Concurrent Constraint Programming at SICS with the Andorra Kernel Language (Extended Abstract)

Seif Haridi Sverker Janson
Johan Montelius Torkel Franzén Per Brand
Kent Boortz Björn Danielsson Björn Carlson
Torbjörn Keisu Dan Sahlin Thomas Sjöland

SICS, Box 1263, S-164 28 KISTA
Tel +46-8-752 15 00, Fax +46-8-751 72 30
E-mail {seif, sverker}@sics.se

Abstract

SICS is investigating a new generation of languages for symbolic processing that are based on the paradigm of concurrent constraint programming. A wide range of pertinent topics are being studied. In particular our efforts are devoted to producing a high quality programming environment based on the Andorra Kernel Language (AKL), a general purpose concurrent constraint language.

1 Introduction

Concurrent constraint programming (CCP) is a powerful paradigm for programming with constraints while being based on simple concepts [9, 11]. A set (or conjunction) of constraints, regarded as formulas in first-order logic, forms a *constraint store*. A number of *agents* interact with the store using the two operations *tell*, which adds a constraint to the store, and *ask*, which tests if the store either entails or disentails the asked constraint, otherwise waiting until it does. Telling and asking correspond to sending and receiving "messages", thereby providing the basic means for communication and synchronisation for concurrent programming.

The Andorra Kernel Language (AKL) is a concurrent constraint programming language which generalises the above functionality using a small set of powerful combinators [5]. The basic paradigm is still that of agents communicating over a constraint store, but the combinators make possible also other readings, depending on the context, where agents compute functions or relations, serve as user-defined constraints, or as objects in object-oriented programs. A major point of AKL is that its paradigms can be combined. For example, it is quite natural to have a reactive process- or object-oriented top-level in a program, with other components performing constraint solving using don't know nondeterminism. The nondeterminism can be encapsulated, so that it does not affect the process component.

Nondeterminism in AKL is controlled using *stability*, a generalisation of the Andorra principle, which has proven its usefulness in the context of constraint programming (see e.g., [4]).

AKL offers a large potential for parallel execution, no less than that of logic programming languages and dataflow languages [10]. Arbitrary parallel algorithms can be programmed in AKL, which is exemplified by its ability to simulate a PRAM. This ability is given by ports, an extension of the language for process communication, but in the spirit of concurrent constraint programming [6]. Pure functional and logic languages do not have this ability.

SICS is currently developing a programming environment for AKL [7]. Among our goals are to: (1) develop the necessary implementation technology for efficient (sequential and parallel) execution, (2) develop an execution model which allows different constraint systems to be easily incorporated, (3) offer good interoperability with conventional languages such as C, and (4) investigate large scale applications where combinations of paradigms naturally occur.

Our efforts are divided into the following main lines of activities.

Language Design A language meeting our requirements has been designed: AKL. Possible extensions and generalisations are being investigated. AKL features

- Combinators for (parallel) composition, hiding, conditional choice, nondeterminate choice, committed choice, and solution aggregation
- Improved control and synchronisation, e.g., encapsulation of nondeterminism controlled by stability
- Subsumption of Prolog, CLP, and committed-choice languages such as GHC, Parlog, and Strand
- Support for concurrent object-oriented programming
- Support for arbitrary parallel algorithms

Current and future work includes investigation of maximizing (minimising) combinators, and *engines*, which provide an AKL computation with the ability to inspect and control another computation.

Implementation A prototype implementation of full AKL has been developed. Experiments indicate that performance, using an optimising compiler being developed, will be no less than that of state of the art implementations of Prolog (e.g., SICStus Prolog). The implementation consists of the following main parts.

- A compiler (written in AKL) to an abstract machine
- A threaded emulator (written in C)
- A Prolog-style debugger

Current and future work includes developing an optimising compiler and a parallel implementation.

Constraint Systems A number of different constraint systems are being considered. The prototype implementation is parameterised with constraint systems, which may be added as separate modules. The following constraint systems are addressed. (See also the following section.)

- Herbrand and rational trees serve as a foundation.
- Feature trees have been implemented.
- Finite domains are being investigated.
- Complete Herbrand (closed under logical combinators) is being designed.

Current work also includes the definition of a generic constraint interface.

Formal Aspects Soundness and completeness results for a logical interpretation of AKL have been shown and a proof system for programs is being investigated.

Analysis and Transformation Program analysis by abstract interpretation and program transformation by partial evaluation are studied, both with the aim to improve performance.

In the remainder of this paper a short section will first summarise the constraint systems that have been considered, and the rest is devoted to an explanation of AKL, since it is assumed that this language is largely unknown to the present readership.

2 Constraint Systems

A *constraint system* for AKL may in principle be any first-order theory, closed under conjunction and existential quantification, for which decision procedures for satsifiability and entailment can be provided. Of course, a large body of work on practically motivated constraint systems exists. We have so far considered Herbrand and rational trees, feature trees, finite domains, and complete Herbrand. This investigation will surely be extended to other systems in the future.

2.1 Herbrand and Rational Trees

Herbrand is a theory of equality of terms (the Clark equality theory). The difference between Herbrand and rational trees is that the latter provides solutions to constraints of the form '$X = f(X)$'. Traditionally, logic programming is based on Herbrand or rational trees.

For example 'T = tree(I,L,R)' and 'L = [1,2,3|S]', as may be found in Prolog programs, are Herbrand constraints.

2.2 Feature Trees

Feature trees are formed from constraints of the form 'X f Y', where 'f' is a *feature*, a feature being anything which may serve as a label [1]. Intuitively, the feature constraint associates with X a "property" f having the value Y, thus regarding X as a map from properties to values.

2.3 Finite Domains

Finite domain constraints encodes efficiently properties for finite sets [12, 13].

For example, 'X in 1..5' restricts X to be in the range 1 to 5, 'Z in 4..8' correspondingly for Z, and by adding '$X > Z$', we may conclude '$X = 5$' and '$Z = 4$'.

2.4 Complete Herbrand

Complete Herbrand is Herbrand closed under the usual (first-order) logical combinators [8]. This makes the decision procedures for satisfiability and entailment considerably more complex, while offering, among other things, the potential for a more powerful treatment of negation.

3 Language Design

In this section AKL is introduced step by step, introducing one language construct at a time, also explaining its behaviour. There is no space for a formal definition of the computation model (even though it is fairly concise). An explanation of *solution aggregates* has also been omitted for reasons of space.

For a formal computation model see [5], and [3] which also defines the logical interpretation. Observe that these are based on a clausal syntax which is equivalent to the current combinator syntax. Both are available in the AKL programming system.

3.1 Basic Constructs

The agents of concurrent constraint programming correspond to statements being executed concurrently.

Constraints, as discussed in the previous section, appear as atomic statements known as *constraint atoms* (or just *constraints*).

A *program atom* of the form

$\langle$name$\rangle$(X_1, ..., X_n)

is a defined agent. For example,

plus(X, Y, Z)

is a (plus/3) atom.

The behaviour of atoms is given by *(agent) definitions* of the form

$\langle$name$\rangle$(X_1, ..., X_n) := $\langle$statement$\rangle$.

The variables X_1, ..., X_n must be different. During execution, any atom matching the left hand side will be replaced by the statement on the right hand side. For example,

plus(X, Y, Z) := Z = X + Y.

is a definition (of plus/3).

A *composition statement* of the form

$\langle$statement$\rangle$, ..., $\langle$statement$\rangle$

builds a composite agent from a number of agents. Its behaviour is to replace itself with the concurrently executing agents corresponding to its components.

A *hiding statement* of the form

X_1, ..., X_n : $\langle$statement$\rangle$

introduces variables with local scope. The behaviour of a hiding statement is to replace itself with its component statement, in which the variables X_1, ..., X_n have been replaced by new variables.

The *conditional choice statement*

($\langle$statement$\rangle$ $\rightarrow$ $\langle$statement$\rangle$
; ...
; $\langle$statement$\rangle$ $\rightarrow$ $\langle$statement$\rangle$)

is used to express conditional execution. Its components are called *(guarded) clauses* and the components of a clause *guard* and *body*. A clause may be enclosed in hiding.

The behaviour of a conditional choice statement is as follows. Its guards are executed with corresponding local constraint stores. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If the first (remaining) guard is successfully reduced to a store which is entailed by the union of external stores, the conditional choice statement is replaced with the composition of the constraints with the body of the corresponding clause.

If a variable Y is hidden in a clause, then, when testing for entailment, the local constraint store is preceded by the expression 'for some Y' (or logically, '$\exists$Y'). For example, in

X = f(a), (Y : X = f(Y) $\rightarrow$ q(Y))

the asked constraint is '$\exists$Y (X = f(Y))' ('for some Y, X = f(Y)'), which is entailed, since there exists a Y (namely 'a') such that X = f(Y) is entailed.

It is now time for a first small example, illustrating the nature of concurrent computation in AKL. The following definitions will create a list of numbers, and add together a list of numbers, respectively.

```
list(N, L) :=
        ( N = 0 → L = []
        ; L1 : N > 0 → L = [N|L1], list(N - 1, L1) ).

sum(L, N) :=
        ( L = [] → N = 0
        ; M, L1, N1 : L = [M|L1] → sum(L1, N1), N = N1 + M ).
```

The following computation is possible. In the examples, computations will be shown by performing rewriting steps on the state (or configuration) at hand, unfolding definitions and substituting values for variables, etc., where appropriate, which should be intuitive. In this example we avoid details by showing only the relevant atoms and the collection of constraints on the output variable N. Intermediate computation steps are skipped. Thus,

```
        list(3, L), sum(L, N)
```

is rewritten to

```
        list(2, L1), sum([3|L1], N)
```

by unfolding the list atom, executing the choice statement, and substituting values for variables according to equality constraints. This result may in its turn be rewritten to

```
        list(1, L2), sum([2|L2], N1), N = 3 + N1
```

by similar manipulations of the list and sum atoms. Further possible states are

```
        list(0, L3), sum([1|L3], N2), N = 5 + N2
        sum([], N3), N = 6 + N3
        N = 6
```

with final state N = 6.

The list/2 agent produces a list, and the sum/2 agent is there to consume its parts as soon as they are created. If the tail of the list being consumed by the sum/2 call is unconstrained, the sum/2 agent will wait for it to be produced (in this case by the list/2 agent).

The simple set of constructs introduced so far is a fairly complete programming language, quite comparable in expressive power to, e.g., functional programming languages.

In the following sections, we will introduce constructs that address the specific needs of important programming paradigms, such as processes and process communication, object-oriented programming, relational programming, and constraint satisfaction. In particular, we will need the ability to choose between alternative computations in a manner more flexible than that provided by conditional choice.

3.2 Don't Know Nondeterminism

Many problems, especially frequent in the field of Artifical Intelligence, and also found elsewhere, e.g., in operations research, are currently solvable only by resorting to some form of search. Many of these admit very concise solutions if the programming language abstracts away the details of search by providing don't know nondeterminism.

For this, AKL provides the *nondeterminate choice* statement.

```
        ( ⟨statement⟩ ? ⟨statement⟩
        ; . . .
        ; ⟨statement⟩ ? ⟨statement⟩ )
```

The symbol '?' is read *wait*. The statement is otherwise like the conditional choice statement.

The behaviour of a nondeterminate choice statement is as follows. Its guards are executed with corresponding local constraint stores. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If only one clause remains, and its guard is successfully reduced to a store which is consistent with the union of external stores, the choice statement is said to be *determinate*. Then, the nondeterminate choice statement is replaced with the composition of the constraints with the body of the corresponding clause. Otherwise, if there is more than one clause left, the choice statement is said to be *nondeterminate*, and it will wait. Subsequent telling of other agents may make it determinate. If eventually a state is reached in which no other computation step is possible, each of the remaining clauses may be tried in different copies of the state. The alternative computation paths are explored concurrently.

Let us first consider a very simple example, an agent that accepts either of the constants a or b, and then does nothing.

```
p(X) :=
        ( X = a ? true
        ; X = b ? true ).
```

The interesting thing happens when the agent p is called with an unconstrained variable as an argument. That is, we expect it to produce output. Let us call p together with an agent q examining the output of p.

```
q(X, Y) :=
        ( X = a → Y = 1
        ; true → Y = 0 ).
```

Then the following is one possible computation starting from

```
        p(X), q(X, Y)
```

First p and q are both unfolded.

```
        ( X = a ? true ; X = b ? true ),
        ( X = a → Y = 1 ; true → Y = 0 )
```

At this point in the computation, the nondeterminate choice statement is nondeterminate, and the conditional choice statement cannot establish the truth or falsity of its condition. The computation can now only proceed by trying the clauses of the nondeterminate choice in different copies of the computation state. Thus,

```
        X = a, ( X = a → Y = 1 ; true → Y = 0 )
        Y = 1
```

and

```
        X = b, ( X = a → Y = 1 ; true → Y = 0 )
        Y = 0
```

are the two possible computations. Observe that the nondeterminate alternatives are ordered in the order of the clauses in the nondeterminate choice statement.

The constructs introduced so far give us (constraint) logic programming in addition to functional programming. Nondeterminism is introduced only lazily. Propagation of known constraints is always given priority. This simple functionality gives us means to solve many constraint satisfaction problems efficiently [4].

Up to this point, the constructs introduced belong to the strictly logical subset of AKL, which has a straightforward interpretation in first-order logic both in terms of success and failure.

3.3 Don't Care Nondeterminism

In concurrent programming, processes should be able to react to incoming communication from different sources. In constraint programming, constraint propagating agents should be able to react to different conditions. Both of these cases can be expressed as a number of possibly non-exclusive conditions with corresponding branches. If one condition is satisfied, its branch is chosen.

For this, AKL provides the *committed choice* statement

```
    ( ⟨statement⟩ | ⟨statement⟩
    ; . . .
    ; ⟨statement⟩ | astatementn )
```

The symbol '|' is read commit. The statement is otherwise like the conditional choice statement.

The behaviour of a committed choice statement is as follows. Its guards are executed with corresponding local constraint stores. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If any of the (remaining) guards is successfully reduced to a store which is entailed by the union of external stores, the committed choice statement is replaced with the composition of the constraints with the body of the corresponding clause.

List merging may now be expressed as follows, as an example of an agent receiving input from two different sources.

```
merge(X, Y, Z) :=
    ( X = [] | Z = Y
    ; Y = [] | Z = X
    ; E, X1, Z1 : X = [E|X1] | Z = [E|Z1], merge(X1, Y, Z1)
    ; E, Y1, Z1 : Y = [E|Y1] | Z = [E|Z1], merge(X, Y1, Z1) ).
```

A merge agent can react as soon as either X or Y is given a value. In the last two guarded statements, hiding introduces variables that are used for "matching" in the guard, as discussed above. These variables are constrained to be equal to the corresponding list components.

3.4 Encapsulated Computations

To avoid unwanted interactions between don't know nondeterministic and process-oriented parts of a program, the nondeterministic part can be encapsulated in a statement that hides nondeterminism. Nondeterminism is encapsulated in the guard of a conditional or committed choice and in the solution aggregation constructs provided by AKL.

The scope of don't know nondeterminism in a guard is limited to its corresponding clause. New alternative computations for a guard will be introduced as new alternative clauses. This will be illustrated using the following simple nondeterminate agent.

```
or(X, Y) :=
    ( X = 1 ? true
    ; Y = 1 ? true ).
```

Let us start with the statement

```
    ( or(X, Y) | q )
```

The or atom is unfolded, giving

```
    ( ( X = 1 ? true ; Y = 1 ? true ) | q )
```

Since no other step is possible, we may try the alternatives of the nondeterminate choice in different copies of the closest enclosing clause, which is duplicated as follows.

```
( X = 1 | q
; Y = 1 | q )
```

Other choice statements are handled analogously.

Before leaving the subject of don't know nondeterminism in guards, it should be clarified exactly when alternatives may be tried. A (possibly local) state with agents and their store is (locally) stable if no computation step other than splitting a nondeterminate choice is possible, and no such computation step can be made possible by adding constraints to external constraint stores (if any). Splitting may then be applied to the leftmost possible nondeterminate choice in a stable state.

3.5 Ports for Concurrent Objects

The combinators mentioned above are adequate to model concurrent objects in a style known from concurrent logic programming. A standard example of an object definition is a bank account providing services such as withdrawals, deposits, etc.

```
make_bank_account(S) := bank_account(S,0).

bank_account(Ms, State) :=
    ( Ms = [] → true
    ; A,R : Ms = [withdraw(A)|R] →
          bank_account(R, A-State)
    ; A,R : Ms = [deposit(A)|R] →
          bank_account(R, A+State)
    ; A,R : Ms = [balance(A)|R] →
          A = State, bank_account(R, State)).
```

However it known from experience that communication between objects using streams and merger agents is very awkward.

In AKL we use another communication medium between objects, called *ports*. A port is a binary constraint on a bag (a multiset) of messages and a corresponding stream of these messages. It simply states that they contain the same messages in any order. A bag connected to a stream by a port is usually identified with the port, and is referred to as a port. The open_port(P,S) operation relates a bag P to a stream S, and connects them through a port.

The stream S will typically be connected to an object of the above form. Instead of using the stream to access the object, we will send messages by adding them to the port. The constraint send(P,M) sends the message M to the port P. To satisfy the port constraint, a message sent to a port will immediately be added to its associated stream, first come first served. In this sense the port constraints have the same status as the committed choice statement by committing to a single arbitrary order of messages in the stream associated with the port constraint.

When a port is no longer referenced from other parts of the computation state, when it becomes garbage, it is assumed that it contains no more messages, and its associated stream is automatically closed. When the stream is closed, any object consuming it is thereby notified that there are no more clients requesting its services.

A simple example follows.

```
open_port(P,S), send(P,a),send(P,b)
```

yields

```
P = <a port>, S=[a,b]
```

Ports solve a number of problems that are implicit in the use of stream. Here is a summary, for a detailed description of these see [6].

- Several clients can access the same objects without the need to explicitly merge messages into a single stream.
- Objects can be embedded freely in other data structures provided by AKL.
- Message sending conceptually takes constant time in the computational model of AKL.
- Automatically closing the stream associated to a port, when the port is no longer accessible, provides a good method for garbage collecting concurrent objects.

3.6 Ports for State and Parallel Algorithms

Ports provide AKL with means to incorporate various types of mutable data structures, such as arrays and hashtables. These structures are modelled in AKL as objects connected to a port, but may be implemented very efficiently at a lower level.

For example, a memory cell can be modelled in AKL as an object connected to a port that accepts the messages read(V), write(V), and exchange(V1,V2), to read a value V, to write a value V, and to atomically exchange the current value V1 with V2, respectivley.

This ability allows us to model a parallel random access memory, or parallel access to hash tables in the language, and as a consequence allows us to write many parallel algorithms that cannot be efficiently expressed in pure functional and concurrent logic languages. The following is a typical example.

First we define a shared memory as follows.

```
memory(M) :=
        M = m(C1, ..., Cn),
        cell(C1), ..., cell(Cn).
```

where cell/1 agents are specified as above. M becomes a tuple of ports to cells.

The problem is to, given a binary tree in which the leaves contain numbers in the range $1, \ldots, n$, count the occurrences of each number by a parallel algorithm (as parallel "as possible"). In our solution the occurrences are collected in a table of counters, with indices in the given range. Assume that the memory agent defined above defines a memory of this size. The program traverses the tree, incrementing the counter corresponding to each number found. To guarantee that all nodes have been counted, the program performs the computation in a guard, making the table visible to other agents only when the computation has completed.

```
histogram(T, M) :=
        memory(M), count(T, M) ? true.

count(Tree, Table) :=
        (I,C,K : Tree = leaf(I) →
                arg(I, M, C), send(exchange(K,K+1),C)
        ; L,R : Tree = node(L,R) →
                count(L,M),count(R,M)).
```

This example is due to [2].

4 Concluding Remarks

Current and planned topics at SICS include efficient sequential and parallel implementations parametrised with user-definable constraint systems (in C), implementations of various constraint systems, extensions of the basic framework, such as engines for meta-level programming, program analysis and program transformation, inter-operability with conventional languages and operating systems, and investigation of formal properties.

An experimental AKL programming system is available from SICS for research purposes. The system consists of a compiler (in AKL) from AKL to an abstract machine, an emulator written in C (including a copying garbage collector), and a Prolog style debugger.

References

[1] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*. ICOT, 1992.

[2] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: extending a parallel, non-strict, functional language with state. In *Functional Programming and Computer Architecture '91*, 1991.

[3] Torkel Franzén. Logical aspects of the Andorra Kernel Language. SICS Research Report R91:12, Swedish Institute of Computer Science, October 1991.

[4] Steve Gregory and Rong Yang. Parallel constraint solving in Andorra-I. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*. ICOT, 1992.

[5] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*. MIT Press, 1991.

[6] Sverker Janson, Seif Haridi, and Johan Montelius. *Research Directions in Concurrent Object-Oriented Programming*, chapter Ports for Objects in Concurrent Logic Programs. MIT Press, 1993. To appear.

[7] Sverker Janson and Johan Montelius. The design of the AKL/PS 0.0 prototype implementation of the Andorra Kernel Language. ESPRIT deliverable, EP 2471 (PEPMA), Swedish Institute of Computer Science, 1992.

[8] Torbjörn Keisu. Hcl. SICS research report, Swedish Institute of Computer Science, 1993. Fortcoming.

[9] Michael J. Maher. Logic semantics for a class of committed choice programs. In *Logic Programming: Proceedings of the Fourth International Conference*. MIT Press, 1987.

[10] Remco Moolenaar and Bart Demoen. A parallel implementation of AKL. CW-report, Department of Computer Science, Katholieke Universiteit Leuven, 1991.

[11] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1990. To be published by MIT Press.

[12] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[13] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Technical report, Computer Science Department, Brown University, 1991.