# A general theory of action languages

Alexander Letichevsky
Glushkov Institute of Cybernetics
National Academy of Sciences of Ukraine
let@d105.icyb.kiev.ua

David Gilbert
Department of Computer Science
City University, UK
drg@cs.city.ac.uk

July 13, 1998

**Abstract**

We present a general theory of action-based languages as a paradigm for the description of those computational systems which include elements of concurrency and networking, and extend this approach to describe distributed systems and also to describe the interaction of a system with an environment.

As part of this approach we introduce the Action Language as a common model for the class of nondeterministic concurrent programming languages and define its intensional and interaction semantics in terms of continuous transformation of environment behaviour. This semantics is specialised for programs with stores, and extended to describe distributed computations.

Keywords: interaction, semantics, behaviour, environments, distribution.

## Contents

**1 Introduction** **2**

**2 The Action Language** **3**
2.1 Actions . . . . . 4
2.2 Procedure calls . . . . . 5

**3 Semantics** **5**
3.1 Behaviours . . . . . 7
3.2 Behaviours and transition systems . . . . . 9
3.3 Examples . . . . . 10
3.3.1 Finite (rational) behaviours . . . . . 10
3.3.2 Algebraic behaviours . . . . . 10
3.3.3 Parameterised algebraic behaviours . . . . . 11
3.3.4 Behaviours over state spaces . . . . . 11
3.4 Syntactic algebras . . . . . 11
3.5 Intensional semantics . . . . . 12
3.6 Interaction semantics . . . . . 13
3.7 Compositionality of parallel and sequential compositions . . . . . 16
3.8 Computational semantics . . . . . 16

**4 Environments** **17**

**5 Action languages over stores** **19**
5.1 Synchronous and asynchronous communication . . . . . . . . . . . . . . . . . . 19
5.2 Local store . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
5.3 Channels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

**6 Distributed action languages** **22**

**7 Conclusions** **23**

## 1 Introduction

We present a general theory of action-based languages as a paradigm for the description of those computational systems which include elements of concurrency and networking, and extend this approach to describe distributed systems and also to describe the interaction of a system with an environment. Our claim is that we are able to characterise most existing computational and interactive systems with our approach, and to relate the concepts of computation and interaction. We hope that our approach will facilitate the design and construction of new computational and interactive systems in the future.

Our basic notion is that of an *action* which transforms the state of a world; actions are performed by agents[1] whose behaviour is changed as a result. We distinguish between an agent and its state and thus speak of an agent in a given state; special types of agents are programs (which have a syntactical representation) and environments (which usually are not syntactically represented, and into which programs can be inserted). Behaviours are agents in a given state considered up to bisimilarity, or possibly to a weaker equivalence. Each agent may be represented as a transition system labelled by actions from a corresponding action domain and whose action algebra describes combination, non-deterministic and sequential composition. Thus we distinguish between *primitive actions* and *compound actions*, the latter being formed from combination of other actions.

Interaction between agents is of two types. The first is expressed by the parallel composition of agents over the same action domains and is characterised by the combination of actions or interleaving. The second is expressed by the insertion of an agent into some environment and results in the transformation of the environment into a new environment. Some informal examples of environments are:

- a computer, or interpreter for some programming language (which does not perform global analysis and only considers actions performable at some moment of time),
- a server on a computer network, or a software system which manipulates queries considered as actions of programs, where some actions can be performed immediately and others are suspended,
- an interactive interface connecting a program with a user, where the user may interrupt the execution of the program and perform his own action.

Interactive computing is a well-established technique applied to many problem domains, for example in the construction of controllers, operating systems, programming environments,

[1]The term "agent" is used as a notion which formalises real objects such as programs, environments, users, clients, servers and agents as in the meaning of "software agents"[6]

expert systems etc. However this paradigm has not been regarded as fundamental until the relatively recent advent of widely available communications facilities which abstract from physical locations and networks, coupled with a very high rate of accessibility to computers. Peter Wegner has documented and explored the paradigm shift from algorithms to interaction in his recent CACM articles [30, 31]. Our view is of computation *and* interaction as two somewhat orthogonal concepts as opposed to the view of computation *as* interaction characterised by Milner in [24].

We believe that descriptions of interactive systems should be made using formalisms based on very generalised (abstract) languages, and that a sound semantics needs to be given for them. This paper presents our first attempt in this direction. We base our approach on that of a general abstract Action Language (AL) as a common model for the class of nondeterministic concurrent programming languages [19] (ncpl) and consider them as interactive programming languages by giving a compositional semantics for them. The set of continuous transformations of the behaviour algebra is used as a semantic domain for this purpose. This approach is in some sense a generalization of the idea of discrete transformer introduced by V.Glushkov in [10] and considered as a model of computation (see also [11]). As a further extension, we give a distribution semantics for those ncpl languages which have the notion of store components.

Nondeterministic concurrent programming languages (ncpl) are languages which employ as primitive constructs nondeterministic choice, parallel and sequential composition. The best known are the languages based on CCS [21] and the $\pi$-calculus [22] of R. Milner, CSP [16] of C.A.R. Hoare, and process algebra [5] which were designed to study communication and interaction in concurrent processes. Another class are the concurrent constraint programming languages [7, 9, 26, 27], which appeared during the last decade and are very popular nowadays, and combine the properties of computation (over relations) and interaction in a very high level and abstract manner. All of these languages use all three characteristic constructs of ncpl. Nondeterministic choice is an important feature of declarative programming and also of specification languages, although it may be present implicitly. For example the choice of rewriting rules and redexes in algebraic specifications as well as the choice of clauses in logic programs considered as specifications of subject domains, are nondeterministic. Parallel composition may also be present implicitly as the possibility of simultaneously computing the values of subexpressions of algebraic expressions or simultaneously solving constraints.

An important advantage of our approach is that it can easily be specialised to describe specific features of languages belonging to the ncpl class. For example some special features of the cc family: a store, variables and synchronisation mechanisms. Another advantage lies in the use of our model to design tools such as interpreters, simulators and workbenches for ncpl languages.

## 2 The Action Language

The general abstract Action Language (AL) is a common model for the class of ncpl languages. The abstract syntax of the Action Language is as follows, where the syntax of `Act` and `ProcedureCall` are the parameters of each particular language of the class.

$$\mathtt{Prog} ::= \ \mathtt{Act} \ | \ \mathtt{ProcCall} \ | \ (\mathtt{Prog} + \mathtt{Prog}) \ | \ (\mathtt{Prog} \| \mathtt{Prog}) \ | \ (\mathtt{Prog} ; \mathtt{Prog})$$

## 2.1 Actions

The meaning of actions is defined by some algebra of actions $A$ (action algebra) and if it does not result in any contradictions then action expressions (considered up to their equivalence) will be identified with the actions themselves. The language is called a language over $A$ if all action expressions are interpreted in the algebra $A$.

The main operation of the algebra of actions is a binary ac-operation (associative and commutative) denoted as $\times$ and called the combination of actions. There is also the empty action $\delta$ which is the neutral element for combination and the zero element 0 (the impossible action). Therefore the algebra of actions is a commutative monoid. It may include also some other operations, as for instance in [23]. Among the different operations we are interested in are nondeterministic choice and sequential composition of actions. The main properties of these operations are illustrated in Figure 1 (nondeterministic choice of $a$ and $b$ is $a + b$, sequential composition is $ab$).

$$\begin{gathered}
a \times b = b \times a \\
(a \times b) \times c = a \times (b \times c) \\
a \times \delta = \delta \times a = a \\
a \times 0 = 0 \times a = 0 \\
\\
a + a = a \\
a + b = b + a \\
(a + b) + c = a + (b + c) \\
a + 0 = a \\
\\
(ab)c = a(bc) \\
a\delta = \delta a = a \\
a0 = 0a = 0 \\
\\
(a + b) \times c = a \times c + b \times c \\
(a + b)c = ac + bc \\
c(a + b) = ca + cb
\end{gathered}$$

Figure 1: Relations of an algebra of actions

An action is called *deterministic* if it cannot be represented as a nondeterministic choice of two different nonzero actions. A deterministic action is called *primitive* if it is $\delta$ or 0, or it cannot be split into a combination of actions different to $\delta$ and 0. Nondeterministic or nonprimitive actions are called *compound* ones. The algebra of actions is called *primitive* if

1. It is generated by primitive actions;
2. The representation of a nonzero action as a (finite) sum of nonzero deterministic actions is unique up to the commutativity and idempotence relations for sum.

**Theorem 1** *Each action algebra is a homomorphic image of a primitive action algebra.*

A free action algebra defined by equations of Figure 1 is a primitive one and any action algebra is a homomorphic image of some free action algebra.

In the sequel we shall consider only primitive action algebras without explicit reference to this fact.

In real languages the combination of actions is usually either parallel (simultaneous) performance of information independent computations, or interaction (for example send and receive operations for the exchange of data between two processes). Combinations expressing multiparty communications which are performed in parallel with communication and interaction are also possible. The complexity of actions and their compositions depends on the point of view and level of abstractness desired.

The sequential composition of two actions is a nontrivial (different from 0) action if these actions are interpreted as functions or relations and in this case the new action is equivalent to the sequential performance of two actions. Nondeterministic composition of actions is used for technical reasons, and as it will be shown later usually can be eliminated at the level of program transformations.

The simplest examples of action algebras are Hoare action algebra ($a \times a = a, a \times b = 0$ if $a \neq b$) and Milner action algebra ($a \times \overline{a} = \tau, a \times b = 0$ if $b \neq \overline{a}$). Relations on some set of states or transformations defined by assignments on a set of memory states are also the examples of action algebras. Others include Milne's Circal calculus [20], the algebra of Hennessy [15], LOTOS [17, 29] and its extensions (e.g. LOTCAL [8]).

### 2.2 Procedure calls

The syntax of procedure calls is another parameter of action language and each program is associated with a set of procedure definitions and also an algorithm which *unfolds* any procedure call to a program. We do not consider the details of procedure definitions, parameter passing, etc. in order not to restrict generalisations. As an example the utilisation of rewriting technique for the definition of unfolding algorithm is quite useful.

Thus a program may be considered as an infinite object which can be obtained by means of infinite (in the case of nontrivial recursive definitions) unfolding of procedure calls. Formally this infinite program may be considered as the limit of a directed set of finite programs using an approximation relation with a bottom element added to the set of all programs. If the unfolding algorithm is partially defined then the resulting program may contain occurrences of the bottom element. The unfolding process will be formally defined later.

## 3 Semantics

Semantics is a function defined on the expressions of a language and which maps the program expressions of a language to their meaning in some semantic domain. Different semantic functions reflect different levels of abstractions and different properties of a program. We are interested in two kinds of semantics: computational and interaction ones; we also want our semantical functions to be compositional which means that the meaning of a composition of programs is a corresponding composition of their meanings.

A semantic domain is usually equipped with some topology which provides the possibility of constructing infinite objects using passage to the limits. Classical examples of such domains are Scott functional domains [14], see also [3]. In this paper we shall use domains which are continuous algebras [12, 13] or algebras with approximation [18]. The latter is a poset with partial order called an approximation relation, a minimal element $\perp$, and operations which are continuous w.r.t. the approximation relation. We shall also assume that in each algebra with approximation which we consider there is given a subalgebra of finite elements which contains the bottom element $\perp$ and that all other elements are the limits of ordered sets of finite elements. In [18] it has been shown how an arbitrary algebra with approximation can be completed by such limits.

We speak about *computational semantics* of a program if it has been designed to compute some function or relation. In this case the meaning of a program is that function or relation itself. This corresponds to the traditional denotational semantics of programs. However the execution of any program takes place in some environment which interacts with the program, performing the sequences of actions defined by this program or allowing these sequences to be performed. If the environment only supports the computational properties of a program it is passive and does not change the operational meaning of a program. This interaction is described by the traditional operational semantics of programs [25]. However the environment may be more active, and change the predefined behaviour of a program within wide limits. For example, it may contain some other programs designed independently and intended to interact and communicate with the given program at its run time. Therefore the interaction semantics must include an environment as the main parameter. The classical theories of communication (CCS, CSP, $\pi$-calculus) are based on the notions of transition systems and bisimulation, and consider interaction within the scope of the parallel composition of agents. The influence of the environment sometime are expressed as an explicit language operation such as restriction in CCS or hiding in CSP.

Our approach in describing the interaction semantics of the Action Language is also based on the notion of bisimilarity, but the environment is considered as a semantic notion and is not explicitly included in the program. The meaning of an interactive program is defined as a transformation of an environment which corresponds to inserting the program into its environment. When this action is performed the environment changes and this change is considered as the main property of a program which is to be described by its meaning.

In order to realize this approach first we formalise the notion of *behaviour* in terms of algebras with approximation. Each behaviour is an element of some behaviour algebra over an algebra of actions. This behaviour defines some transition system (with a given initial state); two behaviours are equal iff the initial states of corresponding transition systems are bisimilaryly equivalent. Therefore behaviours are the invariants of transition systems considered up to the bisimilarity relation. Then each program is assigned its behaviour which is defined independently of its environment. The behaviour of a program is called its *intensional meaning*. The construction of the intensional meaning of a program is built in two steps. The first step is to convert the syntactic algebra of the AL to a continuous syntactic algebra by eliminating procedures calls. This conversion is realized by homomorphism which identifies equivalent procedure calls (having the same infinite unfoldings). Then the continuous syntactic algebra is homomorphically mapped to the behaviour algebra by means of continuous homomorphism which provides programs of the AL by behavioral meaning.

After introducing the intensional semantics of programs the notion of an *environment* is presented. The environment is defined as a four-tuple which includes as a component a subset of some behaviour algebra (the algebra of environment behaviours) over the action algebra different from the action algebras of the languages which are accepted by this environment. This subset is closed over transitions. Then the algebra of continuous transformations of environment behaviours is introduced and the continuous homomorphism of intensional semantics algebra of the AL to the algebra of transformations is defined providing each program with its interaction meaning. The homomorphism is determined by a residual function which sets the relationships between the actions of a program and those of an environment.

## 3.1 Behaviours

A behaviour over an action algebra $A$ is considered as an element of an *algebra of behaviours* over $A$ (sometimes called a behaviour algebra). This algebra is an algebra with approximation (poset with a minimal element and continuous operations[2]). It has two operations, the first being denoted by $+$ and is the internal binary aci-operation (idempotent ac-operation). This operation corresponds to nondeterministic choice. The second operation is prefixing $au$, $a$ being an action, $u$ being a behaviour. The minimal element of a behaviour algebra is denoted $\perp$. The empty behaviour $\Delta$ performs no actions and usually denotes the termination of a process. The impossible behaviour 0 is denoted by the same symbol as the impossible action and is the neutral element for nondeterministic choice.

Generating relations of any algebra of behaviours are shown in Figure 2. The symbols $a, b$ are actions, $u, v, w$ are behaviours. All other relations are consequences of these ones.

$$\begin{gathered} u + v = v + u \\ (u + v) + w = u + (v + w) \\ u + u = u \\ u + 0 = 0 + u = u \\ \\ (a + b)u = au + bu \\ 0u = 0 \end{gathered}$$

Figure 2: Relations of an algebra of behaviours

The approximation relation of the algebra of behaviours over $A$ is the minimal partial order which satisfies the relations presented in Figure 3.

$$\begin{gathered} \perp \sqsubseteq u \\ \\ u \sqsubseteq v \Rightarrow u + w \sqsubseteq v + w \\ \\ u \sqsubseteq v \Rightarrow au \sqsubseteq av \end{gathered}$$

Figure 3: Approximation for behaviours

The elements of the minimal sub-algebra $F_{\mathtt{fin}}(A)$ of the algebra of behaviours over $A$ that is a sub-algebra generated by the empty behaviour, the impossible behaviour and the bottom element are called *finite behaviours.* All other behaviours are assumed to be the limits (least upper bounds) of the countable directed sets of finite elements. The algebra of behaviours which includes all such limits is denoted $F(A)$. It is defined uniquely up to the continuous isomorphism and all behaviour algebras considered in the paper are assumed to be subalgebras of this algebra.

[2]A function $f : D \to D$ on a poset $D$ is called continuous if it is monotone and for each directed set $\{x_i | i \in I\}$ if this set is convergent i.e. has the least upper bound $\coprod_{i \in I} x_i$ then the set $\{f(x_i) | i \in I\}$ is also convergent and $f(\coprod_{i \in I} x_i) = \coprod_{i \in I} f(x_i)$. An operation is continuous if it is continuous as a function of each of its arguments.

From the primitivity of an action algebra it follows that each behaviour $u$ can be represented in the form

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

where $a_i$ are nonzero deterministic actions, $u_i$ are behaviours, $I$ is a finite (for finite elements) or infinite (but countable) set of indices, $\varepsilon = \Delta, \perp, \Delta + \perp, 0$ (*termination constants*).

**Theorem 2** *If all summands in representation (1) are different then this representation is unique up to the associativity and commutativity of nondeterministic choice.*

For a finite behaviour $u$ the statement of this theorem is true because the set of behaviours of a type $av$ with $a$ deterministic such as $u = av + v'$ does not depend on the representation of $u$ as an expression of the behaviour algebra considered up to the commutativity and assosiativity of nondeterministic choice. The same is true for the termination constants. For infinite behaviours the theorem follows from the uniqueness of the representation of $u$ as an infinite sum

$$u = \sum_{a \in A_0 \wedge P(av)} av + \varepsilon$$

where $A_0$ is the set of deterministic actions, $\varepsilon$ is the termination constant and the predicate $P$ is defined as follows:

$$P(z) \Leftrightarrow \exists x \in F_{\mathtt{fin}}(A).x + \perp \sqsubseteq u \wedge z = \coprod_{x \sqsubseteq y \in F_{\mathtt{fin}}(A), y + \perp \sqsubseteq u} y$$

Parallel and sequential compositions are introduced as derived operations using the recursive definitions presented in Figure 4 where $u, v, w$ are behaviours, $a$ and $b$ are deterministic actions. Parallel composition is denoted by $\|$ and sequencing by ; (however we will sometimes omit this latter operator as in Figure 4).

$$\begin{aligned}
(u+v)\|w &= u\|w + v\|w \\
u\|(v+w) &= u\|v + u\|w \\
(au)\|(bv) &= (a \times b)(u\|v) + a(u\|bv) + b(au\|v) \\
\Delta\|u &= u\|\Delta = u \\
0\|u &= u\|0 = 0 \\
\perp \|u &= u\| \perp = \perp
\end{aligned}$$

$$\begin{aligned}
(u+v)w &= uw + vw \\
(au)v &= a(uv) \\
\Delta u &= u\Delta = u \\
0u &= 0 \\
\perp u &= \perp
\end{aligned}$$

Figure 4: The definition of parallel and sequential composition of behaviours

These definitions uniquely determine sequential and parallel composition on finite elements and may be uniquely extended to all others by continuity if the corresponding limits are in the algebra of behaviours [3] under consideration.

**Theorem 3** *Sequential composition is associative, parallel composition is associative and commutative.*

The theorem is proved first for finite behaviours and then extended to the infinite ones. The proofs for finite behaviours use induction on the length of a behaviour which is defined so that $\mathtt{length}(\varepsilon) = 0$ for the termination constant $\varepsilon$, $\mathtt{length}(au) = \mathtt{length}(u) + 1$, and $\mathtt{length}(u + v) = \mathtt{max}(\mathtt{length}(u), \mathtt{length}(v))$. The following (expansion) theorem gives the explicit form of parallel composition. In this theorem $E(u)$ is a termination constant for a behaviour $u$, and $u$ and $v$ are finite behaviours.

**Theorem 4** *Let* $u = \sum a_i u_i + E(u), \quad v = \sum a_j v_j + E(v)$. *Then*

$$u \| v = \sum (a_i \times b_j)(u_i \| v_j) + \sum a_i (u_i \| b_j v_j) + \sum b_j (a_i u_i \| v_j) + E(u) \| v + u \| E(v)$$

Proof is by induction on the sum of the lengths of $u$ and $v$. From this theorem the associativity of parallel composition is proved by direct computation (other properties of compositions are trivial). To simplify the computations it is useful to distinguish between final and nonfinal behaviours. A behaviour $u$ is called *final* if it is equal to 0 or $E(u) \neq 0$ and *nonfinal* otherwise. The associativity law is first proved for nonfinal behaviours, then for parallel composition of three behaviors at least one of which is a termination constant, and then for a general case.

## 3.2 Behaviours and transition systems

We present the well known notions of a transition system and (partial) bisimulation, adapted to our collection of termination constants.

**Definition 1** *A transition system over the set of actions $A$ is a set $S$ of states with a transition relation $s \xrightarrow{a} s', \quad s, s' \in S, \quad a \in A$ and two subsets $S_\Delta$ and $S_\perp$ called correspondingly sets of terminal and divergent states.*

**Definition 2** *A binary relation $R \subseteq S \times S$ is called a partial bisimulation if for all $s$ and $t$ such that $sRt$ and for all $a \in A$*

- $s \in S_\Delta \Rightarrow t \in S_\Delta$
- $s \xrightarrow{a} s' \Rightarrow \exists t'.t \xrightarrow{a} t' \wedge s'Rt'$
- $s \notin S_\perp \Rightarrow (t \notin S_\perp \wedge t \xrightarrow{a} t' \Rightarrow \exists s'.s \xrightarrow{a} s' \wedge s'Rs)$

[3]The behaviour algebra plays the same role in the theory of interactive programs as the Kleene algebra does in the theory of automata. In fact the only difference from the Kleene algebra is the absence of right distributivity (if nondeterministic choice and sequential composition are considered as the only operations of the algebra of behaviours) and the Kleene algebra may be obtained as the homomorphic image of the corresponding algebra of behaviours.

A state $s$ of a transition system $S$ is called a *bisimilar approximation* of $s'$ denoted as $s \sqsubseteq_B s'$ if there exists a partial bisimulation $R$ such that $sRs'$. Symmetric closure of partial bisimulation is a *bisimulation* equivalence denoted $s \sim_B s'$.

To each state $s$ of a transition system there is a corresponding behaviour $u_s$ which is a component of a minimal solution of a system of equations

$$u_s = \sum_{s \xrightarrow{a} s'} a u_{s'} + \varepsilon_s$$

**Theorem 5** *$s \sqsubseteq_B s' \Leftrightarrow u_s \sqsubseteq u_{s'}$ and $s \sim_B s' \Leftrightarrow u_s = u_{s'}$*

These are standard domain theoretic constructions. Detailed proofs of similar statements based on Plotkin power domains can be found in [1].

The *transition closure* $\mathtt{Tr}(u)$ of behaviour $u$ is the minimal set of behaviours which includes $u$ and for any $v \in \mathtt{Tr}(u)$ if $v = aw$ for some action $a$ then $w \in \mathtt{Tr}(u)$. If $\mathtt{Tr}(u) = \{u_i | i \in I\}$ then $u$ may be represented as a component of the minimal solution of a system of equations in an algebra of behaviours (equational representation):

$$u_i = \sum_{(i,j,k) \in M_i} a_{ijk} u_j + \varepsilon_i, \quad i \in I, M_i \subseteq I^2 \times K \tag{2}$$

The notion of a transition closure can be naturally extended to sets of behaviours. The set $U$ is called *transition closed* if it coincides with its transition closure. A transition closed set $U$ can be considered as a set of states of a transition system with transitions defined by the following rule:

$$v \xrightarrow{a} v' \Leftrightarrow v = av' + v''$$

The state $u$ is terminal if $E(u) = \Delta + \varepsilon$ and divergent if $E = \perp + \varepsilon$. In all such representations we assume that $a$ is a deterministic action.

## 3.3 Examples

Let us consider some special cases of behaviours which cover the majority of classical examples and are useful for applications.

### 3.3.1 Finite (rational) behaviours

We obtain finite state transition systems by taking the sets $I$ and $M_i$ in the equational representation (2) as finite. This corresponds to the "linear case", the behaviours $u_i$ constituting the minimal solution of a system of linear equations. Bisimilarity is algorithmically recognisable and many theoretical and practical problems such as proving and recognising properties, model checking and so on may be solved completely [4].

### 3.3.2 Algebraic behaviours

Equational representation:

$$u = F_i(u_1, \dots, u_n), \quad i = 1, \dots, n$$

Here $F_i(u_1, \dots, u_n)$ are expressions of some behaviour algebra, which use not only prefixing and nondeterministic choice but also parallel and sequential compositions. This is the simplest way to introduce constructive transition systems with infinite sets of states and this corresponds to the "nonlinear case". The continuity of parallel and sequential compositions provides the minimal solution. An interesting special case occurs when parallel composition is not used, and corresponds to "context free behaviours" [28].

### 3.3.3 Parameterised algebraic behaviours

Equational representation:

$$u_i(x_1, ..., x_m) = F_i(v_1, \ldots, v_k), \quad i = 1, \ldots, n$$

$$v_j = u_{i_j}(f_1(x_1, \ldots, x_m), \ldots, f_m(x_1, \ldots, x_m)), \quad i_j \in [1:n]$$

Here $x_1, ..., x_m$ are variables (formal parameters), $f_j(x_1, \ldots, x_m)$ are expressions of some algebra where the variables are assigned values (data algebra). $F_i(v_1, \ldots, v_k)$ are again behaviour algebra expressions. Each equation is in fact a set of equations indexed by value tuples from data algebra (compare with value passing in CCS). This is another more powerful way to introduce the infinite state behaviours.

### 3.3.4 Behaviours over state spaces

Action $a \in A$ is interpreted as a partial transformation $f_a \subseteq S \to S$ of a state space. It may be, for instance, conventional memory states or stores in concurrent constraint programming [27] (conjunctions of primitive constraints). The equality of transformations performed by actions must be a congruence w.r.t. combination: $f_a = f_b \Rightarrow f_{a \times c} = f_{b \times c}$. Now a behaviour over a state space $S$ may be defined in the equational form in the following way:

$$u_i(s) = \sum_{(i,j,k) \in M_i, s \in \texttt{Dom}(a_{ijk})} a_{ijk} u_j(s a_{ijk}) + \varepsilon_i, \quad i = 1, \ldots, n$$

Here $\texttt{Dom}(a)$ is the domain of $f_a$, $sa = f_a(s)$. Usually if the behaviour of a program and information environment is considered, it is split into the behaviour of a program

$$u_i = \sum_{(i,j,k) \in M_i} a_{ijk} u_j + \varepsilon_i, \quad i = 1, \ldots, n$$

and the behaviour of a whole system which is defined by the following rule:

$$\frac{u_i \xrightarrow{a} u_j, \quad s \in \texttt{Dom}(a)}{u_i(s) \xrightarrow{a} u_j(sa)}$$

## 3.4 Syntactic algebras

The main syntactic compositions of programs define the algebra of syntactic expressions which is called the syntactic algebra of a language. In the case of the Action Language there are three main compositions (nondeterministic choice, parallel and sequential composition). Actions and procedure calls are the generators of the syntactic algebra. We may construct the syntactic algebra with approximation and delete procedure calls in the following way.

Extend the syntax of programs by adding the undefined program $\perp$ to the definition of `Prog`. Define the approximation relation $\sqsubseteq$ on the set `Prog` as the minimal partial order satisfying the rules in Figure 5.

$$\perp \sqsubseteq P$$

$$P \sqsubseteq Q \Rightarrow P\|R \sqsubseteq Q\|R$$
$$P \sqsubseteq Q \Rightarrow P+R \sqsubseteq Q+R$$
$$P \sqsubseteq Q \Rightarrow PR \sqsubseteq QR$$

Figure 5: Approximation for programs

Let `Fprog` be the set of all programs without procedure calls. For each $p \in$ `ProcCall` and each integer $n = 0, 1, \ldots$ define the $n$ step unfolding $p^{(n)}$ of $p$ and the substitution $\sigma_n :$ `ProcCall` $\rightarrow$ `Fprog` by the definition in Figure 6.

$$p^{(0)} = \perp$$

$$p^{(n+1)} = (\mathtt{unfold}(p))\sigma_n$$
$$\sigma_n(p) = p^{(n)}$$

Figure 6: Unfolding procedure calls

For each program $P$ define its complete unfolding `Unfold`$(P)$ as the least upper bound of the set $\{P\sigma_n\}_{n=0,1,\ldots}$. Completing the algebra `Fprog` by these limits we obtain the continuous algebra `Prog`$^*$. This algebra is a homomorphic image of `Prog` with homomorphism `Unfold` which obviously identifies the procedure calls with the same complete unfoldings. In the rest of this paper we shall identify program with its complete unfolding and consider it as a member of a continuous syntactic algebra.

## 3.5 Intensional semantics

The intensional meaning of a program in AL is its behaviour defined independently of any external environment. If the language is a language over $A$, the meaning of its program is an element of a behaviour algebra $F(A)$. This algebra also will be called an intensional algebra of the language. The formal definitions are presented in Figure 7.

$$[\![P+Q]\!] = [\![P]\!] + [\![Q]\!]$$

$$[\![P\|Q]\!] = [\![P]\!] \| [\![Q]\!]$$
$$[\![P;Q]\!] = ([\![P]\!];[\![Q]\!])$$
$$[\![a]\!] = a\Delta$$

Figure 7: Intensional semantics of Action Language

$P$ and $Q$ in this figure are programs, $a$ is an action. The same operation symbols on the left and right hand sides of equations denote the operations in different algebras. The left hand side operations are the operations of the syntactic algebra, the right hand side are operations of the intensional algebra of the language. The mapping $[\![.]\!]$ is obviously a continuous homomorphism so the intensional semantics is compositional.

The intensional meaning of a program can also be presented as a labelled transition system defined up to the bisimilarity relation where labels are actions. First we define the equivalence relation on a set of programs as an equivalence generated by the identities of the algebra of behaviours (Figure 2), associativity of sequential composition, associativity and commutativity of parallel composition and relations among termination constants and other compositions (Figure 4). Now the reduction relation $\rightarrow$ is defined on a set of programs and the transition system is defined by only one inference rule in Figure 8. The relation $\xrightarrow{*}$ is the transitive closure of the reduction relation defined on programs and $\xrightarrow{a}$ denotes a labelled transition on programs.

$$p \rightarrow \texttt{unfold}(p)$$

$$p \text{ is a procedure call}$$

$$((P+Q);R) \rightarrow (P;R)+(Q;R)$$

$$((P+Q)\|R) \rightarrow P\|R+Q\|R$$

$$(a;P)\|(b;Q) \rightarrow ((a\times b);(P\|Q))+(a;(P\|(b;Q)))+(b;((a;P)\|Q))$$

$$P \xrightarrow{*} (a;Q)+R \Rightarrow P \xrightarrow{a} Q$$

$$a,b \text{ are actions}$$

Figure 8: Reductions and labelled transitions of programs

Note that our definition of parallel composition is weaker than that usually defined in the process algebra using the so called left merge operator [5]. The system corresponding to this more strong definition is presented in Figure 9. Nondeterministic choice and parallel composition are considered here as a commutative operations.

## 3.6 Interaction semantics

The interaction or extensional semantics of AL over an action algebra $A$ is defined for a given environment $\langle E, A, C, \texttt{res}\rangle$. Here $E$ is a transition closed subset of behaviour algebra $F(C)$ over an algebra of action $C$ called the behaviour algebra of an environment, and

$$\texttt{res} : C \times A \rightarrow 2^C$$

is called a *residual function.* The set $E$ is also called a set of behaviour states of an environment, and its symbol sometimes is used as a symbol of the environment instead of a four-tuple.

$$p \to \mathtt{unfold}(p)$$

$$\frac{P \overset{*}{\to} Q,\ \ Q \overset{a}{\to} R}{P \overset{a}{\to} R}$$

$$a \overset{a}{\to} \Delta$$

$$\frac{P \overset{a}{\to} Q,\quad S \neq \perp, 0}{P + R \overset{a}{\to} Q, PR \overset{a}{\to} QR, P \| S \overset{a}{\to} Q \| S}$$

$$\frac{P \overset{a}{\to} Q, P' \overset{a'}{\to} Q', a \times a' \neq 0}{P \| P' \overset{a \times a'}{\longrightarrow} Q \| Q'}$$

Figure 9: Transition system representing strong intensional semantics of Action language

The interaction meaning $[\![P]\!]_E$ of a program $P$ is the continuous transformation of $F(C)$ restricted to the set $E$. This transformation is defined by means of a residual function `res`.

**Definition 3** *If* $\mathtt{res}(c, a) \neq \emptyset$ *then action $a$ is said to be* conformant *with the environment action $c$, and any action $d \in \mathtt{res}(c, a)$ is called a* residual *of $c$ generated by $a$.*

**Definition 4** *An action of the environment is said to be* complete *if it has no conformant actions, otherwise it is* incomplete.

Instead of considering a function, one may consider a ternary relation $\mathtt{res} \subseteq C \times A \times C$. This relation defines a label transitions system on the set $C$ with transitions

$$d \in \mathtt{res}(c, a) \Leftrightarrow c \overset{a}{\to} d$$

The function `res` induces the equivalence relation on $A$:

$$a \sim b \Leftrightarrow \forall c \in A \quad \mathtt{res}(c, a) = \mathtt{res}(c, b)$$

The important restrictions on `res` are the following:

1. The relation $a \sim b$ is a congruence w.r.t. combination, that is for all $c \in A$ $a \sim b \Rightarrow a \times c \sim b \times c$;

2. $\mathtt{res}(c, 0) = \mathtt{res}(0, a) = \emptyset$;

3. $a \neq 0 \Rightarrow \exists c \in C(\mathtt{res}(c, a) \neq \emptyset \wedge \mathtt{res}(c, a) \neq \{c\}$.

An environment and an action algebra $A$ are said to be *compatible* if they satisfy the restrictions above.

The domain for the interaction semantics is the algebra $T_{\mathtt{res}}(A, C, E)$ of continuous behaviour transformations of the type $\varphi : E \to F(C)$. This algebra has the same type as an intensional algebra, that is nondeterministic choice and prefixing by actions from $A$ are defined for behaviour transformations, but this algebra may posses more relations among

behaviours. It is also an algebra with approximation built up in the following way. First generate a finite element algebra using the following as generators (basic transformations):
(i) the identity transformation $\mathtt{I}$, such that $\mathtt{I}(u) = u$,
(ii) the zero transformation $\varphi_0$, such that $\varphi_0(u) = 0$,
(iii) the bottom transformation $\varphi_\perp$, such that $\varphi_\perp(u) = \perp$ for all $u \in E$.
Then complete this algebra by all necessary limits. Nondeterministic choice and prefixing in this algebra are defined in Figure 10. Action $a$ in this definition is supposed deterministic, the definition of prefixing is recursive and must be understood as the minimal fixed point.

$$(\varphi + \psi)(u) = \varphi(u) + \psi(u),$$

$$(a\varphi)(u + v) = (a\varphi)(u) + (a\varphi)(v)$$

$$(a\varphi)(cu) = \begin{cases} \sum\limits_{c' \in \mathtt{res}(c,a)} c'\varphi(u) & \text{if } \mathtt{res}(c,a) \neq \emptyset \\ c(a\varphi)(u) & \text{otherwise} \end{cases}$$

$$a \neq 0$$

$$(a\varphi)(\Delta) = \Delta$$

$$(a\varphi)(0) = 0$$

$$(a\varphi)(\perp) = \perp$$

Figure 10: The definition of nondeterministic choice and prefixing in the algebra of behaviour transformations

The (informal) meaning of this definition is the following. The interaction between a program and an environment at the current moment of time consists of choosing a conformant pair of actions $a \in A$ and $c \in C$ from all possible ways defined by the nondeterminism of their current states. From the point of view of game semantics [2] a program and an environment are partners in a game and this choice is a choice of moves. We do not fix the order of moves "splitting the atom of interaction" and leave it for applications. Both cases are possible.

If an environment moves first then the choice of an action $c$ defines the set of actions of a program which are conformant with the action of an environment and which a program may chose as an answer. The residual action $c'$ of an environment is selected as a result of an interaction of a program and an environment. If this residual is complete it means that no other programs may interact with the new environment and the given program at this moment of time. Moreover, if some of these programs are ready to interact, this interaction will be postponed and may only be performed in the future. Otherwise a program may interact at the current moment with other programs according to the rules of the game defined by the conformance relation.

This is of course only one interpretation of the interaction semantics introduced here. Other interpretations may include many programs and many environments to describe more complex multiparty hierarchical interaction.

Now the meaning $[\![P]\!]_E$ of a program $P$ in the environment $E$ is defined by the equation:

$$[\![P]\!]_E = \mathtt{trans}[\![P]\!]$$

The function `trans` is the continuous homomorphism of the intensional semantic algebra $F(A)$ to the algebra of behaviour transformations defined by the equations: $\mathtt{trans}(\Delta) = \mathtt{I}, \mathtt{trans}(0) = \varphi_0, \mathtt{trans}(\perp) = \varphi_\perp$. This provides the compositionality of the extensional semantics w.r.t prefixing and nondeterministic choice.

## 3.7 Compositionality of parallel and sequential compositions

To prove compositionality, parallel and sequential composition must be defined in the algebra of behavior transformations so that $\mathtt{trans}(\varphi \| \psi) = \mathtt{trans}(\varphi) \| \mathtt{trans}(\psi)$, $\mathtt{trans}(\varphi\psi) = (\mathtt{trans}(\varphi))(\mathtt{trans}(\psi))$.

Both compositions are defined by the same equations as for behaviours (i.e. Figure 4 with $\Delta$ changed to $I$, 0 changed to $\varphi_0$, $\perp$ to $\varphi_\perp$ and behaviour transformations considered instead of behaviours).

Now the problem is to prove the uniqueness of this definition. For this purpose we introduce the normal form for finite behaviour transformations. Each finite behaviour transformation can be presented in the form $\sum_{i \in I} a_i \varphi_i + \varepsilon$ where $a_i \neq 0$ and $\varepsilon = I, \varphi_0, \varphi_\perp$ or $I + \varphi_\perp$. Then we can apply the relation $a\varphi + b\varphi = (a+b)\varphi$ and all $\varphi_i$ will be different. Such a representation is called the normal form of a transformation. The main theorem is on the uniqueness of this normal form.

**Theorem 6** *If the set of states of an environment which is compatible with an algebra $A$ is a subalgebra of the environment algebra $F(C)$ then the normal form of behavior transformations is unique up to the equivalence of prefixes and the order of summands.*

To prove the theorem first we prove that for $a \neq 0$, $a\varphi = b\psi \Leftrightarrow a \sim b$ and $\varphi = \psi$. Denote $\mathtt{Res}(c, a) = \sum_{c' \in \mathtt{res}(c,a)} c'$ and let $\mathtt{res}(c, a) \neq \emptyset$ (such $c$ exists because of compatibility). Then $(a\varphi)(cu) = \mathtt{Res}(c, a)\varphi(u) = \mathtt{Res}(c, b)\psi(u)$, (note that $\mathtt{res}(c, b) \neq \emptyset$ because the equality must be true for $u = \Delta$). The arbitrariness of $u \in E$ implies the required result.

Similar reasoning can be applied to the sum of guarded transformations and for the transformations of the type $\varphi + \varepsilon$, which then completes the proof.

The compositionality of parallel and sequential composition is proved using this theorem together with the congruence property of $\sim$. We must prove the independence of the definition of parallel and sequential composition from the representation of transformation in normal (now canonical) form. It can be done first for finite behaviors and then extended to their limits.

## 3.8 Computational semantics

In the definition of interaction semantics, arbitrary combinations of choices of actions for program and environment are possible. This reflects the situation in which a given program may interact with arbitrary other programs which were inserted to the environment before the choices under consideration had to be made. But in reality there may be some restrictions or commitments which the choices made by a program and an environment must satisfy. Specifically, if the program has been developed as a computational one, that is for computation of some function or relation then only interaction with the environment, not with other programs must be considered for the definition of its computational meaning.

The aim of this section is to define the computational meaning of a program in an abstract and possibly general form. For this purpose the notion of completeness of environment action will be used. Namely, if the action of a program forces the transition $u \xrightarrow{d} v$ and $d$ is complete then a transition $u \to v$ can be considered as a transition which is the result of the interaction of an action and an environment only, otherwise some other actions produced by other programs could participate in generating this transition.

So the computational meaning can be defined as a relation $\mathtt{comp}[\![P]\!]_E \subseteq E \times E$. By definition $(u, v) \in \mathtt{comp}[\![P]\!]_E$ iff there exists a sequence of transitions

$$(P, u) = (P_1, u_1) \xrightarrow{d_1} \dots \xrightarrow{d_{m-1}} (P_m, u_m) = (\Delta, u)$$

such that all $d_i,\ \ i = 1, \dots, m-1$ are complete. Computational semantics can be expressed also in the form of a transition system. This presentation is given on Figure 11.

$$\frac{P \xrightarrow{a} Q, u \xrightarrow{c} v, c \xrightarrow{a} d,\ \ \mathtt{complete}(d), d \neq 0}{(P, u) \xrightarrow{d} (Q, v)}$$

$$\frac{P \xrightarrow{a} Q, u \xrightarrow{c} v,\ \ \mathtt{complete}(c)}{(P, u) \xrightarrow{c} (aQ, v)}$$

Figure 11: Transition representation of a computation semantics

An important property of a computation semantics is the following.

**Theorem 7** *Computation semantics is compositional w.r.t. sequential composition.*

The computation meaning of a sequential composition of programs is the sequential composition of their meanings considered as relations over environment behaviours.

The operational representation of computation semantics (Figure 11) also allows us to distinguish between different forms of termination, which is important for studying the computational properties of a program.

If the state of a computation system is $(Q, u)$ then if $P = \Delta$ this is a successful termination of a computation process. If $Q \neq \Delta$ and there are no moves from $(Q, u)$ the termination is unsuccessful. We can also distinguish between a case when there exists such an action $c$ of an environment that $u \xrightarrow{c} v$ (deadlock) or there is no such an action (fail).

## 4 Environments

If $\varphi$ is the transformation defined by a given program over an environment $E$, then all behaviours $\varphi(u),\ u \in E$ may be represented as states of a transition system defined on a set of pairs $(P, u)$ where $P$ is a state of a program. This is illustrated by the rules in Figure 12 where the first rule describes a move of a program in a state $P$ and the second rule describes the situation when a program is suspended (with the already selected action $a$). We also identify states of a type $(\Delta, u)$ with $u$, so a system continues its performance after a program has finished. The terminal state of a system is therefore $\Delta$ (not $(\Delta, u)$) and there are three types of divergent states: $(\bot, u)$, $(P, \bot)$, and $\bot$.

$$\frac{P \xrightarrow{a} Q, u \xrightarrow{c} v, c \xrightarrow{a} d, \quad d \neq 0}{(P, u) \xrightarrow{d} (Q, v)}$$

$$\frac{P \xrightarrow{a} Q, u \xrightarrow{c} v, \mathtt{res}(c, a) = \emptyset}{(P, u) \xrightarrow{c} (aQ, v)}$$

Figure 12: Interaction semantics of AL, transition representation

Let us consider some useful examples of environments. Each example consider an environment $\langle E, A, C, \mathtt{res} \rangle$ and define this environment by the properties of its components. Sometimes we refer to an environment instead of its behaviour having in mind an environment in a given initial or intermediate state which defines this behaviour.

**Example 1**. Let $A \cup \{e\} \subseteq C$ and residual function satisfy the equations $\mathtt{res}(e, a) = \{a\}$, $\mathtt{res}(a, b) = \emptyset$, $a, b \in A$. Let

$$u = eu$$

be a behaviour of the environment ($u \in E$), $P$ be a program and $\varphi_P$ the interaction meaning of this program. Then

$$\varphi_P(u) = (P; u)$$

and if $\varphi_Q$ is the interaction meaning of another program $Q$ then

$$\varphi_Q(\varphi_P(u)) = (P; Q; u)$$

**Example 2**. Generalisation to multi-threaded computing. For an arbitrary action algebra let us define $a^n = a \ldots a$ and $[a]^n = a \times \ldots \times a$, $n$ times; note that $a^0 = [a]^0 = \delta$. Assume the following properties of the residual function (for this example):

$$\mathtt{res}([e]^n, a) = \{[e]^{n-1} \times a\}, \quad n > 0$$

$$\mathtt{res}([e]^n \times a, b) = \{[e]^{n-1} \times a \times b\}, \quad n > 0$$

Now if the state of an environment is

$$u = \sum_{n=0}^{\infty} [e]^n u$$

then several programs are permitted to be inserted to it and interact using combinations of actions. The environment

$$u = \sum_{n=0}^{\infty} [e]^n u + \sum_{a \in A} au$$

not only permits several programs to be inserted, but can itself interrupt the execution of a program and perform its own action, changing the intensional meaning of a program.

## 5 Action languages over stores

Usually (especially for computational programs) some actions of a program are interpreted as functions or relations over some state space, and the algebra of actions of a program generated by these actions is the algebra of relations. The states can be memory states if there are names and values, or some abstract structures describing all the possible values which names (or variables) may take in a given state as in the constraint programming paradigm. We shall use the general term *store* to denote the state space and sometimes say "store" instead of "state of a store". If the actions of a language are interpreted on some store the language is called a *language over stores*.

If a program $P$ comprises the parallel composition of some other programs, then the store is a common store for these programs and therefore it must be considered as a part of an environment into which $P$ will be inserted before execution.

After inserting a program into this environment, a new environment will be obtained whose behaviour can be described by a transition system with states $(P, s, u)$. In this triple, $P$ represents a state of a program, $s$ is a state of a store and $u$ is a state of the control part of an environment. Of course the environment must be compatible with the algebra of actions of a program. As before we suppose that $(\Delta, s, u) = (s, u)$. The transition system which produces this behaviour can be defined by the rules in Figure 13

$$\frac{P \xrightarrow{a} Q,\ \ (s,u) \xrightarrow{c} (t,v),\ \ c \xrightarrow{a} d\ \ d \neq 0}{(P,s,u) \xrightarrow{d} (Q,t,v)}$$

$$\frac{P \xrightarrow{a} Q,\ \ (s,u) \xrightarrow{c} (t,v),\ \ \mathtt{res}(c,a) = \emptyset}{(P,s,u) \xrightarrow{c} (aQ,t,v)}$$

Figure 13: Transitions for programs over store

### 5.1 Synchronous and asynchronous communication

The above presentation of interaction semantics of programs over stores is very general and hence cannot serve to provide a good understanding of what happens in reality when the program interacts with its environment. Especially it does not describe synchronous and asynchronous communication as well as the computational connections between the actions of a program and an environment. So let us consider a more specific case in which a store is used as a common memory for exchanging the information between a program and an environment. Let the action space of the environment includes the following three types of actions:

1. $(\mathtt{prog} : a),\ a \in A$, computational program actions;
2. $(\mathtt{env} : a),\ a \in A$, computational environment actions;
3. $(\mathtt{inter} : a, b),\ a, b \in A$, interactions.

Let actions of $A$ be interpreted as relations on a store and computational actions as relations on the environment states. The desired properties of a residual function and relations between the actions of an environment and a program are presented in Figure 14.

$$\begin{array}{l}
\texttt{res}((\texttt{prog}:a),a)=\{a\} \text{ for all } a\in A\\
\texttt{res}((\texttt{prog}:a),b)=\emptyset, \text{ if } b\neq a\\
\texttt{res}((\texttt{env}:a),b)=\emptyset \text{ for all } b\in A\\
\texttt{res}((\texttt{inter}:a,b\times c),b)=\{(\texttt{inter}:a\times b,c)\}\\
\texttt{res}((\texttt{inter}:a,d),b)=\emptyset \text{ if there is no } x \text{ such that } d=b\times x
\end{array}$$

$$\begin{array}{l}
(s,u)\overset{(\texttt{prog}:a)}{\rightarrow}(t,v)\Rightarrow s\overset{a}{\rightarrow}t\\
(s,u)\overset{(\texttt{env}:a)}{\rightarrow}(t,v)\Rightarrow s\overset{a}{\rightarrow}t\\
(s,u)\overset{(\texttt{inter}:a,b)}{\rightarrow}(t,v)\Rightarrow s\overset{a\times b}{\rightarrow}t
\end{array}$$

Figure 14: Residual and transition functions of the environment with common store

The special cases when computational actions “add something to a store” or “remove something” can be considered as asynchronous communication. Interactions are obviously synchronous ones.

## 5.2 Local store

In real programs only a part of a store may be used as a shared item, other parts are localised in a program and the external part of an environment can not access them. To express this partitioning on the semantic level the notion of variables or names must be introduced on the language level and then used in the definition of the extensional (interactive) semantics of programs. Action expressions of a language and procedure calls are called *primitive programs.* Actions of an environment are represented by means of syntactic expressions of an extended AL and also are considered as primitive programs.

Let $V$ be a set of variables, and assume that for each primitive program $q$, a set of variables $\texttt{Var}(q)\subseteq V$ on which it depends on is given. If primitive programs may contain operators with bound variables such as for instance lambda abstraction, only free occurrences must be considered, and they also contain all free variables which can appear under the unfolding of this procedure call. We also need the renaming substitutions $\sigma=[X/Y]$ where $X$ and $Y$ are ordered sets of the same numbers of variables (if the order is not given it must be chosen in an arbitrary way). These substitutions are defined on primitive programs and extended to arbitrary ones by renaming of all occurrences of primitive programs.

The notion of program is extended by adding the notion of *local program components* with the syntactical form $\texttt{Loc}(X,P)$ where $X$ is a finite set of variables and P is a program. Local program components are considered up to renaming. It means that if $\sigma=[Y/X]$ is a substitution which changes symbols from $X$ to symbols from $Y$ different from all symbols which occur free in $P$ $(Y\cap V(P)=\emptyset)$ then $\texttt{Loc}(X,P)$ is equivalent to $\texttt{Loc}(Y,P\sigma)$. This assumption extends the equivalence of programs. We also define the reduction relation for local program components by Figure 15 which extends the reductions in Figure 8.

We define an interaction semantics for programs with a local store and the control part

$$\texttt{Loc}(X,P)+Q \to \texttt{Loc}(Y,P\sigma+Q)$$

$$\texttt{Loc}(X,P)\|Q \to \texttt{Loc}(Y,P\sigma\|Q)$$

$$(\texttt{Loc}(X,P);Q) \to \texttt{Loc}(Y,(P\sigma;Q))$$

$$\texttt{Loc}(X,\texttt{Loc}(Y,P)) \to \texttt{Loc}(X\cup Z,P\sigma)$$

$$P\to Q \Rightarrow \texttt{Loc}(X,P) \to \texttt{Loc}(X,Q)$$

Figure 15: Reductions of local program components

of an environment given by the equation:

$$u = eu + \sum_{a\in A} au$$

For this case the state of the control part of an environment must not be considered in the definition of the interactive semantics of AL and the state of a transition system for interactive semantics is *a local store component* $\texttt{Loc}(X,P,s)$. We assume that the set of free variables and renaming are also defined for store states. Local store components are considered up to the equivalence relation defined in Figure 16. The renaming $\sigma$ renames all variables $y\in Y$ to those which are different from $X$.

$$\texttt{Loc}(X,\texttt{Loc}(Y,P),s) = \texttt{Loc}(X\cup Z,P,(s\sigma))$$

$$(P,s) = \texttt{Loc}(\emptyset,P,s)$$

Figure 16: Equivalence of local store components

The transition system for the interactive semantics is presented in Figure 17. Renaming $\sigma$ protects the local variables of a program when the transition is defined by an environment. Hiding operator $h$ also hides the local information of a program action from other programs which can be inserted to the transformed environment as well as from an environment itself.

$$\frac{P\xrightarrow{a}Q,\quad s\xrightarrow{a}t}{\texttt{Loc}(X,P,s)\overset{h(X,a,s,t)}{\to}\texttt{Loc}(X,Q,t)}$$

$$\frac{s\sigma\xrightarrow{a}t}{\texttt{Loc}(X,P,s)\xrightarrow{a}\texttt{Loc}(Y,P\sigma,t)}$$

Figure 17: Transition system for programs with local store

### 5.3 Channels

Communication between a program and an environment in AL with a local store (as well as between programs inserted to an environment) can be realized via common (global) memory or as a synchronous interaction (rendez-vous, hand shaking etc.). Channels may be represented by variables in the common memory which have values assigned to them or changed by actions. The use of these actions can be restricted by an environment according to the information which the program gives to its environment or introduced by special compositions equivalent to the declaration of properties of variables.

## 6 Distributed action languages

Distributed computation is a very important area of modern computer science and its applications. The main characteristic of this area is the use of local sites for distributing memory and programs. These local sites may be separate processors in the network or the components of a multiprocessor system as well as persistent software components (software agents) which perform concurrent computation sharing the time of a central processor (multi-threading) or other resources.

The AL with local store could be used for distributed programming but its interactive semantics defined in the previous section has some disadvantages which limit this use. The main disadvantage of the semantics for programs with a local store is that it loses the structure of a program state defined by nesting program components and partitioning a global store on local spaces for parallel composition. In reality this information could be used for the organisation of a distributed implementation of a given program. Moreover, if the programmer is aware of the strategy for the distributed implementation, he could use this information for the development of efficient distributed programs using the localisation operator as a tool for expressing his algorithmic ideas for the distribution of data and actions.

Another use of a local component structure could be a more adequate description of components of real systems as programs which simulate their activity or create communities of software components.

To eliminate the disadvantage mentioned above, a new distributed interaction semantics is introduced for the AL with local store. We call this new language the distributed action language (DAL). The main semantic notion of the DAL is the notion of a distributed action component (dac) which is used to describe the state of computation with distributed programs and stores (local stores). The definition is the following:

1. A program is a (simple) dac;
2. Parallel composition of dacs $C \| D$ is a (parallel) dac;
3. If $C$ is a dac and $s$ is a store then $\mathtt{Loc}(X, C, s)$ is a (local) dac.

The equivalence of dacs includes the equivalence of programs, renaming, and extra equivalences introduced by Figure 18.

Instead of reductions of local components in Figure 15 we introduce only one reduction rule on action components (Figure 19) where $\sigma$ is a protective renaming.

Transitions of dacs which define the distributed interaction semantics are introduced in Figure 20.

$$\texttt{Loc}(X, \texttt{Loc}(Y, \Delta, s) \| C, t) = \texttt{Loc}(X \cup Z, C, (s\sigma) \times t)$$

$$\texttt{Loc}(X, \texttt{Loc}(Y, C, s), t) = \texttt{Loc}(X \cup Z, C\sigma, (s\sigma) \times t)$$

$$\texttt{Loc}(X, 0, s) = 0$$

Figure 18: Equivalence of components

$$\texttt{Loc}(X, P)Q \to \texttt{Loc}(Y, (P\sigma)Q, \perp)$$

Figure 19: Reductions for components

The transition rules for dacs cover some of the transition rules of programs as a special case of a dacs. Therefore these rules can be reduced to those presented in Figure 21.

## 7 Conclusions

We have presented a general theory of action-based languages as a paradigm for the description of those computational systems which include elements of concurrency and networking, and extended this approach to describe distributed systems and also to describe the interaction of a system with an environment. As part of this approach we have introduced the Action Language as a common model for the class of nondeterministic concurrent programming languages and defined its intensional and interaction semantics in terms of continuous transformation of environment behaviour. This semantics has been specialised for programs with stores, and extended to describe distributed computations.

In the future we intend to specialise our theory in order to obtain a working semantics for reasoning about and designing programs in different paradigms, including concurrent constraint languages. We have started on this work, and the ideas presented in this paper are being used as the basis for the design of a workbench for action languages.

We plan to study the semantics of distributed languages. Our present approach to semantics does not describe the structure of distributed programs, and will investigate the possibility of preserving this structure using equivalent transformations of distributed components.

## Acknowledgments

This research was carried out as part of a joint project between City University and the Glushkov Institute of Cybernetics, funded by the Royal Society, UK.

## References

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

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

$$\frac{C \xrightarrow{a} C'}{C \| C'' \xrightarrow{a} C' \| C''}$$

$$\frac{C \xrightarrow{a} C', \ \ D \xrightarrow{b} D', \ \ a \times b \neq 0}{C \| D \xrightarrow{a \times b} C' \| D'}$$

$$\frac{C \xrightarrow{a} C', \ \ s \xrightarrow{a} t}{\mathtt{Loc}(X, C, s) \xrightarrow{h(X,a,s,t)} \mathtt{Loc}(X, C', t)}$$

$$\frac{s\sigma \xrightarrow{a} t}{\mathtt{Loc}(X, C, s) \xrightarrow{a} \mathtt{Loc}(Y, C\sigma, t)}$$

Figure 20: Transitions of distributed action components

$$a \xrightarrow{a} \Delta$$

$$G \xrightarrow{a} H \Rightarrow G + F \xrightarrow{a} H$$

$$G \xrightarrow{a} H \Rightarrow GF \xrightarrow{a} HF$$

Figure 21: Transitions of programs

[2] S. Abramsky. Semantics of interaction. In *Trees in Algebra and Programming – CAAP'96, Proc. 21st Int. Coll., Linköping*, volume 1059 of *Lecture Notes in Computer Science*, page 1. Springer-Verlag, 1996.

[3] S. Abramsky and A. Jung. Domain Theory. In *Handbook of Logic in Computer Science*, volume 3, pages 2–168. Clarendon Press, 1994.

[4] A. Arnold. *Finite Transitions Systems.* Masson, Prentice Hall, 1994.

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

[6] J. Bradshaw. *Software Agents.* AAAI / MIT Press, 1997. ISBN 0-262-52234-9.

[7] L. Brim, J-M. Jacquet, D. R. Gilbert, and M. Křetínský. A process algebra for synchronous concurrent constraint programming. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Proceedings of ALP96: Fifth International Conference on Algebraic and Logic Programming*, pages 165–178, Sep 1996. ISBN 3-540-61735-3.

[8] E. Brinksma. *On the Design of Extended LOTOS; a Specification Language for Open Distributed Systems.* PhD thesis, Department of Informatics, University of Twente, Enschede, Netherlands, 1988.

[9] F. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP91*, Lecture Notes in Computer Science, pages 296–319. Springer-Verlag, 1991.

[10] V. M. Glushkov. Automata theory and the design of computers. *Kibernetika*, (1), 1965.

[11] V. M. Glushkov and A. A. Letichevsky. Theory of algorithms and descrete processors. In *Advances in Information Sciences*, volume 1, pages 1–58. Plenum Press, 1969.

[12] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, January 1977.

[13] I. Guessarian. *Algebraic semantics.* Lecture Notes in Computer Science v.99. Springer-Verlag, 1981.

[14] C. A. Gunter and D. S. Scott. Semantic Domains. In *Handbook of Theoretical Computer Science*, volume B, pages 633–674. MIT Press, 1994.

[15] M. Hennessy. *Algebraic Theory of Processes.* MIT press, 1988.

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

[17] ISO. *ISO IS 8807 Information Processing Systems, Open Systems Interconnection, LOTOS*, 1989.

[18] A. A. Letichevsky. Algebras with approximation and recursive data structures. *Kibernetika*, (5):32–37, September-October 1987.

[19] A. A. Letichevsky and D. R. Gilbert. Toward an implementation theory of nondeterministic concurrent languages. Technical Report 1996/09. ISSN 1364-4009, Department of Computer Science, City University, 1996. Also presented at the Second workshop of the INTAS-93-1702 project Efficient Symbolic Computing St Petersburg, Russia, October 1996.

[20] G. J. Milne. CIRCAL and the Representation of Communication, Concurrency and Time. *ACM TOPLAS*, 7(2):270–298, April 1985.

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

[22] R. Milner. The polyadic $\pi$-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, oct 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[23] R. Milner. Action calculi, or syntactic action structures. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93*, pages 105–121. Lecture Notes in Computer Science, Vol. 711, Springer, 1993. ISBN 3-540-57182-5.

[24] R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):78–89, 1993. Turing Award Lecture.

[25] G. Plotkin. A structured approach to operational semantics. Technical report, Tech.Rep. DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.

[26] F. Rossi and U. Montanari. Concurrent semantics for concurrent constraint programming. In B. Mayoh, E. Tyugu, and J.Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 181–220. Springer-Verlag, 1994.

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

[28] C. Stirling. Decidability of bisimulation equivalence for normed pushdown processes. In *CONCUR96*, pages 217–232. LNCS 1119, 1996.

[29] Peter van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. North-Holland, 1989.

[30] P. Wegner. Interaction as a Basis for Empirical Computer Science. *ACM Computing Surveys*, 27(1):45–48, Mar 1995.

[31] P. Wegner. Why interaction is more powerful than algorithms. *CACM*, 40(5):80–91, 1997.