A Note on Computational Meaning Attribution in Rewriting Logic

Ph. Gerner [1]

LSIIT-ICPS, CNRS
Université Louis Pasteur
Strasbourg, France

Abstract

Rewriting logic (RWL) has been proposed as a model of concurrency and a precise semantics for it has been given in the form of categories. Because of that, we have choosed to use RWL in our work on the semantics of data parallel languages. But this application reveals a problematic point w.r.t. using a category for representing states and transitions. The paper describes briefly our application and then discusses the problem of attributing a computational meaning to a model of a rewrite theory. Finally a resolution is proposed, which can benefit any application of RWL which makes a computational interpretation of it.

1 Introduction

Rewriting logic (RWL) [9] has shown to be a good framework for modeling a great variety of systems in computer science, and in particular systems with concurrency. As we are working on clarifying the semantics of data-parallel languages like HPF (High Performance Fortran [5]), we were inspired in using RWL to this end by such reports as [8].

A description of the semantics of a programming language has to be written with a consideration of the intended readership. In our case, we are interested in how one may write a *standard*, i.e., a document which describes the semantics of the language for both the programmer and the implementer of a compiler for this language, so as to be sure they have the same understanding of the language. Both are programmers, not computer science theoreticians;

[1] Email: `gerner@icps.u-strasbg.fr`

in the case of data parallel programming, programmers are not even professional programmers, but mainly engineers or scientists in the physical sciences. Therefore, the text which defines the standard has to rely more on natural language and drawings than on abstract mathematical formulations. So we do not intend to provide the programmer and the compiler implementer directly with a RWL description of the language. But we are interested in the fact that the models of rewriting logic are categories, and as such, have a natural graphical quality (this very quality which makes the efficiency of diagram drawing in category theory). Our strategy is to exploit this by letting the category expose the *control flow* of a program in our language. Thus, one can both garantee for ourselves a precise formulation of the semantics, and translate the form of the program categories into self-explanatory drawings in the text which describes the standard.

However, as we do that, we find that the computational meaning of the category is not so clear. Since we have chosen RWL for clarifying the semantics of data-parallel languages, we are interested in sheding light on this computational meaning. For this, we are drawn to distinguish between the category of the model and the category which represents the intented meaning of the specification, and explicitating the relation between the two. As the question of computational interpretation concerns all the uses of RWL in modeling dynamical systems, we propose to use this distinction as a way to deal with the problem without having to modify the now standard semantics of RWL.

The paper is structured as follows. In section 2 rewriting logic is exposed, with an emphasis on the aspects of it which are of interest to us. The following section provides the context for our discussion of the computational interpretation of RWL models: it is the RWL description of a mini data parallel language. In the 4th section, the interpretation problem is exposed, and our solution of distinguishing the model from its computational meaning is exposed. The 5th section concludes the discussion.

2 Rewriting Logic

The basics of rewriting logic are recalled here. For simplicity we treat only RWL with unconditional equations and transitions. The reader can find a treatment of the conditional case in [9].

Rewriting logic, like classical algebraic specification, relies on the model-theoretic concept of *satisfaction*: when one writes a rewrite theory, one describes its *models*. A model of a rewrite theory is a family of *categories* [6], functors, and natural transformations, which satisfies the description. The relationship between a rewrite theory and a model of it is stated in the following. Let $[\![.]\!]$ be the denotation function with respect to this chosen model.

A rewrite theory consists of:

- *Sort* declarations. A sort s is a name which denotes a category $[\![s]\!]$ in the model.
- *Operation* declarations. An operation σ is a name, and has an arity $s_1 \dots s_n, s$ of sorts. It denotes a *functor* $[\![\sigma]\!]$ from the product category $[\![s_1]\!] \times \cdots \times [\![s_n]\!]$ to category $[\![s]\!]$.
- *Equations.* An equation has form $t = t'$, with t and t' being two terms constructed from some operations and from some variable names $x_1, \dots, x_n$, and having the same sort s. Any instanciation $t(u_1, \dots, u_n)$ of term t with ground terms $u_1, \dots, u_n$ (where $u_1, \dots, u_n$ substitute for the variables $x_1, \dots, x_n$) denotes an object $[\![t(u_1, \dots, u_n)]\!]$ from category $[\![s]\!]$. That is, t denotes a functor from a product category (the domain for the substitution of the variables $x_1, \dots, x_n$) to $[\![s]\!]$. Equation $t = t'$ states the equality of functors $[\![t]\!]$ and $[\![t']\!]$, and hence the equality, for any substitution $(u_1, \dots, u_n)$, between objects $[\![t(u_1, \dots, u_n)]\!]$ and $[\![t'(u_1, \dots, u_n)]\!]$ of $[\![s]\!]$.
- *Transitions.* A transition has form $r : t \to t'$, where r is a *label*, and with t and t' being, as above, two terms of same sort, constructed from some operations and some variable names $x_1, \dots, x_n$. The transition states that there is a *natural transformation*, which is named r, from functor $[\![t]\!]$ to functor $[\![t']\!]$. This implies that there is in category $[\![s]\!]$, for any subtitution $u_1, \dots, u_n$ of the variables, an arrow $r_{(u_1,\dots,u_n)}$ from object $[\![t(u_1, \dots, u_n)]\!]$ to object $[\![t'(u_1, \dots, u_n)]\!]$.

3 Data Parallelism

Our application of RWL is in describing the semantics of HPF-like data parallel languages. This section gives a taste of this application, with the RWL description of a mini data parallel language which we call TDP (for "*T*oy *D*ata *P*arallel language"). For denoting the rewrite theories, we use the well-defined *CafeOBJ* notation [2]. (The specification below is directly executable with the CafeOBJ system [10], which has been used for testing TDP.)

3.1 TDP Programs

TDP is not a language in the usual sense. We follow the proposition of Bjarne Stroustrup, of "the syntax being the user interface of a language," as "the only really important constant [...] is the basic semantics of the language" [11, p. 208]. So for us real syntax will just be an interface for TDP, and TDP is a rewrite theory which is used (imported) by TDP programs, which are themselves rewrite theories. Thus the following CafeOBJ module is an example of a TDP program. (The module `TDP` it imports is defined later.)

```
module! PROG1 {
  using (TDP)
  ops A B C D : -> ParVarName
  eq domain A = [ 2 ] .  eq domain B = [ 2 ] .
  eq domain C = [ 2 ] .  eq domain D = [ 2 ] .
  op algo-prog1 : -> TDPAlgo
  eq algo-prog1 = (B [ 1 ] := B [ 1 ] + A [ 2 ]) ;
                  (C := A + B [ 1 ]) ;
                  (D [ 1 ] := C [ 1 ] + C [ 2 ]) .
}
```

`PROG1` is a data parallel program, in that all four variables `A`, `B`, `C`, `D` are *parallel variables*. Each has index domain `[ 2 ]`, which denotes (as seen later in the specification) the set of integers $\{1, 2\}$, so they can be seen as arrays (of size 2) in the classical sense. The second instruction of the program, `C := A + B [ 1 ]`, is a true data parallel assignment, in that it signifies that all the indices of `C` (that is, indices 1 and 2) are affected. Making this assertion precise is the motivation for writing the theory `TDP`.

The difference between this approach and usual approches to the algebraic semantics of programming language like [3] is that we do not try to describe the syntax of the language, which eases the specification of our theory `TDP`. However, if we want to define a particular syntactic interface, we could still stay in the RWL framework, thanks to the reflectivity of languages like Maude [1].

3.2 Semantics of TDP

3.2.1 TDP Syntax

The rewrite theory `TDP` is structured along the line of the classical syntax/semantics division. The following theory, which is imported by module `TDP`, is called `TDP-SYNTAX`, but in fact is just the signature whose sorts and operations are to be used by TDP programs like `PROG1`.

```
module TDP-SYNTAX {
  [ ParVarName ] [ Domain ]
     op domain_  : ParVarName  -> Domain
  protecting (INT)
     op [_]      : Int         -> Domain
  [ SingleAssignt ParAssignt ]
     op _[_]:=_[_]+_[_] : ParVarName Int  ParVarName Int  ParVarName Int
                                -> SingleAssignt
     op _:=_+_[_]      : ParVarName ParVarName ParVarName Int
                                -> ParAssignt
  [ TDPAlgo ]
  [ SingleAssignt ParAssignt < TDPAlgo ]
     op noTDPAlgo :                    -> TDPAlgo
     op _;_       : TDPAlgo TDPAlgo  -> TDPAlgo   { assoc id: noTDPAlgo }
}
```

3.2.2 The Interpretation of TDP programs

Our idea is to have a machine for executing program `PROG1`, such that the category of the machine states has a form which exhibits the control flow of the algorithm of `PROG1`. This machine is defined the following way:

```
module* MACHINE {
  [ MachineName ]
  op M : -> MachineName

  [ Inst ] [ Trace ]
  ops (toDo:_) (done:_) : Inst -> Trace
  op null      :                -> Trace
  op (_;_)     : Trace Trace    -> Trace  { assoc id: null }
  op (_||_)    : Trace Trace    -> Trace  { assoc id: null }

  [ MachineState ] [ Pgm ]
  op (_wTrace:_wAlgo:_) : MachineName Trace Pgm -> MachineState

  var inst : Inst
  trans toDo: inst => done: inst .    ** [ e ]
}
```

The last line of the above module declares the automatic transition from any `toDo` instruction to its `done` version. This transition has label `e` for "execution"; *it is the only transition in our whole specification.* It generates transitions between machine states, as the `Trace` component of a state can contain `toDo` instructions.

The translation of assignment sequences in TDP algorithms into series of arrows in the category $[\![$`MachineState`$]\!]$ of machine states is made by the following module `TDP0` which is imported by module `TDP`. It has the machine accepting single assignments as instructions: this is done with the sort ordering declaration `SingleAssignt < Inst`. There is one equation for the translation of single assignments, and one for parallel assignments. Operation "`finished`" is used by both equations to check that the trace of the machine contains only "`done`" tasks—that is, the machine has finished executing its previous tasks. (The definition of operation "`finished`" is straitforward and is not included here).

```
module* TDP0 {
  protecting (TDP-SYNTAX)
  using      (MACHINE)
  [ SingleAssignt < Inst ]
  [ TDPAlgo       < Pgm  ]

  var tr : Trace            var algo : TDPAlgo
  vars X Y Z : ParVarName   vars i j k : Int
  op distrAssignts_____ : Domain ParVarName ParVarName ParVarName Int
                                                                -> Trace
```

```
  ceq   M wTrace: tr  wAlgo: ((Z [ k ] := X [ i ] + Y [ j ]) ; algo)
    =   M wTrace: tr ; (toDo: (Z [ k ] := X [ i ] + Y [ j ])) wAlgo: algo
        if finished tr .

  ceq   M wTrace: tr wAlgo: ((Z := X + Y [ j ]) ; algo)
    =   M wTrace: tr ; (distrAssignts (domain Z) Z X Y j) wAlgo: algo
        if finished tr .

  var n : Int
  eq  distrAssignts [ 1 ] Z X Y j = toDo: (Z [ 1 ] := X [ 1 ] + Y [ j ]) .
  ceq distrAssignts [ n ] Z X Y j = toDo: (Z [ n ] := X [ n ] + Y [ j ])
                                    || distrAssignts [ n - 1 ] Z X Y j
                                    if n > 1 .
}
```

The resulting drawing of the control flow of our example program `PROG1` by the category of machine states is indicated by the following diagram:

$$\begin{tikzcd}
& z_0 \arrow[d, "\{\mathsf{a}_1\}"] & \\
& z_1 \arrow[dl, "\{\mathsf{a}_2\}"'] \arrow[dr, "\{\mathsf{a}_3\}"] \arrow[dd, "\{\mathsf{a}_2 \,||\, \mathsf{a}_3\}"] & \\
z_2 \arrow[dr, "\{\mathsf{a}_3\}"'] & & z_3 \arrow[dl, "\{\mathsf{a}_2\}"] \\
& z_4 \arrow[d, "\{\mathsf{a}_4\}"] & \\
& z_5 &
\end{tikzcd}$$

where $z_0 = [\![\texttt{M wTrace:null wAlgo:algo-prog1}]\!]$,
"a_1," "a_2," "a_3" and "a_4" are names for the following terms of sort `SingleAssignt`:

$\mathsf{a}_1 =$ `B [ 1 ] := B [ 1 ] + A [ 2 ]`,
$\mathsf{a}_2 =$ `C [ 1 ] := A [ 1 ] + B [ 1 ]`,
$\mathsf{a}_3 =$ `C [ 2 ] := A [ 2 ] + B [ 1 ]`,
$\mathsf{a}_4 =$ `D [ 1 ] := C [ 1 ] + C [ 2 ]`;

and "$\{\mathsf{a}_i\}$" as an arrow name indicates that this arrow from category $[\![\texttt{MachineState}]\!]$ is generated by the arrow $[\![\mathsf{e}]\!]_{[\![\mathsf{a}_i]\!]} : [\![\texttt{toDo:}\,\mathsf{a}_i]\!] \to [\![\texttt{done:}\,\mathsf{a}_i]\!]$ from category $[\![\texttt{Trace}]\!]$. Arrow $\{\mathsf{a}_2 \,||\, \mathsf{a}_3\}$ is generated by the $[\![\texttt{Trace}]\!]$-arrow $[\![\mathsf{e}]\!]_{[\![\mathsf{a}_2]\!]}[\![\,||\,]\!][\![\mathsf{e}]\!]_{[\![\mathsf{a}_3]\!]} : \{\texttt{toDo:}\,\mathsf{a}_2 \,||\, \texttt{toDo:}\,\mathsf{a}_3\} \to \{\texttt{done:}\,\mathsf{a}_2 \,||\, \texttt{done:}\,\mathsf{a}_3\}$.

The way this works is explained with the following explicitation of going through one of the paths from state z_0 to state z_5. The equalities come from the interpretation equations in TDP0, while the transitions come from the transition in MACHINE.

$$
\begin{array}{lll}
 & [\![\texttt{M wTrace: null wAlgo: algo-prog1}]\!] & = z_0 \\
= & [\![\texttt{M wTrace: toDo:}\,\mathsf{a}_1\ \texttt{wAlgo: (C := A + B [1]) ;}\ \mathsf{a}_4]\!] & \\
\xrightarrow{\{\mathsf{a}_1\}} & [\![\texttt{M wTrace: done:}\,\mathsf{a}_1\ \texttt{wAlgo: (C := A + B [1]) ;}\ \mathsf{a}_4]\!] & = z_1 \\
= & [\![\texttt{M wTrace: (done:}\,\mathsf{a}_1\texttt{);(toDo:}\,\mathsf{a}_2\,\texttt{||}\,\texttt{toDo:}\,\mathsf{a}_3\texttt{) wAlgo:}\,\mathsf{a}_4]\!] & \\
\xrightarrow{\{\mathsf{a}_3\}} & [\![\texttt{M wTrace: (done:}\,\mathsf{a}_1\texttt{);(toDo:}\,\mathsf{a}_2\,\texttt{||}\,\texttt{done:}\,\mathsf{a}_3\texttt{) wAlgo:}\,\mathsf{a}_4]\!] & = z_3 \\
\xrightarrow{\{\mathsf{a}_2\}} & [\![\texttt{M wTrace: (done:}\,\mathsf{a}_1\texttt{);(done:}\,\mathsf{a}_2\,\texttt{||}\,\texttt{done:}\,\mathsf{a}_3\texttt{) wAlgo:}\,\mathsf{a}_4]\!] & = z_4 \\
= & [\![\texttt{M wTrace: (done:}\,\mathsf{a}_1\texttt{);(done:}\,\mathsf{a}_2\,\texttt{||}\,\texttt{done:}\,\mathsf{a}_3\texttt{);toDo:}\,\mathsf{a}_4\ \texttt{wAlgo: noTDPAlgo}]\!] & \\
\xrightarrow{\{\mathsf{a}_4\}} & [\![\texttt{M wTrace: (done:}\,\mathsf{a}_1\texttt{);(done:}\,\mathsf{a}_2\,\texttt{||}\,\texttt{done:}\,\mathsf{a}_3\texttt{);done:}\,\mathsf{a}_4\ \texttt{wAlgo: noTDPAlgo}]\!] & = z_5
\end{array}
$$

3.2.3 The Semantics of Assignments

Until now, we have not defined the effect on states of executing an assignment. This is done by the following module TDP, which completes our specification. It uses an operation .val for observing the values of parallel variables.

```
module* TDP {
  protecting (TDP0)
  op (_.val__)  : MachineState ParVarName Int -> Int

  vars tr tr'  : Trace        var algo : TDPAlgo
  vars X Y Z W : ParVarName   vars i j k l : Int
  var s-assgnt : SingleAssignt

  eq (M wTrace: tr ; (tr' || (toDo: s-assgnt)) wAlgo: algo) .val W l
      = (M wTrace: tr ; tr' wAlgo: algo) .val W l .

  ceq (M wTrace: tr ; (tr' || (done: (Z [ k ] := X [ i ] + Y [ j ])))
        wAlgo: algo) .val W l
      =   ((M wTrace: tr wAlgo: algo) .val X i)
        + ((M wTrace: tr wAlgo: algo) .val Y j)
          if W == Z and-also l == k .

  ceq (M wTrace: tr ; (tr' || (done: (Z [ k ] := X [ i ] + Y [ j ])))
        wAlgo: algo) .val W l
      = (M wTrace: tr ; tr' wAlgo: algo) .val W l
          if not (W == Z and-also l == k) .
}
```

Each of the above equations includes the case of the non-concurrent tasks, because for any assignment assignt one has $[\![\texttt{assignt}]\!] = [\![\texttt{null}]\!][\![\texttt{||}]\!][\![\texttt{assignt}]\!]$, null being the identity for operation ||. The second equation indicates that a task is not dependent on the effect of the tasks that have been launched in parallel with it.

Notice that no state equality is declared by these equations: they only involve relationships between values. In fact, the first equation simply declares some states to be *observationally equivalent* through observation operation `.val`. In the context of hidden algebra [4] it would have been written

```
beq M wTrace: tr ; (tr' || (toDo: assgnt)) wAlgo: algo
  = M wTrace: tr ; tr' wAlgo: algo .
```

where "`beq`" declares a *behavioural equation*. A detailed commentary of the relationship of our specification approach with hidden algebra is out of the scope of this paper. One can remark however that the use of operation `.val` naturally permits concurrent access to a value from a state. Another solution to the problem of concurrent access in RWL is proposed in [8]: it does not need observation operations but has the inconvenient of leading to non-executable theories.

4 On the Computational Reading of the Models

The literature says that the role of the models is "[to make] mathematically precise the idea [...] that the models of rewriting logic theories are concurrent systems" [7]. So in the previous section we have relied entirely onto the models of RWL to give the semantics of TDP. However, the literature does not give many details on *how* the models are concurrent systems. It says that "a system can naturally be regarded as a *category*, whose objects are the system's transitions" [7, p. 16]. But while we agree with this observation, it happens that its converse, that any category can naturally be regarded as a system, is false, and an example of this is given by the models of rewriting logic. As our semantics of TDP depends on it, we seek below to expose and resolve the problem, which simply comes from the question of the computational interpretation of the composite arrows. For example, in the category for program `PROG1`, which is evoqued by the diagram in the previous section, what is the computational meaning of the arrow $\{\mathsf{a}_1\};\{\mathsf{a}_2\}$ (where ";" is composition in diagrammatic order)? We show below that finding an answer is not so easy.

4.1 Preliminaries

RWL models can be divided into two classes as regards their computational interpretation. The first class contains the models whose objects and arrows "are" the states and change of states of some system. By constrast, the models in the second class must have a computational meaning externally attributed to them. And example from this second class is the term model: the objects are equivalence classes of terms, and the arrows are equivalence classes of proof terms. As equivalence classes of terms are not intuitive "states", and equivalence classes of proof terms are hardly "computations", it is clear that the computational meaning of this model is only "in the eye of the beholder". Because models are usually exposed and discussed through the term model, we place the following discussion in the context of this second class of models. The discussion will then be extended to models of the first class.

For the needs of the discussion, we make explicit here the basics of the computational meaning attribution which are left implicit in the literature. First, the objects of the category of states are interpreted as *states* of a particular system, and any arrow between two objects is interpreted as an *action*. For simplifying the following discussion, we call Rf (for "computational reading of f") the computation meaning which is attributed to arrow f, and, similarly, Rz the state we take an object z to represent. Then the other basic rules are:

[**Preservation of source and target**] For any arrow $f : w \to w'$, action Rf starts from state Rw and leads to state Rw'.

[**Choice and single choice**] If several arrows $f_1, f_2, \dots$ start from an object w, then for an action starting from state Rw one can choose among actions $Rf_1, Rf_2, \dots$. Moreover, only one of these actions can be followed.

[**Sequence**] If actions Rf and Rf' have been chosen, with $f : w \to w'$ and $f' : w' \to w''$, then it assumed that action Rf has occurred *before* action Rf'. (For simplicity, we do not consider here the case of cycles in the underlying graph of the category).

4.2 Discussion

Given the preceding meaning attribution rules, let us now consider the following problem: if, in the category of states, one has arrows $f : w \to w'$ and $f' : w' \to w''$, then what is $R(f; f')$, the computational meaning of arrow $f; f'$?

Try 1 Our discussion uses for argumentation the category $[\![\texttt{MachineState}]\!]$ of the program `PROG1` (see the diagram in section 3.2.2). There, an instance of the question is: what is $R(\{\mathsf{a}_2\}; \{\mathsf{a}_3\})$?

The "obvious" answer would have $R(\{\mathsf{a}_2\}; \{\mathsf{a}_3\})$ be "$R\{\mathsf{a}_2\}$ and then (in a time frame) $R\{\mathsf{a}_3\}$"—with $R\{\mathsf{a}_2\}$ and $R\{\mathsf{a}_3\}$ the execution of assignments a_2 and a_3 respectively. However, due to functoriality of $[\![\,||\,]\!]$, one has the equalities

$$\{\mathsf{a}_2\}; \{\mathsf{a}_3\} = \{\mathsf{a}_2 \,||\, \mathsf{a}_3\} = \{\mathsf{a}_3\}; \{\mathsf{a}_2\}$$

from which $R(\{\mathsf{a}_2\}; \{\mathsf{a}_3\}) = R\{\mathsf{a}_2 \,||\, \mathsf{a}_3\} = R(\{\mathsf{a}_3\}; \{\mathsf{a}_2\})$ follows. As we are willing to preserve for $R\{\mathsf{a}_2 \,||\, \mathsf{a}_3\}$ the meaning "concurrent execution of assignements a_2 and a_3", the above equality states the equality of sequence and concurrency, which is clearly not satisfying for a model of concurrency. Therefore, interpretation "$R(f; f')$ is Rf followed by Rf'" does not make the job.

Try 2 If there is a clash between sequence and concurrency interpretation for composite arrows, then why not have $R(f;f')$ always be "Rf and Rf' concurrently"?

This does not work either. For example, there is a dependency—through value `B [ 1 ]`—between assignments a_1 and a_2, so action $R(\{\mathsf{a}_1\};\{\mathsf{a}_2\})$, "concurrent execution of a_1 and a_2", cannot lead to the same state Rz_2 as "$R\{\mathsf{a}_1\}$ and then $R\{\mathsf{a}_2\}$": this contradicts the "preservation of source and target" rule.

Try 3 One can also try to have action $R(f;f')$ left unspecified. Then an exception has to be made for the "parallel arrows" of the kind of $\{\mathsf{a}_2 \,||\, \mathsf{a}_3\}$, because while these arrows are composite (as examplified by the equalities in Try 1 above), we want to keep their concurrency interpretation.

We find that this solution is not satisfying either. Indeed, the "choice" rule allows the unspecified actions to be chosen: but in our application, as in most uses of RWL for specifying dynamical systems, the specified transitions are those which are *meant*. In our example, we mean—it is our semantics for TDP—going through assignment $R\mathsf{a}_1$, and then only through the other assigments; we do not mean going straight from state Rz_0 to state Rz_2 through an unknown computation which just happens to lead to the same state as doing computation $R\{\mathsf{a}_1\}$ and then $R\{\mathsf{a}_2\}$.

Try 4 If we can't leave unspecified the computational meaning of the arrows we do not know how to interpret, then why not refuse to attribute a meaning to them? That is, let R be a *partial* function with domain all non-composite arrows—*plus* all the "parallel arrows"—, and then let us declare illegal to choose in an execution path an arrow which is not in this domain. This solution breaks the interpretation of arrows rule and the choice rule, but it works. Thus, for example, arrow $\{\mathsf{a}_1\};\{\mathsf{a}_2\}$ cannot be chosen from object z_0.

4.3 Synthesis

A convenient way to recapitulate the preceding discussion is to use the well-known notion of functor.

The idea is to follow the observation cited at the beginning of the section, that a dynamical system can be seen as a category, with states as objects and transitions between states as arrows. We take the composition of arrows (computations) g and g', $g;g'$, to be the sequence (in time) of computations g and g'. Then we situate the actions Rf and states Rw of the previous discussion into such a category, let us call it **Sys**. Let **C** be the category $[\![\texttt{MachineState}]\!]$ of machine states for `PROG1`. Then R is a function from the arrows of **C** to the arrows of **Sys**, and from objects of **C** onto objects of **Sys**, with the property that, for any $f : w \to w'$ in **C**, one has $Rf : Rw \to Rw'$.

Then Try 1 above shows that R **is not a functor** because we cannot have $R(f; f') = Rf; Rf'$ (the later arrow meaning in **Sys** "Rf and Rf' in sequence"). Starting from this, Tries 2 and 3 show that it is not possible to have then $R(f; f')$ be simply distinct from $Rf; Rf'$. Try 4 concludes by abandonning altogether the idea of having R interpret every arrow in **C**.

The discussion can now be extended to the class of the RWL models which "already are" systems. If they really were systems, then their identity functor *Id* should be a computational reading functor for them. But we have shown that the computational reading cannot be a functor, so *Id* cannot be a computational reading, from which we conclude that these models cannot be systems in the sense of **Sys**.

4.4 Keeping the Categorical Models

So we cannot interpret in a simple way the models RWL as concurrent systems: a precise meaning attribution function has to be explicitated.

The usual way to code mathematically a meaning is to have the structure of the mathematical object reflect the structure of the intended meaning. If we were to follow this trend for RWL, we would have to change the models of RWL so that they better convey the intended computational meaning.

But adapting the models to each particular computational situation could lead to a proliferation of *semantics dialects* for RWL, while RWL users would probably prefer to be able to refer to a semantic *standard*. Since the present categorical models have been used for more than 10 years, they can be considered as such a standard. An important advantage of this standard, which is used in our application, is the graphical aspect of categories, as they can be discussed about through diagrams.

Therefore, it would probably be best to keep the present models unchanged, and, for each use of RWL in specifying a particular computational situation, to explicitate the attribution of a computational meaning to the models. This explicitation can be made either in "natural language", or by mathematically coding a relation between the category and another mathematical structure which is more isomorphic to the intended meaning of the specification.

5 Conclusion

We have examined the computational meaning of the models of rewriting logic. First, we have exposed our application context, by using RWL to give the semantics of a small data parallel language. In this application we arrange to have the control flow of a program exhibited by the structure of a category in the RWL models: this lead us to ask the computational meaning of the composite arrows. For addressing this question we have distinguished between the arrows and their computational meaning, the later being different from their logical meaning because two computations which have the same effect on states are still different. An example is the difference between a sequential and a concurrent execution, which is an essential point in our application. This discussion should be beneficial to any application that uses RWL to specify a dynamical system and where the nature of actions is important.

Acknowledgements
I wish to thank Pascal Schreck for our discussions on meaning attribution, and for his comments on previous versions of the manuscript.

References

[1] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. *Maude: Specification and Programming in Rewriting Logic.* SRI International, March 1999.

[2] Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. Submitted to *Theoretical Computer Science*, 2000.

[3] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, 1996.

[4] Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, August 2000. Special issue on Algebraic Engineering, edited by Chrystopher Nehaniv and Masami Ito.

[5] HPF Forum. *High Performance Fortran Language Specification*, 1.0 edition, 1993.

[6] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[7] Narciso Martí-Oliet and José Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhöfer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, volume 12 of *Applied Logic Series*, pages 1–53. Kluwer Academic Publishers, 1999.

[8] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proc. CONCUR'96*, number LNCS 1119, pages 331–372. Springer, August 1996.

[9] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[10] Ataru T. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. *CafeOBJ User's Manual –ver.1.4–.*

[11] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, January 1994.