ELSEVIER

Information and Software Technology 38 (1996) 539–547

INFORMATION AND SOFTWARE TECHNOLOGY

Concurrent object-oriented programming using term graph rewriting techniques

George A. Papadopoulos

Department of Computer Science, University of Cyprus, 75 Kallipoleos Str., Nicosia, P.O.B. 537, CY 1678, Cyprus

Received 24 December 1994; accepted 5 December 1995

Abstract

The generalized computational model of Term Graph Rewriting Systems is used as the basis for expressing concurrent object-oriented programming techniques exploiting the fine grain highly parallel features of TGRS in a language independent fashion that renders it able to act as the basis for developing specific languages based on object-orientation but also to study and compare existing approaches to the modelling of object-oriented programming techniques such as delegation, various forms of inheritance, etc.

Keywords: Concurrent object-oriented programming; Term graph rewriting systems (TGRS); Implementation techniques; Dactl

1. Introduction

The generalized computational model of Term Graph Rewriting Systems (TGRS) [1,2] has been used extensively as an implementation vehicle for a number of, often divergent, programming paradigms ranging from the traditional functional programming ones [3,4] to the (concurrent) logic programming ones [5–7]. Recent studies have shown that TGRS are also able to act as a means for implementing languages based on computational models such as Concurrent Constraint Programming [8], Linear Logic [9,10] and π-calculus [11,12].

In this paper we use TGRS and we exploit the high degree of fine grain parallelism available in the model in expressing a variety of concurrent object-oriented programming techniques. Being our framework language independent, it can serve as a basis for designing new concurrent object-oriented languages, implementing existing ones or act as a point of reference in comparing different approaches proposed by various languages to modelling certain OOP techniques such as delegation, inheritance, etc. but also to solving any associated problems encountered [13].

The rest of the paper is organized as follows. The next section introduces the model of Term Graph Rewriting Systems with some emphasis on the associated language Dactl which we will be using as our implementation platform. The following section introduces an extension to the model and shows how this extended TGRS framework can be used to model OOP techniques. The paper ends with some conclusions and a short discussion on further and related research.

2. 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 $L \rightarrow R$ 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 [1], a graph G is represented as the tuple $\langle N_G, root_G, Sym_G, Succ_G \rangle$ 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 $Succ(N) = (N_1 \cdots N_k)$ then k is the arity of N and $N_1 \cdots N_k$ are the arguments of N.

* email:george@turning.cs.ucy.ac.cy

SSDI 0950-5849(96)01093-9

Note that the arguments of a graph node are identified by position and in fact we write Succ(N,i) to refer to the ith 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.

In the associated compiler target language Dactl [14,15], 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 $\{\epsilon, *, \#^n\}$;
- $AMark_G$ which is a function from N_G to the set of tuples of arc markings $\{\epsilon, \hat{}\}^*$.

A Dactl rule is of the form:

Pattern $\rightarrow$ 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 Pattern is of the form $F[x_1 : P_1 \ldots x_n : P_n]$ where F is a symbol name, x_1 to x_n are node identifiers and P_1 to P_n are patterns. In particular a pattern P_i can be, among others, of the following forms with associated meanings:

ANY	matches anything;
INT, CHAR, STRING	with obvious meanings;
READABLE	matches a symbol name which can only be matched;
CREATABLE	matches a symbol name which can be matched and created;
REWRITABLE	matches a symbol name which can be rewritten with root overwrites;
OVERWRITABLE	matches a symbol name which can be overwritten with non-root overwrites;
$(P_1 + P_2)$	matches a symbol name which is *either* P_1 or P_2;
$(P_1 \P P_2)$	matches a symbol name which is *both* P_1 and P_2;
$(P_1 - P_2)$	matches a symbol name which is P_1 *but not* P_2.

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 : OVERWRITABLE

v2 : OVERWRITABLE]

→ 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-roots nodes d1 and d2 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:OVERWRITABLE v2:OVERWRITABLE]

⇒ *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.

The following piece of code implements a non-deterministic merge program:

```
MODULE Merge;
IMPORTS Lists;
SYMBOL REWRITABLE PUBLIC CREATABLE Merge;
SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE Var;

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[11 : Var 12 : Var zs] ⇒ #Merge[^xs ^ys zs];

ENDMODULE Merge;
```

The module starts with a declaration of all the new symbols to be used in the program and the way they are supposed to be used. So, for instance, Merge can be rewritten in this module but can only be created in some other module whereas Var, playing the role of a "variable", can be overwritten anywhere.

The first four Dactl rules implement the actual merging of the two lists. Note here the use of := to model assignment. The fifth rule models the suspension of the process if none of its first two input arguments is instantiated yet. Note here the use of two notification markings and just one suspension marking. In general, a node of the form $\#P[\,\hat{}p_1 \ldots \hat{}p_n]$ will be activated in a non-deterministic way when some p_i notifies. In our example this technique models the required non-deterministic merging of the lists. Rules separated by a | can be tested in any order whereas those separated by a ; will be tested sequentially. Non-determinism in this program is modelled by means of the | rule separator in the first two rules which have overlapping input patterns (so if both the first two arguments of Merge are instantiated to a list, one of them will be selected arbitrarily) and by the way the suspension rule is written (which will be activated when either of the two arguments get instantiated to a list).

We should also stress again the point that the nodes of a graph are labelled with symbols for which an associated access class is specified. In particular, a REWRITABLE symbol (such as Merge) can be rewritten only by means of ordinary root redirections whereas an OVERWRITABLE symbol (such as Var) can be rewritten only by means of non-root redirections; also a CREATABLE symbol can only be used as the name implies. An overwritable symbol can be "assigned" values by means of non-root overwrites as many times as it is required, and can thus play the role of either a declarative single-assignment variable or the usual imperative one.

It should be apparent by now that TGRS is a powerful *generalized* computational model able to accommodate the needs of a number of languages, often with divergent operational semantics such as lazy functional languages, "eager" concurrent logic languages or combinations of them. In addition, the implementation of TGRS themselves on a variety of (data-flow and graph rewriting) machines such as Flagship [16] has been extensively studied. Thus, TGRS can be viewed as playing the role of an interface between a variety of programming languages and computer architectures.

In this paper particular emphasis is paid on languages based on object-oriented programming. In the rest of the paper we discuss language independent concurrent object-oriented programming techniques which can be used as the basis for designing new TGRS based languages with concurrent object-oriented features or act as an implementation model and comparison framework for existing languages [17,18], especially functional [19] and concurrent logic ones [20].

3. Multi-headed rules

Every TGRS rule as defined in the previous section is in fact translated to a kernel form where the relationship between the nodes becomes explicit. For example, the kernel form of the first rule of the merge program is as follows

1 : Merge[11 12 13], 11 : Cons[111 112],
111 : ANY, 112 : ANY, 12 : ANY, 13 : Var
→ r : Merge[112 12 13], r1 : Cons[111 r12];
r12 : Var, 1 :=* r, 13 :* r1 |

In traditional TGRS it is customary to enforce the restriction that all graph nodes comprising the LHS of some rule should be accessible from the special symbol root (in our example above the root is the node 1 : Merge). Here we propose extending the TGRS framework with the lifting of this restriction in the sense that all graph nodes should be accessible from *some* root node (there can be more than one) which effectively allows the formation of multi-headed rules. In particular, a rule now is of the form:

Pattern → Contractum, $x_1 := y_1, \ldots, x_i := y_i$,

$\mu_1 z_1 \cdots \mu_j z_j$

where Pattern ::= node, ..., node

The enhancement of the model with multi-headed rules requires also extending its operational semantics. Whereas in a traditional rewrite rule:

Head[...] ⇒ NewHead[...]

Head is rewritten to NewHead and garbage collection is an implicit activity, in the case of a multiple-head rule:

$Head_1[\ldots], Head_2[\ldots], \ldots, Head_i[\ldots] \Rightarrow RHS$

the fate of the heads comprising the LHS of the rule (i.e. whether they should remain active or removed) must be specified in the RHS. This can be accomplished in our framework by allowing the symbols $Head_1$ to $Head_i$ to be OVERWRITABLE rather than REWRITABLE symbols and use the redirection operator : = to overwrite them to a special symbol not used in the program, thus effectively eliminating them. As an example the following rule:

r1 : F[...], r2 : G[...]
→ *H[...],*r1, r2 :=* GARBAGE;

creates a new potential redex H, retains F and removes G by redirecting it to the special symbol GARBAGE with an obvious meaning. Thus garbage collection becomes an active process triggered at pattern matching.

Effectively, we have introduced a "linear" behaviour to the model where the left hand side of a rewrite rule denotes resources to be consumed and the right hand side denotes resources to be produced as in the following general rule

P1 ⊗ P2 ⊗ P3 – Q1 ⊗ Q2

of Linear Logic [10] where resources P1 to P3 must be consumed in order to produce Q1 and Q2. (Note that in our example above the resource r1 has been consumed and then immediately produced; the inclusion of it in the RHS of the rule is an obvious optimization to the alternative approach of redirecting it to GARBAGE only to create again a copy of it in the RHS of the rule.) This allows our model to be used in other frameworks such as planning [21] but we do not pursue this subject here any further.

4. Object-oriented programming using multi-head 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). In particular, we can view such a set of multi-headed rewrite rules as defining some object class as follows:

Class_Name, Message_Queue, Attribute1, ...,
AttributeN → Class_Name,
Updated_Message_Queue,
Updated_Attribute1, ...,
Updated_AttributeN|
...
Class_Name, Message_Queue, Attribute1, ...,
AttributeN → Class_Name,
Updated_Message_Queue,
Updated_Attribute1, ...,
Updated_AttributeN;

where messages sent to the object invoke object's methods and we have a particular rule for each such

method. Note that the second argument is effectively a message queue where each message triggers a particular method by invoking the corresponding rewrite rule.

We use the typical 2-D point example to show a particular case of the above modelling apparatus.

```
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]];
```

The first head plays the role of class id (an object of type Point), the second is a stream of messages to the object, and the rest are this object's arguments; finally, each rule denotes a method invocation. 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 one is also able to define methods for a coloured 2-D point.

```
Point, Cons[GetColour[m] 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 the colour of the 2-D point. Note that all previously defined methods for any 2-D point are still applicable. 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 2-D coloured point are applicable to a 3-D coloured one.

4.1. Broadcast-like object communication

If desired, a more flexible object invocation and communication strategy can be used where even messages themselves are heads in a multi-headed rewrite rule. We illustrate the repercussions of such an approach by means of the following example, similar to the previous one, and modelling a drawing agent. This example serves also in showing how a not fully defined class (what is referred to as *deferred* class in Eifel or *abstract* superclass in Smalltalk) can be modelled in the enhanced TGRS framework.

```
Create_New_Figure → *Drawing,* Noshape,
    *ID[s : VAR],* Centre[point : P[0 0]];
Drawing, ID[s], Centre[point : P[x y]],
    Move[s newcoord : C[dx dy]] → Drawing, ID[s],
    *Ack[s], #Centre[^##P[^*IAdd[x dx]
    ^*IAdd[y dy]]]|
Drawing, ID[s], Noshape, Make_Square[s a]
    → Drawing, ID[s],* Ack[s],* Square,* Side[a]|
Square, ID[s], Side[a], Centre[point], Print[s]
    → *Square,* ID[s],* Side[a],* Centre[point],
    *Line[m1 : Var m2 : Var],* Line[m2 m3 : Var],
    *Line[m3 m4 : Var],* Line[m4 m1];
```

The first rule creates a drawing figure comprising a class name (Drawing), an initial shape which is not defined yet (Noshape), an id (whose purpose will be explained shortly) and an initial position on the screen (coordinates 0,0). The second rule moves the (still abstract in form) object, the third one gives it a concrete shape (a square) and the final one displays it (here we assume the presence of some other agent receiving the Line messages and displaying the sides of the square after computing their length using the rest of the information available for the object). Note that this time there is no explicit stream of method invocation messages consumed by an object. Instead, the method messages (Move, Make_Square, Print, etc.) is posted on the forum by the requesting agent and captured by the pattern matching performed in the left hand sides of the rewrite rules. So how is it possible now to send messages to a specific object? This is achieved by means of the *pointer equality* and *node sharing* mechanisms available in TGRS [1]. In particular, we associate with every object creation a unique new graph node playing the role of an object id. In order to send a message to a specific object, the message is posted to the forum (i.e. the whole graph apparatus) but it carries with it the id of the object it is meant to be consumed by. The receiver

of the message is discovered by means of the pointer equality performed in the left hand side of the corresponding rewrite rule. So the rule

```
Drawing, ID[s], Noshape, Make_Square[s a]
  → Drawing, ID[s], * Ack[s], * Square,
  *Side[a]|
```

will be selected for the benefit of that object which is sharing the graph node s with the posted message Make_Square. The benefit of this approach is that it is more flexible than the traditional stream-based approach described in the previous section since it does not require possibly complex stream manipulation operations such as merging, one-to-many specific communication patterns, etc. However, there is still need for developing some sort of send-acknowledgment protocol between the concurrently executing agents. This is achieved by means of the Ack message sent back to the forum by the object receiving a method invocation message which denotes that the requested operation has been performed.

Note that this approach is reminiscent of the Linda-type "tuple space"; in fact one can model the fundamental Linda operations as illustrated by the following elementary rules:

```
Data1[x : INT] → ...;
  r : Data2[x : INT] → ..., r :=* GARBAGE;
  LHS →* Data3[42];
  r : Data1[x : Var] → #r;
  r : Data2[x : Var] → #r;
```

which correspond to the operations rd("data1",?x), in("data2",?x) and out("data3",42) respectively i.e. read a tuple from the tuple space with an integer value and suspend if it is unavailable, remove a similar tuple and again suspend if it is unavailable and finally add to the forum a tuple.

A final point regarding this particular approach which must be addressed is what happens to the messages received (by means of pattern matching) by some object. These messages should obviously be removed from the forum. So the actual operational interpretation of a rule such as

```
Drawing, ID[s], Noshape, Make_Square[s a]
  → Drawing, ID[s], * Ack[s], * Square, * Side[a]|
```

is that after matching the rule, all the nodes comprising the left hand side are removed from the graph (i.e. consumed) and those specified in the right hand side are created. This is done by means of the usual technique involving redirection i.e.:

```
r1 : Drawing, r2 : ID[s], r3 : Noshape,
  r4 : Make_Square[s a] → *r1, * r2, * Ack[s],
  *Square, * Side[a], r3 :=* GARBAGE,
  r4 :=* GARBAGE|
```

To recapitulate, we can think of this framework as modelling a public forum (the graph structure) and a number of messages travelling and "interacting" with objects by means of rewrite rules, causing their elimination from the forum and the changing of the object's state.

4.2. Tuple-based object handling

More restrictive approaches to the issue of object representation, manipulation and invocation are also possible to be developed within our framework. In particular the rules could consist of two heads only where the first one denotes the received message and the second one is a tuple with all the information pertinent to an object. Here again sharing and pointer equality are used to denote the intended receiver object of some message as in the following bounded buffer example.

```
MODULE Buffer;
IMPORTS Messages; Objects;
SYMBOL OVERWRITABLE PUBLIC OVER-
WRITABLE Buffer;
PATTERN PUBLIC BUFFER = Buffer[ANY];

Message[object : BUFFER Put[item]],
Object[object contents : LIST limit : INT in : INT
  out : INT] → #IF[^ILt[^*ISub[in out] limit]
  then], then : #Object[object Cons[item contents]
  limit ^*IAdd[in 1] out]|
Message[object : BUFFER Get[rep_object]],
Object[object Cons[item rest] limit : INT in : INT
  out : INT] → #Object[object rest limit in
  ^*IAdd[out 1]], * Message[rep_object
  Reply[item]];
ENDMODULE Buffer;
```

The buffer object comprises five arguments being the class name, a list of items (its contents), and three integer values representing the bound, the number of items inserted in the buffer and the number of items removed

from the buffer so far. The first rewrite rule handles a put message and before servicing it checks whether the buffer is not full (in-out should be less than limit). The second rewrite rule handles a get message; here the checking on whether the buffer is empty is done implicitly by the pattern matching performed in the left hand side of the rewrite rule which with respect to the second argument of Object should match a Cons-[item rest]. Note that a message is represented by a tuple of the form Message[object_id message] and an object by a tuple of the form Object[object_id : Class_name attributes]. Note also that the class name is the pattern BUFFER whose exact format is the overwritable node Buffer with one argument of unspecified type (ANY). The reason for not using just Buffer (without an argument) for class id and for having it declared as an overwritable node (which is usually done for variables) will become apparent in the rest of this section and the following one.

To illustrate how inheritance is achieved in this more restrictive framework we give below the implementation of a BUFFER2 subclass to BUFFER which, in addition, can also serve a get2 message requesting the handling of two elements at once.

```
MODULE Buffer2;
IMPORTS Messages; Objects; Buffer;
SYMBOL OVERWRITABLE PUBLIC OVERWRI-
TABLE Buffer2;
PATTERN PUBLIC BUFFER2 = Buffer [Buffer2];

Message[object : BUFFER2 Get2[rep_object]],
Object[object Cons[item1 Cons[item2 rest]]
   limit : INT in : INT out : OUT] → *Object
   [object rest limit in out], * Message[rep_object
   Reply[item1 item2]];
ENDMODULE Buffer2;
```

To understand how inheritance has been implemented in this case, compare the (more restrictive) pattern of BUFFER2 with the (more general) one of its superclass BUFFER and note that the rewrite rules for the get and put messages can still be used in BUFFER2. In particular messages of the form:

```
Message[object : BUFFER2 Put[item]]
   Message[object : BUFFER2 Get[rep_object]]
```

will be accepted by the rewrite rules of Buffer without any modifications.

4.3. Class name as first class citizen

In all the previous approaches for modelling object-oriented behaviour we must note that the class names are in fact first class citizens. It has recently been shown [13] that this approach helps in solving the so called *inheritance anomaly* problem which arises often when inheritance is combined with concurrency [22]. In particular, in order to enforce a proper way of an object receiving messages there is a need to impose *synchronization constraints* by means of associated *synchronization code*. However, it is often the case that this synchronization code cannot be inherited by other objects without requiring extensive modifications. A typical case, which incidentally requires multiple inheritance, is the bounded buffer example just presented. Consider having a class Lockable of lockable objects in general; a lockable bounded buffer object then, which ignores get or put messages when it is locked, could be defined by multiple inheritance from bounded buffers and lockable objects where the standard approach is to add a boolean value attribute to ascertain whether the state of the object is locked or unlocked. It becomes immediately apparent, however, that although the issue of a buffer being locked is orthogonal to that of receiving get or put messages, the methods for these two messages must be redefined in order to test the value of the boolean attribute before accepting any of those messages.

A way to solve this problem along the technique proposed in [13] is to note that class names are first class citizens, whose value can in fact change. We illustrate this solution by first defining a class of lockable objects.

```
MODULE Lockable;
IMPORTS Messages; Objects;
SYMBOL OVERWRITABLE PUBLIC OVERWRI-
TABLE Lockable;
PATTERN PUBLIC LOCKABLE = Lockable[ANY];

Message[obj_class Lock], r : Object[obj_class . . .]
   → *r, obj_class :=* Locked[obj_class]|
Message[obj_class UnLock],
   r : Object[c1 : Locked[obj_class] . . .]
   → *r, c1 :=* obj_class;
ENDMODULE Lockable;
```

where . . . denotes a variable number of arguments. Here it becomes apparent why the class names are declared as overwritable nodes; a lock or unlock message causes the graph node representing the class name to be overwritten to Locked[class_name] and back to class_name respectively. Thus, a lockable bounded buffer can now be defined as follows.

```
MODULE Lockable_BB;
IMPORTS Messages; Objects; Buffer; Lockable;
```

```
SYMBOL OVERWRITABLE PUBLIC OVER-
WRITABLE Lockable_Buffer;
PATTERN PUBLIC LOCKABLE_BUFFER ≔
(Buffer[ANY] + Lockable[ANY]);
ENDMODULE Lockable_BB;
```

where we recall that + is the union pattern operator. The trick here is that if a buffer object receives a lock message, its class name changes from Buffer[ANY] to Locked[Buffer[ANY]] and the rewrite rules for get, get2 or put cannot match, thus preventing the object from accepting messages until it receives an unlock message which will change its class name back to Buffer[ANY]. Note, however, that this is achieved without having to modify at all the code for these particular methods. Note also that there are no new rules for the class Lockable_Buffer which simply inherits those of Buffer and Lockable without having to modify them in any way. All messages such as

```
Message[object : LOCKABLE_BUFFER
    Put[item]],
Message[object : LOCKABLE_BUFFER Lock]
```

can be handled by the existing rules.

5. Conclusions—related and further work

We have presented a highly parallel execution model for Object-Oriented Programming based on the Term Graph Rewriting Systems computational model. The advantages of using the TGRS based model for Object-Oriented Programming which was presented in this paper can be summarized as follows:

- The model is highly parallel at all levels of interaction between the concurrently executing entities (agents, objects, messages, etc.).
- The proposed framework is completely language independent; for instance, there is no commitment to adhering to, say, specific synchronization mechanisms; thus, it can act as a basis for both implementing a variety of concurrent object-oriented languages and comparing various approaches to object-orientation using TGRS as a common intermediate representation.
- Since TGRS languages have been implemented on parallel configurations [5,23] the mappings we have described in this paper form effectively an implementation apparatus.

There are a couple of additional areas where we believe this work is making some contribution. By mapping some other computational model onto TGRS or by showing how the latter can model the behaviour of the former one does not only provide an implementation route for the computational model in question but in addition a further appreciation is gained of TGRS' potential as a computational model and understanding and re-interpretation of the model's behaviour from other computational models' points of view. For instance, we have seen how (multiple) root overwriting can be used to model a linear behaviour needed in consuming messages to objects and also the benefits of modelling classes as OVERWRITABLE nodes and finally the viewing of the union (+) operator as a form of overloading.

The relationship between multi-headed rules and OOP has been exploited in a number of other computational models such as [24] and in fact our framework can be used as a TGRS-based implementation route for a subset of the model presented there without backtracking and splitting of context. In the TGRS framework we first introduced multi-headed rules in [8] albeit for a rather different reason, namely to show that they can be used to model the entailment relationship in concurrent constraint programming.

We are currently examining ways to implement efficiently the multi-headed pattern matching and in the process we draw expertise from other models [4,20,24] which use similar mechanisms. There are a number of problems that must be addressed; one is the issue of pattern matching since graph nodes participating in a multi-head rewriting can now be anywhere in the graph forum. Another problem is atomicity of rewriting and in particular whether upon matching a rule of the form

$$\mathrm{Head}_1[\ldots], \mathrm{Head}_2[\ldots], \ldots, \mathrm{Head}_i[\ldots] \rightarrow \mathrm{RHS}$$

all heads must be matched atomically or they can be matched asynchronously. If the latter is allowed then it is possible to collapse the multi-headed pattern matching to the more efficient single-head pattern matching. We believe that it is possible to enforce this restriction without compromising heavily the model's expressiveness. Furthermore, a number of additional restrictions may be enforced leading effectively to a model similar to MONSTR [5,11] but allowing multi-headed pattern matching and a form of interaction not only between overwritable nodes [9] but also during multi-headed pattern matching.

An interpretive approach is the most straightforward way to provide a first implementation. This would be very useful not only in further understanding the repercussions of the proposed extensions but also because (term) graphs can be used as object-oriented based intermediate forms [23] for many applications.

Reference

[1] 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,

Proc. PARLE'87, Eindhoven, The Netherlands, 15–19 June, 1987. LNCS 259, Springer-Verlag, pp. 141–158.

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

[3] K. Hammond, Parallel SML: A Functional Language and its Implementation in Dactl, PhD Thesis, School of Information Systems, University of East Anglia, Norwich, UK, Pitman Publishers, 1990.

[4] J.R. Kennaway, Implementing term rewrite languages in Dactl, Theoretical Computer Science 72 (1990) 225–250.

[5] R. Banach and G.A. Papadopoulos, Parallel term graph rewriting and concurrent logic programs, Proc. Parallel and Distributed Processing '93, Sofia, Bulgaria 4–7 May, 1993, Bulgarian Academy of Sciences, pp. 303–322.

[6] J.R.W. Glauert and G.A. Papadopoulos, A parallel implementation of GHC, Proc. FGCS'88, Tokyo, Japan, 28 Nov.–2 Dec., 1988, pp. 1051–1058.

[7] G.A. Papadopoulos, A fine grain parallel implementation of parlog, Proc. TAPSOFT'89, Barcelona, Spain, 13–17 March, 1989, LNCS 352, Springer-Verlag, pp. 313–327.

[8] R. Banach and G.A. Papadopoulos, A highly parallel model for object-oriented concurrent constraint programming, Proc. IEEE 1st ICA3PP, Brisbane, Australia, 19–21 April, 1995, IEEE Press, pp. 61–70.

[9] R. Banach and G.A. Papadopoulos, Linear behaviour of term graph rewriting programs, Proc. ACM SAC'95, Nashville, TN, USA, 26–28 Feb., 1995, ACM Press, pp. 157–163.

[10] J.-Y. Girard, Linear Logic, Theoretical Computer Science, 50 (1987) 1–102.

[11] R. Banach, J. Balazs and G.A. Papadopoulos, Translating the pi-calculus into MONSTR, Journal of Universal Computer Science, 1 (1995) 335–394.

[12] J.R.W. Glauert, Asynchronous mobile processes and graph rewriting, Proc. PARLE'92, Paris, France, 15–18 June, 1992, LNCS 605, Springer-Verlag, pp. 63–78.

[13] J. Meseguer, Solving the inheritance anomaly in concurrent object-oriented programming, Proc. ECOOP'93, Kaiserslautern, Germany, 26–30 July, 1993, LNCS 707, Springer-Verlag, pp. 220–246.

[14] J.R.W. Glauert, K. Hammond, J.R. Kennaway and G.A. Papadopoulos, Using Dactl to implement declarative languages, Proc. CONPAR'88, Manchester, UK, 12–16 Sept, 1988, Cambridge University Press, pp. 116–124.

[15] J.R.W. Glauert, J.R. Kennaway and M.R. Sleep, Dactl: an experimental graph rewriting language, Proc. GRA GRA, LNCS 532, Springer-Verlag, 1990, pp. 378–395.

[16] J.A. Keane, An overview of the flagship system, Journal of Functional Programming 4 (1994) 19–45.

[17] G. Agha, P. Wegner and A. Yonezawa (eds.), Research Directions in Object-Based Concurrency, MIT Press, Cambridge, MA, 1993.

[18] A. Yonezawa and M. Tokoro (eds.), Object-Oriented Concurrent Programming, MIT Press, Cambridge, MA, 1987.

[19] J. Sargeant, Uniting functional and object-oriented programming, Proc. 1st JSST, Kanazawa, Japan, 4–6 Nov., 1993, LNCS 742, Springer-Verlag, 1993, pp. 1–26.

[20] Y. Goldberg, W. Silverman and E.Y. Shapiro, Logic programs with inheritance, Proc. FGCS'92, Tokyo, Japan, 1–5 June, 1992, 2, pp. 951–960.

[21] A. Guglielmi, Concurrency and plan generation in a logic programming language with a sequential operator, Proc. ICLP'94, Santa Margherita, Italy, 13–18 June, 1994, MIT Press, pp. 240–254.

[22] S. Matsuoka, K. Taura and A. Yonezawa, Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages, Proc. 8th OOPSLA'93, Washington DC, USA, 26–28 Sept., 1993, ACM Press, 1993, pp. 109–126.

[23] J.F.Th. Kamperman, GEL, a Graph Exchange Language, Technical Report, CWI, Amsterdam, The Netherlands, 1994.

[24] J.-M. Andreoli and R. Pareschi, Linear objects: logical processes with built-in inheritance, Proc. ICLP'90, Jerusalem, Israel, 18–20 June, 1990, MIT Press, pp. 495–510.