An execution environment for language formal definitions

A. Fantechi, S. Gnesi, P. Inverardi

I.E.I. - C.N.R. - Pisa

Abstract

In this work we describe an enviroment for the execution of the formal definitions of concurrent languages. The proposal applies to formal definitions given using the SMoLCS method; the execution environment is obtained by expressing in Prolog the main parts in which the formal definition is structured, namely the denotational clauses and the concurrent algebra. The advantages of the logic programming approach are discussed especially with respect to novel techniques like metaprogramming which allow to provide a wide range of execution strategies.

The method is applied on a simple concurrent language showing the use of the available environment functionalities.

1. Introduction

In this paper we describe the guidelines we have followed in the realization of an environment for the execution of programming language formal definitions. The work has been partly developed inside the project "The Draft Formal Definition of ANSI-MIL-STD-1815A Ada®", whose purpose is the production of a draft formal definition of full Ada together with the development of some tools for its use. In this framework an environment for the execution of the Ada formal definition is being studied. The proposal is based on the use of a SMoLCS based abstract operational model [AB86, AGMRZ86] for giving the definition of the language.

The basic idea of the SMoLCS methodology is that a concurrent system is modelled as a labelled transition system, the state of which consists of a set of states (one for each component process) and of some global information, which corresponds to the storage shared among the processes. Transitions of a state of the concurrent system are inferred from the transitions of the component subsystems in three passes: 1) synchronization, i.e. given the transitions of single components, the synchronized transitions of groups of components and their global effect on the global information are defined; 2) parallelism, i.e. a set of synchronized transitions are composed in parallel allowing disjoint groups of components to evolve concurrently; 3) monitoring, i.e. some global scheduling strategy is imposed on the system.

This work is partially funded by the Commission of the European Communities under the Multi-Annual Programme in the field of Data-Processing, project No. 782. "The Draft Formal Definition of ANSI/MIL-STD 1815A Ada".

A two step approach has been followed to give the dynamic semantics part of the Ada Formal Definition and can in general be followed to give the dynamic semantics of concurrent languages [AR86]. In the first step it is used an applicative style typically denotational which associates to each well defined language construct a structure (a term on a signature), whose value is defined by an appropriate semantic algebra, the concurrent algebra, in the second step. In the concurrent algebra a specification of a SMoLCS concurrent system, i. e. an abstract data type representing a transition system, is given. The sort of interest of this algebra is referred to as behaviours, whose term are similar to the behaviours expressions of CCS [Mil 80]. Then an observational semantics is directly associated to the concurrent system.

This method is particular suited to give the semantics of languages which show a strong interference between the sequential and the concurrent aspects, which is the case of Ada.

Summarizing, a language definition along the discussed method is structured in two main parts which are the "first step" denotational clauses, defined in an applicative metalanguage and the concurrent algebra, expressed via axioms on the possible transitions of the behaviours.

The execution environment for a language formal definition can be obtained using a logic programming approach, i.e. translating in Prolog the denotational clauses and the concurrent algebra.

The logic programming approach is followed mainly for two purposes: on one hand the use of metaprogramming techniques allows to control the execution of a program, that is a meta-level control of the performed actions is possible; on the other hand, the translation of a formal definition in Prolog is almost straightforward thus guaranteeing a certain degree of correctness in the translation process.

In the following, examples are made in reference to an example language, EL, already used in [AR86] as an example of how the SMoLCS methodology works.

The appendix presents a complete example of a Prolog translation of the denotational clauses and of the concurrent algebra of the semantics of EL.

2. Denotational clauses

In order to make the first step executable, the translation of the denotational clauses in Prolog is performed. The translation will try to reflect as much as possible the style in which the clauses has been defined, e.g. in the case of the EL language the continuation style is maintained. Although the translation is quite direct, there is no such a straightforward correspondence as it exists for the operational interpretation of the concurrent algebra. Anyhow, due to the inductive (homomorphic) nature of the denotational description, we expect to be able to perform an (almost) automatic translation.

In the classical denotational style, a semantic function is associated to each syntactic category, similarly, in the translation, in correspondence to each semantic function definition a predicate definition exists.

S[[st1; st2]]ρθ = S[[st1]]ρ(S[[st2]]ρθ) is translated in:

s(seq(St1,St2),Env,Cont,Res) :- s(St1,Env,Res1,Res),s(St2,Env,Cont,Res1).

Note that the elements of the syntactic domain take the form of terms with functional syntax, instead of the infix syntax used in the denotational clauses.
Each predicate takes the same input parameters of the corresponding clause, while its last parameter (Res) acts as an output parameter.
Moreover, since functions are implemented as relations, the application operation, which is primitive in a functional setting, needs to be explicitly defined as a proper predicate; also, predicates defining the environment (and its operations) are defined.

3. Concurrent algebra

The SMoLCS definition of a transition system is given (operationally) by means of rules of inference. Such rules of inference can be directly expressed in Prolog, resulting in the definition of a Prolog predicate that gives the possible moves of a behaviour. For example, the following rule giving the moves of a simple behaviour (which performs the action a and then behaves like bh):

$$a\Delta bh \xrightarrow{a} bh$$

becomes a fact about the predicate trans:

trans(A^Bh, A, Bh).

and one of the rules of inference defining the nondeterministic choice:

$$bh1 \xrightarrow{a} bh3 \quad \supset \quad bh1+bh2 \xrightarrow{a} bh3$$

becomes the Prolog clause:

trans(plus(Bh1, Bh2), A, Bh3):- trans(Bh1,A,Bh3).

Obviously, the syntax of behaviour has been modified in order to be acceptable by Prolog (again most infix operations on behaviours should become prefix ones).
The same direct translation in Prolog can be done for the more complex rules of inference that define the synchronization step of SMoLCS, which have to take in account the global information; for example tthe following rule, corresponding to the synchronization between two processes in the EL language:

$$b_1 \xrightarrow{SEND(ci,v)} b_1' \wedge b_2 \xrightarrow{REC(ci,v)} b_2' \supset \langle b_1 | b_2, st\rangle \rightsquigarrow^{\tau} \rightsquigarrow \langle b_1' | b_2', st\rangle$$

becomes (note that again the functions has to be transformed in predicates about the result):

```
synchr(par(B1,B2),STG1,tau,par(B3,B4),STG1):-
        trans(B1,send(Cid,V),B3),
        trans(B2,rec(Cid,V),B4).
```

In the same way, the other rules defining the parallelism and monitoring step of SMoLCS can be expressed in Prolog.

In the end, we obtain a Prolog program which defines predicates about the possible moves of a given set of behaviours. The following recursively defined predicate *run* can execute a behaviour set P, starting from a shared storage value Stg0 (this predicate hence constitutes an interpreter for the set of behaviours):

```
run(P,Stg0):-moves(P,Stg0,Act,P1,Stg1),run(P1,Stg1).
```

4. Advantages of the Logic Programming Approach

In the following some of the advantages of the use of logic programming are discussed:

Metaprogramming

The use of logic programming allows to experiment the use of metaprogramming techniques in order to control the first step translation and the execution of a given behaviours set. Metaprogramming techniques (easy to implement in Prolog [Sha 82, SS 86, St 85]) allow to express the control at the meta-interpreter level instead of in the interpreted Prolog program, hence without modifying the text of the program, i.e., in our case, of the translation of the denotational clauses and the concurrent algebra.

A "bare" Prolog meta-interpreter can be defined as follows:

```
solve(true).
solve((Goal1,Goal2)):- solve(Goal1),solve(Goal2).
solve(Goal):- clause(Goal,Body),solve(Body).
solve (Goal):- sys(Goal),Goal.
```

where clause is a system predicate which is true if there exists a clause with the given goal and the given body and sys is a (not predefined) predicate which is true if the argument is a system predicate.

At the meta-level is possible, see for example [St 85], by simply enhancing the metainterpreter, to record the history of the proof of a goal or to query the user about new facts to be added in the database. It is also possible to change the control strategy of Prolog, i.e. to modify the built-in strategy for nondeterministic choices, at the meta-interpreter

level.

As shown in [SS86], the overhead introduced by the meta-interpreter level can be overcome by the use of partial evaluation techniques. Hence, metaprogramming should help to build in a flexible, modular and systematic way all the facilities offered to the user of the system, while partial evaluation can help in maintaining the efficiency within acceptable bounds.

Translation algorithms

Since we are dealing with formal definitions, maintaining the consistency between the original definition and its executable version is a primary goal. To this respect the use of Horn clause logic (i.e. pure Prolog) as the target of the translation process allows to produce translation algorithms, for the first step and for the second step respectively, for which a proof of correctness can be given quite easily. The first partial results in this area which is still under study, can be found in [FGI 86].

Invertibility

Due to the well known invertibility property of the procedural interpretation of a predicate, it is possible to solve a goal which has as a ground parameter a behaviour expression and as a free term the program whose first step transformation is the given behaviour.

This allows to achieve a limited form of program synthesis, which can be useful in the implementation of programs starting from their behavioural specification; this feature could be better exploited in a framework of automated program production. Some examples and a discussion of this topic are presented in [FGI 86].

5. The execution environment

The topics discussed in the previous sections can be organized in an environment which allows the execution of a program along the formal definition of the language. This can help in learning the critical points of the language, or in testing a program, for example examining the possible actions of the program, or looking for unforeseen side effects.

The following is a possible scenario of the use and the working of the execution environment.

Suppose the user wishes to study the meaning and the behaviour, following the formal definition, of the following small EL program:

```
program
        X
        Y
begin
        X:=5;
        while X>0 do
            X:=X-1;
            Y:=Y+2
end
```

The analysis of this program can begin with the result of the first step transformation: for example the user can issue a command like:

> firststep

which calls the Prolog goal: writefstep(program).
where program stands for the program text above, and which is solved by the clause:
writefstep(Prog):-fstep(Prog,Res), write(Res).
The behaviour expression corresponding to the EL program is displayed, after a prettyprintig phase. In our case it displays:

```
  + λloc.ALLOC(loc) Δ  + λloc'.ALLOC(loc') Δ
LOC                    LOC

   τ Δ WRITE(loc,5) Δ
   fix λ y. +  λ v.READ(loc,v) Δ τ Δ τΔ
            VAL

       cond (v>0,  +  λ v.READ(loc,v) Δ τΔ τΔ
                   VAL

            WRITE (loc,v-1) Δ +  λ v'.READ(loc',v') Δ τΔ τΔ
                              VAL

            WRITE (loc',v'+2) Δ y, nil )
```

In order to investigate this result the user is allowed to issue further requests, for example:

>nextactions

which shows a list of the next possible actions of the program; this command calls the Prolog goal:
nextactions(behexp).
where behexp is the result of the fstep goal (in the internal, Prolog suitable syntax), and which is solved by the clause
nextactions(B):-setof(Act,trans(B,Act,B1),S),write(S).
In our case, since there is not an immediate nondeterministic choice among several action, but a nondeterministic choice of a value in a set, the user is shown the action:

ALLOC(loc)

where loc is a value internally chosen for the location.

In the case there is a nondeterministic choice, the user can choose to perform one of the shown actions - e.g. by pointing at it with a mouse, which automatically invokes the goal trans(behekp,chosenaction,B).
The resulting B is the continuation, to which the user can apply again the nextaction query, for a step by step execution.
Note however that the actions returned by the previous goals are the possible atomic actions of the program or of a single process in the program. Each of these actions is effectively possible depending on the global information and, in general, on other processes. Choosing one of them as shown before means that the user forces one particular execution path.
Applying instead the three steps of the SMoLCS methodology, the choices are guided by the global information - in the case of our example program all the choices are deterministic - and the actions will become internal, unobservable actions; the meaning of the program is given, in the case of EL, just by the global state value at the termination of the program execution.
Thus, if our user issues the command :

>run

which runs the interpreter *run* given at the end of section 3, no actions are displayed, and the final value of Info is the result of the program. More generally, for a program with nondeterministic behaviour, such value is one of the possible results.
However, for the user of the system is not satisfying to know just one of the possible results; rather, the ability to

interact more closely with the program, in a "debugging" style, should be provided, by proper queries to the Prolog program clauses.

For example, the ability to know how the program is arrived at certain status is easily provided by a meta-interpreter which records the history of the proof; such history can be backward analyzed in detail to see why the execution of a program has chosen some path and not other ones.

6. Overall architecture of the system

The possible interactions of the user with the system discussed above can be implemented by invoking proper queries to interpreters, partial interpreters, and meta-interpreters. Such queries should be invoked via a command environment which interpretes the commands given by the user. The results of the queries should be displayed through a prettyprinting phase.

The overall architecture of the system is hence that sketched in Fig. 1. The user interface may provide aids to the user interaction, such as windowing, mouse interaction, etc.

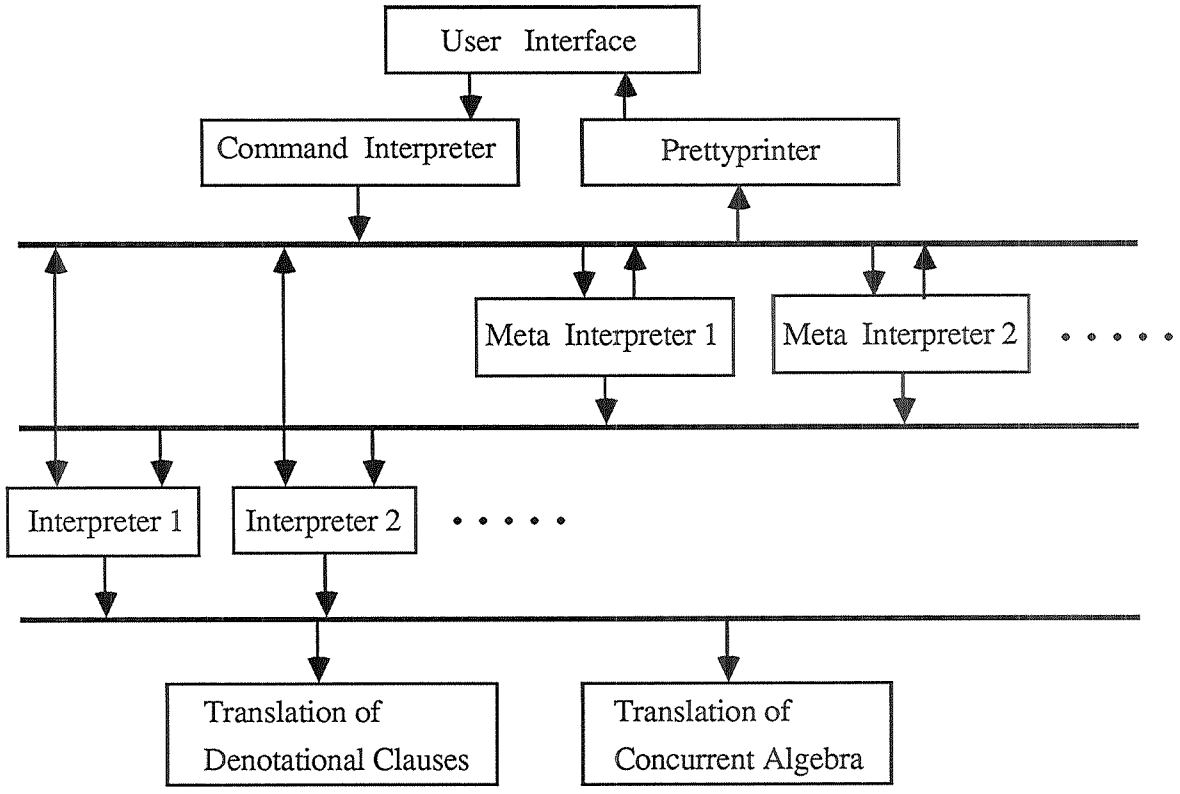

Figure 1.

7. Conclusions

The aim of the proposal presented in this paper is to investigate the feasibility of using logic programming techniques for building up environments to execute formal definitions of concurrent languages based on operational semantics. In this sense, the proposal easily extends to formalisms or specification languages based on labelled transition systems, notably CCS [Mil 80].

In the framework of the Ada Formal Definition project, this approach is used to study problems related to the efficiency of the execution of the formal definition, to the interaction of the user with the system and to the evaluation of the use of metaprogramming techniques. The described facilities are supposed to be integrated in the tool set under development [GMP 86]. The translation algorithms will be automated since it is planned to integrate them with the Ada Formal Definition tool set.

Up to now the execution environment for the EL language has been implemented in CProlog on VAX/UNIX; it is going to be moved on Sun workstations using Quintus Prolog in order to improve efficiency and to exploit the graphic interface of the Sun.

REFERENCES

[AB86] Astesiano, E., Bendix Nielsen, C. et al., "The Trial Definition of Ada", Deliverable 7 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada.

[AGMRZ86] Astesiano, E., Giovini, A., Mazzanti, F., Reggio, G. and Zucca, E. "The Ada Challenge for new Formal Semantic Techniques". Proc. Ada-Europe Conference, Edinburgh, (1986).

[AR86] Astesiano, E. and Reggio, G. "A Syntax-Directed Approach to the Semantics of Concurrent Languages". Proc. 10th IFIP World Congress, Dublin, (1986).

[FGI 86] Fantechi, A., Gnesi, S., Inverardi, P. "Proposal for an execution environment for the Ada formal definition", IEI Internal Report, Draft version , August 1986.

[GMP 86] Gallo, T., Manfredi, F., Papa, M.P.,"Requirements for an Ada FD Tool set", Deliverable19 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada.

[Mil 80] Milner, R. "A Calculus of Communicating Systems", LNCS 92, 1980.

[Sha 86] Shapiro, E., "Algorithmic Program Debugging", MIT Press, Cambridge, Massachussetts, (1982).

[SS 86] Safra, S., Shapiro, E., "Meta Interpreters for Real". Proc. 10th IFIP Wold Congress, Dublin, (1986).

[St 85] Stirling, L., "Expert System = Knowledge + Meta-interpreter", Dept. of Applied Mathematics, The Weizmann Institute of Science, Internal Report CS84-17.