Describing Execution Graphs with Rewriting Logic

Ph. Gerner [1]

LSIIT-ICPS
Université Louis Pasteur
Strasbourg, France

Abstract

The description of a language has to be written with a consideration for the backgrounds and interests of the readers. So, if the reader is someone who wants to write a compiler, the description should ease somehow the implementation. In this paper we propose a method for producing such an adequate description. The method uses rewriting logic [11], and the categorical nature of the models of rewriting logic is exploited. The description so obtained is executable, so that it is possible to get an intuitive understanding of it. The question of the refinement of descriptions is also discussed.

1 Introduction

It is generally admitted that being able to give the semantics of a language is important. However, the contexts where it is important vary, from verification that a program written in the language satisfies its specification, to proof that it terminates, proof that it denotes the same function as another program, or definition of the operational semantics, when it has to be implemented with a compiler.

The present article is concerned with the latter context. IOHO, the problem of writing the formal semantics of a language in the context of its implementation has not been much explored: the default method is, writing the formal semantics, and then searching for how to implement it. Exceptions exist, such as for the functional language SML [12], where a formal definition of the language and its implementation went together. But in general, the method for implementing a compiler for a real language is: borrowing the programmer's

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

reference manual, and implementing the language so that the execution of the object code does "what it is supposed to do", with respect to what was understood from the manual. And, since efficiency of the produced object code is at stake, many "liberties" have to be taken here, w.r.t. the semantics of the manual, which thus becomes the *official* semantics of the language.

The question of the liberty of the compiler with respect to the semantics exposed in the manual, is present in the context of programming with any real high-level programming language, because programmers tend to program with efficiency in mind, and consequently to base some of their design choices onto their guessing of what is done by the compiler.

This question is particularly a matter of concern with high-level parallel languages, when the possibility for the programmer, to write so-called *directives* in the program is added, as in HPF [8], or with the OpenMP model (see, e.g., [1]). These directives are indications to the compiler, which are supposed to help it to derive an efficient code. Since the distributions of compilers for these language are not clear about how the compiler incorporates these indications, tuning the program through the directives is not easy.

So, the aim in this article is to obtain a framework where it is possible to express the "amount of liberty" of the compiler w.r.t. the official semantics.

Rewriting logic [11] is used for that purpose. Besides being an adequate semantic framework for our purpose, rewriting logic allows to "run" the specifications. In this article, the well-defined notation CafeOBJ [3] is used for writing the specification. They can thus be run with the CafeOBJ system [13].

1.1 Overview of the Paper

Section 2 explains the elements of the method. Section 3 illustrates them with an example. In this section, an official semantics, and another one designed for a compiler writer, are described. Then the relationship between the two is made explicit. Section 4 compares our approach with existing ones. Finally, section 5 summarises the characteristics of the method, and explains its possible applications.

2 The Method

2.1 The Method: Overview

What is called, in this article, official semantics of a programming language, is the semantics which is exposed in a reference manual. The operational semantics used by the compiler (different from the official one) will be called an *effective* semantics —because it is closer to the final execution of the program, which is described by the object code resulting from the compilation process.

In this article, one wants to describe the official semantics and the effective semantics, and then relationship between them, using the language of category theory.

The method which is used for describing the operational meaning of a program is to describe an execution graph. This graph is described by a category, which is itself described by a rewriting logic (RWL) specification. This specification includes the pre-defined specification of a *machine*, whose function is to execute *instructions*.

Official and effective semantics will both use the same generic machine, plus a chosen machine language, whose semantics is defined through how the machine executes programs written in this language. Then defining semantics of our high-level language is just adding equations that define translation of program texts into machine language.

The translation equations look like compiling algorithms: this makes the operational semantics easy to understand for the compiler writer. And also, a compiler writer can re-use these algorithms.

2.2 Rewriting Logic Specifications

The basics of rewriting logic are recalled here. A RWL specification acts like a classical algebraic specification. When one writes a RWL specification, one describes its *models*. A model of the specification is a family of categories, functors, and natural transformations, which satisfies the description. The relationship between a specification, and a model of it, is stated in the following. A specification 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 $s_1 \times \cdots \times s_n \to s$ 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 with operations and variable names, and having the same sort. A term t of sort s denotes an object $[\![t]\!]$ from category $[\![s]\!]$. The equation states that $[\![t]\!] = [\![t']\!]$.
- Transitions. A transition has form $t \to t'$, with t and t' being two terms of same sort. The transition means that there is an arrow in category $[\![s]\!]$, from object $[\![t]\!]$ to object $[\![t']\!]$.

2.2.1 The Initial Model

In this article, one will use *initial models* of a specification. This will make the comparison between different models easier. An initial model of the specification is a model in which the only categories, functors, and natural transformations considered are those declared in the specification (with, respectively, sorts, operations, and transitions). Also, each category is inhabited only by objects which are the interpretation of at least one term, and with arrows which come from the transitions declared in the specification. In the rest of the article, when one speak about "the" category for a specification, it will mean that one considers any model among the initial models of the specification.

2.3 The Specification of the Machine

As said above, the method consists in using the specification of a machine, which is given below. The specification is written with the CafeOBJ notation.

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

  [ Inst ] [ Trace ]
  ops (toDo:_) (done:_) : Inst  -> Trace
  var inst : Inst
  trans toDo: inst => done: inst .    ** [ execution ]

  op null     :              -> Trace
  op (_;_)    : Trace Trace -> Trace  { assoc id: null }
  op (_||_)   : Trace Trace -> Trace  { assoc comm id: null }

  [ MachineState ] [ MPgm ] [ Pgm ]
  op (_withTrace:_MpgmdWith:_pgmdWith:_) :
      MachineName Trace MPgm Pgm          -> MachineState
}
```

The above specification is now explained, based on the explanation of the sorts which are declared.

- `MachineName` is for naming the machine, with an operation declaration `op M : -> MachineName`. From a purely technical point of view, naming the machine is not necessary, since only one machine is used. However, doing without a name for the machine would have made the semantics descriptions difficult to read.
- `Inst` is for instructions, written in machine language. This language can be chosen, from compiler intermediate code to languages closer to the considered machine.
- `Trace` is for the traces of instructions which are `toDo` or `done`. The transition

`toDo: inst => done: inst` says that when an instruction is `toDo`, it can be executed by the machine, thus becoming `done`.

A trace is a list, built with operation `_;_`. Its elements are either instruction, or sets of parallel instructions; A set of parallel instructions is built with operation `_||_`. The intended meaning is that these instructions can be executed concurrently. An example of a trace is:
`done:` $inst_1$ `; (done:` $inst_2$ `|| done:` $inst_3$`) ; toDo:` $inst_4$
where each $inst_i$ is an instruction.

- `Pgm` is for the programs written in the chosen high-level language.
- `MPgm` is for the programs written in the chosen machine language.
- `MachineState` is for the states of the machine. A machine state is described by a program to be executed, a machine program, and a trace. This is said with operation

```
_withTrace:_MpgmdWith:_pgmdWith:_ :
          MachineName Trace MPgm Pgm -> MachineState .
```

  A transition of an instruction in the trace becomes a transition of machine state. Several such transitions can occur concurrently: the corresponding transition from a machine state to another represents then a concurrent transition.

The preceeding specification describes categories of machine states. The initial models are chosen for our method, because they do not identify two machine states which are denoted by two different terms: these models are used below for safely comparing different semantics.

2.4 Machine Language and Translation Equations

The method for defining the operational semantics for a language consists in augmenting the machine with a language for it; and then defining the translation from the high-level language to this machine language: this automatically defines an operational semantics for the high-level language. An example of a machine language definition is given in section 3 below. The translation is expressed with equations (instead of transitions) between machine state terms, so that the only arrows in the category of machine states (which is called **MS** below) are execution arrows.

2.5 Execution Graphs

The result of adding a translation equation to the machine specification is: identifying some states of category **MS**. The execution graph for a program is then extracted from this category.

So this can be done, this execution graph is taken from a set of execution graphs, which is built from a subcategory of **MS**, which is called **subMS**.

This subcategory is constituted from all objects and arrows found on the way, when, starting from an *initial state*, one follows the arrows of the category. An initial state is any object denoted by a machine state term with a null trace.

The set of execution graphs is the set of components of the graph which is now extracted from **subMS**.

This graph is obtained from **subMS** by forgetting all composite arrows and identities, except *parallel arrows*. (This is a variation on the well-known operation of forgetting composite arrows and identities which is described in [9, pp. 48–49].) Here parallel arrows are defined. An arrow is **subMS** is a parallel arrow, if and only if it is an arrow from a state $[\![t]\!]$ to a state $[\![t']\!]$, and there exist at least two instructions $inst_1$ and $inst_2$ such that t contains subterms $\texttt{toDo:}\ inst_1$ and $\texttt{toDo:}\ inst_2$, and t' contains subterms $\texttt{done:}\ inst_1$ and $\texttt{done:}\ inst_2$. An example of a parallel arrow is:

$$\begin{array}{c}
[\![\texttt{M withTrace:}\ tr\ \texttt{; (toDo:}\ inst_1\ \texttt{||}\ \texttt{toDo:}\ inst_2\texttt{)}\ \texttt{MpgmdWith:}\ mp\ \texttt{pgmdWith:}\ p]\!] \\
\Big\downarrow \\
[\![\texttt{M withTrace:}\ tr\ \texttt{; (done:}\ inst_1\ \texttt{||}\ \texttt{done:}\ inst_2\texttt{)}\ \texttt{MpgmdWith:}\ mp\ \texttt{pgmdWith:}\ p]\!]
\end{array}$$

In the following, the function from categories to graphs which strips away the non-parallel composite arrows is called *stripArrows*.

2.6 Comparing the Semantics

The operational semantics of a program is its execution graph. Two operational semantics for the same program will be compared as a comparison of two graphs G and G', and this is done, in our method, via a comparison of the categories which represent these graphs.

So, if $G = \mathit{stripArrows}(\mathbf{subMS})$ and $G' = \mathit{stripArrows}(\mathbf{subMS}')$ are to be compared, this is done by showing a relation between **subMS** and $\mathbf{subMS}'$. In **subMS** and $\mathbf{subMS}'$ one has the composite arrows: since these correspond to paths in the graph, the relationship between the two graphs can be exposed easily. The relation between **subMS** and $\mathbf{subMS}'$ is a *functor* from **subMS** to $\mathbf{subMS}'$: this kind of relation is rich enough to express *refinement* of semantics. An example of a refinement according to this sense is adding intermediate steps in an execution path. Another example is adding alternative paths in the execution graph. This latter example will be illustrated in the following example section.

3 Example

In this section, our method is illustrated with a simple language, which we call "ATL" (*A Toy Language*). First, a machine language is defined. Then the syntax of ATL is given, and then its official semantics is defined. A running test gives some feedback for this definition. Then, an effective semantics for ATL is given. Finally, official and effective semantics are compared.

3.1 Choosing The Machine Language

The language of the machine can be chosen, high or medium or low-level, depending on one's needs, as with intermediate languages in a compiler. In the present example, the language of the machine is chosen to be medium-level, so as to shorten the presentation. It is called TMIR (*T*oy *M*edium *I*ntermediate *L*anguage). It is defined in module `MACHINE-WITH-TMIR`, which is shown in its entirety in appendix 6. Below the main points are explained.

First, the syntax of TMIR is defined, with operations, as BNF-like rules written from right to left:

```
[ Inst < ParInst ]
  protecting (STRING)
  op (_<-_+_) : String String String      -> Inst
  op (_||_)   : Inst Inst                 -> ParInst
  op (_||_)   : Inst ParInst              -> ParInst
  op noMPgm   :                           -> MPgm
  op (_;_)    : ParInst MPgm              -> MPgm
```

It is a language with only one instruction, namely addition plus assignment. Sort `String` is for variable names (module `STRING` is pre-defined in CafeOBJ). `Inst < ParInst` means that a single instruction is considered as a parallel instruction with only one instruction.

As can be guessed from its syntax, TMIR allows building sets of parallel assignment instructions; and a TMIR program is a sequence of single or parallel assignment instructions. This semantics is defined in module `MACHINE-WITH-TMIR` (whose code is given in the appendix section 6). This module is twofold. It includes:

- an equation that says how TMIR programs are executed by the machine: the next TMIR assignment instruction, or set of parallel instructions, to be executed, is given as `toDo` to the machine, when preceeding instructions are all `done`. This expresses the sequential execution of the sequence in the TMIR program.
- Some equations that define the effect on the values of variables, of executing an assignment instruction.

3.2 The Syntax of a Little Language

The syntax of ATL is now defined:

```
module! ATL-SYNTAX {
  [ ATLPgm ]
  op noATLPgm :                          -> ATLPgm
  op _;_     : ATLPgm ATLPgm             -> ATLPgm
               { assoc id: noATLPgm }
  protecting (STRING)
  op _:=_+_ : String String String      -> ATLPgm
}
```

Sort ATLPgm is for ATL commands and sequences of commands.

3.3 Official Semantics

As can be guessed from its syntax, ATL is an imperative language, with sequences of assignments. The official operational semantics of ATL is: assignments are executed one after the other, in the order shown by the program text. This is described by the following specification, which defines the translation from ATL programs to TMIR programs, whose semantics was given above.

```
module* ATL-OFFICIAL-SEM {
  using (MACHINE-WITH-TMIR)
  protecting (ATL-SYNTAX)
  [ ATLPgm < Pgm ]   ** ATL is accepted as a programming language
  var tr : Trace  var mp : MPgm var p : ATLPgm vars x y z : String

  eq  M withTrace: tr MpgmdWith: mp pgmdWith: ( ( z := x + y ) ; p)
    = M withTrace: tr MpgmdWith: (addAtEnd mp ( z <- x + y ))
          pgmdWith: p .
}
```

The above equation states that the next assignment instruction in the ATL program which is to be to be translated, is translated into its TMIR version, and this new TMIR instruction is appended to the end of the current TMIR program mp, with operation AddAtEnd. From the operational semantics of TMIR shown above, this means that this assignment instruction will be executed just after mp has been executed.

3.4 Testing the Description

In this section an example of an ATL program is given. This program is tested, using the above definition of the semantics of ATL.

The above specification is executable in the CafeOBJ system, because the translation equation of the official semantics and the equation in the module MACHINE-WITH-TMIR which states how TMIR programs are executed can both be viewed as rewrite rules.

Below are written the CafeOBJ commands which define and test a short ATL program `prog1`, in the context of the official semantics of ATL. The values of variables `a` and `b` are set in the initial state. Then, the value of variable `f` after execution of `prog1` is looked at with operation `.val`. For this, the CafeOBJ system is asked, with command `exec`, to reduce the term
`(M withTrace: null MpgmdWith: noMPgm pgmdWith: prog1) .val "f"`
using equations plus transitions.

```
open ATL-OFFICIAL-SEM
op prog1 : -> ATLPgm .
eq prog1 =  ("c" := "a" + "b") ; ("d" := "a" + "a") ;
            ("e" := "c" + "d") ; ("f" := "a" + "e") .
var p : ATLPgm .  var mp : MPgm .
eq (M withTrace: null MpgmdWith: mp pgmdWith: p) .val "a" = 1 .
eq (M withTrace: null MpgmdWith: mp pgmdWith: p) .val "b" = 2 .
exec (M withTrace: null MpgmdWith: noMPgm pgmdWith: prog1) .val "f" .
** should be: 6
close
```

Feeding the CafeOBJ system with the preceeding text yields the following result:

```
processing input : ./test-official-sem.mod
-- opening module ATL-OFFICIAL-SEM.. done.__*
-- execute in %ATL-OFFICIAL-SEM : (M withTrace: null MpgmdWith: noMPgm
    pgmdWith: prog1) .val "f"
6 : NzNat
(0.000 sec for parse, 85 rewrites(0.120 sec), 347 matches)
CafeOBJ>
```

3.5 An Effective Semantics

In this section an effective semantics for ATL is given—that is, a semantics which is closer to what a compiler would do. The difference between this effective semantics and the official semantics is that an optimization is introduced: when two successive assignment instructions (in the program) are found to be independent, they can be executed concurrently. An example of such successive independent instructions is the first two instructions of program `prog1` in the preceeding test. This effective semantics is described by the following specification:

```
module* ATL-OPT-SEM {
  using (MACHINE-WITH-TMIR)
  protecting (ATL-SYNTAX)
```

```
[ ATLPgm < Pgm ]   ** ATL is accepted as a programming language

vars x x' y y' z z' w : String
var tr : Trace  var mp : MPgm var p : ATLPgm

** sequence of two independent assignment instructions:
ceq M withTrace: tr MpgmdWith: mp
      pgmdWith: ((z := x + y) ; (z' := x' + y') ; p)
    = M withTrace: tr
        MpgmdWith: addAtEnd mp ((z <- x + y) || (z' <- x' + y'))
        pgmdWith: p
    if not
   ((z' == z) or (x' == z) or (y' == z) or (x == z') or (y == z')) .
```

This equation states that when the next two assignment instructions in the ATL program are independent, a parallel TMIR instruction is constructed from them and is appended to the end of the current TMIR program `mp`. This parallel instruction will be executed just after the TMIR program `mp` has been executed. The operational semantics of TMIR says that the two assignments of a parallel instruction $a_1 \,||\, a_2$ can be executed either concurrently, or first a_1 and then a_2, or else first a_2 and then a_1. The compiler, or the operating system of the machine, will choose among those three possibilities.

There are two more equations which define the effective semantics: they are shown in the appendix section 6. The first equation states that if the two ATL instructions are not independent, the first one is translated into its TMIR version, and this new TMIR instruction is appended to the end of `mp`. The second equation treats the case where the ATL program to be translated contains only one assignment instruction: it states that this last instruction is translated into its TMIR version, and this new TMIR instruction is appended to the end of `mp`.

3.6 Comparing the Semantics

Our method allows an easy comparison of the official and effective semantics. In this section one shows that official and effective semantics of an ATL program define the same input-output relation on variable values.

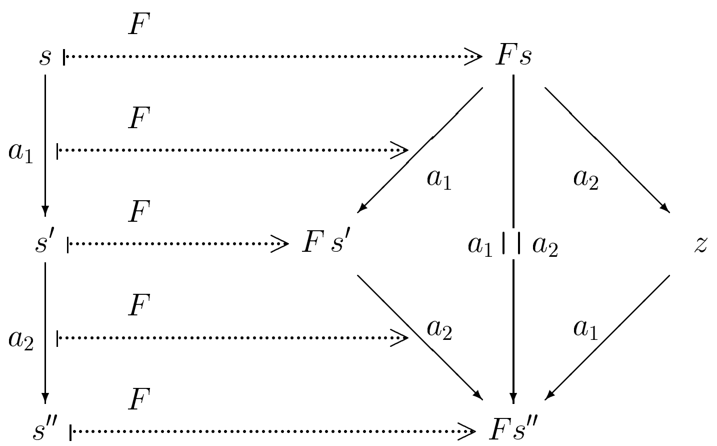
This is done by showing that the official execution graph is a subgraph of the effective execution graph, with same initial and final states.

As explained in sections 2.5 and 2.6, this is realized by defining a functor F from **subO** to **subE**, where **subO** and **subE** are the subcategories of categories of machine states **O** and **E**, of the official and effective semantics, respectively.

Now, let us define functor F. Let us note $[\![.]\!]_{\mathbf{O}}$ and $[\![.]\!]_{\mathbf{E}}$, respectively, for the functions which sends machine states terms into **subO** and **subE**.

First, any root r in **subO**, where
$r = [\![$`M withTrace: null MpgmdWith:` mp `pgmdWith:` $p]\!]_{\mathbf{O}}$,
where p is an ATL program and mp is a TMIR program, is sent onto
$Fr = [\![$`M withTrace: null MpgmdWith:` mp `pgmdWith:` $p]\!]_{\mathbf{E}}$.
From the root, objects and arrows from `subO` are sent into `subE`, with the following process: let s be the current considered object in `subO` (set to r at the beginning of the process). Then two cases may occur:

- Fs has only one arrow starting from it, $f : Fs \to w$. Then there is an arrow $e : s \to s'$ in `subO`. Then one sets $Fs' = w$, and $Fe = f$. And s' is the new current considered object.
- Fs has three arrows starting from it. This is the case where two successive independent assignment instructions a'_1 and a'_2 are found in the ATL program text, leading to the TMIR instruction $a_1 \,||\, a_2$, where a_i is the TMIR translation if a'_i. The behaviour of F in this case is shown by the correspondence between the following two diagrams, the left one in `subO`, and the right one, commutative, in **subE**:



where what is indicated on each arrow is the instruction whose execution is represented by the arrow.
And s'' becomes the new current considered object in the process of construction of F.

4 Related Works

The use of algebra in specifying the compiling process has been advocated by Teodor Rus [14], who proposes an infrastructure for doing that, but with a focus on code transformation, not on the operational semantics of the code, while our method emphasizes the latter aspect. In the present section some other attempts at modeling the operational semantics of languages in an algebraic manner are mentioned.

4.1 Abstract State Machines

The *Abstract State Machines* (ASMs) [6] formalism has been designed especially for describing algebraically the operational semantics of programming languages, in the form of a graph of states. For example, a semantics could be given of the C programming language [7]. ASMs allow description of step-by-step progression in the execution, and of concurrency. The difference with our method is that in ASMs, comparison of semantics, as a comparison of graphs, is not easy, because execution graphs are defined directly, not through categories, while paths, which are not first-class citizens in graphs, are important in the comparison (see, e.g., [5]). In categories, paths are composite arrows, so, as arrows, they are first-class citizens.

4.2 RWL and the Semantics of Programming Languages

[10] describes how rewriting logic can be used to give the semantics of programming languages, with a mini-ML as an example. Use of rewriting modulo equations for the same purpose is exposed in [15], for describing the operational semantics of the π-calculus.

Our specification of the machine looks like an object-oriented one, with a class of machine states, and only one attribute for that class, namely, `.val`. The rewriting logic system Maude [2] allows writing directly object-oriented specifications with RWL, but with attributes coded "in hard", as part of the syntactic term. We prefered accessing values via an operation. This allows, in particular, an easy treatment of comparison of execution graphs, independent of values. Moreover, since values are not put into the syntactic term which describes the machine state, the term is kept small, which is good when we are interested in the description of data-parallel languages like HPF, which are designed to treat big amounts of data.

4.3 Order-Sorted Algebra and The Semantics of Programming Languages

J. Goguen and G. Malcolm used order-sorted algebra specifications (that is, specifications without transitions) for describing the semantics of an imperative language [4]. The operational semantics of the language is given by interpreting the equations as rewrite rules. The problem of this approach is that no distinction is made between the compilation/interpretation rules, and the execution rules.

5 Conclusion

A method for describing different operational semantics for a programming language, and for comparing them, has been presented. An operational semantics is given by an execution graph, which is described with a rewriting logic specification. The first principal characteristic of the method is that comparison of different operational semantics is not made at the level of the execution graph, but at the level of categories, whose structures are adapted for this kind of comparison. The second principal characteristic is the explicitation of compilation mechanisms in the description, which ease both the comparison of semantics, and the understanding of the description in the context of writing a compiler. Another interest of the method, which comes through the use of algebraic specifications, is clean structuration of the description into several modules.

Our method could be used to structure and modelize the compilation process, whose nature is to transform operational semantics, from the semantics for the programmer (the official semantics), to machine-level operational semantics. An important special application will be expressing the meaning of compiling directives, in the context of high-level parallel programming languages. Directives are a solution in such languages, to get an efficient compiled code, but this works only if programmer and compiler agree on the meaning of directives.

Another possible application is in the modeling of the interplay between static and dynamic compilations, by the addition of compilation transitions in the method.

References

[1] *OpenMP Fortran Application Program Interface*, November 2000. Version 2.0.

[2] Manual 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.

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

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

[5] G. Goos and W. Zimmermann. Verifying compilers and ASMs —or: ASMs for uniform description of multistep transformations. In Ph.W. Gurevich, Y. Kutter, M. Odersky, and L. Thiele, editors, *Proceedings of ASM'2000 Workshop: Abstract State Machines, Theory and Applications*, volume LNCS 1912, pages 177–202. Springer, 2000.

[6] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide*, pages 9–36. Oxford University Press, 1995.

[7] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In *Selected papers from CSL'92 (Computer Science Logic)*, volume LNCS 702, pages 274–308. Springer Verlag, 1992.

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

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

[10] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, 1993. To appear in *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

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

[12] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

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

[14] Teodor Rus. Algebraic processing of programming languages. *Theoretical Computer Science*, 199(1–2):105–143, 1998.

[15] Patrick Viry. A rewriting implementation of pi-calculus. Technical Report R-96-30, March 1996.

6 Appendix: CafeOBJ Code

.

The intermediate language TMIR is defined by the following CafeOBJ module:

```
module* MACHINE-WITH-TMIR {
  using (MACHINE)

** Syntax of TMIR:
  [ Inst < ParInst ]
  protecting (STRING)
  op (_<-_+_) : String String String      -> Inst
  op (_||_)   : Inst Inst                 -> ParInst
  op (_||_)   : Inst ParInst              -> ParInst
  op noMPgm   :                           -> MPgm
  op (_;_)    : ParInst MPgm              -> MPgm

** Semantics of TMIR:
  vars tr tr' : Trace   vars mp mp' : MPgm  var p : Pgm
  vars pinst pinst' : ParInst   vars inst inst' : Inst    vars x y z w : String
```

```
  op finished_  : Trace -> Bool
  op prepare_   : ParInst -> Trace
  ceq M withTrace: tr MpgmdWith: (pinst ; mp) pgmdWith: p
    = M withTrace: (tr ; (prepare pinst)) MpgmdWith: mp pgmdWith: p
          if finished tr .
                ** execution of TMIR programs by the machine

 ** the effect onto the values of variables, of the machine executing an assignment:
  protecting (INT)
  op (_.val_) : MachineState String -> Int
  ceq (M withTrace: (tr ; (done: ( z <- x + y ))) MpgmdWith: mp pgmdWith: p) .val w
      =   ((M withTrace: tr MpgmdWith: mp pgmdWith: p) .val x)
        + ((M withTrace: tr MpgmdWith: mp pgmdWith: p) .val y)
           if w == z .

  ceq (M withTrace: (tr ; (done: (z <- x + y))) MpgmdWith: mp pgmdWith: p) .val w
      = (M withTrace: tr MpgmdWith: mp pgmdWith: p) .val w if not (w == z) .

  eq (M withTrace: (tr ; (toDo: inst)) MpgmdWith: mp pgmdWith: p) .val w
      = (M withTrace: tr MpgmdWith: mp pgmdWith: p) .val w .

  ** parallel instructions: artificially sequentialized:
  eq (M withTrace: tr ; ((toDo: inst) || tr') MpgmdWith: mp pgmdWith: p) .val w
      = (M withTrace: tr ; tr' MpgmdWith: mp pgmdWith: p) .val w .
  eq (M withTrace: tr ; ((done: inst) || tr') MpgmdWith: mp pgmdWith: p) .val w
      = (M withTrace: tr ; (done: inst) ; tr' MpgmdWith: mp pgmdWith: p) .val w .

** definition of the auxiliary operations:
  eq finished null = true .
  eq finished (tr ; (done: inst)) = (finished tr) .
  eq finished (tr ; (toDo: inst)) = false .
  eq finished (tr ; ((toDo: inst) || tr' )) = finished (tr ; (toDo: inst) ; tr') .
  eq finished (tr ; ((done: inst) || tr' )) = finished (tr ; (done: inst) ; tr') .

  eq prepare inst = toDo: inst .
  eq prepare (inst || inst') = (toDo: inst) || (toDo: inst') .
  eq prepare (inst || pinst) = (toDo: inst) || (prepare pinst) .

** used by translation equations:
  op addAtEnd__ : MPgm Inst -> MPgm
  op addAtEnd__ : MPgm ParInst -> MPgm
  eq addAtEnd noMPgm pinst = pinst ; noMPgm .
  eq addAtEnd (pinst' ; mp) pinst = pinst' ; (addAtEnd mp pinst) .
}
```

The last two equations of module `ATL-OPT-SEM`, which describes the effective semantics of ATL, are:

```
** sequence of two non-independent assignment instructions:
ceq M withTrace: tr MpgmdWith: mp pgmdWith: ((z := x + y) ; (z' := x' + y') ; p)
  = M withTrace: tr MpgmdWith: addAtEnd mp (z <- x + y)
      pgmdWith: ((z' := x' + y') ; p)
    if (z' == z) or (x' == z) or (y' == z) or (x == z') or (y == z') .

** when only one assignment instruction is left:
eq M withTrace: tr MpgmdWith: mp pgmdWith: (z := x + y)
  = M withTrace: tr MpgmdWith: addAtEnd mp (z <- x + y) pgmdWith: noATLPgm .
```