A simulator for concurrent constraint languages

Tatyana Valkevych (tanya@soi.city.ac.uk)
Department of Computer Science
City University, UK

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

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

Abstract

A simulator for concurrent constraint languages has been generated from a generic tool for specification, modelling, exploring and analysing the behaviour of concurrent systems. The design of the tool has been based on the general Action language theory for the class of nondeterministic concurrent programming languages. It has been developed under the new action-based approach using algebraic foundation and executable specification technique in the algebraic programming system APS. The traditional simulation scheme has been enriched by the probabilistic framework on the base of the probabilistic concurrent constraint programming language pccp. The simulator allows to explore the computation of traditional, interactive and probabilistic models of concurrent constraint languages.

1 Introduction

Most computational systems in use today explore the distributed computation scheme. A strong element of concurrency is involved in the computations to permit the expressiveness required to construct such systems. Automated tools based on well-founded theories are required for the support of the specification and reasoning about concurrent systems during their investigation and design development as well as for computation coordination during execution. Recent research work has shown that the constraint paradigm [11] is well suited to the description of concurrent systems, and the aim of this research is to construct such tools for the design and description of systems whose basis is that of concurrent constraints.

The main objective of the research was to design and implement a simulator for a class of concurrent constraint languages, as a first step towards the design of a workbench for these languages. It is intended to explore concurrency and distribution aspects of computations.

The simulator enables the user explore computations, providing a record of the computations performed and performable ones at each step of computation. It permits to navigate the computational history in different directions. The traditional simulation scheme has been enriched with the probabilistic framework on the base of the probabilistic concurrent constraint language pccp [8]. It includes a transition probability, reachability probability and probabilistic matrix of transition probabilities. The simulator allows to explore the computation of traditional models of concurrent constraint languages (with eventual and atomic **tell**), an interactive variant of ccp with a user as a participant of the computation. The probabilistic version of the workbench permits the exploration of probabilistic computations.

This work is based on the general Action language theory for the class of nondeterministic concurrent programming languages (ncpl), which has been proposed as a paradigm for the specification of networked concurrent interactive computational systems [7]. The main advantage of the underlying theory and the Action-based tools is its general semantic-based approach to concurrent computation in open distributed environment, that encompasses parallelism and concurrency, synchronous and asynchronous schemes, interaction between processes and interactive dependence on environment. This new unified approach for the definition of the computational semantics for ncpl is in a step-based computation of the behaviour of a program and generation of computational histories [6]. The approach may be used for different languages from the class of ncpl as well as for the development different tools for investigating program behaviour such as interpretors, simulators, workbenches.

The simulator has been developed using algebraic foundation and executable specification technique in the algebraic programming system APS [3]. This system generates platform independent (and thus portable) persistent code, permitting the interruptible, distributed and mobile simulation and analysis of concurrent programs.

Structure of the paper

The Action language framework are briefly introduced in Section 2. In Section 3 we describe the architecture, formal specification and algebraic-based design and development of the generic Action-based simulator. Generation of concurrent constraint action languages and fragments simulation of a ccp program in the simulator are described in Section 4. The probabilistic extension of the simulator and simulation of pccp programs are described in Section 5. Finally, in Section 6, we summaries our research and mention other applications of the Action approach and designate further work.

2 The Action language as a general semantic basis of concurrent computation

2.1 The Action language

The general abstract *Action language* (AL) [7] represents a common model for action-based languages, and the associated theory is a paradigm for the description of those computational systems which includes elements of concurrency, distribution and networking. It is based on the notions of an *action* and *interaction* and includes as basic operations *nondeterministic choice*, *parallel* and *sequential composition*, indicated respectively by $+$, $\|$ and ;. Any concrete ncpl can be obtained by specializing the generic parameters of the Action language.

The abstract syntax of the AL is as follows, where the syntax of actions `Act` and procedure calls `ProcedureCall` are the parameters defined for each particular language of the class.

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

Each program `Prog` is associated with a set of procedure definitions and also an algorithm which *unfolds* any procedure call to a program. 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. Kinds of termination of a program are expressed by *terminal constants* `TermConst`, like successful termination, suspension and impossible action.

An *action* is a basic notion of this approach to consideration of ncpl. The meaning of actions is defined by some *algebra of actions*, generated by *primitive* actions. The main operation of this algebra is a binary associative and commutative operation (ac-operation) denoted as $\times$ and called the *combination of actions*; the empty action δ is the neutral element for the combination and the impossible action 0 is the zero element.

Actions transform the state of a world and are performed by *agents*. 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 loaded and performed interacting with it). *Behaviours* are agents in a given state considered up to bisimilarity, or possibly to a weaker equivalence. These are defined by the *intensional semantics* of AL. It is represented as an action-labelled transition system in Figure 1; p, P, Q, R in this figure are programs, a and b are actions. Thus the behaviour of any Action program can be represented by means of an action-tree whose nodes represent computational states and whose branches are labelled by actions.

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

$$\frac{P \xrightarrow{*} Q, \quad Q \xrightarrow{a} R}{P \xrightarrow{a} R}$$

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

$$\frac{P \xrightarrow{a} Q}{P + R \xrightarrow{a} Q, PR \xrightarrow{a} QR, P\|R \xrightarrow{a} Q\|R}$$

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{b} Q', a \times b \neq 0}{P\|Q \xrightarrow{a \times b} P'\|Q'}$$

Figure 1: Transition system representing strong intensional semantics of the Action language

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. Definition of these types of interaction are expressed by *residual functions* for a *program* and *environment*, which define the *interaction* or *extensional semantics* of AL – see [7] for more details.

2.2 Action languages over stores

The Action language can be used to describe languages over stores like concurrent constraint languages. This is achieved by interpreting actions of a program as functions or relations over some states space; the algebra of actions of a program is generated by these actions being 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 may take in a given state as in the constraint programming paradigm. We use the general term *store* to denote the state space and sometimes say "store" instead of "state of a store".

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), where 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.

3 The Action-based simulator

The Action-based approach for the definition of the computational semantics for ncpl is in a step-based computation of the behaviour of the program in its environment and generation of computational histories. The approach may be applied for various languages from the class of ncpl, which become corresponding action languages in the Action language framework. Various languages are defined in the general Action language framework, comprising its *intensional* and *extensional semantics*, by means of specification of its generic parameters, that adjust the syntax and the semantics to a corresponding language. These include *action algebra* and *procedures* that define syntax for a concrete *action language* and are the parameters of the *intensional semantic*, and *environment* and *residual functions*, which interpret actions over environment and are the parameters of the *interaction (extensional) semantic*.

The experimental application of the abstract Action model to concrete concurrent models has been accomplished using the Action language workbench. The *Action language workbench* is a generic tool for modelling, exploring and analysing the behaviour of concurrent systems [9]. It enables creation of the Action model for a concrete language, generating a particular *action language* and the simulation of the behaviour of programs in action languages.

3.1 Architecture of the generic simulator

The Action-based simulator consists of three levels: the Action language framework general level, the application level including libraries for action languages and the level of exploration of computation.

The **Action language framework**, that is a prototype of action languages, comprises the definitions of the intensional and interactive semantics.

The **application level** for action languages includes libraries of concrete action languages with appropriate specifications of syntax and both intensional and interactive semantics.

Each application library comprises:

• an *environment* that defines a kind of a program domain (for example, a constraint system as propositional logic) by functions on environment;

• an *action algebra*, defined by terminal symbols of actions for a concrete action language and by the combination of actions for specifying interaction between parallel processes;

• *procedure* definitions that extend a programming language by additional operations for composition, and

• *residual functions* for a *program* and *environment*, which interpret actions over environment by defining an action transition step and a corresponding environment transformation.

Various languages with different semantics are defined by specification of the general Action framework: environment, actions and their combinations, procedures and the residual functions for a program and an environment are defined by a user.

The **exploration level** provides environment for simulation of programs in action languages, namely records about the current and next states of computations and its history, and permits the navigation and exploration of the computations in different directions.

The current state of computation is represented by the:
• current state of a *program* (*agent* or *goal*, up to different programming paradigms),
• current state of the *environment* and
• *behaviour* of the program represented by *labelled* (by numbers) alternatives for *next* possible computations with the following fields: a unique *address* denoting the state's coordinates in the tree of computations, an *action* for transition to the next state, a *residual program* to perform next (next *agent* or *goal*), the next state of the *environment* and the next *behaviour*.

For exploration of the computations the workbench provides the following commands to:
• *display* the computation, namely,
current state of computation and its behaviour as a record described before,
history as a sequence of transformation of a program and environment,
trace as a sequence of actions,
tree of computations explored so far,
• *navigate* the computations, namely,
move one step **back** and one step **next**,
return to the state enumerated in the history of a current agent,
jump to a node of tree,
forward (one path) automated simulation.

3.2 Development of the simulator

The design of the workbench is represented by the executable algebraic specifications written in the algebraic programming language APLAN. The derived APLAN specification is then executed in the algebraic programming system APS [3]. APS is intended for the study of mathematical models of information processing and their application for the development of efficient algorithms in the field of symbolic and algebraic computation [4], [2], [5].

3.2.1 The algebraic programming language APLAN

The algebraic data structures are terms in a signature defined by the environment of a program. The main computational technique is term rewriting defined by systems of rewriting rules (for example at the Figure 4). A system of rewriting rules in general has the form $\mathtt{rs}(x_1, \ldots, x_n)(p_1, \ldots, p_m)$, where $x_1, \ldots, x_n$ are the variables of a system, $p_1, \ldots, p_m$ - rewriting rules. A rewriting system may be applied to a term using different strategies of rewriting. There is a library of strategies in the system, which may be called in functional or procedural form, and the user may develop his/her own strategies transforming various examples and using some basic strategies, two of which are the most important.

The basic strategy is that of recursive rewriting, which uses a system of rewriting rules as a recursive function definition (with matching). In recursive rewriting a term to be rewritten is matched with the left hand sides of a system in the order they occur in a system. Therefore, the next rule may be applied only if neither of the previous are applicable. After substituting the right hand side some additional reductions may be performed, among them standard canonization for built-in operations and call of functions presented by their names. Functions may be presented in rewriting or procedural forms.

For specification of the simulator the following signatures are used:
• standard signature of arithmetical and algebraic operations and functions, predicates, logic operations, procedural operators (implication or conditional operator `->`, set of a name to a value `-->`, assignment a value to a name `:=`, `do`, `while` and `loop` iterative operators), statement separators (`;` and `,`) and special functions for processing algebraic objects like `equ` checking equivalence of two algebraic objects, `arity` giving arity of an operation, `mrg` used for ordering (merging) ac-operations, e.g.: `mrg(x+x+...` *(y times)* `...+x) = x$y`, etc.
• additional signatures for AL operators and definitions of ac-operations.
Particularly, the AL algebraic environment is defined by the following APLAN *operations*: `+` – for nondeterministic choice, `||` – for parallel composition, `><` – for combination of actions; and by the following APLAN *atoms*: `bot` – suspension, `0` – impossible action, `Delta` – success.

3.2.2 Algebraic specification

Computations are modelled by a *tree* of program behaviours. The state of a computation is represented by a *node* `st(prog,env,next)` in the tree. It is marked by `st` and represents the state of a (residual) program `prog` under simulation, the current state of environment `env` transformed by `prog`, and possible transitions `next` from this step of computation. A history `Hist` of the transformation of a program is represented as a path in a tree of behaviour by `(Node,Back)`, where `Node` is a current state of computation and `Back` is a pointer to a previous state of computation.

The top-level specification of the workbench, which is presented in Figure 2,
• assembles the *definitions of an action language* under investigation and setting for *initial states* of *program* and *environment*,
• extends the basic language definition with *procedure* definitions,
• sets the *initial configuration of computations* that includes a program for exploration, initial state of environment, empty behaviour and empty history of computation,
• performs *initial user actions*, and
• iteratively explores *computations* of the program by *interpretation steps* realized by the simulator under guidance of a user.

```
task:=( intr_init();
        extend_rs(prog_unfold,proc_def);
        Hist-->(st(prog,env,Nil),Nil);
        do(user_action);
        loop( step_intr();
              finish->do(Finish);
              finish:=0 )         );
```

Figure 2: Top-level algebraic specification of the Action language workbench

The simulator (Figure 3) creates the current *state* of computation in a `Node` and a *history* `Hist` of computations as a path in the tree of behaviour from the root to the current node `Node` by means of the pointer `Back` to the previous state of computation. For a new (unvisited) node the simulator computes all possible behaviours `Next` by means of the `Intens()` function, which computes the intensional semantics, and by the `Trans()` function, which computes the interaction semantics. Then it *displays* the new state of the computation by `prn_node()` and

```
step_intr:=proc()loc(Node,Prog,Env,Next,Back,p)(
    let(Hist,(Node,Back));
    let(Node,st(Prog,Env,Next));
    equ(Next,Nil)->(
                    p-->Intens(Prog);
                    Next:=Trans(p,Env)          );
    prn_node(Node);
    ((Next==0)|/equ(Next,bot))->Move_back(Back)
                        else Move(Next,Back)   );
```

Figure 3: Algebraic specification of the Action-based simulator

suggests a selection of numbered alternatives of behaviour for the next step of exploration by the `Move()` function, in which a new configuration is formed.

`Intens()` (Figure 4) is a generic function of the simulator that is parameterized by an action algebra defined in the functions `combine()` and `is_act()` by a user. The function `Intens()` transforms an arbitrary program into a canonical form which is the sum of performable programs and termination constants. It employs a parallel (horizontal) strategy of transforming alternative branches of computations with automatic simplification of terminal actions by the function `simpl()`. The functions `Intens_par()` and `Intens_seq()` define transformation of the parallel and sequential composition of behaviours. Computations for `Intens_par()` are structured into synchronization, fulfilled by the `synchr()` function (for parallel interaction), and interleaving acting. To achieve *strong* intensional semantics of concurrent computation left-shifted transformation of the parallel operator is done in interleaving computation of `Intens_par()`. The function `prog_unfold()` refers to procedure and action definitions in order to unfold the procedure calls.

The transition function `Trans()` (Figure 3) uses the transition rewriting system `trans()` (Figure 5) to compute transition steps and transformations of a program and environment by means of `solve_prog()` and `solve_env()` defined by a user. It instantiates a new state of a program by `ins_prog()`.

The *movement* functions `Move()` and `Move_back()` fulfill transition to a new state depending on user selection of transition to one of the alternatives.

3.2.3 Implementation in the APS

The algebraic design of the Action language simulator is implemented in the APS [3], and there are versions in both DOS and UNIX. The APS is a flexible interpreted system with interpreters for algebraic operations, rewriting rules, procedures. The system generates *persistent* and thus *portable* internal algebraic data structure of an APLAN program in *graph representation*. It can be processed by a corresponding interpreter on any machine that has the APS, and a user is able to send images of agents simulated to different sites where one can then continue to explore these behaviours.

The simulator has a hierarchical structure of algebraic programs corresponding to the architectural levels. Each level accompanied by documentation, user manual and demonstrations. Libraries of some action languages have been developed in the system.

```
Intens:=rs(P,Q)(         P+ Q  = simpl mrg(Intens P +  Intens Q),
                         P||Q  = Intens_par (Intens P || Intens Q),
                        (P; Q) = Intens_seq (Intens P; Q),
                         Delta = Delta,
                        act Q  = Q,
                            Q  = Intens prog_unfold new(Q)                    );
Intens_par:=rs(P,Q)( P||Q  = simpl mrg(synchr(P||Q)+ mrg(lmrg(P||Q)+lmrg(Q||P))));
synchr:=rs(P,Q,E,G)(    (Delta||    Q) = Q,
                        (P    ||Delta) = P,
                        (bot  ||    Q) = bot,
                        (P    ||  bot) = bot,
                        (E;P) || (G;Q) = Intens_seq(combine(E><G);(P||Q)),
                        (E  ) || (G;Q) = Intens_seq(combine(E><G);Q),
                        (E;P) || (G  ) = Intens_seq(combine(E><G);P),
                        (E  ) || (G  ) = combine(E><G)                       );
lmrg:=rs(P,Q,R,E)(       (P+R)||Q  = simpl mrg(lmrg(P||Q)+lmrg(R||Q)),
                        (Delta||Q) = Q,
                          (bot||Q) = bot,
                         (E;P)||Q  = (E;P||Q),
                         (E  )||Q  = (E;Q)
                         (P+R)||Q  = simpl mrg(lmrg(P||Q)+lmrg(R||Q)),
                        (Delta||Q) = Q,
                          (bot||Q) = bot,
                         (E;P)||Q  = (E;P||Q),
                         (E  )||Q  = (E;Q)                                   );
Intens_seq:=rs(P,Q,R)( ((P+ Q);R) = simpl mrg(Intens_seq(P;R)+Intens_seq(Q;R)),
                       ((P; Q);R) = Intens(P;Q;R),
                       (Delta; R) = Intens R,
                       (  bot; R) = bot,
                       (    0; R) = 0                                        );
```

Figure 4: Algebraic specification of strong intensional semantics for the Action language

4 Application to concurrent constraint languages

CCP is one of the well structured programming paradigms for ncpl, easily captured by the Action model. Let assume that P denotes processes, D – declarations, A – agents, p – a procedure name; X – a subset of set of variables V; x, t – variables from a corresponding subset of V; $C = \langle S, \vdash, \&, true, false, V, \exists \rangle$ - a cylindric constraint system. Syntax of a general model of ccp are described by the following grammar:

$$P ::= D.A$$
$$D ::= \epsilon \mid D.D \mid p(x) : -A$$
$$A ::= \textbf{Success} \mid \textbf{Fail} \mid \textbf{ask}(c) \rightarrow A \mid \textbf{tell}(c) \mid A + A \mid A \| A \mid \exists X.A \mid p(t)$$

Success and **Fail** represent successful and failing termination. Basic actions are **tell** and **ask** parameterized by a constraint c. The process $\textbf{ask}(c) \rightarrow A$ waits until the store entails c and then behaves like A. The process $\textbf{tell}(c)$ adds c to the store. (If c is inconsistent with the store, the process fails.) $\|$ denotes parallel composition of processes (with interleaving semantic); $+$ – nondeterministic choice between processes; $\exists X.A$ – a process A with local

```
trans:=rs(P,Q,a,env)(
    ( P+Q, env) = simpl mrg(rtrans(P,env)+rtrans(Q,env)),
    ((a;P),env) = simpl mrg(ins_prog(   P, solve_prog(a,env)) +
                          ins_prog((a;P),solve_env (a,env))),
    (Delta,env) = 0,
    ( a,   env) = rtrans((a;Delta),env));

ins_prog:=rs(P,Q,d,env)(
    (P,(d,env)+Q) = (d:st(P,env,Nil))+ins_prog(P,Q),
    (P,(d,env)  ) = (d:st(P,env,Nil)),
    (P,        0) = 0,
    (P,      bot) = bot);
```

Figure 5: Algebraic specification of interactive semantics of action languages

variables X; $p(t)$ – a procedure call, in accordance with procedure declaration as $p(x) : -A$. We consider the operational semantics of ccp which is described in [1].

Ccp action languages, that form *ccp.al* library, encode two traditional models of ccp with and without consistency checking and an interactive variant of ccp including a user as participant of the computation in an environment. A *constraint* c is a propositional variable, and a *store* s is represented by a conjunction of constraints.

The environment for ccp is a constraint system over a store of constraints, where a constraint c is a propositional variable, and a store s is represented by a conjunction of constraints. It is defined in Figure 6 by the following functions:
• *entailment* function `Entail(s,c)` of a constraint c from the store s, which is realized as implication by function `isin()`;
• *consistency* function `consis(c,s)` of the store s with constraint c, which means that conjunction of constraints is not equal to 0 ($false$);
• *increment* function `Store_uni(s,c)` for a store s with a constraint c.

```
Entail:=rs(s,c)(       (s,c) = isin(c,s)         );
Store_uni:=rs(x,y)(    (x,y) = mrg(x&y)          );
consis:=rs(c,s)(       (c,s) = ~(equ(mrg(c&s),0)));
isin:=rs(x,y,a)(
                       (x,x    ) = 1,
                       (x,x  +z) = 1,
                       (x,x$a  ) = 1,
                       (x,x$a+z) = 1,
                       (x,y  +z) = isin(x,z),
                       (x,y    ) = 0);
```

Figure 6: ccp environment: constraint system

The syntax and intensional semantics of the ccp (Figure 7) as an action language is realized by parameters of AL in the simulator and derived from the AL general framework by:
• definition of its action algebra specifying
(1) ccp-actions **tell** and **ask** (other ones are impossible) in the `is_act` rewriting system and
(2) an *asynchronous* scheme of ccp as *zero* combination operation over actions in the `combine`

rewriting system;
• inheriting a parallel operator `||` and a choice operator `+` from the general AL framework;
• defining the *guard* `->` using the sequential operator `;` from the general AL framework;
• defining the *hiding operator* as an extension of the basic Action model by the `Ex` procedure through `Fresh()` function, generating new variables in `proc_def` rewriting system for definition of procedure definition;
• defining other ccp procedures (like `proc1` for example).

```
/*                     ccp.al action algebra                   */
combine:=rs(x,y)(      x >< y = 0  );
is_act:=rs(x)(         tell x = 1,
                        ask x = 1,
                            x = 0  );
/*                     ccp.al procedure composition            */
proc_def:=rs(x,y)( x -> y     =  (x;y),
                   Ex x y     =  Fresh(Ex x y),
                   proc1(x,y) =  (ask(x) -> tell(y));
```

Figure 7: Action algebra and syntax of ccp

The generic interaction semantic (see Figure 5) are parameterized with `solve_prog()` and `solve_env()` functions for definition of interpreting the actions over a store. There are two functions used for `solve_prog()`: `solve_atomic()` which realizes the atomic tell and `solve_eventual()` which realizes the eventual tell which are represented in Figure 8. A user can switch among these using `user_action` name and assigning the appropriate value to it like in the *set* statement `user_action:=(solve_prog-->solve_eventual);` which is performed before simulation (see Figure 2). There is no residual actions over environment and `solve_env() = 0`.

```
/*     ccp.al interaction semantic : two traditional models     */
/*                      1) Atomic tell, consistency check       */
solve_atomic:=rs(s,c)(
    consis(c,s)->((tell(c),s) = (tell(c),Store_uni(s,c))),
                  (tell(c),s) = 0,
    Entail(s,c)->((ask (c),s) = (ask(c),s)),
    consis(c,s)->((ask (c),s) = bot),
                  (ask (c),s) = 0       );
solve_env:=rs(a,s)(     (a,s) = 0       );
/*                      2) Eventual tell, no consistency check   */
solve_eventual:=rs(s,c)(
                   (tell(c),s) = (tell(c),Store_uni(s,c)),
     Entail(s,c)->((ask (c),s) = (ask(c),s)),
                   (ask (c),s) = 0      );
```

Figure 8: Interaction semantic of concurrent constraint action languages

The interactive variant of ccp, including a user as participant of computation in an environment, represents one of the traditional models of ccp [1] and a nontrivial interaction with an environment which is implemented by the `solve_env()` function (see Figure 9). It has the same basic definition. If asked information is not entailed from the store but consistent with the store, the computation is suspended and an environment are controlled by a user, which can add some new information into a store. Otherwise, if asked information is inconsistent with the store, the computations are failed.

```
/*          ccp.al interaction semantic : interaction with a user        */
solve_prog:=rs(s,c,a)(
                        (a, 0) = 0,
    consis(c,s)->( (tell(c),s) = (tell(c),Store_uni(s,c))),
                   (tell(c),s) = 0,
    Entail(s,c)->( (ask (c),s) = (ask(c),s)),
    consis(c,s)->( (ask (c),s) = bot),
                   (ask (c),s) = 0);
solve_env:=rs(a,s,c)(
       Entail(s,c)->( (ask (c),s) = 0),
       consis(c,s)->( (ask (c),s) = AskUser(c,s)),
                      (     a, s) = 0);
AskUser:=proc(c,s)(
    put("\nProgram asks "c);
    put("\nThis constraint is consistent but not entailed by a store");
    put("\nStore is "s);
    put("\nSay ok or add new constraint to a store\n    >");
    answer(Ans);
    equ(Ans,ok)->return 0; ord(Ans);
    Ans-->((Env:tell(Ans)),mrg(s&Ans));
    return(Ans)     );
```

Figure 9: Interactive concurrent constraint action language

In Figure 10 we give an example of a ccp program of the interactive variant and a fragment of a simulation on exploring its behaviour.

5 Probabilistic extension of the simulator

The experimental development [10] of an environment for probabilistic concurrent computation as well as modelling of the probabilistic concurrent language and simulation probabilistic programs has been carried out using the Action-based simulator on the base of the probabilistic concurrent constraint programming language pccp [8]. The extension includes the computation of a *transition probability* from a previous state to a current one, a *reachability probability* from an initial state to a current one and the construction of the *probabilistic matrix* of transition probability distribution as a snapshot the computation history.

In the probabilistic version of the simulator the computational state includes also:

- *a transition probability* for next computational states and
- *a reachability probability* for current and next computational states.

A *matrix of transition probabilities* is generated for computations explored so far with the **matrix** command.

The probabilistic version of the simulator enables the exploration of pccp computations based on the operational semantics of this language. A fragment of the simulation of a pccp program is given in Figure 11.

The Action language model for pccp is described by the algebraic specifications of the functions that are parameters of the probabilistic AL framework. It is created by the basic constructs of the application level of the simulator and forms the *pccp.al* library. There are two versions of probabilistic concurrent constraint computation in the simulator:
(1) one is implemented as an extension to ccp with pccp semantics but ccp syntax; it accepts pure ccp programs and demonstrates the probabilistic interpretation of ccp programs with implicit probability, as well as of ccp programs with explicit input probability in the form of **ask**(c, p); this version processes two models of ccp (with and without consistency check);
(2) another one is the implementation of pccp language [8].

A fragment of the simulation of a pccp program is given in Figure 11. The procedures `A` and `B` are `A = (tell(c))`, `B = ( ((1,0.5) -> tell(d)) + ((1,0.5) -> tell(e)) )`. Initial states for a program and environment are set in `prog` and `env`. The final successful state is designated as `Delta`. The computational state is displayed by the record described before, by the symbolic tree of computations and by the probabilistic matrix. The symbolic tree, the reachability probability and the probabilistic matrix are generated dynamically for each state of computation using the main computational structure.

In the tree the following denotations are used. A node is represented by its address name showing a path in the tree and the mode of exploration in brackets: computation which is not explored yet is marked by `(?)`, intermediate state already explored `(n)` denotes quantity of transitions from this state, successful termination is denoted by `(+)`. The current position in the simulation is designated in a tree by the corresponding note. Correspondence between tree nodes and matrix coordinates is designated by bracketed numbers in the tree.

6 Summary, present and future work

We have described generation of the simulator for concurrent constraint languages from general Action language framework for the definition of the syntax and intensional and interaction semantics of ncpl. In the Action language framework there have been modelled two traditional models of ccp with and without consistency check, interactive ccp with a user as a participant of a computation.

Probabilistic framework for the simulation and analysis of probabilistic computations has been introduced on the base of probabilistic concurrent constraint language. It includes a transition probability, reachability probability and probabilistic matrix of the transition probability distribution as a snapshot the computation history. This permits reasoning about probabilities including operating on the matrix, and is a step towards probabilistic reasoning. The probabilistic environment enables to perform integrated quantitative-qualitative analysis for nondeterministic concurrent models via process algebras.

The generic Action-based simulator has been applied also to coordinational languages to model distributed computation and simulate agents behaviour.

Future work is planned in the following directions:
- to create a uniform integrated environment for quantitative-qualitative analysis of the computational models, which supports a general formalism for modelling multiple aspects of concurrent distributed systems,
- to develop a language for such analysis and management of the systems behaviour and
- to produce expert systems for study and verification of the computational models, prediction behaviour of systems, evaluation of performance and assessment reliability of these.

7 Acknowledgements

Tatyana Valkevych has been supported by a Royal Society/NATO fellowship grant, and Alexander Letichevsky by a Royal Society joint project between the Glushkov Institute and City University.

References

[1] F. S. 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.

[2] J. V. Kapitonova, A. A. Letichevsky, and S. V. Konozenko. Optimization of algebraic programs. In *ISSAC' 91*, pages 370–376. ACM Press, 1991.

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

[4] J.V. Kapitonova, A.A.Letichevsky, V.A.Volkov, and M.S.Lvov. Tools for solving problems in the scope of algebraic programming. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Integrating Symbolic Mathematical Computation and Artificial Intelligence*, pages 30–47. Lecture Notes in Computer Science, Vol. 958, Springer, 1995.

[5] A. A. Letichevsky. Development of rewriting strategies. In *PLILP'93*, volume 714. LNCS, 1993.

[6] 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.

[7] A. A. Letichevsky and D. R. Gilbert. A general theory of action languages. *Cybernetics and System Analysis*, 34(1):32–51, January-Fabruary 1998.

[8] A. Di Pierro and H. Wiklicky. An operational semantics for probabilistic concurrent constraint programming. In *ICCL'98 - IEEE Computer Society International Conference on Computer Languages (Loyola University Chicago)*, 1998.

[9] T. Valkevych, D. Gilbert, and A. Letichevsky. A generic tool for modelling and simulating the behaviour of concurrent systems. Technical Report 1998, Department of Computer Science, City University, 1998.

[10] T. Valkevych, D. Gilbert, and A. Letichevsky. Simulating pccp program in the action language workbench. In *Workshop on Probabilistic Logic and Randomised Computation, held as part of the 10th European Summer School in Logic, Language and Information ESSLLI-98*, 1998.

[11] P. van Hentenryck and V. Saraswat et al. Strategic directions in constraint programming. *ACM Computing Survey*, 28(4):701–726, December 1996.

```
prog: (tell c || proc1 (a , b)) + (tell k || tell m ; ask (k & m) ; tell OK)
env:  1

WBAL:
Program asks a
This constraint is consistent but not entailed by a store
Store is 1
Say ok or add new constraint to a store
WBAL>a;

prog: (tell c || proc1 (a , b)) + (tell k || tell m ; ask (k & m) ; tell OK)
env:a

1:
   act:  tell c
   prog: ask a ; tell b
   env:  c
   next: 0

2:
   act:  tell m
   prog: tell k ; ask (k & m) ; tell OK
   env:  m
   next: 0

3:
   act:  tell k
   prog: tell m ; ask (k & m) ; tell OK
   env:  k
   next: 0

4:
   act:  Env : tell a
   prog: ask a ; tell c || tell b
   env:  a
   next: 0
WBAL: You are in a root, no move back
      select direction 1...4
WBAL>3;

prog: tell m ; ask (k & m) ; tell OK
env:  k

1:
   act:  tell m
   prog: ask (k & m) ; tell OK
   env:  k & m
   next: 0   Ent

WBAL: select direction -1(back) or 1 down
```

Figure 10: Fragment of the simulation of a ccp program in the basic Action model

```
/*   i n i t i a l   s t a t e   o f   c o m p u t a t i o n */
/*   prog:=(A||B);   */
/*   env:=1;         */


WBAL: current state
normalized probabilistic
agent:
 tell c

environment:
 e

at the expenses of 0.125

behaviour:

1:
   name:  1 # 3 # 2 # 1
   prob:  1
   act:   tell c
   agent: Delta
   env:   e & c
   next:  0
   at the expenses of 0.125

WBAL: matrix of probability

  (1)  1(3)
  (2)  1 # 1(?)
  (3)  1 # 2(?)
  (4)  1 # 3(2)
  (5)  1 # 3 # 1(1)
  (6)  1 # 3 # 1 # 1(+)
  (7)  1 # 3 # 2(1)  YOU ARE HERE
  (8)  1 # 3 # 2 # 1(+)

   1      2      3      4      5      6      7      8
   ^      ^      ^      ^      ^      ^      ^      ^
1< 0      0.5    0.25   0.25   0      0      0      0
2< 0      0      0      0      0      0      0      0
3< 0      0      0      0      0      0      0      0
4< 0      0      0      0      0.5    0      0.5    0
5< 0      0      0      0      0      1      0      0
6< 0      0      0      0      0      1      0      0
7< 0      0      0      0      0      0      0      1
8< 0      0      0      0      0      0      0      1
```

Figure 11: Fragment of the simulation of the probabilistic concurrent computation in the probabilistic AL workbench