On the automatic generation of events in Delta Prolog

Veroniek Dumortier Maurice Bruynooghe
Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3030 Heverlee

Abstract

Delta Prolog is a concurrent logic programming language, which extends Prolog with AND-parallelism. Communication and synchronization between parallel processes is established by using special control structures: event goals.
To relieve the programmer from inserting these structures at the correct program points, we propose a method for automatically generating them, departing from a Prolog program together with the top-level call pattern and indications of which goal-expressions should be executed in parallel. This method is based on the technique of abstract interpretation. The gathered abstract information allows to decide at which program points an event goal has to be generated and how it is composed.
However, it turns out that only a limited class of programs can be transformed automatically. Problems impeding general automatic transformation are discussed.

Keywords: Delta Prolog, distributed Prolog, abstract interpretation.

1. Introduction

Delta Prolog [6,11,12,13] is a concurrent logic programming language, which extends Prolog to include AND-parallelism and interprocess communication and synchronization. The latter concepts are expressed by means of special language constructs, namely (synchronous) event goals.
This paper presents a method to derive these constructs automatically for programs that are executable under the Prolog computation rule. Given such a program, its top-level call pattern, and annotations concerning the desired AND-parallelism, the method derives the event goals to be inserted. The technique of abstract interpretation [4,5,7,8,9,10] is used to gather the necessary information upon which the generation of event goals can be based. The kind of abstract interpretation applied here is an extension of the application developed by Bruynooghe and Janssens [3]; this is itself an instance of the general framework described in [4].

However, automatic program transformation is only applicable in a limited number of cases. We mention several problems and identify a class of treatable programs.

The paper starts with a brief overview of the Delta Prolog language and concepts. Afterwards, the method for automatic generation of event goals is presented. The core of the method is the abstract interpretation application. The complete procedure is obtained by augmenting this core with the procedure for event-generation analysis. In the last section we evaluate the method and discuss its applicability.

2. Delta Prolog: overview of the language

2.1 Delta Prolog control constructs

Delta Prolog contains some special language constructs: split goals and event goals.

- *Split goals* are used to express AND-parallelism. They are of the form **S1 // S2**, where S1 and S2 are arbitrary Delta Prolog goal expressions. Logically, S1 // S2 is true iff S1 and S2 are both true. Operationally, S1 and S2 are solved in parallel.
- *Event goals* are of the form X ! E : C or X ? E : C. Their components are:
  — X: the event term or the message (a Prolog term).
  — ! or ?: the communication mode (infix binary predicate symbol).
  — E: the event name (a Prolog constant or variable).
  — C: the event condition (a conjunction of goals whose evaluation invokes no Delta Prolog goals)

  Two event goals X1!E:C1 and X2?E:C2 with the same event name and different communication modes are called *complementary* event goals.
  Event goals deal with interprocess communication as well as synchronization.
  — *Communication* is realized through two event goals by unifying their event terms under the restrictions given by the event conditions, e.g. X ? e: X > 0.
  — *Synchronization* is accomplished by the fact that an event goal can only be solved if its complementary one is available in a parallel derivation; otherwise, resolution of the goal suspends. Event goals thus establish a rendez-vous between two processes. If a complementary event goal is never derived, a deadlock occurs.

  Remarks:
  — To be able to run programs on a parallel machine without common memory, the implementors add some restrictions which we adopt in the sequel.
    1. In a split goal S1 // S2, S1 and S2 should not share free variables. If they do have variables in common, these are unified at completion of the split goal.
    2. The event terms are required to be ground after solving complementary event goals.

  — An event name may be shared only by *two* active parallel processes, to be able to define a complete traversal of the search space. If multiple consumers or producers are desired, asynchronous events or a communications manager may be used.

2.2 Distributed backtracking

In contrast to committed-choice languages, Delta Prolog incorporates don't-know non-determinism. Executing a program means searching for *all* answers to a given query. The search rule involves sequential clause search and backtracking. Backtracking occurs at two levels. As long as no interaction point (i.e. a split or event goal) is involved, *local* backtracking within one process occurs. *Distributed* backtracking is triggered to undo a joint derivation step. It is a kind of intelligent backtracking that affects only the processes (directly or indirectly) related to a failure. Insight into the rules for distributed backtracking allows one to place events so that the communication overhead is reduced and the overall efficiency is increased.

3. Automatic generation of events

3.1 Analysing the problem

Suppose we are given a Prolog program whose execution does not fail under the standard Prolog computation rule. The program's efficiency may be increased if some of its goal expressions are solved concurrently. The user of our system indicates this parallelism by incorporating the split operator // into the program, as illustrated here.

```
sumofsquares(L,S):- squares(L,SqL) // sum(SqL,0,S).
(s1) squares([],[]).              (r1) sum([],S,S).
(s2) squares([X|L],[SX|SL]):-     (r2) sum([X|L],AccS,S):-
       SX is X*X,                        NewAccS is AccS + X,
       squares(L,SL).                    sum(L,NewAccS,S).
```

This program computes the relation sumofsquares(L,S) where S is the sum of the squares of the elements of L. We use this example throughout the rest of this text.
The goal is to derive an executable and efficient Delta Prolog program which is equivalent to the original one. This mostly requires the addition of event goals:

```
sumofsquares(L,S):- squares(L,SqL) // sum(SqL,0,S).
(s1) squares([],[]):- [] ! ev.    (r1) sum([],S,S):- [] ? ev.
(s2) squares([X|L],[SX|SL]):-     (r2) sum([X|L],AccS,S):-
       SX is X*X,                        X ? ev: integer(X),
       SX ! ev,                          NewAccS is AccS + X,
       squares(L,SL).                    sum(L,NewAccS,S).
```

Event goals need to be added only within parallel processes sharing a variable which is non-ground upon initialisation of the processes. For the moment we consider only the case in which a variable is common to *two* processes (extensions will be discussed in section 3.4). One of the processes (the *receiver* or *consumer*) uses (part of) the value of the common variable to

1. be able to continue its execution, as occurs in the example where the builtin +/2 in

sum/2 causes an error when the value of X is not available, or to
2. avoid wasteful computations because of non-determinism in one of the processes, as would be the case in the above example with + defined by an infinite set of ground facts.

The effect of event goals is to suspend the receiver until the *sender* or *producer* provides the information needed to proceed. Although this restricts the parallelism, it can have a beneficial effect. In case 1, the event is necessary to *avoid failure*. In case 2, the event increases *efficiency*: with non-determinism in the receiver, it avoids local backtracking; with non-determinism in the sender, it avoids the expensive distributed backtracking. The latter is typical for generate-and-test programs where the generate part is the sender.
Although bi-directional communication between parallel processes is possible in Delta Prolog, it is never called for in programs written with the standard Prolog computation rule in mind. Consequently, we will always be able to distinguish a real sender and a real receiver process.

Having motivated the use of event goals, we now discuss some conditions on their generation.

First, the derived program should be *equivalent* to the original Prolog program: exported predicates should retain their input-output behaviour. The main problem to preserve the set of solutions is to avoid deadlock. This implies that, for each event goal, a complementary event goal must exist at run-time.

Second, complementary event goals should satisfy certain properties:

—An event within a receiver clause often contains an event condition that tests the type of the value to be received. This test is necessary if some receiver goal, following the event, would cause a run-time error in case a value of a wrong type was accepted. For example, in the above program, the condition integer(X) within the event of (r2) guarantees that +/2 will have integer operands.
The use of the type test may also prevent further useless execution of an incorrect combination of a sender and receiver clause. For example, when combining (s1) and (r2), simply unifying the event terms would succeed; due to the *explicit* type test, the combination will fail when trying to solve the event goals.

—In correct clause combinations, each pair of complementary event goals should contain event terms having not only the same type, but also the same enclosing structure: the event terms should each refer to the original common variable itself or to components at the same position within the value of the common variable, with that value structure being the same in sender and receiver. For example, the type of common variable values treated by (s2) and (r2) is a non-empty list of integers and both event terms refer to the first component of the list.
The correspondence of event terms has to be established when generating the event goals (i.e. at compile-time). It is not —and even cannot be— checked at run-time, since, when reaching complementary events during execution, no information is available about the way in which the event terms were selected within the other

parallel process. This is due to an implicit assumption in Delta Prolog, which states that no information is passed from sender to receiver about the upper functors of the common variable value. The reason for this restriction is twofold. Adding events to pass the information would increase the communication overhead. Also, the event terms would still contain free variables after solving the event goals, e.g. this would be the case if (s2) and (r2) contained [SX|SL] ! ev and [X|L] ? ev respectively.

3.2 Core of the method for automatic event generation

The heart of the method consists of gathering information to examine the instantiation of the common variables. This is achieved by using the technique of abstract interpretation [4,5,7,8,9,10]. Guided by the observations made in the previous section, we derive what kind of abstract information is needed.

- **mode information:**
  This enables us to decide whether events should be generated at all, and, if so, when to generate them within the sender. According to the implementation restriction stated in section 2.1, event terms must be ground after solving complementary events. The event term of the receiver process usually is a free variable, which is used afterwards in a goal requiring it to be ground. So, the sender should only be allowed to send *ground* data.

- **type information:**
  Type information enables us to incorporate in the receiver event goals that are complementary to those in the sender, ensuring that their event terms correspond. Type information is also used to derive the type test that is part of the event condition.

An application of abstract interpretation that integrates type- and mode-inferencing and supplies sufficiently detailed information for our purpose has already been developed in [3]. To start the abstract interpretation process, the user has to specify the top-level call-pattern (or abstract query). While resolving this query, the abstract interpretation procedure builds an abstract AND/OR graph. This graph represents a set of concrete proof trees for any concrete call satisfying the abstract query. The solved goals appear as (OR)nodes in the graph. We obtain a *graph* rather than a tree because of the way in which recursive calls are treated: if a recursive call together with its abstract call substitution already occurs within the abstract tree, a back reference is introduced to that previous call. Otherwise, the call is elaborated; it leads to a new version of the procedure defining the predicate.

Each of the goal nodes is annotated with an abstract call- and success-substitution containing a normalized type graph for each argument of the goal. The call (resp. success) type graph represents the set of all terms which can appear at that argument position before (resp. after) that goal is executed. A type graph can contain three kinds of nodes: functor nodes, nodes representing predefined types (such as integer, atom, variable) and OR-nodes, which unite several type variants. The type of each argument of a functor appears as a subgraph of that functor node. Type graphs are normalized, which essentially means that, pairwise, the sons of an OR-node must have nonoverlapping principal functor sets. The set of principal functors of a node is the set of all functors that can be reached directly of indirectly through a sequence of OR-nodes. Normalization not only allows unambiguous selection within type graphs, but it also eases the development of algorithms to manipulate them.

3.3 Basic method

The basic method for automatic generation of event goals consists of three major steps, which will be elaborated below:
—normalization of the input program
—abstract interpretation, with a subprocedure for analysing where and when an event should be generated and, if necessary, for performing event generation
—derivation of a full Delta Prolog program from the abstract interpretation graph

3.3.1 Program normalization

Program normalization consists of reducing all predicates (except builtins) to flat structures of the form p(X1,...,Xn) where all Xi are distinct variables. This is done by introducing explicit unifications of the form X = Y or X = f(Y1,...,Yn) or X = t, where X, Y and Y1,...,Yn are distinct variables and t is a ground term. The third kind of unification prevents splitting up ground terms.
The sumofsquares program is normalized as:

```
sumofsquares(Y1,Y2):-
      Y1 = L, Y2 = SqL, squares(L,SqL) // (Y3 = 0, sum(SqL,Y3,S)).
(s1) squares(X1,X2):-               (r1) sum(X1,X2,X3):-
      X1 = [], X2 = [].                   X1 = [], X2 = S, X3 = S.
(s2) squares(X1,X2):-               (r2) sum(X1,X2,X3):-
      X1 = [X|L], X2 = [SX|SL],           X1 = [X|L], X2 = AccS, X3 = S,
      SX is X*X,                          NewAccS is AccS + X,
      squares(L,SL).                      sum(L,NewAccS,S).
```

Normalization simplifies the abstract interpretation procedure and the subprocedure for generating event goals. It allows us to deal with unification just once, by ruling out full call-head unification and reducing it to a simple renaming of variables. To prevent the need for triggering the event generation procedure at procedure entry, it is important that all arguments in the head of a clause are replaced by *fresh* variables (cfr. following section about the receiver analysis).

3.3.2 Abstract interpretation and event generation

3.3.2.1 Overall description

This step of the method is an extension of the abstract interpretation application for type and mode inferencing. The event-generation analysis will be carried out as a subprocess *during* abstract interpretation, since it is based only upon the abstract information gathered up to the point where the event is introduced. A generated event will be saved as a node in the abstract graph, just like an ordinary goal node.

The abstract interpretation procedure starts off from a top-level call-pattern, e.g. sumofsquares(τ_1,υ) where τ_1= [] | $\iota.\tau_1$. ('|' is used to separate several type alternatives (or OR-branches); '.' is the list constructor. Greek letters are used to represent type denotations; ι represents the integer type and υ represents the type of variables.)
Abstract interpretation before and after a split goal proceeds as in [3]. To deal with split goals, the abstract AND/OR-graph is replaced by an abstract AND/OR/*SPLIT*-graph: a SPLIT-node is created whenever a split goal is reached.

If the goal expressions of a split goal contain non-ground common variables, event generation analysis must be carried out during abstract interpretation of the split branches. A special label is attached to the common variables to distinguish them from non-common ones. The label consists of the indication *C*(ommon), followed by a unique number. In the sumofsquares example, SqL gets the label C1. This identifier is used to generate the correct event name within all event goals concerning (parts of) that specific variable. So, variables that get bound to (part of) the original common variable during execution of the sender or receiver process inherit its label.

A split goal initiates two subprocesses: a sender and a receiver. The leftmost process, L, can always be considered as the sender. It does not need information from the rightmost process, R, to proceed with its execution, since we assumed the given program to be executable under the standard Prolog computation rule. This determination of the dataflow will be justified in a following section concerning efficiency.

Another general aspect concerns the order in which sender and receiver are treated. Abstract interpretation in the existing application [3] is purely sequential and proceeds from left to right. So, the sender will be treated first. Then, the sender's success substitution will become the call substitution of the receiver.

For the purpose of event generation, this purely sequential interpretation and analysis is convenient. Analysing the sender before the receiver respects the natural relation between those processes. A receiver goal may depend on the sender to obtain the type of a common variable during *abstract* interpretation, just as it expects the value of this variable to be supplied by the sender during a *concrete* execution.
Passing the results of analysing the sender on to the receiver provides a way to ensure that correct complementary event goals will be generated. During the construction of the common variable type graph within the sender, we will put an *event marker* at the root of a subgraph when a component having the type of that subgraph is sent over. A complementary event goal will be generated as soon as a component with an event marker at the root of its type graph is reached during the analysis of the receiver. So, event terms will automatically correspond. We will also add a *complementary event marker* to the root of the type graph of the event term.

At each stage of abstract execution, the current abstract substitution will contain two components for each accessible variable:
—a variable label, Ci, or _ (i.e. empty) if the variable is not a common variable.
—a normalized type graph, where functor and predefined nodes contain a node label, a field to contain an event marker, and one to contain a complementary event marker.

We now discuss in detail the event-generation analysis within the sender and the receiver. For the moment we will ignore whether the sender and receiver are part of a larger system of parallel processes; we return to this briefly in section 3.4.

3.3.2.2 Sender analysis

The aim of the analysis within the sender is to detect when (part of) a common variable becomes ground. At that point, an event goal has to be generated. Since the input program has been normalized, the analysis can be focussed on treating builtins and explicit unifications: these are the only goals which may cause a change of type. The analysis procedure is triggered as soon as their success substitution is computed.
The analysis will be illustrated on the sumofsquares example. We will show what abstract information is derived at each program point (Pi), i.e. immediately after call-head unification and after each body goal, and how it is used.

In our example, the top-level call pattern is 'sumofsquares(τ,υ)' with τ = [] | $\iota.\tau$. So, the abstract call substitution of the sender call 'squares(L,SqL)' is: { L: (_,τ), SqL: (C1,υ) }. The first component of the tuple attached to a variable may contain a common variable label; the second one indicates the type of the variable.

The first clause defining squares/2 is (in normalized form):

squares(X1,X2):- (P1) X1 = [], (P2) X2 = [] (P3) .

(P1): { X1: (_,τ), X2: (C1,υ) }.
(P2): { X1: (_,[]), X2: (C1,υ) }.
Since there is no change of the mode of the common variable X2 from (P1) to (P2), no event goal has to be generated.
(P3): { X1: (_,[]), X2: (C1,[]) }.
Comparing (P2) and (P3) reveals that X2 has become completely ground. An event *X2 ! ev1* is generated and stored in the abstract AND/OR/SPLIT graph. We choose '!' as the communication mode of an event within the sender; the event name is derived from the common variable label and there is no event condition. Consequently, there is a program point (P4), which is the success substitution of the event. An event marker is put at the root of the type graph of X2 to indicate the generation of the event:
(P4): { X1: (_,[]), X2: (C1,[]#e) }.

The above clause is transformed into its desired form:

squares(X1,X2):- (P1) X1 = [], (P2) X2 = [], (P3) *X2 ! ev1* (P4) .

We now consider the second clause defining squares/2.

```
squares(X1,X2):-
   (P1) X1 = [X|L], (P2) X2 = [SX|SL],
   (P3) SX = X * X, (P4) squares(L,SL) (P5).
```

(P1): { X1: (_,τ), X2: (C1,υ), X: (_,υ), L: (_,υ), SX: (_,υ), SL: (_,υ) }
(P2): { X1: (_,$\iota.\tau$), X2: (C1,υ), X: (_,ι), L: (_,τ), SX: (_,υ), SL: (_,υ) }
No event goal has to be generated, since no common variable has become ground.
(P3): { X1: (_,$\iota.\tau$), X2: (C1,$\upsilon.\upsilon$), X: (_,ι), L: (_,τ), SX: (C1,υ), SL: (C1,υ) }.

The common variable label of X2 is passed on to its components SX and SL. None of the common variables has become completely ground; so, no event goal has to be generated yet.
(P4): { X1: (_,$\iota.\tau$), X2: (C1,$\iota.\upsilon$), X: (_,ι), L: (_,τ), SX: (C1,ι), SL: (C1,υ) }.
A change in the type of SX also affects X2, because of the binding between SX and X2. We just generate the event goal *SX ! ev1*, since SX is the only common variable that has become *completely* ground. The success substitution of the event goal contains an event marker to reflect this generation; so, for the extra program point (P4') we have:
(P4'): { X1: (_,$\iota.\tau$), X2: (C1,ι#e.υ), X: (_,ι), L: (_,τ), SX: (C1,ι#e), SL: (C1,υ) }.

The recursive call 'squares(L,SL)' has the same call pattern as the original call in the split goal, i.e. { (_,τ), (C1,υ) }. Subsequent interpretation of the call can thus be avoided; a back reference is introduced to the previous, identical call. The success substitution (P5) will be:
(P5): { X1: (_,$\iota.\tau$), X2: (C1,τ'), X: (_,ι), L: (_,τ), SX: (C1,ι#e), SL: (C1,τ') }
with τ' = []#e | ι#e.τ'

The transformed version of the second clause is:

```
squares(X1,X2):-
   (P1) X1 = [X|L], (P2) X2 = [SX|SL],
   (P3) SX = X * X, (P4) SX ! ev1,
   (P4') squares(L,SL) (P5).
```

Finally, we obtain the success substitution of the original sender call in the split goal:
{ L: (_,τ), SqL: (C1,τ') } with τ = [] | $\iota.\tau$ and τ' = []#e | ι#e.τ'

3.3.2.3 Receiver analysis

One can distinguish two situations that should lead to the generation of an event goal within the receiver. The first one is that an event goal should be generated in front of each goal which makes use of (part of) the common variable value (thus requiring it to be ground). The second requirement is that event goals have to be generated complementary to the ones in the sender. Therefore, we can use the type graphs, constructed during interpretation of the sender and passed on to the receiver.

We first focus on fulfilling the second requirement. The analysis will be based on finding common variables with an event marker at the root of their type graph. We only need to consider explicit unifications and builtins, since these are the only goals which can select common variable components and their associated type graphs. Even original common variables that only appeared within the head of a clause now appear within an explicit unification, due to the introduction of fresh head variables during program normalization.

We will use the sumofsquares example to illustrate the receiver analysis. The call substitution of the receiver call 'sum(SqL,Y3,S)' is derived from the success substitution of the sender call, giving { SqL: (C1,τ'), Y3: (_,ι), S: (_,υ) } with τ' = []#e | ι#e.τ'

The first clause defining sum/3 is:

sum(X1,X2,X3):- (P1) X1 = [], (P2) X2 = S, (P3) X3 = S (P4).

(P1): { X1: (C1,τ'), X2: (_,ι), X3: (_,υ) }
(P2): { X1: (C1,[]#e), X2: (_,ι), X3: (_,υ) }
The common variable X1 contains an event marker at its root. So, an event *X1 ? ev1* is generated. We already noticed that an event goal within the receiver should contain an explicit type test as event condition. The test can be derived from the type of X1 in (P2), yielding 'X1 = []'. Since this is identical to the interpreted unification, we can simply take the unification itself as event condition (which will always be the case for a unification X = t, where t is a ground term).
To prevent the generation of another event for X1, a complementary event marker is put at the root of its success type graph:
(P2'): { X1: (C1,[]#ec), X2: (_,ι), X3: (_,υ) }

The subsequent unifications in the clause body do not contain or affect common variables. The first clause is thus transformed into:

sum(X1,X2,X3):- (P1) *X1 ? ev: X1 = []*, (P2') X2 = S, (P3) X3 = S (P4).

The second clause of the sum/3 procedure is:

```
sum(X1,X2,X3):-
    (P1) X1 = [X|L], (P2) X2 = AccS, (P3) X3 = S,
    (P4) NewAccS is AccS + X,
    (P5) sum(L,NewAccS,S) (P6).
```

(P1): { X1: (C1,τ'), X2: (_,ι), X3: (_,υ), X,L,AccS,NewAccS,S: (_,υ) }
(P2): { X1: (C1,ι#e.τ'), X2:(_,ι), X3: (_,υ), X: (C1,ι#e), L: (C1,τ'),
AccS,NewAccS,S: (_,υ) }
The label of X1 is passed on to its components X and L. X contains an event marker at the root of its type graph; so, a complementary event has to be generated.
The predefined predicate integer(X) will be used to check the type of the received event-term value. We hereby want to make a remark about the predefined type atom: if this is the expected type of the event term, we will not use the builtin test predicate atom/1, but real_atom/1 where real_atom(X) is defined as real_atom(X):- atom(X), X \= []. It is important to make the distinction between [] and another atom, because of the special meaning attached to []: [] is the empty list symbol.

The sum/3 clause is so far transformed into:

```
sum(X1,X2,X3):-
    (P1) X1 = [X|L], (P2) X ? ev1: integer(X), (P2') X2 = AccS, (P3) X3 = S,
    (P4) NewAccS is AccS + X,
    (P5) sum(L,NewAccS,S) (P6).
```

with (P2') containing a complementary event marker at the root of the type graph of X and within the type graph of X1 (because of the binding between X and X1):
(P2'): { X1: (C1,ι#ec.τ'), X2: (_,ι), X3: (_,υ), X: (C1,ι#ec), L: (C1,τ'),
AccS,NewAccS,S: (_,υ) }

The subsequent unifications and builtin will not cause the generation of another event.
(P5): { X1: (C1,ι#ec.τ'), X2: (_,ι), X3: (_,υ), X: (C1,ι#ec), L: (C1,τ'),
AccS: (_,ι), NewAccS: (_,ι), S: (_,υ) }
The recursive call sum(L,NewAccS,S) has the same abstract call substitution as the original sum call, i.e. { (C1,τ'), (_,ι), (_,υ) }. A back reference is thus introduced in the abstract AND/OR/SPLIT graph.
The final success substitution of the original receiver call is:
{ SqL: (C1,τ"), X2: (_,ι), X3: (_,ι) } with τ" = []#ec | ι#ec.τ"

Note that, by generating event goals complementary to the ones in the sender, all requirements for some common variable value are automatically satisfied. This is at least so in the usual situation, where the amount of information contained in the event terms of the sender is exactly the same as the amount required by the receiver.

3.3.3 Program derivation

After the entire input program has been interpreted and analysed, a complete abstract AND/OR/SPLIT graph will have been built up. During the interpretation and analysis, the generated event goals were inserted at the right places within the program graph. A full Delta Prolog program can thus be derived from the abstract graph.

3.4 Extensions

Up to now we only examined the case of *two* parallel processes out of their context. The presented method can be extended to deal with nested split goals where each common variable is shared by *at most two* processes. The essential point is that a process may now contain some common variables with a sending role and others with a receiving role; so, the sender/receiver distinction has to be made on variable level, rather than on process level. A common variable has a sending role within the leftmost of the two processes in which it occurs, and a receiving role within the rightmost one. The labels CSi and CRi have to be used instead of Ci to characterize an occurence of a common variable. Both parts of the event generation procedure may now be executed within the same process, depending on the kind of common variable.
If a variable is common to *more than two* processes, *a*synchronous events are needed, since synchronous ones may only be shared by *two* active processes. We will not further explore this case.

A final remark concerns the type of variable sharing between parallel processes. *Static* sharing is the sharing of variables with identical names or what we called common variables. *Dynamic* sharing is established through variable binding at run time: variables with different names in the split parts became bound during execution up to the split goal. Our method presented so far uses information about static sharing and will thus yield Delta Prolog programs which closely resemble hand-written ones. However, information about dynamic sharing is derived during abstract interpretation and can be used to generate additional events. This may contribute to the efficiency of the final program: in the absence of events, compatibility of variable bindings created by each of the parallel processes is not checked until completion of the split goal; by adding events, failure can often be derived much sooner.

4. Discussion

Based on the abstract interpretation application for type- and mode-inferencing, we developed a method for automatic event goal generation that consists of 3 steps: program normalization, abstract interpretation and event-generation analysis, and program derivation. The analysis procedure to decide when an event has to be generated and how it is constructed can be split in two parts:

—analysis within the sender, where event goals and associated event markers are generated as soon as a common variable becomes completely ground; and

—analysis within the receiver, where complementary event goals and complementary event markers are generated, for each common variable having an event marker at the root of its type graph.

4.1 Efficiency considerations

Efficiency of a Delta Prolog program depends largely on how communication between parallel processes is established. We cannot assure that the position of generated events will be optimal. However, by applying some obvious principles one can reduce the communication and synchronization overhead considerably.

Reducing the number of synchronization points implies that:

—one should send as much information as possible within one event goal, thus reducing the number of events, and

—one should avoid to trigger distributed backtracking as much as possible, replacing it by local backtracking.

We contributed to meeting the first principle by a careful normalization of the original program. If we had only allowed explicit unifications of the form X = Y and X = f(Y1,...,Yn) (with X,Y,Y1,...,Yn all variables), this would have led to a lot of events each sending a small amount of information, e.g. X = f(1,g(2)) in the sender would be transformed into X = f(Y1,Y2), Y1 = 1, Y1 ! ev1, Y2 = g(Y3), Y3 = 2, Y3 ! ev1, although all information could be sent at once through X ! ev. This is achieved by introducing the third kind of explicit unification, X = t where t is a ground term.

Attempts we made to realize the second principle are as follows.

—Unifications and their associated events in the sender are put after tests affecting the correct clause selection and thus the validity of the associated common variable instantiation (cfr. next section on correctness); failure of the tests then implies only local backtracking.

—Within the receiver, the type test and the builtin that caused the generation of an event can be made part of the event condition. Failure of the tests will then be covered while still trying to solve the event goals, whereas distributed backtracking would be triggered -thus involving a second synchronization of sender and receiver- if the tests were executed sequentially after solving the events.

The degree of distributed backtracking also depends on which of both split processes is chosen to be the sender. In some cases, efficiency may be increased by considering the rightmost process R as the sender instead of sticking to a left-to-right dataflow. This occurs when R is more deterministic than the leftmost process L with regard to possible values of the common variable. Selecting sender and receiver has an influence on the position of events: an event in the sender occurs *after* the instantiation of the variable, whereas in the receiver it will be put *in front* of the goal which now just tests the variable value. This in turn determines whether local or distributed backtracking is applied in the case of incompatible sender and receiver bindings: first, local backtracking is performed to try out other receiver possibilities; only when none of those turns out to be compatible with the sender instantiation, distributed backtracking is triggered to explore another sender choice. Having the largest number of choices within the receiver ensures that distributed backtracking is avoided as much as possible. However, since analysing the degree of non-determinism is difficult and expensive, we always consider the leftmost process to be the sender.

4.2 Correctness considerations - applicability of the method

We first summarize some important issues which have to be taken care of in order to obtain a correct Delta Prolog program - not only during automatic transformation but also when manually developing such a program. All of these criteria eventually serve the main goal, namely to preserve the input-output behaviour of exported goals.

The primary concern is to avoid deadlock. One always has to ensure that *complementary* events are generated. Moreover, generating superfluous events —in particular duplication of events— must be avoided. Secondly, complementary events can only establish useful communication between sender and receiver if their event terms correspond. Thirdly, a type check is often needed as event condition to avoid run-time errors.

We have stated above how those basic issues are approached within our method, but some points still need to be mentioned.
A first point concerns the prevention of duplicating events: within the sender, an event for some common variable component is generated only when the component becomes completely ground, which happens at one specific point in the program; within the receiver, duplication is prevented by using complementary event markers.
Secondly, the generation of complementary events requires precise abstract information to be passed from sender to receiver. Here, a problem may occur due to the approximate nature of the abstract interpretation procedure. In order to have a finite abstract domain, a depth restriction is introduced: a functor symbol may only appear a limited number of times on a path starting from the root of a type graph. This restriction implies that the sender's final success type graph may be only an approximation of the real type. For example, if the depth restriction is taken to be 1, the sender's success type graph of a matrix represented as a list of rows will be $\tau = []\#e \mid (\tau|\iota\#e).\tau$ instead of the more precise type $\tau = []\#e \mid \tau_2.\tau$ with $\tau_2 = []\#e \mid \iota\#e.\tau_2$. When a matrix element is selected within the receiver, the root of its type graph will not contain an event marker; so, no complementary event will be generated, causing deadlock at run-time. We also note that the depth needed to obtain the required precision cannot be derived automatically from the program text.

Beyond this lack in the analysis, a major share of problems are related to the structure of the input programs. Within the rest of this section we will illustrate some of the most important problems.

A first problem concerns the position of events within the *sender*. In some cases, the basic sender analysis procedure would cause event goals to be generated too soon. An example of one such case is the following procedure (which is part of a larger system):

```
proc(N):- filter(N,X) // consumer(X).
filter(N,N):- N > 0, N < 100.       consumer(X):- write(X).
```

Assume the call substitution of proc(N) is { N: $(\_,\iota)$ }. Applying the basic sender analysis yields:

```
proc(N):- filter(N,X) // consumer(X).
filter(X1,X2):- X1 = N, X2 = N, X2 ! ev, N > 0, N < 100.
consumer(X1):- X1 ? ev: integer(X1), X1 = X, write(X).
```

Note that the input-output behaviour of proc(N) is not preserved: a value which does not lie between the given bounds may appear on the output stream ! The reason is twofold: the common variable value is passed to the receiver before it is tested within the sender and the receiver uses this value while performing a side-effect; this side-effect cannot be undone upon failure of the sender test. To solve this problem, the test part of sender clauses has to be determined such that the common variable unification and associated event goal can be put *after* the sender test, e.g.: filter(X1,X2):- X1 = N, N > 0, N < 100, X2 = N, X2 ! ev. This is entirely analogous to the derivation of guards in the context of committed-choice languages [1,2] and cannot easily be automated in general.

Another case concerns a non-tail-recursive sender procedure, where the non-tail-recursive clause contains an analogous test after its recursive call. For example, consider the predicate minimum/2 with call pattern minimum(τ,υ) where τ = [] | $\iota.\tau$; we immediately present the derived Delta Prolog version:

```
minimum(X1,X2):-
    X1 = [X|L], X2 = MinL, minimum(L,MinL), X > MinL, !.
minimum(X1,X2):-
    X1 = [X|L], X2 = X, X2 ! ev.
```

Note that a value that is not the required minimum can be passed on to the receiver. One possible solution is to transform the non-tail-recursive procedure into a tail-recursive one (which is again not obvious), e.g.:

```
minimum(Y1,Y2):- Y1 = [X|L], Y2 = Min, minimum_tr(L,X,Min).
minimum_tr(X1,X2,X3):-
    X1 = [], X2 = Min, X3 = Min, X3 ! ev.
minimum_tr(X1,X2,X3):-
    X1 = [X|L], X2 = AccMin, X3 = Min, X > AccMin, !, minimum_tr(L,AccMin,Min).
minimum_tr(X1,X2,X3):-
    X1 = [X|L], X2 = AccMin, X3 = Min, minimum_tr(L,X,Min).
```

A second possible solution consists of putting the event goal after the original call to the non-tail-recursive procedure.

Another kind of problem occurs when the common variable is not instantiated incrementally. Consider the following program fragment:

```
append(X1,X2,X3):-                      sum(X1,X2,X3):-
   X1 = [], X2 = L, X3 = L.                X1 = [], X2 = S, X3 = S.
append(X1,X2,X3):-                      sum(X1,X2,X3):-
   X1 = [X|L1], X2 = L, X3 = [X|L3],       X1 = [X|L], X2 = AccS, X3 = S,
   append(L1,L2,L3).                       NewAccS is AccS + X, sum(L,NewAccS,S).
```

In the first clause of append, the generated event term would not be a list *element* but a list, which is not what is expected by the receiver sum/3. A correct Delta Prolog program can be obtained by changing the structure of the sender procedure such that all list elements will be passed one by one. For example,

```
append(X1,X2,X3):-
   X1 = [], copy(X2,X3).
copy(X1,X2):-
   X1 = [], X2 = [].
copy(X1,X2):-
   X1 = [X|L1], X2 = [X|L2], copy(L1,L2).
```

Finally, an even more complex situation concerns the order in which the common variables or their components are instantiated: the sender should send information in the same order as it is expected by the receiver. It is not at all obvious how to change the structure of the program automatically in order for this property to hold.

All of this shows that it is very hard to obtain a general procedure for automatic transformation. The main difficulty is that the program structure often has to be changed significantly. At least, we can characterize a class of programs to which our method can safely be applied. These programs have the following properties: they define a simple system of two parallel processes, sharing one non-ground common variable of the list type whose elements are successively instantiated by the sender (note that the problem of event order is avoided here). Stream-communication is then established between the parallel processes.
For the subclass of generate-and-test programs (e.g. permsort, N-queens), efficiency will be increased significantly by allowing parallel execution of the generate and the test part; hereby, the possibility of distributed backtracking is fully exploited.

Acknowledgements

We are indebted to Gerda Janssens, whose work on type- and mode-inferencing constituted the base for this research. We also want to thank S.K. Debray, W. Winsborough and anonymous referees for their comments on a draft of this paper. Veroniek Dumortier is supported by the contract RFO/AI/02 and Maurice Bruynooghe is supported by the Belgian National Fund for Scientific Research.

References

[1] Bansal, A.K., Incorporating parallelism in logic programs using program transformation, Ph.D. Thesis, Dept. of Computer Science, Case Western Reserve University, Cleveland, Ohio, July 1988.
[2] Bansal, A.K., Sterling, L., On source-to-source transformation of sequential logic programs to AND-parallelism, Proc. Int. Conf. on Parallel Processing, St. Charles, Illinois, Aug. 1987, pp. 795-802.
[3] Bruynooghe, M., Janssens, G., An instance of abstract interpretation integrating type and mode inferencing, Proc. 5th Int. Conf. and Symp. on Logic Programming, Seattle, Aug. 1988, pp. 669- 683.
[4] Bruynooghe, M., A practical framework for the abstract interpretation of logic programs, revised version of Report CW 62, Dept. of Computer Science, K.U.Leuven, 1989. (to appear in Journal of Logic Programming)
[5] Cousot, P., Cousot, R., Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, Proc. 4th ACM POPL Symp., Los Angeles, June 1977, pp. 238-252.
[6] Cunha, J. C., Ferreira, M. C., Pereira, L. M., Programming in Delta Prolog, Proc. 6th Int. Conf. on Logic Programming, Lisboa, 1989, pp. 487-502.
[7] Debray, S.K., Warren, D.S., Automatic mode inferencing for Prolog Programs, Proc. 3rd Symp. on Logic Programming, Salt Lake City, Sept. 1986, IEEE Society Press, pp.78-88
[8] Jones, N.D., Sondergaard, H., A semantics-based framework for the abstract interpretation of Prolog, in Abstract Interpretation of Declarative Languages, eds. S. Abramsky and C. Hankin, Ellis Horwood, 1987, pp. 123-142.
[9] Kanamori, T., Kawamura, T., Analysing success patterns of logic programs by abstract hybrid interpretation, ICOT technical report TR 279, 1987.
[10] Mellish, C.S., Abstract interpretation of Prolog programs, Proc. 3rd Int. Conf. on Logic Programming, London, July 1986, Springer Verlag, pp. 463-474, revised in Abstract Interpretation of Declarative Languages, eds. S. Abramsky and C. Hankin, Ellis Horwood, 1987, pp. 181-198.
[11] Pereira, L. M., Monteiro, L. F., Cunha, J. H., Aparício, J. N., Concurrency and Communication in Delta-Prolog, IEE International Specialist Seminar: the design and application of parallel digital processors, 1988, No 298, pp. 94-104.
[12] Pereira, L. M., Monteiro, L., Cunha, J., Aparício, J. N., Delta Prolog, a distributed backtracking extension with events, Proc. 3rd Int. Conf. on Logic Programming, London, July 1986, Springer-Verlag, pp. 69-83
[13] Pereira, L. M., Nasr, R., Delta-Prolog : a distributed logic programming language, Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, 1984, pp. 283-291.