as software components, programs, users, clients, servers, active components of distributed knowledge bases and so on. More specifically each notion of agent is also used in so called agent programming, an engineering discipline devoted to the design of intelligent interactive systems.

Theories of communication, interaction, and concurrency have a long history, that starts from the structural theory of automata (50-th years of the last century). Petri Nets is another very popular general model of concurrency. However Petri Nets is very specific, and structural automata theory requires too many details to represent interaction in a sufficiently abstract form. Theories of communication and interaction which appeared in 70-th captured the fundamental properties of the notion of interaction and constitute the basis of modern research in this field. They include CCS (Calculus of Communicated Processes) [19, 20] and the π-calculus [21] of R. Milner, CSP (Communicated Sequential Processes) of T. Hoar [13], ACP (Algebra of Communicated Processes) [5] and many branches of these basic theories. The current state-of-the-art is well represented in recently edited Handbook of Process Algebra [6]. A new look at the area appeared recently in connection with the coalgebraic approach to interaction [24, 4].

The main notion of the theory of interaction is the behavior of agents or processes interacting with each other within some environment. At the same time traditional theories of interaction do not formalize the notion of an environment where agents are interacting or consider very special cases of it. The usual point of view is that an environment for a given agent is the set of all other agents surrounding it. In the theory of interaction of agents and environments developed in [15] and presented in the lectures, the notion of environment is formalized as an agent supplied by an insertion function, which describes the change of behavior of an environment after the agent is inserted. After inserting one agent an environment is ready to accept another one and, considered as an agent, it can itself be inserted into another environment of higher level. Therefore multi-agent and multilevel environments can be created using insertion functions.

Both agents and environments are characterized by their behaviors represented as the elements of a continuous behavior algebra, a kind of ACP with approximation relation [14]. The insertion function takes the behavior of an agent and the behavior of environment as arguments and returns a new behavior of this environment. Each agent therefore can be considered as a transformer of environment behaviors and a new equivalence of agents is defined in terms of the algebra of behavior transformations. This idea comes from Glushkov discrete transformers (processors) [9, 10], an approach considering programs and microprograms as the elements of the algebra of state space transformations. In the theory of agents and environments the transformations of a behavior space is considered instead, so this transition is similar to the transition from point spaces to functional spaces in mathematical analysis.

Arbitrary continuous functions can be used as insertion functions and rewriting logic is applied to define computable ones. The theory has applications for the study of distributed computations and multi agent systems. It is used for the development specification languages and tools for the design of concurrent and distributed software systems. Four lectures correspond to the four main sections of the paper and contain the following.

The first section gives an introduction to the algebraic theory of processes considered as agent behaviors. Agents are represented by means of labeled transition systems with divergence and termination, and considered up to bisimilarity or other (weaker) equivalences. The theorem characterizing bisimilarity in terms of a complete behavior algebra (cpo with algebraic structure) is proved and the enrichment of a behavior algebra by sequential and parallel compositions is considered. The second section introduces algebras of behavior transformations. These algebras are classified by the properties of insertion functions and in many cases can be considered as behavior algebras as well. The enrichment of a transformation algebra by parallel and sequential composition can be done only in very special cases. Two aspects of studying transformation algebras can be distinguished.

Mathematical aspect The study of algebras of environments and behavior transformations as mathematical objects, classifying insertion functions and algebras of behavior transformations generated by them, developing specific methods of proving properties of systems represented as environments with inserted agents.

Application aspect The insertion programming, that is a programming based on agents and environments. This is an answer to the paradigm shift from computation to interaction and consists of developing a methodology of insertion programming (design environment and agents inserted in it) as well as the development of tools supporting insertion programming. The application of the insertion programming approach to the development of proof systems is considered in the last section.

2 Behavior algebras

2.1 Transition systems

Transition systems are used to describe the dynamics of systems. There are several kinds of transition systems which are obtained by the enrichment of an ordinary transition system with additional structures.

An *ordinary transition system* is defined as a couple

$$\langle S, T\rangle, \quad T \subseteq S^2$$

where S is the set of states and T is a *transition relation* denoted also as $s \to s'$. If there are no additional structures, perhaps the only useful construction is the transitive closure of the transition relation denoted as $s \xrightarrow{*} s'$ and expressing the reachability in the state space S.

A *labeled transition system* is defined as a triple

$$\langle S, A, T\rangle, \quad T \subseteq S \times A \times S$$

where S is again a set of states, A is a set of actions (alternative terminology: labels or events), and T is a set of labeled transitions. Belonging to a transition relation is denoted by $s \xrightarrow{a} s'$. This is the main notion in the theory of interaction. We can consider the external behavior of a system and its internal functioning using the notion of labeled transitions. As in automata theory two states are considered to be equivalent if we cannot distinguish them by observing only external behavior, that is actions produced by a system during its functioning. This equivalence is captured by the notion of bisimilarity discussed below. Both the notion of transition system and bisimilarity go back to R. Milner and, in its modern form, were introduced by D. Park [22] who studied infinite behavior of automata.

The *mixed* version

$$\langle S, A, T\rangle, \quad T \subseteq S \times A \times S \cup S^2$$

combines unlabeled transitions $s \to s'$ with labeled ones $s \xrightarrow{a} s'$. In this case we discuss unobservable or hidden and observable transitions. However as it will be demonstrated later, the mixed version can be reduced to labeled systems. Technically sometime it is easier to define a mixed system and then reduce it to labeled one.

The *attributed transition systems*:

$$\langle S, A, U, T, \varphi \rangle, \quad \varphi : S \to U.$$

This kind of transition system is used when not only transitions but also states should be labeled. The function φ is called a state label function. Usually a set of state labels is structured as $U = D^R$, where the set R is called the set of attributes and the set D is a set of attribute values. These sets can also be typed and in this case $U = (D_\xi^{R_\xi})_{\xi \in \Xi}$ (Ξ is the set of type symbols).

Adjusted transition systems are obtained distinguishing three kinds of subsets,

$$S_0, S_\Delta, S_\perp \subseteq S$$

in a set S of system states. They are *initial states*, *states of successful termination* and *undefined (divergent) states*, respectively. The supposed meaning of these adjustments is the following: from initial states a system can start in an initial state and terminate in a state of successful termination, undefined states are used to define an approximation relation on the set of states; the behavior of a system can be refined (extended) in undefined states. The states of successful termination must be distinguished from the *dead lock states*, that is the states from which there are no transitions but which are neither states of successful termination nor undefined states. The property of a state having no transitions is denoted as $s \not\to$.

Other important classes of transition systems are stochastic, fuzzy, and real time transition systems. All of them are obtained by introducing some additional numeric structure to different kinds of transition systems and will not be considered here. Attributed transition systems as well as mixed systems can be reduced to labeled ones, so the main kind of system will be labeled an adjusted transition system (usually with $S_0 = S$) and other kinds will be used only in examples.

Let us consider some useful examples (without details which the reader is encouraged to supply himself/herself).

Automata The set A of actions is identified with an input (output) alphabet or with the set of pairs input/output.

Programs The set A of actions is an instruction set or only input/output instructions according to what should be considered as observable actions. The set S is the set of states of a program (including memory states). If we want some variables to be observable, a system can be defined with a state label function mapping the variable symbols to their values in a given state.

Program schemata Symbolic (allowing multiple interpretations) instructions and states are considered. The set of actions are the same as in the model of a program.

Parallel or distributed programs and program schemata The set A of actions is a set of observable actions performed in parallel or sequentially (with interleaving) in different components; communications are usually represented by hidden transitions (as in CCS). The states are composed with the states of components by parallel composition. This example will be considered below in more details.

Calculi States are formulas; actions are the names of inference rules.

Data and knowledge bases Actions are queries.

There are two kinds of non-determinism inherent in transition systems. The first one is the existence of two transitions $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ for some state s with $s' \neq s''$. This non-determinism means that, after performing an action a, a system can choose the next state non-deterministically. The second kind of non-determinism is the possibility of different adjustment of the same state, that is a state can be at the same time a state of successful termination as well as undefined or initial.

A labeled transition system (without hidden transitions) is called *deterministic* if for arbitrary transitions from $s \xrightarrow{a} s' \wedge s \xrightarrow{a} s''$ it follows that $s' = s''$ and $S_\Delta \cap S_\perp = \emptyset$.

2.2 Trace equivalence

A *history* of system performance is defined as a sequence of transitions starting from some initial state s_1 and continuing at each step by application of transition relation, to a state obtained at this step:

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} \cdots$$

A history can be finite or infinite. It is called *final* if it is infinite or cannot be continued. A *trace* corresponding to a given history is a sequence of actions performed along this history:

$$a_1 a_2 \ldots a_n \ldots$$

For an attributed transition system the trace includes the state labels:

$$\varphi(s_1) \xrightarrow{a_1} \varphi(s_2) \xrightarrow{a_2} \cdots \xrightarrow{a_n} \cdots$$

Different sets of traces can be used as invariants of system behavior. They are called *trace invariants*. Examples of trace invariants of a system S are the following sets: $L(S)$—the set of all traces of a system S; $L_S(s)$—the set of all traces starting at the state s; $L_\Delta(S)$—the set of all traces finishing at a terminal state, $L^0_\Delta(S)$—the set of all traces starting at an initial state and finishing at a terminal state, etc. All these invariants can be easily computed for finite state systems as regular languages.

We obtain the notion of *trace equivalence* considering $L^0_\Delta(S)$ as the main trace invariant: systems S and S' are trace equivalent ($S \sim_T S'$) if $L^0_\Delta(S) = L^0_\Delta(S')$. Unfortunately trace equivalence is too weak to capture the notion of transition system behavior. Consider the two systems presented in Fig. 1.

Both systems in the figure start their activity by performing an action a. But the first of the two systems has a choice at the second step. It can perform action b or c. At the same

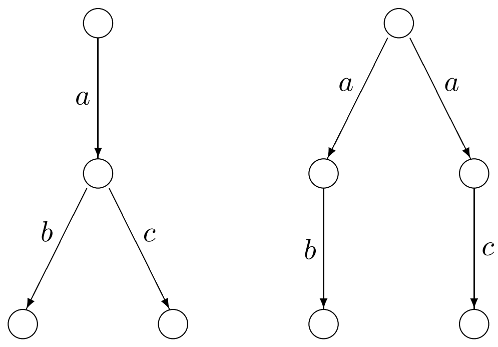

Figure 1: Trace equivalent systems with different behaviors

time the second system will only perform an action b and can never perform c or it can only perform c and never perform b, dependent on what decision was made at the first step. The equivalence, stronger than trace equivalence, that captures the difference between the two systems in Fig. 1 is bisimilarity. It is considered in the next section.

2.3 Bisimilarity

2.1 Definition A binary relation $R \subseteq S^2$ is called a *bisimulation* if:

(1) $(s, s') \in R \implies (s \in S_\Delta \iff s' \in S_\Delta,\ s \in S_\perp \iff s' \in S_\perp)$;

(2) $(s, s') \in R \wedge s \xrightarrow{a} t \implies \exists\, t'\ ((t, t') \in R \wedge s' \xrightarrow{a} t')$;

(3) $(s, s') \in R \wedge s' \xrightarrow{a} t' \implies \exists\, t\ ((t, t') \in R \wedge s \xrightarrow{a} t)$.

States s and s' are called *bisimilar* ($s \sim_B s'$) if there exists a bisimulation R such that $(s, s') \in R$. For attributed transition systems an additional requirement is: $(s, s') \in R \implies \varphi(s) = \varphi(s')$. We can also extend this definition to mixed transition systems if $\exists\, s'\ (s \xrightarrow{*} s' \xrightarrow{a} t)$ will be used instead of $s \xrightarrow{a} t$ and use $\exists\, s'\ (s \xrightarrow{*} s' \wedge s' \in S_\Delta(S_\perp))$ instead of $s \in S_\Delta(S_\perp)$.

2.2 Proposition *Bisimilarity is an equivalence relation.*

Proof Note that $\{(s, s) \mid s \in S\}$ is a bisimulation. If R is a bisimulation then R^{-1} is a bisimulation and if R and R' are bisimulations then $R \circ R'$ is also a bisimulation. $\square$

2.3 Proposition *Bisimilarity is a maximal bisimulation on S.*

Proof An arbitrary union of bisimulations is again a bisimulation; therefore a bisimulation is a union of all bisimilarities on S. $\square$

Bisimilarity of two states can be extended to the case when they are the states of different systems in a usual way (consider the disjoint union of the two systems). The bisimilarity of two systems can also be defined so that each state of one of them must be bisimilar to some state in the other.

Reduction of mixed transition systems Let S be a mixed transition system. Add new rules to define new labeled transitions and extend termination states in the following way.

$$\frac{s \xrightarrow{*} s',\ s' \xrightarrow{a} s''}{s \xrightarrow{a} s'}$$

$$s \xrightarrow{*} s',\ s' \in S_\Delta(S_\perp) \implies s \in S_\Delta(S_\perp).$$

Now delete unlabeled transitions. The new labeled system is called a reduction of the system S.

2.4 Proposition *A mixed transition system and its reduction are bisimilar.*

Proof The relation $s' \xrightarrow{*} s$ between s, considered as a state of a reduced system, and s', considered as a state of a mixed system, is a bisimulation. □

For a deterministic system the difference between trace equivalence and bisimilarity disappears.

2.5 Proposition *For deterministic systems* $s \sim_T s' \implies s \sim_B s'$.

Th spectrum of different equivalences, from trace equivalence to bisimilarity, can be found in the paper of Glabbeek [8]. Bisimilarity is the strongest; trace equivalence is the weakest.

To define an approximation relation on the set of states of a transition system, the notion of partial bisimulation will be introduced.

2.6 Definition The binary relation $R \subseteq S^2$ is called a *partial bisimulation* if:

(1) $(s, s') \in R \implies (s \in S_\Delta \implies s' \in S_\Delta,\ s \notin S_\perp \implies s' \notin S_\perp) \wedge (s \notin S_\perp \wedge s' \in S_\Delta \implies s \in S_\Delta)$;

(2) $(s, s') \in R \wedge s \xrightarrow{a} t \implies \exists t'\ ((t, t') \in R \wedge s' \xrightarrow{a} t')$ (the same as for bisimilarity);

(3) $(s, s') \in R \wedge s \notin S_\perp \wedge s' \xrightarrow{a} t' \implies \exists t\ ((t, t') \in R \wedge s \xrightarrow{a} t)$ (the same as for bisimilarity with the additional restriction $s \notin S_\perp$).

We say that s is less defined then s' or s *approximates* s' ($s \sqsubseteq_B s'$), if there exists a partial bisimulation such that $(s, s') \in E$. A partial bisimulation is a preorder and from the definitions it follows that:

2.7 Proposition $s \sim_B s' \iff s \sqsubseteq_B s' \sqsubseteq_B s$.

2.4 Behavior algebras

The invariant of a trace equivalence is a language. What is the invariant of a bisimilarity? To answer this question one should define the notion of behavior of a transition system (in a given state). Intuitively it is a node of a diagram of a transition system unfolded into a (finite or infinite) labeled tree (synchronization tree), with some nodes of this tree being identified. More precisely, two transitions from the same node labeled by the same action should be identified if they lead to bisimilar subtrees. Different approaches are known for studying bisimulation. Among them are Hennessy-Milner logic [12], the domain approach of

S. Abramsky [1], and the final coalgebra approach of Aczel and Mendler [3]. A comparative study of different approaches to characterize bisimilarity can be found in [23]. Here we shall give the solution based on continuous algebras [11] or algebras with an approximation [14]. The variety of algebras with approximation relation will be defined and a minimal complete algebra $F(A)$ over a set of actions A will be constructed and used for the characterization of bisimilarity. It is not the most general setting, but the details of direct constructions are important for the next steps in developing the algebra of transformations.

Behavior algebra $\langle U, A\rangle$ is a two sorted algebra. The elements of sort U are called *behaviors*, the elements of A are called *actions*. The signature and identities of a behavior algebra are the following.

Signature *Prefixing* $a.u$, $a \in A$, $u \in U$, *non-deterministic choice* $u + v$, $u, v \in U$, *termination constants* Δ, $\perp$, 0, called *successful termination*, *divergence* and *dead lock* correspondingly, and *approximation relation* $u \sqsubseteq v$ (u approximates v), $u, v, \in U$.

Identities Non-deterministic choice is an associative, commutative, and idempotent operation with 0 as a neutral element ($u+0 = u$). Approximation relation $\sqsubseteq$ is a partial order with minimal element $\perp$. Both operations (prefixing and non-deterministic choice) are monotonic with respect to the approximation relation:

$$\begin{aligned} &\perp \sqsubseteq u,\\ &u \sqsubseteq v \implies u + w \sqsubseteq v + w,\\ &u \sqsubseteq v \implies a.u \sqsubseteq a.v. \end{aligned}$$

Continuity Prefixing and non-deterministic choice are continuous with respect to approximation, that is they preserve least upper bounds of directed sets of behaviors if they exist.

More precisely, let $D \subseteq U$ be a directed set of behaviors, that is for any two elements $d', d'' \in D$ there exists $d \in D$ such that $d' \sqsubseteq d$, $d'' \sqsubseteq d$. The least upper bound of the set D if it exists will be denoted as $\bigsqcup D$ or $\bigsqcup_{d\in D} d$. The continuity condition for U means that

$$\begin{aligned} a.\bigsqcup D &= \bigsqcup_{d\in D} a.d,\\ \bigsqcup D + u &= \bigsqcup_{d\in D} (d + u). \end{aligned}$$

Note that monotonicity follows from continuity.

Some additional structures can be defined on the components of a behavior algebra.

Actions A combination $a \times b$ of actions can be introduced as a binary associative and commutative (but in general case not idempotent) operation to describe communication or simultaneous (parallel) performance of actions. In this case an impossible action $\emptyset$ is introduced as unnulator for combination and unit action δ with identities

$$\begin{aligned} a \times \emptyset &= \emptyset, \\ a \times \delta &= a, \\ \emptyset.u &= 0. \end{aligned}$$

In CCS each action a has a dual action $\overline{a}$ ($\overline{\overline{a}} = a$) and the combination is defined as $a \times \overline{a} = \delta$ and $a \times b = \emptyset$ for non-dual actions (the symbol τ is used in CCS instead of δ; it denotes the observation of hidden transitions and two states are defined as weakly bisimilar if they are bisimilar after changing τ transitions to hidden ones). In CSP another combination is used: $a \times a = a$, $a \times b = \emptyset$ for $a \neq b$.

Attributes A function defined on behaviors and taking values in an attribute domain can be introduced to define behaviors for attributed transition systems.

To characterize bisimilarity we shall construct a complete behavior algebra $F(A)$. Completeness means that all directed sets have least upper bounds. We start from the algebra $F_{\text{fin}}(A)$ of finite behaviors. This is a free algebra generated by termination constants (an initial object in the variety of behavior algebras). Then this algebra is extended to a complete one adding the limits of directed sets of finite behaviors. To obtain infinite, convergent (definition see below), non-deterministic sums this extension must be done through the intermediate extension F_{fin}^{∞} of the algebra of finite depth elements.

Algebra of finite behaviors $F_{\text{fin}}(A)$ is the algebra of behavior terms generated by termination constants considered up to identities for non-deterministic choice, and with the approximation relation defined in the following way.

2.8 Definition (Approximation in $F_{\text{fin}}(A)$) $u \sqsubseteq v$ if and only if there exists a term $\varphi(x_1, \ldots, x_n)$ generated by termination constants and variables $x_1, \ldots, x_n$ and terms $v_1, \ldots, v_n$ such that $u = \varphi(\bot, \ldots, \bot)$ and $v = \varphi(v_1, \ldots, v_n)$.

2.9 Proposition *Each element of $F_{\text{fin}}(A)$ can be represented in the form*

$$u = \sum_{i \in I} a_i.u_i + \varepsilon_u$$

where I is a finite set of indices and ε is a termination constant. If the $a_i.u_i$ are all different and all u_i are represented in the same form, this representation is unique up to commutativity of non-deterministic choice.

Proof The proof is by induction on the height $h(u)$ of a term u defined in the following way: $h(\varepsilon) = 0$ for the termination constant ε, $h(a.u) = h(u) + 1$, $h(u + v) = \max\{h(u), h(v)\}$. □

A termination constant ε_u can possess the following values: 0, Δ, $\bot$, $\bot + \Delta$. Behavior u is called *divergent* if $\varepsilon_u = \bot, \bot + \Delta$, otherwise it is called *convergent.* For *terminal* behaviors $\varepsilon_u = \Delta, \bot + \Delta$ and behavior u is *guarded* if $\varepsilon_u = 0$.

2.10 Proposition $u \sqsubseteq v$ *if and only if*

(1) $\varepsilon_u \sqsubseteq \varepsilon_v$;

(2) $u = a.u' + u'' \implies v = a.v' + v'',\ u' \sqsubseteq v'$;

(3) $v = a.v' + v''$ *and* u *is convergent* $\implies u = a.u' + u'',\ u' \sqsubseteq v'$.

2.11 Proposition *The algebra* $F_{\text{fin}}(A)$ *is a free behavior algebra.*

Proof Only properties of approximation need proof. To prove that approximation is a partial order and that prefixing and nondeterministic choice are monotoniv is an easy exercise (to prove antisymmety use Proposition 2.10). To prove that the operations are continuous note that each finite behavior has only a finite number of approximations and therefore only finite directed sets have least upper bounds. The property $\varphi(\perp, \dots, \perp) \sqsubseteq \varphi(v_1, \dots, v_n)$ is true in an arbitrary behavior algebra (induction); therefore the approximation in $F_{\text{fin}}(A)$ is a minimal one. □

Note that in $F_{\text{fin}}(A)$
$$x = y \iff x \sqsubseteq y \sqsubseteq x.$$

Algebra of finite height behaviors $F_{\text{fin}}^{\infty}(A)$ is defined in the following way. Let

$$F_{\text{fin}}^{(\infty)}(A) = \bigcup_{n=0}^{\infty} F^{(n)},$$
$$F^{(0)}(A) = \{\Delta, \perp, \Delta + \perp, 0\},$$
$$F^{(n+1)}(A) = \{\textstyle\sum_{i \in I} a_i.u_i + v \mid u_i, v \in F^{(n)}\},$$

where I is an arbitrary set of indices, but expressions $\sum_{i\in I} a_i.u_i$ and $\sum_{j \in J} b_j.v_j$ are identified if $\{a_i.u_i \mid i \in I\} = \{b_j.v_j \mid j \in J\}$. Therefore one can restrict the cardinality of infinite I to be no more then $2^{|A|}$ for $F^{(1)}(A)$ and no more then $2^{|F^{(n)}|}$ for $F^{(n+1)}(A)$.

Take Proposition 2.10 as the definition of an approximation relation on the set $F_{\text{fin}}^{\infty}(A)$. Taking into account the identification of infinite sums we have again that $x = y \iff x \sqsubseteq y \sqsubseteq x$. Define prefixing as $a.u$ for $u \in F^{(n)}(A)$ and we have $\sum_{i \in I} a_i.u_i + \sum_{j\in J} b_j.v_j = \sum_{k \in I \cup J} c_k.w_k$ where $I \cap J = \emptyset$ and $c_k.w_k = a_k.u_k$ for $k \in I$ and $c_k.w_k = b_k.v_k$ for $k \in J$ (disjoint union).

2.12 Proposition *The algebra* $F_{\text{fin}}^{\infty}(A)$ *is a behavior algebra.*

Proof Use induction on the height. □

However the algebra $F_{\text{fin}}^{\infty}(A)$ has the same identities as $F_{\text{fin}}(A)$. It is not free because it has no free generators and the equality

$$\sum_{i \in I} a_i.u_i + \sum_{j \in J} b_j.v_j = \sum_{k \in K} c_k.w_k$$

for $\{a_i.u_i \mid i \in I\} \cup \{b_j.v_j \mid j \in J\} = \{c_k.w_k \mid k \in K\}$ does not follow from the identities when at least one of I or J is infinite. But they are the only equalities except for identities in $F_{\text{fin}}^{\infty}(A)$ (infinite associativity).

In the algebra $F_{\text{fin}}^{\infty}(A)$ the canonical representation of Proposition 2.9 is still valid for infinite sets of indices.

Let X be a set of variables. Define the set $F_{\text{fin}}^{\infty}(A, X)$ in the same way as $F_{\text{fin}}^{\infty}(A)$, but redefine $F^{(0)}(A)$ as $F^{(0)}(A, X) = \{\sum_{i\in I} \varepsilon_i \mid \varepsilon_i \in F^{(0)}(A) \cup X,\ i \in I\}$ so that, besides the set of termination constants, it also includes the sums of variables. The set $F_{\text{fin}}^{\infty}(A, X)$ is a behavior algebra with operations and approximation defined in the same way as for $F_{\text{fin}}^{\infty}(A)$.

Define substitution $\sigma = \{x_i := v_i \mid i \in I\}$ as a homomorphism $u \mapsto u\sigma$ such that $x_i\sigma = v_i$, $\varepsilon\sigma = \varepsilon$ for termination constants, and $(\sum u_i)\sigma = \sum u_i\sigma$. If $u, v \in F_{\text{fin}}^{\infty}(A, X)$ then $u(v)$ denotes $u\{x := v,\ x \in X\}$.

2.13 Proposition *For elements $u, v \in F_{\text{fin}}^{\infty}(A, X)$ the approximation relation satisfies the following statement: $u \sqsubseteq v$ if and only if there exists $\varphi \in F_{\text{fin}}^{\infty}(A, X)$ and substitution $\sigma = \{x_i := v_i \mid i \in I, x_i \in X\}$ such that $\varphi(\perp) = u$, and $\varphi\sigma = v$.*

Proof By induction on the height of u. □

Complete behavior algebra $F(A)$

The elements of $F(A)$ are directed sets of $F_{\text{fin}}^{\infty}(A)$ considered up to the following equivalence.

2.14 Definition Directed sets U and V in $F_{\text{fin}}^{\infty}(A)$ are called equivalent ($U \sim V$) if for each $u \in U$ there exists $v \in V$ such that $u \sqsubseteq v$ and for each $v \in V$ there exists $u \in U$ such that $v \sqsubseteq u$.

Define operations and approximation on directed sets in the following way.

- Prefixing: $a.U = \{a.u \mid u \in U\}$;
- Non-deterministic choice: $U + V = \{u + v \mid u \in U, V \in V\}$;
- Approximation: $U \sqsubseteq V \iff \forall(u \in U)\ \exists(v \in V)\ (u \sqsubseteq v)$.

These operations preserve equivalence and therefore can be extended to classes of equivalent directed sets.

2.15 Proposition *The algebra $F(A)$ is a behavior algebra. It is a minimal complete conservative extension of the algebra $F_{\text{fin}}^{\infty}(A)$.*

Proof The least upper bound of a directed set of elements of $F(A)$ is a (set theoretical) union of these elements. Also the algebra $F_{\text{fin}}^{\infty}(A)$ can be isomorphically embedded into $F(A)$ by the mapping of $u \in F_{\text{fin}}^{\infty}(A)$ to $\{v \mid v \sqsubseteq u\}$. Minimality means that if H is another complete conservative extension of $F_{\text{fin}}^{\infty}(A)$ then there exists a continuous homomorphism from $F(A)$ to H such that the following diagram is commutative:

$$\begin{tikzcd}
F_{\text{fin}}^{\infty}(A) \arrow[rr] \arrow[dr] & & F(A) \arrow[dl] \\
& H &
\end{tikzcd}$$

For details see [14]. □

Let us define $\sum_{i\in I} u_i$ for $u_i \in F(A)$ and infinite I as $\{\sum_{i\in I} v_i \mid v_i \in u_i\}$. Note that $F(A,X)$ can be defined as a free complete extension of $F(A)$ and Proposition 2.13 can be proved for $F(A,X)$

2.16 Proposition *Each $u \in F(A)$ can be represented in the form $u = \sum_{i\in I} a_i.u_i + \varepsilon_u$ and this representation is unique if all $a_i.u_i$ are different.*

Proof Let $M(a)$ be the set of all solutions of the equation $a.x + y = u$ with unknowns $x, y \in F(A)$ and $S(a)$ the set of all x such that, for some y, $(x,y) \in M(a)$. Let $I = \{(a,u) \mid a \in A,\ u \in S(a)\}$ and $a_{(a,u)} = a$, $u_{(a,u)} = u$. Then $u = \sum_{i\in I} a_i.u_i + \varepsilon_u$ and uniqueness is obvious. □

Another standard representation of behaviors is through the definition of a minimal solution of the system of equations

$$x_i = F_i(X), \quad i \in I$$

where $F_i(X) \in F^{\infty}_{\text{fin}}(A,X)$ and $x_i \in X$. As usually, this minimal solution is defined as $x_i = \bigsqcup_{i=0}^{(\infty)} x_i^{(n)}$ where $x_i^{(0)} = \perp$, $x_i^{(n+1)} = (F_i(X))\sigma_n$, $\sigma_{(n+1)} = \{x_i := x_i^{(n)},\ i \in I\}$. Note that the first representation is used in the co-algebraic approach and the second is a slight generalisation of the traditional fixed point approach.

2.5 Behaviors of transition systems

Let S be a labeled transition system over A. For each state $s \in S$, define the behavior $\texttt{beh}(s) = u_s$ of a system S in a state s as a minimal solution of the system

$$u_s = \sum_{s \xrightarrow{a} t} a.u_t + \varepsilon_s$$

where ε_s is defined in the following way:

$$\begin{aligned}
s \notin S_\Delta \cup S_\perp &\implies \varepsilon_s = 0,\\
s \in S_\Delta \backslash S_\perp &\implies \varepsilon_s = \Delta,\\
s \in S_\perp \backslash S_\Delta &\implies \varepsilon_s = \perp,\\
s \in S_\Delta \cap S_\perp &\implies \varepsilon_s = \Delta + \perp.
\end{aligned}$$

Behaviors as states

A set of behaviors $U \subseteq F(A)$ is called *transition closed* if

$$a.u + v \in U \implies u \in U.$$

In this case U can be considered as a transition system if transitions and adjustment are defined in the following way:

$$\begin{aligned}
a.u + v &\xrightarrow{a} u,\\
U_\Delta &= \{u \mid u = u + \Delta\},\\
U_\perp &= \{u \mid u = u + \perp\}.
\end{aligned}$$

2.17 Theorem *Let s and t be states of a transition system, u and v behaviors. Then*

(1) $s \sqsubseteq_B t \iff u_s \sqsubseteq u_t$;

(2) $s \sim_B t \iff u_s = u_t$;

(3) $u \sqsubseteq v \iff u \sqsubseteq_B v$;

(4) $u = v \iff u \sqsubseteq_B v$.

Proof The first follows from the bisimilarity of s and u_s considered as a state. (1) $\Rightarrow$ (2) because $\sqsubseteq$ is a partial order, and (2) $\Rightarrow$ (3) because $\texttt{beh}(u) = u$. $\square$

An *agent* is an adjusted labeled transition system. An *abstract agent* is an agent with states considered up to bisimilarity. Identifying the states with behaviors we can consider an abstract agent as a transition closed set of behaviors. Conversely, considering behaviors as states we obtain a standard representation of an agent as a transition system. This representation is defined uniquely up to bisimilarity. We should distinguish an agent as a set of states or behaviors from an agent in a given state. In the latter case we consider each individual state or behavior of an agent as the same agent in a given state adjusted to have the unique initial state. Usually this distinction is understood from the context.

2.6 Sequential and parallel compositions

There are many compositions enriching the base process algebra or the algebra of behaviors. Most of them are defined independently on the representation of an agent as a transition system. These operations preserve bisimilarity and can be considered as operations on behaviors. Another useful property of these operations is continuity. The use of definitions in the style of SOS semantics [2] or the use of conditional rewriting logic [18] always produces continuous functions if they are expressed in terms of behavior algebras. The mostly popular operations are *sequential* and *parallel* compositions.

Sequential composition is defined by means of the following inference rules and equations:

$$\frac{u \xrightarrow{a} u'}{(u; v) \xrightarrow{a} (u'; v)},$$
$$((u + \Delta); v) = (u; v) + v,$$
$$((u + \bot); v) = (u; v) + \bot,$$
$$(u; 0) = 0.$$

These definitions should be understood in the following way. First we extend the signature of the behavior algebra adding new binary operation $((\ ); (\ ))$. Then add identities for this operation and convince yourself that no new equation appears in the original signature (conservation of extension). Then a transition relation is defined on the set of equivalence classes of extended behavior expressions (independence of the choice of representative must be shown). These classes now become the states of a transition system, and the value of the expression is defined as its behavior. In the sequel we shall use the notation uv instead of $(u; v)$.

2.18 Exercise Prove identities $\Delta u = u\Delta = u$, $\perp u = \perp$, $(uv)w = u(vw)$, $(u+v)w = uw+vw$. Hint: Define bisimilarity (for non-trivial cases).

Sequential composition can be also defined explicitly by the following recursive definition:

$$uv = \sum_{u \xrightarrow{a} u'} a(u'v) + \sum_{u=u+\varepsilon} \varepsilon v,$$

$$0v = 0, \qquad \Delta v = v, \qquad \perp v = \perp.$$

If an action a is identified with the agent $a.\Delta$, then we have $a = a.\Delta = (a;\Delta) = a\Delta$.

Parallel composition of behaviors assumes that a combination of actions is defined. It is considered as a associative and commutative operation $a \times b$ with annulator $\emptyset$. Rules and identities for the parallel composition are

$$\frac{u \xrightarrow{a} u',\ v \xrightarrow{b} v',\ a \times b \neq \emptyset}{u\|v \xrightarrow{a\times b} u'\|v'},$$

$$\frac{u \xrightarrow{a} u',\ v \xrightarrow{b} v'}{u\|v \xrightarrow{a} u'\|v,\ u\|v \xrightarrow{b} u\|v',\ u\|(v+\Delta) \xrightarrow{a} u',\ (u+\Delta)\|v \xrightarrow{b} v'},$$

$$(u+\Delta)\|(v+\Delta) = (u+\Delta)\|(v+\Delta) + \Delta,$$

$$(u+\perp)\|v = (u+\perp)\|v + \perp,$$

$$u\|(v+\perp) = u\|(v+\perp) + \perp.$$

2.19 Exercise Prove associativity and commutativity of parallel composition.

An explicit definition of parallel composition is

$$u\|v = \sum_{u \xrightarrow{a} u',v \xrightarrow{b} v'} (a \times b)(u'\|v') + \sum_{u \xrightarrow{a} u'} a(u'\|v) + \sum_{v \xrightarrow{b} v'} b(u\|v') + \varepsilon_u\|\varepsilon_v$$

where $\varepsilon_u(\varepsilon_v)$ is a termination constant in the representation $u = \sum a_i u_i + \varepsilon_u$ of a behavior u.

3 Algebra of behavior transformations

3.1 Environments and insertion functions

An *environment* is an abstract agent E over the set C of environment actions together with a continuous *insertion function* $\texttt{Ins}\colon E \times F(A) \to E$. All states of E are considered as possible initial states. Therefore an environment is a tuple $\langle E, C, A, \texttt{Ins}\rangle$. In the sequel C, A, and $\texttt{Ins}$ will be used implicitly and $\texttt{Ins}(e,u)$ will be denoted as $e[u]$. After inserting an agent u (in a given state u), the new environment is ready for new agents to be inserted and the insertion of several agents is something that we will often wish to describe. Therefore the notation

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

will be used to describe this insertion.

Each agent (behavior) u defines a transformation $[u]$ of environment (behavior transformation); $[u]\colon E \to E$ is such that $[u](e) = e[u]$. The set of all behavior transformations of a type $[u]$ of environment E is denoted by $T(E) = \{[u] \mid u \in F(A)\}$. This is a subset of the set $\Phi(E)$ of all continuous transformations of E. A semigroup multiplication $[u] * [v]$ of two transformations $[u]$ and $[v]$ can be defined as follows:

$$([u] * [v])(e) = (e[u])[v] = e[u, v]$$

The semigroup generated by $T(E)$ is denoted as $T^*(E)$ and (for a given insertion function) we have:

$$T(E) \subseteq T^*(E) \subseteq \Phi(E)$$

An insertion function is called a *semigroup insertion* if $T(E) = T^*(E)$. It is possible if and only if for all $u, v \in F(A)$ there exists $w \in F(A)$ such that for all $e \in E$, $e[u, v] = e[w]$.

It is interesting also to fix the cases when $T(E) = \Phi(E)$. Such an insertion is called *universal*. A trivial universal insertion exists if the cardinality of A is not less then the cardinality of $\Phi(E)$. In this case all functions can be enumerated by actions with the mapping $\varphi \mapsto a_\varphi$ and insertion function can be defined so that $e[a_\varphi] = \varphi(e)$.

The kernel of the mapping $u \mapsto [u]$ of $F(A)$ to $T(E)$ defines an equivalence relation on the set F(A) of agents (behaviors). This equivalence is called an *insertion equivalence*:

$$u \sim_E v \iff \forall(e \in E)\ (e[u] = e[v])$$

Generally speaking, insertion equivalence is not a congruence.

Let $\sim_E$ is a congruence. In this case the operations of the behavior algebra $F(A)$ can be transferred to $T(E)$ so that

$$e([u] + [v]) = e[u + v]$$
$$e(a.[u]) = e[a.u]$$

Define an approximation relation on $T(E)$ so that

$$[u] \sqsubseteq [v] \iff \forall(e \in E)\ (e[u] \sqsubseteq e[v])$$

3.1 Theorem *If $\sim_E$ is a congruence, $T(E)$ is a behavior algebra and the mapping $u \mapsto [u]$ is a continuous homomorphism of $F(A)$ on $T(E)$.*

An environment can be also defined as a two sorted algebra $\langle E, T(E)\rangle$ with the insertion function considered as an external operation on E.

Let us consider some simple examples.

Parallel insertion An insertion function is called a *parallel insertion* if it satisfies the following condition:

$$e[u, v] = e[u \| v].$$

A parallel insertion is a semigroup with $[u] * [v] = [u \| v]$. An example of parallel insertion is the insertion function $e[u] = e \| u$ (for $A = C$). This function is called a *strong parallel insertion*. In this case $\sim_E$ is a congruence and if $\Delta \in E$ it coincides with a bisimilarity. Strong parallel insertion models the situation when an environment for a given agent is a parallel composition of all other agents interacting with it.

Sequential insertion An insertion function is called a *sequential insertion* if it satisfies the following condition:

$$e[u, v] = e[uv].$$

A sequential insertion is also a semigroup insertion with $[u] * [v] = [uv]$ and a *strong sequential insertion* defined by the equation $e[u] = eu$ $(A = C)$ is a congruence and in this case the insertion equivalence is a bisimilarity if $\Delta \in E$.

Trace environment A *trace environment* is generated by one state: $E = \{e_0[u] \mid u \in F(A)\}$. An insertion function is defined by the equations $e_0[u, v] = e_0[uv]$, $e_0[\Delta] = e_0$, $e_0[\bot] = \bot$, $e_0[0] = 0$, and

$$e_0\left[\sum_{i\in I} a_i u_i + \varepsilon\right] = \sum_{a\in A} a.e_0\left[\sum_{i\in I,\ a_i = a} u_i\right] + e_0[\varepsilon].$$

It is easy to prove the following.

3.2 Theorem *For a trace environment E, $u \sim_E v$ if and only if $u \sim_T v$.*

In a trace environment we also have a distributive law $[x] * ([y] + [z]) = [x] * [y] + [x] * [z]$ and a Klinee like algebra can be defined by introducing an iteration $[u]^* = \sum_{n=0}^{\infty}[u]^n$. But this algebra contains not only finite but also infinite behaviors and there are equalities like $uv = u$ if u has no termination constant Δ at the end of some history.

Problem Find environments for all the equivalences between trace and bisimulation defined in [8].

3.2 Classification of insertion functions

In this and the following sections assume that the set E of environment behaviors is not only transition closed but, for each behavior e, also contains all of its approximations. That is from $e \in E$ and $e' \sqsubseteq e$ it follows that $e' \in E$.

A continuous insertion function can be represented in the form

$$e[u] = \bigsqcup_{e' \sqsubseteq e,\ u' \sqsubseteq u} e'[u']$$

where $e' \in F_m^\infty(C)$, $u' \in F_m^\infty(A)$. Using this representation the following proposition can be proved.

3.3 Proposition

(1) $e[u] \xrightarrow{c} f \implies$ *there exist, for some m, $e' \in F_m^\infty(C)$, $u' \in F_m^\infty(A), f' \in F(C)$ such that $e'[u'] \xrightarrow{c} f'$, $e' \sqsubseteq e$, $u' \sqsubseteq u$, $f' \sqsubseteq f$;*

(2) $e'[u'] \xrightarrow{c} f'$, $e' \in F_m^\infty(C)$, $u' \in F_m^\infty(A), f' \in F(C) \implies$ *there exist $e \in E$, u, f such that $e[u] \xrightarrow{c} f$ $\implies$ and $e' \sqsubseteq e$, $u' \sqsubseteq u$, $f' \sqsubseteq f$.*

From this proposition and Proposition 2.13 it follows that for each transition $e'[u'] \xrightarrow{c} f'$ there must be a "transition equation"

$$e(X)[u(Y)] \xrightarrow{c} f(X \cup Z)\sigma \tag{3.1}$$

such that $e(\perp) = e'$, $u(\perp) = u'$, $f(\perp) = f'$ and σ substitutes continuous functions of X and Y to Z. To cover more instances the intersection of X and Y must be empty and all occurrences of variables in the left hand side must be different (left linearity). These equations belong to some extended transformation algebra enriched by corresponding functions. More precisely the meaning of (3.1) can be described by the following equational formula:

$$\forall\sigma_1,\ \forall\sigma_2\ \exists\, g(e(X)\sigma_1)[u(Y)\sigma_2] = c.(f(X \cup Z)\sigma_1)\sigma' + g$$

where $\sigma_1 : X \to F(C)$, $\sigma_2 : Y \to F(A)$, $g \in F(C)$, $\sigma' = \sigma(\sigma_1 + \sigma_2)$ (disjoint union of two substitutions in the right hand side of substitution σ). The set of termination equations

$$e(X)[u(Y)] = e(X)[u(Y)] + \varepsilon \tag{3.2}$$

must be considered together with the transition ones. The set of all transition and termination equations uniquely defines an insertion function and can be used for its computation.

The previous discussion results in trying to apply rewriting logic [18] for recursive computation of insertion functions and modeling the behavior of an environment with inserted agents. For this purpose one should restrict consideration to only those substitutions expressed in the form $\sigma = \{z_i := f_i[u_i] \mid z_i \in Z,\ f_i \in F^{\infty}_{\text{fin}}(C, X),\ u_i \in F^{\infty}_{\text{fin}}(A, Y)\}$. This restriction means that the insertion function is defined by means of identities of a two-sorted algebra of environment transformations (with insertion as the operation). Insertions (environments) defined in this way will be called *equationally defined* insertions (environments).

Equationally defined environments can be classified by additional restrictions on the form of rewriting rules for insertion functions. The first classification is on the height of e and u in the left hand side of (3.1):

- One-step insertion: $(1, 1)$; both environment and agent terms have the height 1.
- Head insertion: $(m, 1)$; the environment term is of arbitrary height and the agent term of height 1. This case is reduced to one-step insertion.
- Look-ahead insertion: (m, m); the general case, reduced to the case $(1, m)$.

In addition we restrict the insertion function by the additivity conditions:

$$\left(\sum e_i\right)[u] = \sum(e_i[u]) \tag{3.3}$$

$$e\left[\sum u_i\right] = \sum(e[u_i]) \tag{3.4}$$

Both conditions will be used for a one-step insertion and the second one—for a head insertion. Restictions for termination equations will not be considered. They are assumed to be in a general form.

3.3 One-step insertion

First we shall consider some special cases of one-step insertion and later it will be shown that the general case can be reduced to this one. From the additivity conditions it follows that the transitions for $c.e[a.u]$, $\varepsilon[a.u]$, $(c.e)[\varepsilon]$, and $\varepsilon[\varepsilon']$ should be defined to dependend only on a, c, and $\varepsilon, \varepsilon' \in \mathrm{E} = \{\perp, \Delta, 0\}$. Other restrictions follow from the rules below. To define insertion assume that two functions $D_1 : A \times C \to 2^C$ and $D_2 : C \to 2^C$ are given. The rules for insertion are defined in the following way.

$$\frac{u \xrightarrow{a} u',\ e \xrightarrow{c} e',\ d \in D_1(a,c)}{e[u] \xrightarrow{d} e'[u']} \qquad \text{(interaction)},$$

$$\frac{e \xrightarrow{c} e',\ d \in D_2(c)}{e[u] \xrightarrow{d} e'[u]} \qquad \text{(environment move)}.$$

In addition we must define a continuous function $\varphi_\varepsilon(u) = \varepsilon[u]$ for each $\varepsilon \in E$. This function must satisfy the following conditions. For all $e \in E$ and $u \in F(A)$

$$\perp[u] \sqsubseteq e[u], \qquad e[u] + 0[u] = e[u]. \tag{3.5}$$

The simplest way to meet these conditions is to define $\perp[u] = \perp$ and $0[u] = 0$. There are no specific assumptions for $\Delta[u]$, but usually neither Δ nor 0 belongs to E. Note that in the case when $\Delta \in E$ and $\Delta[u] = u$ the insertion equivalence is a bisimulation.

3.4 Theorem *For a one-step insertion the equivalence on $F(A)$ is a congruence and $T(A)$ is an environment algebra isomorphic to the quotient algebra of $F(A)$ by insertion equivalence.*

First let us prove the following statement.

3.5 Proposition *For a one-step insertion there exists a continuous function $F : A \times T(E) \to T(E)$ such that $[a.u] = F(a, [u])$,*

Proof Let

$$e = \sum_{i \in I} c_i e_i + \varepsilon_e. \tag{3.6}$$

Then

$$e[a.u] = \sum_{d \in D_1(a,c_i)} d.e_i[u] + \sum_{d \in D_2(c)} d.e_i[a.u] + \varepsilon_e[a.u].$$

Therefore for an arbitrary e the value $e[a.u]$ can be found from the minimal solution of the system defined by these equations with unknowns $e[u]$ and $e[a.u]$ (for arbitrary e and u). $\square$

Proof of Theorem 3.4 Define $a.[u] = F(a, [u])$, $[u] + [v] = \lambda e.(e[u] + e[v])$, $[u] \sqsubseteq [v] \iff \forall\, e.(e[u] \sqsubseteq e[v])$, and $[\varepsilon] = \lambda e.\psi_\varepsilon(e)$, where $\psi_\varepsilon(e) = e[\varepsilon]$ for e defined by (3.6) is found from the system of equations

$$\psi_\varepsilon(e) = \sum_{i \in I} \sum_{d \in D_2(c_i)} d.\psi_\varepsilon(e_i) + \varphi_\varepsilon(\varepsilon_e).$$

Now $T(E)$ is a behavior algebra and $u \mapsto [u]$ is a continuous homomorphism. $\square$

3.6 Example Let $A \subseteq C$. Define combinations $c \times c'$ of actions on the set C as arbitrary ac-operation with identities $c \times \delta = c,\ c \times \emptyset = \emptyset$. Now define the functions for a one-step insertion as follows: $D_1(a, c) = \{d \mid c = a \times d\},\ D_2(c) = \{c\}$. It is easy to prove that this is a parallel insertion: $e[u, v] = e[u\|v]$.

Parallel computation over shared and distributed memory The insertion of the previous example can be used to model parallel computation over shared memory. In this case

$$E = \{e[u_1, u_2, \dots] \mid e : R \to D\}.$$

Here R is a set of names, D is a data domain and the environment is called a shared memory over R. Actions $c \in C$ correspond to statements about memory such as assignements or conditions. The combination $c \times c' \neq \emptyset$ if and only if c and c' are consistent. The notion of consistency depends on the nature of actions and intuitively means that they can be performed simultaneously. The transition rules are:

$$\frac{e \xrightarrow{a \times d} e',\ u \xrightarrow{a} u'}{e[u] \xrightarrow{d} e'[u']}.$$

As a consequence

$$\frac{e \xrightarrow{a_1 \times a_2 \times \cdots \times d} e',\ u_1 \xrightarrow{a_1} u_1', \dots}{e[u_1\|u_2\| \dots] \xrightarrow{d} e'[u_1',\ u_2', \dots]}.$$

The residual action d in the transition $e[u_1\|u_2\| \dots] \xrightarrow{d} e'[u_1',\ u_2', \dots]$ is intended to be used by external agents inserted later, but it can be a convenient restricted interaction only with a given set of agents already inserted. For this purpose a shared memory environment can be inserted into a higher level closure environment with the insertion function defined by the equation $g[e[u]][v] = g[e[u\|v]]$ where g is a state of this environment, e is a state of a shared memory environment, and the only rule used is for the transition: $u \xrightarrow{\delta} u' \vdash g[u] \xrightarrow{\delta} g[u']$.

The idea of a two-level insertion can be used to model distributed and shared memory in the following way. Let $R = R_1 \cup R_2$ be divided into two non-intersecting parts (external and internal memories correspondingly). Let C_1 be the set of actions that change only the values of R_1 (but can use the values of R_2). Let C_2 be the set of statements and conditions that change and use only R_2. Generalize the closure environment in the following way:

$$\frac{e[u] \xrightarrow{d} e'[u'],\ d \in C_1}{g[e[u]] \xrightarrow{d'} g[e'[u']]}$$

where d' is the result of substituting the values of R_2 into d. Now closed environments over R_2 can be inserted into the shared memory environment over R_1:

$$e[g[u_1]\|g[u_2]\| \dots]$$

and we obtain a two-level system with shared memory R_1 and distributed memory R_2. This construction can be iterated to obtain multilevel systems and enriched by message passing.

The importance of these constructions for applications is that most problems of proving equivalence, equivalent transformations and proving properties of distributed programs are reduced to the corresponding problems for behavior transformations that have the structure of behavior algebra.

3.4 Head insertion

Again we start with the special case of head insertion. It is defined by two sets of transition equations:

$$\begin{array}{llll} G_{i,a}(X)[a.y] \xrightarrow{d} G'_{i,a}(X)[y], & i \in I(a), & d \in D_{i,a} \subseteq C & \text{(interaction)}, \\ H_j(X)[y] \xrightarrow{c} H'_j(X)[y], & j \in J, & c \in C_j \subseteq C & \text{(environment move)}, \end{array}$$

and the function $\varphi_\varepsilon(u) = \varepsilon[u]$ satisfying conditions (3.5). Here $G_i(X)$, $G'_{i,a}(X)$, $H_j(X)$, $H'_j(X) \subseteq F^{\infty}_{\text{fin}}(C, X)$ and y is a variable running over the agent behaviors. It is easy to prove that one-step insertion is a special case of head insertion. Note that transition rules for head insertion are not independent. An insertion function defined by these rules must be continuous. Corresponding conditions can be derived from the basic definitions.

Reduction of general case

Two environment states e and e' are called *insertion equivalent* if for all $u \in F(A)$ $e[u] = e'[u]$. This definition is also valid if e and e' are the states of two different environments, E and E', over the same set of agent actions. Environments E and E' are called *insertion equivalent* if for all $e \in E$ there exists $e' \in E'$ such that e and e' are equivalent and vice versa.

3.7 Theorem *For each head insertion environment of the general case there exists an equivalent head insertion environment of the special case.*

Proof The transitions of the general case have the form

$$G(X)[a.y] \xrightarrow{d} G'(X \cup Z)\sigma \tag{3.7}$$

where $\sigma = \{z_i := f_i(X)[g_i(y)] \mid z_i \in Z,\ i \in I\}$, $f_i(X) \in F^{\infty}_{\text{fin}}(C, X)$, $g_i(y) \in F^{\infty}_{\text{fin}}(A, \{y\})$, or

$$G(X)[y] \xrightarrow{d} G'(X \cup Z)\sigma \tag{3.8}$$

with the same description of σ. Introduce a new environment E' in the following way. The states of this environment (represented as a transition system) are the states of E and the insertion expressions are $\overline{e}[u]$ where $e = f\sigma$, $f \in F^{\infty}_{\text{fin}}(A, Z)$, $\sigma = \{z_i := f_i[g_i] \mid z_i \in Z,\ f_i \in F(C),\ g_i \in F^{\infty}_{\text{fin}}(A, \{\overline{y}\}),\ i \in I\}$. The symbol $\overline{y}$ is called suspended agent behavior and the insertion expressions are considered up to identity $\overline{e[\overline{y}]}[u] = e[u]$.

Define a new insertion function so that if there is a transition rule (3.7) in E, then there is a rule $G(X)[a.y] \xrightarrow{d} \overline{G'(X \cup Z)\sigma'}[y]$ in E' where $\sigma' = \sigma\{y := \overline{y}\}$ and if there is a transition rule (3.8) in E then there is a rule $G(X)[y] \xrightarrow{d} \overline{G'(X \cup Z)\sigma'}[y]$ in E' with the same σ'. Add also the rule: if $e \xrightarrow{c} e'$ in E then $\overline{e}[u] \xrightarrow{c} \overline{e'}[u]$ in E'. The termination insertion function φ_ε is derived from those transition rules that have ε as the left hand side. The equivalence of the two environments follows from their bisimilarity as transition systems. □

Reduction to one-step insertion

3.8 Theorem *For each head insertion environment there exists a one-step insertion environment equivalent to it.*

Proof Let E be a special case of a head insertion environment with the set X common to all insertion identities (both assumptions do not influence generality). Define E' as an environment with the set of actions $C' = F_m^\infty(C, X)$. Define mapping $\gamma : E \to F(C')$ so that

$$\begin{aligned}\gamma(e) &= \gamma_1(e) + \gamma_2(e),\\ \gamma_1(e) &= \sum_{e=G_{i,a}(X)\sigma+e'} G_{i,a}(X).\gamma(G'_{i,a}(X)\sigma),\\ \gamma_2(e) &= \sum_{e=H_i(X)\sigma+e'} H_i(X).\gamma(H'_i(X)\sigma).\end{aligned}$$

Define E' as the image of γ, the insertion function, by means of equations

$$\begin{aligned}D_1(a, G_{i,a}(X)) &= \{d|G_{i,a}(X)[a.y] \xrightarrow{d} G'_{i,a}(X)[y]\}\\ D_2(H_i(X)) &= \{d|H_i(X)[y] \xrightarrow{d} H'_i(X)[y]\}\end{aligned}$$

for a one-step insertion and a termination function derived from environment moves for E. Equivalence of two environments follows from the statement that $\{(e[u], \gamma(e)[u]) \mid e \in E\}$ is a bisimulation. □

3.5 Look-ahead insertion

A special case of look-ahead insertion is defined by a set of transition equations of the following type:

$$G(X)[H(Y)] \xrightarrow{d} G'(X)[H'(Y)].$$

Reduction of a general case to special one can be done in the same way as for head insertion. Constructions for head insertion reduction to one-step insertion can be used to reduce look-ahead insertion to $(1, m)$ insertion as well. Further reductions have not been considered yet.

If $A = C$, look-ahead can be generalized:

$$G(X)[H(Y)] \xrightarrow{d} G'(X, Y)[H'(X, Y)].$$

3.6 Enrichment transformation algebra by sequential and parallel composition

In this section a transformation algebra of a one-step environment is considered. Some one-step environments allow introducing sequential and parallel compositions of behavior transformations so that they are gomomorphically transferred from corresponding behavior algebras. The following conditional equation easily follows from the definition of one-step insertion:

$$\forall(i \in I)\ ([u_i] = [u]) \implies \left[\sum_{i\in I} a_i u_i\right] = \left[\left(\sum_{i\in I} a_i\right) u\right].$$

Behavior $\sum_{i\in I} a_i$ is called a one-step behavior. Therefore the following normal form can be proved for one-step insertion:

$$[u] = \sum_{i\in I}[p_i u_i] + [\varepsilon], \tag{3.9}$$

$[u_i] \neq [u_j]$ if $i \neq j$, $[p_i u_i] \neq [0]$, p_i are one-step behaviors.

For one-step behavior $p = \sum_{i\in I} a_i$ define $h(p) = \bigcup_{i\in I}\bigcup_{c\in C} D_1(c, a_i)$.

A one-step environment E is called *regular* if:

(1) For all $(e \in E,\ c \in C)$ $(c.e \in E)$;

(2) For all $(a \in A,\ c \in C)$ $(D_1(c,a) \cap D_2(c) = \emptyset)$;

(3) The function $\varphi_\varepsilon(u)$ does not depend on u and all termination equations are consequences of the definition of this function.

3.9 Proposition *For one-step behaviors p and q and a regular environment, $[p] = [q]$ if and only if $h(p) = h(q)$.*

Proof $c.e[p] = \sum_{d\in h(p)} d.e[\Delta] + \sum_{d\in D_2(c)} d.e[p]$. □

3.10 Proposition *For nonempty $h(p)$ and a regular environment there is only one transition $(c.e)[pu] \xrightarrow{d} e[u]$ labeled by $d \in h(p)$.*

Proof It follows from the definition of a regular environment. □

3.11 Proposition *When $h(p) = \emptyset$ and the environment is regular then $e[pu] = e[0]$.*

Proof If $h(p) = \emptyset$ then $(c.e)[pu] = \sum_{d\in D_2(c)} d.e[pu] = (c.e)[0]$, $\varepsilon[pu] = \varphi_\varepsilon(pu) = \varphi_\varepsilon(0)$. □

Using this proposition we can strengthen the normal form excluding in (3.9) one-step behavior coefficients p with $h(p) = \emptyset$ and termination constant $[\varepsilon]$ if $I \neq \emptyset$ (in the case $I = \emptyset$ a constant $[0]$ can be chosen as a termination constant).

3.12 Theorem *For a regular environment, normal form is defined uniquely up to commutativity of non-deterministic choice and equivalence of one-step behavior coefficients.*

Proof First prove that if $h(p) \neq \emptyset$ then $[pu] = [qv] \iff [p] = [q]$ and $[u] = [v]$. If $d \in h(p)$ then for some $a \in A$ and $c \in C$, $d \in D_1(c,a)$. Take arbitrary behavior $e \in E$. Since E is regular, $c.e \in E$ and $d \notin D_2(c)$. Therefore $(c.e)[pu] \xrightarrow{d} e[u]$. From the equivalence of pu and qv it follows that $(c.e)[qv] \xrightarrow{d} e[v]$ and this is the only transition from $c.e[qv]$ labeled by d. Symmetric reasoning gives $d \in h(q)$ implies $d \in h(p)$ and $[p] = [q]$. Therefore $[pu] = [qv] \rightarrow [p] = [q]$ and $[u] = [v]$. The inverse is evident.

Now let $[u] = [v]$ and $[u] = \sum_{i\in I}[p_i u_i]$, $[v] = \sum_{j\in J}[q_j v_j]$ be their normal forms (exclude trivial case when $I = \emptyset$). For each transition $(c.e)[u] \xrightarrow{d} e'$ there exists a transition $(c.e)[v] \xrightarrow{d} e''$ such that $e' = e''$. Select arbitrary $d \in h(p_i)$, c and e in the same way as in the previous part of the proof. Therefore $e' = e[u_i]$ and there exists only one j such that $e'' = e[v_j] = e[u_i]$ and $d \in h(q_j)$. From symmetry and the arbitrariness of e we have $[u_i] = [v_j]$, $[p_i] = [q_j]$ and $[p_i u_i] = [q_j v_j]$. □

3.13 Theorem *For regular environment,*

$$[u] = [u'],\ [v] = [v'] \implies [uv] = [u'v'].$$

Proof Simply prove that relation $\{(e[uv], e[u'v']) =|\ [u] = [u'],\ [v] = [v']\}$, defined on insertion expressions as states, is a bisimilarity. Use normal forms to compute transitions. □

Parallel composition does not in general have a congruence property. To find the condition when it does, let us extend the combination of actions to one-step behaviors assuming that

$$p \times q = \sum_{p=a+p',\ q=b+q'} a \times b.$$

The equivalence of one-step behaviors is a congruence if $h(p) = h(q) \implies h(p\times r) = h(q\times r)$.

3.14 Theorem *Let E be a regular one-step environment and the equivalence of one-step behaviors be a congruence. Then $[u] = [u'] \wedge [v] = [v']$ Longrightarrow $[u\|v] = [u'\|v']$.*

Proof As in the previous theorem we prove that the relation $\{(e[u\|v],\ e[u'\|v'])|[u] = [u'],\ [v] = [v']\}$ defined on the set of insertion expressions is a bisimilarity. To compute transitions, normal forms for the representation of behavior transformations must be used as well as the algebraic representation of parallel composition:

$$u\|v = u \times v + u \lfloor\lfloor v + v \lfloor\lfloor u.$$

□

4 Application to automatic theorem proving

The theory of interaction of agents and environments can be used as a theoretical foundation for a new programming paradigm called *insertion programming* [17]. The methodology of this paradigm includes the development of an environment with an insertion function as a basis for subject domain formalization and writing insertion programs as agents to be inserted into this environment. The semantics of behavior transformations is a theoretical basis for understanding insertion programs, their verification and transformations. In this section an example of the development of an insertion program for interactive theorem proving is considered. The program is based on the evidence algorithm of V. M. Glushkov that has a long history [7] and recently has been redesigned in the scope of insertion programming. According to the present time classification evidence algorithm is related to some sort of tableau method with sequent calculus and is oriented to the formalization of natural mathematical reasoning. We restrict ourselves to consider only first order predicate calculus. However in the implementation it is possibles to integrate the predicate calculus with applied theories and higher order functionals.

4.1 Calculus for interactive evidence algorithm

The development of an insertion program for the evidence algorithm starts with its specification by two calculi: the calculus of conditional sequents and the calculus of auxiliary goals. The formulas of the first one are

$$(X, s, w, (u_1 \Rightarrow v_1) \wedge (u_2 \Rightarrow v_2) \wedge \dots)$$

where, $u_1, v_1, u_2, v_2, \ldots$ are first order formulas; other symbols will be explained later.

All free variables occurring in the formulas are of two classes: fixed and unknown. The first ones are obtained by deleting universal quantifiers, the second ones by deleting the existential quantifier. The expressions of a type $(u_i \Rightarrow v_i)$ are called ordinary sequents (u_i are called assumptions, v_i goals). Symbol w denotes a conjunction of literals, used as an assumption common to all sequents. Symbol s represents a partially ordered set of all free variables occurring in the formula, where partial order corresponds to the order of quantifier deletion (when quantifiers are deleted from different independent formulas new variables are not ordered). It is used to define dependencies between variables. The values of unknowns can depend on variables which only appear before them. A symbol X denotes substitution—partially defined function from unknowns to their values (terms). All logical formulas and terms with interpreted functional symbols and conditional sequents are considered up to some equivalence (associativity and commutativity of logical connectives, deMorgan identities and other Boolean identities excluding distributivity).

The formulas of the calculus of auxiliary goals are:

$$\mathrm{aux}(s, v, u \Rightarrow z, Q)$$

where s is a partial order on variables, v and u are logical formulas, and Q is a conjunction of sequents. The inference rules of the calculus of conditional sequents define the backward inference: from goal to axioms. They reduce the proof of the conjunction of sequents to the proof of each of them and the proof of an ordinary sequent to the proof of an ordinary sequent with literal as a goal. If the reduced sequent has a form $(X, s, w, u \Rightarrow z)$, where z is a literal then the rule of auxiliary goal is used at the next step:

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

In this rule z and x are unifiable literals, Y is the most general unifier extending X, and P is a conjunction of ordinary sequents obtained as an auxiliary goal in the calculus of auxiliary goals. Proving this conjunction is sufficient to prove $(X, s, w, u \Rightarrow z)$. The rule is applicable only if the substitution Y is consistent with the partial order t.

The axioms of the calculus of conditional sequents are:

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

The inference rules are:

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

applied only when (X', s', w', F') is an axiom. This rule is called the sequent conjunction rule.

$$(X, s, w, u \Rightarrow 0) \vdash (X, s, w, 1 \Rightarrow \neg u)$$
$$(X, s, w, u \Rightarrow x \wedge y) \vdash (X, s, w, (u \Rightarrow x) \wedge (u \Rightarrow y))$$
$$(X, s, w, u \Rightarrow x \vee y) \vdash (X, s, w, \neg x \wedge u \Rightarrow y)$$

This rule can be applied in two ways permuting x and y because of commutativity of disjunction.

$$\begin{gathered}
(X, s, w, u \Rightarrow \exists\, xp) \vdash (X, \mathrm{addv}(s, y), w, \neg(\exists\, xp) \wedge u \Rightarrow \mathrm{lsub}(p, x := y))\\
(X, s, w, u \Rightarrow \exists\, x(z)p) \vdash (X, \mathrm{addv}(s, (z, y)), w, \neg(\exists\, xp) \wedge u \Rightarrow \mathrm{lsub}(p, x := y))\\
(X, s, w, u \Rightarrow \forall\, xp) \vdash (X, \mathrm{addv}(s, a), w, u \Rightarrow \mathrm{lsub}(p, x := a))\\
(X, s, w, u \Rightarrow \forall\, x(z)p) \vdash (X, \mathrm{addv}(s, (z, a)), w, u \Rightarrow \mathrm{lsub}(p, x := a))
\end{gathered}$$

In these rules y is a new unknown and a a new fixed variable. The function $\mathrm{lsub}(p, x := z)$ substitutes z into p instead of all free occurrences of x. At the same time it joins z to all variables in the outermost occurrences of quantifiers. Therefore a formula $\exists\, x(z)p$, for instance, appears just after deleting some quantifier and introducing a new variable z. The function $\mathrm{addv}(s, y)$ adds a new element, y, to s without ordering it with other elements of s, and $\mathrm{addv}(s, (z, y))$ adds y, ordering it after z.

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

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

Here a is a new fixed varaible and u is a new unknown as in the calculus of conditional sequents.

4.2 A transition system for the calculus

Each of two calculi can be considered as a non-deterministic transition system. For this purpose the sequent conjunction rule must be split into ordinary rules. First we introduce the extended conditional sequent as a sequence $C_1; C_2; \ldots$ of conditional sequents and the following new rules:

$$\begin{gathered}
(X, s, w, H_1 \wedge H_2) \vdash ((X, s, w, H_1); (X, s, w, H_2));\\
((X, s, w, H_1 \wedge H_2); P) \vdash ((X, s, w, H_1); (X, s, w, H_2); P);\\
\frac{C \vdash C'}{(C; P) \vdash (C'; P)}.
\end{gathered}$$

For axioms (X', s', w', F) the ruless are:

$$\begin{gathered}
((X', s', w', F); (X, s, w, H)) \vdash (X', s', w, H);\\
((X', s', w', F); (X, s, w, H); P) \vdash ((X', s', w, H); P).
\end{gathered}$$

Futhermore the calculus of auxiliary goals always terminates (each inference is finite). Therefore the rule of auxiliary goals can be considered as a one-step rule of the calculus of conditional sequents. Now the transitions of a transition system are transitions of this calculus. Each transition corresponds to some inference rule. The states of successful termination are axioms and a formula P is valid if aqnd only if one of the successful termination states is reachable from the initial state $(X_0, s_0, 1, 1 \Rightarrow P)$.

Let us label the transitions by actions. Each action is a message on which a rule has been applied in the corresponding transition. For example a transition corresponding to the rule of deleting a universal quantifier is:

$$(X, s, w, u \Rightarrow \forall\, xp) \xrightarrow{\text{mes}} (\text{addv}(s, a), w, u \Rightarrow \text{lsub}(p, x := a))$$

where a message "To prove a statement $\forall\, xp$ let us consider an arbitrary element a and prove q" is used as an action mes where $q = \text{lsub}(p, x := a)$. Other rules can be labeled in a similar way. Proving a statement T is therefore reduced to finding the trace which labels the transition of a system from the initial state $(\emptyset, \emptyset, 1, 1 \Rightarrow T)$ to the state corresponding to one of the axioms. If the axioms are defined as the states of successful termination, the problem is to find the trace from the initial state to one of the successfully terminated states. The sequence of messages corresponding to this trace is a text of a proof of a statement T.

This form of an evidence algorithm representation can be implemented in the system of insertion programming using a trivial environment which allows arbitrary behavior of the inserted agent. In automatic mode a system is looking for a trace with successful termination, if possible, and prints the proof when the trace is found. In interactive mode a system addresses to a user each time when it is necessary to make a non-deterministic choice. A user is offered several options to choose an inference rule and a user makes this choice. It is possible to go back and jump to other brunches. An environment which provides such possibilities is a proof system based on an evidence algorithm.

4.3 Decomposition of the transition system

A more interesting implementation of the evidence algorithm can be obtained if a state of a calculus is split into an environment and an agent inserted into this environment. This splitting is very natural if a substitution, partial order, conjunction of literals, or assumption of a current sequent is considered as a state of the environment and the goal of the current sequent, as well as all other sequents, is considered as an agent. Moreover it is possible to change the conjunction of all other sequents to a sequential composition. Call this agent a formula agent. A formula agent is considered as a state of the transition system used for the representation of an agent. To compute the behavior of agents, a recursive unfolding function must be defined on the set of agent expressions. It is easy to extract the unfolding function and the transition relation for a formula agent from the inference rules. Actions produced by formula agent define necessary changes in the environment and are computed by the insertion function. For example a formula agent $(\texttt{prove}\ \forall\, xp; P)$ is unfolded according to its recursive definition to an agent expression $Q = \texttt{fresh}\ C(\forall\, xp).P$, where $\texttt{fresh}\ C(\forall\, xp)$ is an action which substitutes a new fixed variable for p. This substitution is performed by the insertion function with transition

$$e[Q] \xrightarrow{\text{mes}} e'[\texttt{prove}\ \text{lsub}(p, x := a).P],$$

where mes is a message considered before and the transition from e to e' corresponds to the generation of a new fixed a.

In a general extended conditional sequent

$$((X, s, w, u \Rightarrow v); P)$$

is decomposed into an environment state and a sequential composition of agents. The environment state is

$$(X, s, w, u).$$

Initially $(\emptyset, \emptyset, 1, 1)$. The main types of agent expressions are:

- `prove` v; v is a simple sequent or formula,
- `prove` $(H_1 \wedge H_2 \wedge \dots)$; H_i are simple sequents,
- `block` P; P is an agent,
- `end_block` w; w is a conjunction of literals.

The following equations define unfolding and insertion function for blocks.

- `prove` $(H_1 \wedge H_2 \wedge \dots) =$ (`prove` H_1; `prove` $H_2 \wedge \dots$);
- `prove` $(u \Rightarrow v) =$ `block` (`Let` u.(`ask` $0.m_1 +$ `ask` $1.m_2$.`prove` $v).m_3$, where
  - $m_1 =$ `mes` ($\neg u$ `is evident by contradiction`);
  - $m_2 =$ `mes` (`prove` $u \Rightarrow v$);
  - $m_3 =$ `mes` (`sequent proved`);
- e[`block` $P.Q$] $=$ `mes`(`begin`).$e[P;$ `end_block` $w; Q]$, where $e = (X, s, w, F)$;
- e[`end_block` w'] $=$ `mes`(`end`).$e'[\Delta]$, where $e' = (X, s, w', 1)$ if $e = (X, s, w, u)$.

Actions are easily recognized by dots following them. The meaning of actions `mes` x and `block` P is clear from these definitions. Other actions will be explained later.

4.3.1 Unfolding conjunction and disjunction

In this section unfolding rules for conjunction and disjunction are explained as well as some auxiliary rules.

```
prove $(u \Rightarrow 0)$ = prove $(\neg u)$;
prove $(x \wedge y)$ = (prove $(x)$; prove $(y)$);
prove $(x \vee y)$ = block(
    Let $(\neg x)$.(
        ask 0.mes ($x$ is evident by contradiction)
         +
        ask 1.mes(To prove $x \vee y$ let $\neg x$, prove $y$).
        prove $(y)$
    ); mes (disjunction proved)
) + (
    Let $\neg y$.(
        ask 0.mes($y$ is evident by contradiction)
        $+$
        ask 1.mes(To prove $x \vee y$ let $\neg y$, prove $x$).
        prove $x$
     ); mes(disjunction proved)
)
```

4.3.2 Unfolding quantifiers

In these definitions e' is a state of an environment after generating a new unknown (`fresh` V) or fixed (`fresh` C) variable y.

```
prove $\exists xp$ = (fresh $V$ prove $\exists xp$).$\Delta$;
prove $\forall xp$ = (fresh $C$ prove $\forall xp$).$\Delta$;
$e$[fresh $V$ $q$] = $e'$[get_fresh $y$ $q$];
$e$[fresh $C$ $q$] = $e'$[get_fresh $y$ $q$];
get_fresh $y$ prove $\exists xp$ = mes(
   To prove $\exists xp$ find $x = y$ such that $p$
).block(
   Let $\neg\exists xp$.
   prove lsub($p, x := y$);
   mes(existence proved)
);
get_fresh $y$ prove $\forall xp$ = mes(
   Prove $\forall xp$. Let $y$ is arbitrary constant.
).(
   prove lsub($p, x := y$);
   mes(forall proved)
)
```

4.3.3 An auxiliary goal

The calculus of the auxiliary goal is implemented on the level of the environment. It is hidden from an external observer who can only see the result, that is the auxiliary goal represented by a corresponding message.

```
prove $z$ = mes(To prove $z$ find auxiliary goal).start_aux $z$;
$e$[start_aux $z$] = $e'$[prove_aux($z, Q$)];
prove_aux($z$, 1) = mes($z$ is evident);
prove_aux($z, Q$) = mes(auxiliary goal is $Q$).prove $Q$.
```

Note that the rules for insertion of formula agents can be interpreted as one step insertions except for the rules for `start_aux` and `fresh`. Both can be interpreted as an instantiation of the rule:

$$e\frac{\xrightarrow{a} e'[v],\ u \xrightarrow{a} [u']}{e[u] \xrightarrow{a} e'[v; u']}$$

In the case of sequential insertion this rule can also be considered as a one-step insertion rule because in this case $e'[v; u'] = (e'[v])[u]$.

The last step in the development of an insertion program is verification. The correctness of the specification calculi is proved as a sound and completeness theorem comparing it with algorithms based on advanced tableau methods. After decomposition we should prove the bisimilarity of the corresponding initial states of two systems. We can also improve the insertion program. In this case we should do optimization preserving insertion equivalence of agents.

4.4 A proving machine

Formula agent actions can be considered as instructions of a proving machine representing an environment for such kind of agents. Here are some of the main instructions of the proving machine used for the development of the evidence algorithm kernel.

- `Let` <formula> adds the formula to assumptions about the environment. It is used for example for the unfolding sequent: $\texttt{prove}\ (u \Rightarrow v) = \texttt{Let}\ u.\texttt{prove}\ v$ or for the unfolding disjunction
$$\texttt{prove}\ (u \vee v) = \texttt{Let}\ \neg v.\texttt{prove}\ v + \texttt{Let}\ \neg v.\texttt{prove}\ u.$$
An essential reconstruction of the environment is performed each time a new assumption is added to the environment. Formulas are simplified, conjunctive literals are distinguished, substitution is applied etc. Moreover assumptions or a literal conjunction can be simplified up to 0 (false).

- `tell` <literal> adds a literal to conjunction of literals without reconstruction of the environment.

- `ask` 0 checks inconsistency of the environment state (0 in assumptions or in literals).

- `ask` 1 checks that there is no explicit inconsistency.

- `start_aux` <literal> starts the calculus of auxiliary goals:
$$e[\texttt{start\_aux}\ p.Q] = \texttt{Let}\ \neg p.(\texttt{prove}\ P_1 + \texttt{prove}\ P_2 + \cdots)$$
where $P_1, P_2, \ldots$ are auxiliary goals (conjunctions of sequents) extracted from assumptions according to the calculus of auxiliary goals (actually the real relations are slightly more complex, because they include messages about the proof development and they anticipate the case when there are no auxiliary goals at all).

- `fresh V` <formula> substitutes a new unknown into the formula.

- `fresh C` <formula> substitutes a new fixed variable into the formula.

- `solve` <equation> is used for solving equations in the case when equality is used.

- `block` <program> localizes all assumptions within the block. It is used for example when conjunction is proved.

- `Mesg` <text> inserts the printing of messages at the different stages of the proof search.

- `start_lkb` <literal>. It is used to address the local knowledge base for extracting auxiliary goals from assumptions presented in this base. The local knowledge base is prepared in advance according to requirements of the subject domain where the proof is searched for.

The advantages of a proving machine (or more generally an insertion machine for other environment structures) is that it is possible to change in a wide area the algorithms of a proof search without changing the structure of the environment by varying the recursive unfolding rules of formula agents. Moreover the environment itself can be extended by introducing new instructions into the instruction set and adding new components to an environment state.

To run a program on a proving machine some higher level environment should be used to implement back-tracking. Such an environment can work in two modes: interactive and automatic. The following equations demonstrate the approximate meaning of these two modes.

The interactive mode:

$$e[a_1.u_1 + a_2.u_2 + \dots] = \texttt{mes}(\texttt{select}\ a_1, a_2, \dots).(a_1.e'[u_1] + a_2.e'[u_2] + \dots) + \texttt{back}.e''[u] + \Delta.$$

The automatic mode:

$$e[a_1.u_1 + a_2.u_2 + \dots] = e'[(a_1.u_1); (\texttt{return if fail or stop}); (a_2.u_2 + \dots)].$$

This is a depth first search. It works only with restrictions on the admissible depth. A breadth first search is more complex.

5 Conclusions

A model of interaction of agents and environments has been introduced and studied. The first two sections extend and generalize results previously obtained in [15]. The algebra of behavior transformations is a good mathematical basis for the description and explanation of agent behavior restricted by the environment in which it is inserted. System behavior has two dimensions. The first one is a branching time which defines the height of a behavior tree and can be infinite (but no more than countable). The second one is a non-deterministic branching at a given point. Our construction of a complete behavior algebra used for the characterization of bisimilarity allows for branching of arbitrary cardinality.

The restriction of the insertion function to be continuous is too broad and we consider a more restricted classes of equationally defined insertion functions and one-step insertions to which more general head insertion is reduced. The question about reducing or restricting look-ahead insertion is open at this moment. At the same time the algebra of behavior transformations based on regular one-step insertion can be enriched by sequential and parallel compositions.

The algebra of behavior transformations is the mathematical foundation of a new programming paradigm: insertion programming. It has been successfully applied for automatic theorem proving and verification of distributed software systems. In [16] operational semantics of timed MSC (specification language of Message Sequencing Charts) has been defined on the basis of behavior transformations. This semantics has been used for the development of tools for verification of distributed systems.

References

[1] S. Abramsky, A domain equation for bisimulation, *Information and Computation* **92** (2) (1991), 161–218.

[2] L. Aceto, W. Fokking, and C. Verhoef, Structural operational semantics, in: *Handbook of Process Algebra* (J. A. Bergstra, A. Ponce, and S. A. Smolka, eds.), North-Holland, 2001.

[3] P. Aszel and N. Mendler, A final coalgebra theorem, in: *LNCS 389*, Springer-Verlag, 1989.

[4] P. Azcel, J. Adamek, S. Milius, and J. Velebil, Infinite trees and comletely iterative theories: a coalgebraic view, *TCS* **300** (1–3) (2003), 1–45.

[5] J. A. Bergstra and J. W. Klop, Process algebra for synchronous communications, *Information and Control* **60** (1/3) (1984), 109–137.

[6] J. A. Bergstra, A. Ponce, and S. A. Smolka, eds., *Handbook of Process Algebra*, North-Holland, 2001.

[7] A. Degtyarev, J. Kapitonova, A. Letichevsky, A. Lyaletsky, and M. Morokhovets, Evidence algorithm and problems of representation and processing of computer mathematical knowledge, *Kibernetika and System Analysis* (6) (1999), 9–17.

[8] R. J. Glabbeek, The linear time—branching time spectrum i. the semantics of concrete, sequential processes, in: *Handbook of Process Algebra* (J. A. Bergstra, A. Ponce, and S. A. Smolka, eds.), North-Holland, 2001.

[9] V. M. Glushkov, Automata theory and formal transformations of microprograms, *Kibernetika* (5).

[10] V. M. Glushkov and A. A. Letichevsky, Theory of algorithms and descrete processors, in: *Advances in Information Systems Science* (J. T. Tou, ed.), vol. 1, Plenum Press, 1969.

[11] J. A. Goguen, S. W. Thetcher, E. G. Wagner, and J. B. Write, Initial algebra semantics and continuous algebras, *J. ACM* (24) (1977), 68–95.

[12] M. Hennessy and R. Milner, On observing nondeterminism and concurrency, in: *Proc. ICALP'80, LNCS 85*, Springer-Verlag, 1980.

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

[14] A. Letichevsky, Algebras with approximation and recursive data structures, *Kibernetika and System Analysis* (5) (1987), 32–37.

[15] A. Letichevsky and D. Gilbert, Interaction of agents and environments, in: *Resent trends in Algebraic Development technique, LNCS 1827* (D. Bert and C. Choppy, eds.), Springer-Verlag, 1999.

[16] A. Letichevsky, J. Kapitonova, V. Kotlyarov, A. Letichevsky, Jr., and V. Volkov, Semantics of timed msc language, *Kibernetika and System Analysis* (4).

[17] A. Letichevsky, J. Kapitonova, V. Volkov, V. Vyshemirskii, and A. Letichevsky, Jr., Insertion programming, *Kibernetika and System Analysis* (1) (2003), 19–32.

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

[19] R. Milner, *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

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

[21] R. Milner, The polyadic π-calculus: a tutorial, Tech. Rep. ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK (1991).

[22] D. Park, Concurrency and automata on infinite sequences, in: *LNCS 104*, Springer-Verlag, 1981.

[23] M. Roggenbach and M. Majster-Cederbaum, Towards a unified view of bisimulation: a comparative study, *TCS* **238** (2000), 81–130.

[24] J. Rutten, Coalgebras and systems, *TCS* **249**.