A Highly Parallel Model for Object-Oriented Concurrent Constraint Programming

Richard Banach

Department of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL
UK
E-mail: banach@cs.man.ac.uk

George A. Papadopoulos†

Department of Computer Science
University of Cyprus
75 Kallipoleos Str.
Nicosia, P.O.B. 537, CY-1678
CYPRUS
E-mail: george@turing.cs.ucy.ac.cy

Abstract

Two of the currently most promising programming paradigms, namely Object-Oriented Programming and Concurrent Constraint Programming are combined into a single, highly parallel computational model based on Term Graph Rewriting Systems. In particular, we show how multi-headed Term Graph rewrite rules provide a powerful tool able to manipulate Term Graphs which themselves represent in a homogeneous way objects, concurrently executing agents and constraints. Due to the inherent fine grain parallelism of Term Graph Rewriting the proposed model is highly parallel with all activities (object communication, agent execution and constraint solving) executing concurrently.

1. Introduction

The generalised computational model of Term Graph Rewriting Systems (TGRS) ([5]) has been used extensively as an implementation vehicle for a number of, often divergent, programming paradigms ranging from the traditional functional programming ones ([12,15]) to the (concurrent) logic programming ones ([3,10,18]). Recent studies have shown that TGRS are also able to act as a means for implementing languages based on computational models such as Linear Logic ([4]) and π-calculus ([2,7]).

Two of the currently most promising programming paradigms are Object-Oriented Programming (OOP) and Concurrent Constraint Programming (CCP) ([20]), a generalisation and amalgamation of Constraint Logic Programming and Concurrent Logic Programming. The issues of object orientation and constraint solving are rather orthogonal to each other, both offering to the language environment that supports them powerful new features. Typical types of application where both formalisms are needed are, for instance, interactive graphical user interfaces, object-oriented databases, etc. Thus a number of models combining OOP and (possibly Concurrent) Constraint Programming have been proposed ([6,13,22]).

In this paper we use TGRS as the unifying framework for OOP and CCP and we show that both these programming paradigms can be represented homogeneously by Term Graphs which are manipulated by associated sets of rewrite rules. In addition to the ability of TGRS to present objects, agents and constraints in a uniform way, thus allowing the natural intermixing of all of them, the model is highly parallel and all activities involved in a computation such as constraint solving, agent execution and object composition are done concurrently.

The way to achieve all these is to allow the use of multi-headed rewrite rules in addition to the traditional approach of single-headed ones. We show then that such things as single or multiple inheritance, object encapsulation and entailing constraints can be effectively modelled by means of multi-headed rewrite rules. The rest of the paper is organised as follows. The next two sections introduce the models of Concurrent Constraint Programming and Term Graph Rewriting Systems with some

† Temporary attachment: German National Research Centre for Computer Science (GMD-FIRST), Rudower Chaussee 5, D-1199 Berlin, Germany. E-mail: hcm@prosun.first.gmd.de

0-7803-2018-2/95/$4.00 © 1995 IEEE

emphasis on the associated language Dactl which we will be using as our implementation platform. The following two sections discuss the way CCP and OOP can be modelled in a TGRS framework. The paper ends with some conclusions and a short discussion on further and related research.

2. Concurrent Constraint Programming

Concurrent Constraint Programming ([20]) is a generalisation and amalgamation of Constraint Logic Programming and Concurrent Logic Programming. In CCP computation evolves around the notion of a shared store where a number of concurrently executing agents, which themselves share variables, post constraints with respect to these variables. The store evolves monotonically in the sense that posted constraints should be consistent with each other; thus, once some information is posted to the store it cannot be retracted. Note that the agents do not communicate directly with each other even if they share common variables; instead all inter-agent communication is done via the shared store. This is achieved by means of two fundamental operations performed by the concurrently executing agents:

- `ask(c)` succeeds if `c` is entailed by the current store, fails if `c` is incosistent with the store and suspends otherwise; in the latter case some other concurrently executing agent(s) will have to add enough information to make `c` consistent or incosistent with the store so that the `ask` operation can succeed or fail.
- `tell(c)` attempts to add `c` to the current store and succeeds in doing so if `c` is consistent with the information already posted there; otherwise the operation fails.

More formally a concurrent constraint program adheres to the following grammar:

```
P   ::=   D.A
D   ::=   p(X) :: A | D.D
A   ::=   success | fail | tell(c)->A |
          E | A||A | ∃X.A | p(X)
E   ::    ask(c)->A | E+E
```

where `P`, `D`, `A`, `c` and `X` denote respectively programs, procedure declarations, agents, constraints and tuples of variables and in `p(X) :: A` we assume that `vars(A)`$\subseteq$`X`.

The behaviour of the elementary agents comprising the above syntax is as follows:

- `ask(c)->A`, checks if `c` is entailed by the current store and if it does then behaves like agent `A`, otherwise it fails or suspends accordingly.
- `ask(c1)->A1 + ask(c2)->A2`, may behave either like `A1` or `A2` depending on which ones of `c1` and `c2` are entailed or not. If both `A1` and `A2` can evolve the choice is nondeterministic.
- `tell(c)->A`, adds `c` to the current store if it is consistent with the information already posted there and then behaves like `A`; if `c` is incosistent with the store then it fails. Here we employ eventual rather than atomic publication semantics ([20]) meaning that potential inconsistencies may not be detected immediately.
- `A1||A2` behaves like `A1` and `A2` executing concurrently.
- `∃X.A` behaves like `A` with the variables `X` restricted to `A`.
- `p(X)` is simply a call to procedure `p`.

In the CCP framework as described above, the underlying constraint system is a system of partial information of the form <D,$\vdash$> where

- `D` is a set of primitive constraints.
- $\vdash$ is the entailment relation which must satisfy the following axioms:
  — u$\vdash$x if x$\in$u (reflexivity), and
  — u$\vdash$x if u$\vdash$y for all y$\in$v and v$\vdash$x (transitivity).

Thus the state of the computation in a CCP program is a multiset of primitive constraints and a multiset of agents.

3. Term Graph Rewriting and Dactl

The TGRS model of computation is based around the notion of manipulating term graphs or simply graphs. In particular, a program is composed of a set of graph rewriting rules `L->R` which specify the transformations that could be performed on those parts of a graph (redexes) which match some `LHS` of such a rule and can thus evolve to the form specified by the corresponding `RHS`. Usually ([5]), a graph `G` is represented as the tuple <N_G, $root_G$, Sym_G, $Succ_G$> where:

- N_G is the set of nodes for `G`
- $root_G$ is a special member of N_G, the root of `G`
- Sym_G is a function from N_G to the set of all function symbols
- $Succ_G$ is a function from N_G to the set of tuples N_G^*, such that if `Succ(N)` = (N_1...N_k) then `k` is the arity of `N` and N_1...N_k are the arguments of `N`.

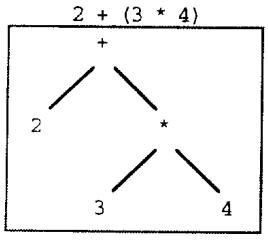
Note that the arguments of a graph node are identified by position and in fact we write `Succ(N,i)` to refer to the `i`th argument of `N` using a left-to-right

ordering. The context-free grammar for describing a graph could be something like

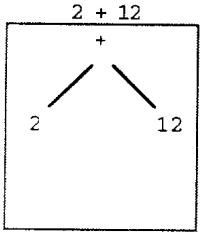
```
graph   ::= node | node+graph
node    ::= A(node,...,node) | identifier |
            identifier:A(node,...,node)
```

where A ranges over a set of function symbols and an `identifier` is simply a name for some node.

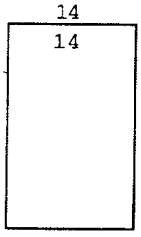
As an illustrative example of how computation in a Term Graph Rewriting System evolves we can view the evaluation of the following expression as rewriting of terms or trees, which are restricted forms of graphs:



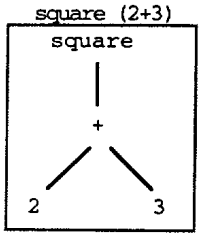
This rewrites to the following one:



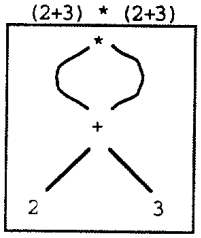
And finally to:



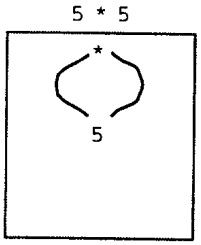
The usefulness of graph representation is revealed if we have a call to a function `square` which is evaluated lazily:



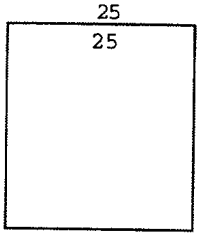
This is rewritten to:



Then to:



And finally to:



In the associated compiler target language Dactl ([9]), a graph G is represented as the tuple $\langle N_G, root_G, Sym_G, Succ_G, NMark_G, AMark_G \rangle$ where in addition to those parts of the tuple described above we also have:

- $NMark_G$ which is a function from N_G to the set of node markings $\{\varepsilon, *, \#^n\}$
- $AMark_G$ which is a function from N_G to the set of tuples of arc markings $\{\varepsilon, \hat{}\}^*$

A Dactl rule is of the form

Pattern -> Contractum, $x_1:=y_1,\ldots,x_i:=y_i$, $\mu_1 z_1 \ldots \mu_j z_j$

where after matching the Pattern of the rule with a piece of the graph representing the current state of the computation, the Contractum is used to add new pieces of graph to the existing one and the redirections $x_1:=y_1,\ldots,x_i:=y_i$ are used to redirect a number of arcs (where the arc pointing to the root of the graph being matched is usually also involved) to point to other nodes (some of which will usually be part of the new ones introduced in the Contractum); the last part of the rule $\mu_1 z_1 \ldots \mu_j z_j$ specifies the state of some nodes (idle, active or suspended).

The Contractum is also a Dactl graph where however the definitions for node identifiers that appear in the Pattern need not be repeated. So, for example, the following rule

```
r:F[x:(ANY-INT) y:(CHAR+STRING)
    v1:REWRITABLE v2:REWRITABLE]
    ->        ans:True, d1:1, d2:2,
              r:=*ans, v1:=*d1, v2:=*d2;
```

will match that part of a graph which is rooted at a (rewritable) symbol F with four descendants where the first matches anything (ANY) but an integer, the second either a character or a string and the rest overwritable symbols. Upon selection, the rule will build in the contractum the new nodes ans, d1 and d2 with patterns True, 1 and 2 respectively; finally, the redirections part of the rule will redirect the root F to ans and the sub-root nodes v1 and v2 to 1 and 2 respectively. The last two non-root redirections model effectively assignment. A number of syntactic abbreviations can be applied which lead to the following shorter presentation of the above rule

```
F[x:(ANY-INT) y:(CHAR+STRING)
  v1:REWRITABLE v2:REWRITABLE]
    => *True, v1:=*1, v2:=*2;
```

where => is used for root overwriting and node identifiers are explicitly mentioned only when the need arises. Finally, note that all root or sub-root overwritings involved in a rule reduction are done atomically. So in the above rule the root rewriting of F and the sub-root rewritings of v1 and v2 will all be performed as an atomic action.

The way computation evolves is dictated not only by the patterns specified in a rule system but also by the control markings associated with the nodes and arcs of a graph. In particular, * denotes an active node which can be rewritten and $\#^n$ denotes a node waiting for n notifications. Notifications are sent along arcs bearing the notification marking ^. Computation then proceeds by arbitrarily selecting an active node t in the execution graph and attempting to find a rule that matches at t. If such rule does not exist (as, for instance, in the case where t is a constructor) notification takes place: the active marking is removed from t and a "notification" is sent up along each ^-marked in-arc of t. When this notification arrives at its (necessarily) $\#^n$-marked source node p, the ^ mark is removed from the arc, and the n in p's $\#^n$ marking is decremented. Eventually, $\#^0$ is replaced by *, so suspended nodes wake when all their subcomputations have notified.

Now suppose the rule indeed matches at active node t. Then the RHS of that rule specifies the new markings that will be added to the graph or any old ones that will be removed. In the example above, for instance, the new nodes ans, d1 and d2 are activated. Since no rules exist for their patterns (True, 1 and 2 are "values"), when their reduction is attempted, it will cause the notification of any node bearing the # symbol and its immediate activation. This mechanism provides the basis for allowing a number of processes to be coordinated with each other during their, possibly concurrent, execution.

Thus, a possible encoding in Dactl of the previous two programs could be the following one, where some other features of the language such as module support are also being illustrated.

```
MODULE ExprEx;
IMPORTS Arithmetic;
RULE
    INITIAL => #IAdd[ 2 ^*IMul[ 3 4 ] ];
ENDMODULE ExprEx;


MODULE SquareEx;
IMPORTS Arithmetic;
SYMBOL REWRITABLE Square;
RULE
    Square[ n ] => #IMul[ ^*n n ];
    INITIAL => *Square[ IAdd[ 2 3 ] ];
ENDMODULE SquareEx;
```

The sequence of evaluation in these cases is as follows (where INITIAL denotes the first rule in a Dactl program to be rewritten); for the first program we have:

```
*INITIAL → #IAdd[ 2 ^*IMul[ 3 4 ] ]
         → #IAdd[ 2 ^*12 ]
         → *IAdd[ 2 12 ]
         → *14
```

and for the second one we also have:

```
*INITIAL → *Square[ IAdd[ 2 3 ] ]
         → #IMul[ ^*n:IAdd[ 2 3 ]  n ]
```

```
→ #IMul[ ^*n:5 n ]
→ *IMul[ n:5 n ]
→ *25
```

Note the use of the identifier n to indicate the sharing of the sub-expression `IAdd[2 3]`.

A final example illustrating some of the points discussed in this section is a nondeterministic merge program:

```
Merge[Cons[x xs] ys zs:Var] ->
    *Merge[xs ys zs1],
    zs:=*Cons[x zs1:Var] |
Merge[xs Cons[y ys] zs:Var] ->
    *Merge[xs ys zs1],
    zs:=*Cons[y zs1:Var] |
Merge[Nil ys zs:Var] -> zs:=*ys |
Merge[xs Nil zs:Var] -> zs:=*xs |
Merge[l1:Var l2:Var zs] ->
    #Merge[^xs ^ys zs];
```

Note here the use of the redirection operation `:=` to redirect all arcs pointing to some node with label `Var` to its "value" thus effectively modelling assignment. Note also the last rule which suspends until either of `Merge`'s first two arguments is instantiated to a list. Nondeterminism in the above program stems: i) from the use of the rule separator `|` indicating a nondeterministic choice in the case of more than one rule being a candidate for matching, and ii) from the last rule where `Merge` will activate again when any one of its first two arguments gets instantiated.

Further details about the language will be mentioned when we discuss particular examples; however, for detailed information on TGRS the reader is advised to consult references [5,21] whereas for Dactl appropriate references are [9,8].

3.1 Multi-Headed Rules

The fundamental difference between the framework just described and the one that will be used in the rest of the paper is the extension of the rewrite rules with multi-headed left hand sides. In particular, a rule now is of the form:

`Pattern -> Contractum,` $x_1:=y_1,\ldots,x_i:=y_i$, $\mu_1 z_1 . . \mu_j z_j$

where

`Pattern ::= node,...,node`

Multi-headed pattern matching allows powerful programming techniques. Compare the following piece of code implementing a Fibonacci function and using multi-headed rules

```
Fib[n m] -> *Last[0], *Current[1],
            *Compute[n], *Result[m];

Compute[n], Last[x], Current[y] ->
    #Compute[^*ISub[n 1]],
    #Current[^*IAdd[x y]], *Last[y];

Compute[0], Current[n], Result[m:Var] ->
    m:=*n;
```

with the following one which is written in a traditional concurrent logic style (a functional style could also be employed instead).

```
Fib[n:(0+1) m:Var] -> m:=*n;
Fib[n:INT m:Var] ->
    #Fib[^*ISub[n 1] m1:Var],
    #Fib[^*ISub[n 2] m2:Var],
    m:=##IAdd[^m1 ^m2];
Fib[n:Var m] -> #Fib[^n m];
```

As discussed in [1], in the former version the size of the proof of `Fib[n m]` is proportional in n and the number of intermediate results which need to be stored is always 2. The trick is to keep the last computed value for the next step of the computation (by passing `Current`'s argument to `Last`) and then get rid of it. In contrast note that in the traditional encoding of the Fibonacci function proofs grow exponentially with respect to n. Incidentally, we recall that + is the union operator; so in the second version of `Fib`, the first rule will be selected if the argument n is either a 0 or a 1 and the second is a variable.

An important point to note in the use of multiple heads is the fact that upon matching a set of nodes forming the LHS of some rule, the rewriting to the corresponding RHS will also cause the removal of these nodes from the graph apparatus. To be more precise, rules such as the second one in the first version of `Fib` are actually written as follows:

```
r1:Compute[n], r2:Last[x], r3:Current[y]
    -> #Compute[^*ISub[n 1]],
       #Current[^*IAdd[x y]], *Last[y],
       r1:=*GARBAGE, r2:=*GARBAGE,
       r3:=*GARBAGE;
```

Thus, the multiple TGR rules exhibit a "linear" behaviour ([1,4]) where heads forming the LHS of a matched rule denote resources which must be consumed whereas those redexes created in the RHS instead denote resources which can be created. This is the sort of behaviour we want when multiple heads are meant to be used in modelling object-oriented behaviour as will be shown in section 5. However, as we shall see when we discuss the modelling of CCP in the next section, we may want to retain these heads after a reduction has been performed by

means of a matched rule. This can be done easily and efficiently by simply including the respective head id (such as `r1`, `r2` or `r3` above) in the RHS of the rule. In general and for reasons of readability we will refrain from redirecting explicitly the heads of a matched rule to the atom GARBAGE, although it should be assumed as taking place for every such head with the exception of those whose head ids appear also in the RHS of the rule.

Finally, note that the overwhelming majority of theoretical results that have developed for TGRS ([5,21]) carry over to the case where multi-headed rewrite rules are involved in a computation. So our underlying theoretical framework requires no semantic extensions.

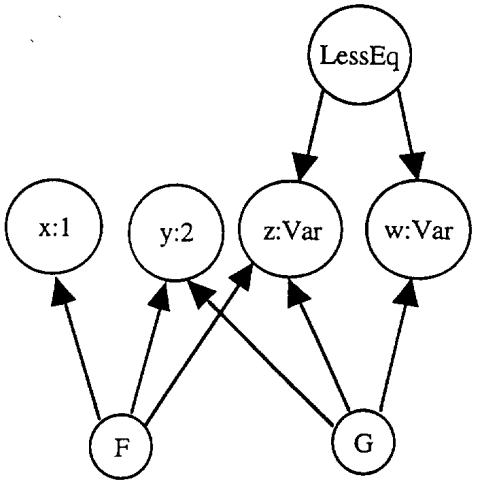
4. Concurrent Constraint Programming Using Multi-Headed TGRS

One of the advantages of mapping a computational model onto TGRS is that all objects involved in the computational model (data values, variables, agents, constraints, etc.) are represented uniformly as graph nodes shared by arcs. In addition, all operations performed by the computational model (instantiation of variables, communication and synchronisation of concurrently executing processes, etc.) are effectively modelled by modifying a representative graph structure as dictated by associated sets of rewrite rules.

The consequence of TGRS exhibiting these characteristics is that in mapping the CCP formalism onto it all activities associated with both agent behaviour and constraint solving can be represented uniformly. The main advantages of such an approach is that the interaction between the two components of a CCP program (agents and constraints), as they both evolve in the concurrent execution of a CCP program, can now be clearly seen; this is contrary to the "black box" approach that it is often adopted where the constraint solving is completely isolated and invisible to the user program. Thus: i) it is possible to examine the way constraint solving influences the execution of the user program, ii) the constraint system, not being fixed, can be tailored to the needs of the user program, iii) reasoning about the (concurrent) execution of a user program can be done more easily, iv) the resultant model increases concurrency a) between agents and constraints and b) within the agents and the constraints themselves.

The ability of TGRS to model both agents and constraints uniformly is illustrated in the following figure which shows the graphical representation of a store containing the agents `F[x y z]` and `G[y z w]` and the constraints `<x=1, y=2, z<=w>`.

Agents, constraints and arguments to them are all represented by graph nodes; relationships between them is represented by arcs connecting the appropriate nodes. Note that the direction of arcs indicates which nodes are arguments to other nodes (so, for instance, `LessEq` has as arguments `z` and `w`).



We show now how all the main activities in the execution of a CCP program as described in section 2 are modelled by graph transformations.

4.1 Entailment Relation

The entailment relation $\vdash$ can be modelled in TGRS by noting that each entailment pair `<t1,...,tn` $\vdash$ `t>` can be simulated effectively by a tell operation. In particular, since if `<t1,...,tn` $\vdash$ `t>` we have that `t1,...,tn` $\cup$ `t` is consistent, we can model the above entailment pair by means of the following TGRS rule

```
r1:t1, ..., rn:tn -> *r1, ..., *rn, *t;
```

making use of multi-headed left hand sides. Note here the repetition of the head ids `r1,...,rn` in the RHS of the rule. We recall from section 3.1 that heads in the LHS of a matched rule are actually removed from the graph upon rewriting to the RHS of the rule. Since constraints in CCP are added to the store monotonically (i.e. an added constraint can never be removed), we want to retain the matched constraints after rule reduction has been performed. This is achieved in the way shown above which in fact is an optimisation for the similar rule

```
t1, ..., tn -> *t1, ..., *tn, *t;
```

where we garbage collect `t1,...,tn` only to create them again immediately.

A more concrete example follows promptly (where $\Rightarrow$ is taken to mean "it translates to").

```
<x=1, x=y ⊢ y=1>
```

$\Rightarrow$

```
x:1, r:EQ[x y] -> *x, *r, y:=*1;
```

4.2 Asking and Nondeterministic Selection

The way the `ask` constraint and the nondeterministic selection operator + is modelled is shown below.

```
f(x,y,z) :: ask(x=1) -> g(y)
            +
            ask(y=2) -> h(z).
```

$\Rightarrow$

```
F[x:Var y z] -> #F[^x y z];
F[x:1 y z] -> *G[y];
F[x y:Var z] -> #F[x ^y z];
F[x y:2 z] -> *H[z];
```

In particular, `ask` is modelled by means of pattern matching and the nondeterministic behaviour of the + operator is captured by the parallel rule separator | which indicates that both rules for F should be tried for matching concurrently (and also by the overlapping patterns in the left hand sides of the two rules as in the original program). Note the suspension in the first and third rules if it cannot be determined whether the corresponding `ask` operations can be satisfied (in this case because the variables involved are still unistantiated).

4.3 Telling

The `tell` constraint is modelled as follows.

```
f(x,y,z) :: tell(x=1,y>5) -> g(z).
```

$\Rightarrow$

```
F[x y z] -> x:=*1, *GT[y 5], *G[z];
```

Note that more complicated constraints such as > (greater than) are translated into equivalent elementary functions which effectively simulate a large (even infinite) number of rewrite rules. Also, we recall that the above rule models eventual publication of told constraints. To model atomic publication the non trivial requirement of locking an arbitrary part of the graph would have to be satisfied.

4.4 Parallel Composition and Existential Quantification

The modelling of the last two fundamental activities of a CCP program is done as follows.

```
f(x) :: ∃y.g(x,y) || ∃z.h(x,z)
```

$\Rightarrow$

```
F[x] -> *G[x y], y:Var, *H[x z], z:Var;
```

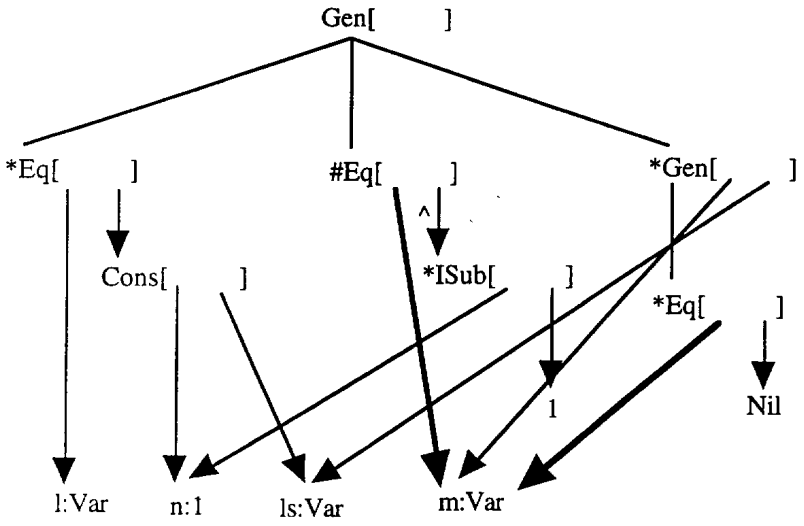
Since in TGRS there is no concept of a variable, a new variable is denoted by creating a node with the special pattern `Var` which is interpreted by all agents as representing an unistantiated variable.

As a final example we consider the following CCP fragment of a program implementing the N Queens problem.

```
queen(n,a) :: ∃l.(gen(n,l) ||
queen1(l,[],[],a,[])).

queen1(cl,ncl,l,s0,s2) ::
    ask(cl=[cf|cr]) ->
          ∃s1.(check(cf,cr,ncl,l,s0,s1) ||
            queen1(cr,[cf|ncl],l,s1,s2))
    +
    ask(cl=[],ncl=[]) -> tell(s0=[l|s2])
    +
    ask(cl=[],ncl=[_|_])) -> tell(s0=s2).

gen(n,x) ::
    ask(n=0) -> tell(x=[])
    +
    ask(n>0) ->
        ∃m xs.((tell(x=[n|xs],
                tell(m=n-1))
                -> gen(m,xs).
```

Its translation to an equivalent set of TGRS rules is shown below.

```
Queen[n a] -> *Gen[n l],
              *Queen1[l Nil Nil a Nil],
              l:Var;

Queen1[cl:Cons[cf cr] ncl l s0 s2] ->
    *Check[cf cr ncl l s0 s1],
    *Queen1[cr Cons[cf ncl] l s1 s2],
    s1:Var|
Queen1[cl:Nil ncl:Nil l s0 s2] ->
    *Eq[s0 Cons[l s2]]|
Queen1[cl:Nil ncl:Cons[ANY ANY] l s0 s2]
    -> *Eq[s0 s2]|
Queen1[cl:Var ncl:Var l s0 s2] ->
    ##Queen1[^cl ^ncl l s0 s2];

Gen[n:0 l] -> *Eq[l Nil]|
Gen[n:(INT-0) l] ->
    *Eq[l Cons[n ls]], #Eq[m ^*ISub[n 1]],
    *Gen[m ls], m:Var, ls:Var|
Gen[n:Var l] -> #Gen[^n l];
```



Incidentally, we recall that the keyword ANY will match any node and thus patterns like (INT-0) will match any integer but 0.

One of the advantages of using TGRS is that the fine grain execution of the CCP program is completely visible to the programmer (in the Dactl environment, for instance, it is possible to trace the execution of a program) and thus a number of properties about the program's behaviour can be revealed. As an example if we trace the execution of the goal Gen[1 Var] we get the graph structure shown above.

Simple lines denote evolution of agents, arrows denote sharing of data structures and the thick arrows highlight the data dependencies between agents during their evolution. By examining the graph, one is able to notice, for instance, that in order for the agent Gen (and hence the associated constraint Eq) in the right hand side of the graph to be able to evolve, its first argument (m) must first be instantiated by the other Eq constraint. Thus one can use this knowledge about the partial ordering of events to derive a better coding, either at a lower than the TGRS level or even at the TGRS one by transforming the second rule for Gen to the following one.

```
Gen[n:(INT-0) l] ->
     *Eq[l Cons[n ls]], #Eq[m ^*ISub[n 1]],
     #Gen[^m ls], m:Var, ls:Var|
```

Here note that the recursive Gen will remain inactive until its first argument is instantiated by the Eq constraint thus saving some uneccessary overhead of activation and immediate suspension.

5. Object-Oriented Programming Using Multi-Headed TGRS

The relationship between multi-headed TGRS rewrite rules of the form

```
H1,...,Hn -> B
```

and Object-Oriented Programming can be understood if one views the multiple heads H1, ..., Hn as object "slots" for method invocation. Encapsulation and hiding is then modelled by having rewrite rules which use only a subset of H1, ..., Hn. Also, inheritance is achieved by creating objects which inherit heads Hi from either a single other object (single inheritance) or many objects (multiple inheritance). The following example, the unavoidable 2-D point, illustrates the above points. Such a point could be represented by means of the following set of head patterns:

```
p:Point, mess:(Var+Cons[ANY ANY]),
        x:X_COORD[INT], y:Y_COORD[INT]
```

The first pattern plays the role of the class id (an object of type Point), the second is a stream of messages to the object being at any time either a list of such messages or an unistantiated variable, and the rest are this object's arguments (in this case an X and a Y coordinate in the form of integers). Thus a 2-D point at coordinates (1,2) is modelled by the context

```
Point, (Var+Cons[ANY ANY]),
       X_COORD[1], Y_COORD[2]
```

Now using multi-headed LHS one can define metthods for such an object as follows.

```
Point, Cons[Clear rest],
X_COORD[x:INT], Y_COORD[y:INT]
    ->
Point, rest, X_COORD[0], Y_COORD[0];

Point, Cons[Move[dx dy] rest],
X_COORD[x:INT], Y_COORD[y:INT]
    ->
Point, rest, #X_COORD[^*IAdd[dx x]],
#Y_COORD[^*IAdd[dy y]];

p:Point, message_list:Var, x, y
    ->
p, ^message_list, x, y;
```

The first rule defines a `Clear` method; note here that the old values of the X and Y coordinates dissapear implicitly due to the linear behaviour of the rule as explained in section 3.1 and are substituted by the 0 value in the RHS of the same rule. The second rule makes use of the built-in functions for arithmetic to define a `Move` method. Finally, the last rule suspends execution of the context if no messages have been sent to the object.

One could also define the following method which projects the target point on the x-axis.

```
Point, Cons[ProjectX rest], Y_COORD[y:INT]
-> Point, rest, Y_COORD[0];
```

Note that the particular method does not need to know the value of the x-coordinate. Now it is also possible to define methods for a coloured 2-D point.

```
Point, Cons[GetColour[m:Var] rest],
X_COORD[x:INT], Y_COORD[y:INT],
Colour[c:COLOUR]
    ->
Point, rest, X_COORD[x], Y_COORD[y],
Colour[c], m:=*c;
```

The above method returns in `m` (which must be a variable) the colour of the 2-D point, thus employing the back-communication technique used in concurrent logic languages. Note that all previously defined methods for any 2-D point are still applicable to the case of a coloured one. Note also that `COLOUR` is a user-defined pattern denoting the way this particular class is represented in the system.

In the same way, methods for a 3-D coloured point can be defined as follows.

```
Point, Cons[Set_3D_Black rest],
X_COORD[x:INT], Y_COORD[y:INT],
Z_COORD[z:INT], Colour[c:COLOUR]
    ->
Point, rest, X_COORD[x], Y_COORD[y],
Z_COORD[z], Colour[Black];
```

Again all methods defined for a 2-D and a 2-D coloured point are applicable to a 3-D coloured one. Thus we see that the use of multi-headed rules implements single and multiple inheritance directly whereas in other similar models such as concurrent logic languages the same functionality can be achieved by the, rather indirect, technique of delegation.

6. Conclusions

We have presented a highly parallel execution model for Object-Oriented Concurrent Constraint Programs using as a common uniform formalism the Term Graph Rewriting computational model. The approach we have used to support Object-Oriented Programming techniques complements the one we propose in [19] where we introduce records. It is also closely related to the LO model proposed in ([1]); however, our model is effectively a subset of that one and restricts computation within a single context (being similar in this sense to the model proposed in [11]). As far as Concurrent Constraint Programming is concerned, our approach is similar to the one presented in [17] where, however, graph grammars are used instead.

The advantages of using the TGRS model for Object-Oriented Concurrent Constraint Programming can be summarised as follows:

- The model is highly parallel at all levels of interaction between the concurrently executing entities (agents, objects, constraints, etc.).
- The uniform representation and handling of the activities associated with both the Object-Oriented and the Concurrent Constraint components offers a natural integration mechanism and facilitates the reasoning about the run-time behaviour of programs.
- Since TGRS languages have been implemented on parallel configurations ([14,21]) the mappings we have described in this paper effectively form an implementation apparatus.

We are currently examining ways to efficiently implement the multi-headed pattern matching and in the process we draw expertise from other models ([1,11,16]) which use similar mechanisms.

Acknowledgements

Part of this work was done while the second author was visiting GMD-FIRST as part of the ERCIM-HCM Fellowship Programme financed by the Commission of the European Community under contract no. ERBCHBGCT930350.

References

[1] J.-M. Andreoli and R. Pareschi, Linear Objects: Logical Processes with Built-in Inheritance, *ICLP'90*, Jerusalem, Israel, June 18-20, MIT Press, pp. 495-510.

[2] R. Banach, J. Balazs and G. A. Papadopoulos, Translating the Pi-Calculus Into MONSTR, submitted to the *Journal of Universal Computer Science*, 1994.

[3] R. Banach and G. A. Papadopoulos, Parallel Term Graph Rewriting and Concurrent Logic Programs, *Parallel and Distributed Processing '93*, Sofia, Bulgaria, May 4-7, Bulgarian Academy of Sciences, pp. 303-322, North Holland (to appear).

[4] R. Banach and G. A. Papadopoulos, Linear Behaviour of Term Graph Rewriting Programs, *ACM Symposium on Applied Computing '95*, Nashville, TN, USA, Feb. 26-28, ACM Computer Society Press (to appear).

[5] H. P. Barendregt, M. C. J. D. Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer and M. R. Sleep, Term Graph Rewriting, *PARLE'87*, Eindhoven, The Netherlands, June 15-19, LNCS 259, Springer Verlag, pp. 141-158.

[6] B. N. Freeman-Benson and A. Borning, Integrating Constraints with an Object-Oriented Language, *ECOOP'92*, June, LNCS 615, Springer Verlag, pp. 268-286.

[7] J. R. W. Glauert, Asynchronous Mobile Processes and Graph Rewriting, *PARLE'92*, Champs Sur Marne, Paris, June 15-18, LNCS 605, Springer Verlag, pp. 63-78.

[8] J. R. W. Glauert, K. Hammond, J. R. Kennaway and G. A. Papadopoulos, Using Dactl to Implement Declarative Languages, *CONPAR'88*, Manchester, UK, Sept. 12-16, Cambridge University Press, pp. 116-124.

[9] J. R. W. Glauert, J. R. Kennaway, and M. R. Sleep, *Final Specification of Dactl*, Internal Report SYS-C88-11, School of Information Systems, University of East Anglia, Norwich, UK, 1988.

[10] J. R. W. Glauert and G. A. Papadopoulos, A Parallel Implementation of GHC, *FGCS'88*, Tokyo, Japan, Nov. 28 - Dec. 2, Vol. 3, pp. 1051-1058.

[11] A. Guglielmi, Concurrency and Plan Generation in a Logic Programming Language with a Sequential Operator, *ICLP'94*, Santa Margherita, Italy, June 13-18, MIT Press, pp. 240-254.

[12] K. Hammond, *Parallel SML: A Functional Language and its Implementation in Dactl*, Ph.D. Thesis, School of Information Systems, University of East Anglia, Norwich, UK, published by Pitman Publishers, 1990.

[13] S. Janson and S. Haridi, Programming Paradigms of the Andorra Kernel Language, *ISLP'91*, San Diego, USA, Oct. 28 - Nov. 1, MIT Press, pp. 167-186.

[14] J. A. Keane, An Overview of the Flagship System, *Journal of Functional Programming 4(1)*, pp. 19-45, January 1994.

[15] J. R. Kennaway, Implementing Term Rewrite Languages in Dactl, *Theoretical Computer Science 72*, 1990, pp. 225-250.

[16] J. Meseguer, Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming, *ECOOP'93*, Kaiserslautern, Germany, July 26-30, LNCS 707, Springer Verlag, pp. 220-246.

[17] U. Montanari and F. Rossi, Graph Rewriting for a Partial Ordering Semantics of Concurrent Constraint Programming, *Theoretical Computer Science 109*, 1993, pp. 225-256.

[18] G. A. Papadopoulos, A Fine Grain Parallel Implementation of Parlog, *TAPSOFT'89*, Barcelona, Spain, March 13-17, LNCS 352, Springer Verlag, pp. 313-327.

[19] G. A. Papadopoulos, Object-Oriented Term Graph Rewriting, to be submitted for publication, 1995.

[20] V. A. Saraswat, *Concurrent Constraint Programming*, Ph.D. Thesis, Carnegie-Mellon University, January 1989, appeared as ACM Doctoral Dissertation Award, MIT Press series on Logic Programming, 1993.

[21] M. R. Sleep, M. J. Plasmeijer and M. C. J. D. Eekelen (eds.), *Term Graph Rewriting: Theory and Practice*, John Wiley, New York, 1993.

[22] G. Smolka, M Henz and J. Würtz, *Object-Oriented Concurrent Constraint in Oz*, Research Report RR-93-16, DFKI, Germany, April 1993.