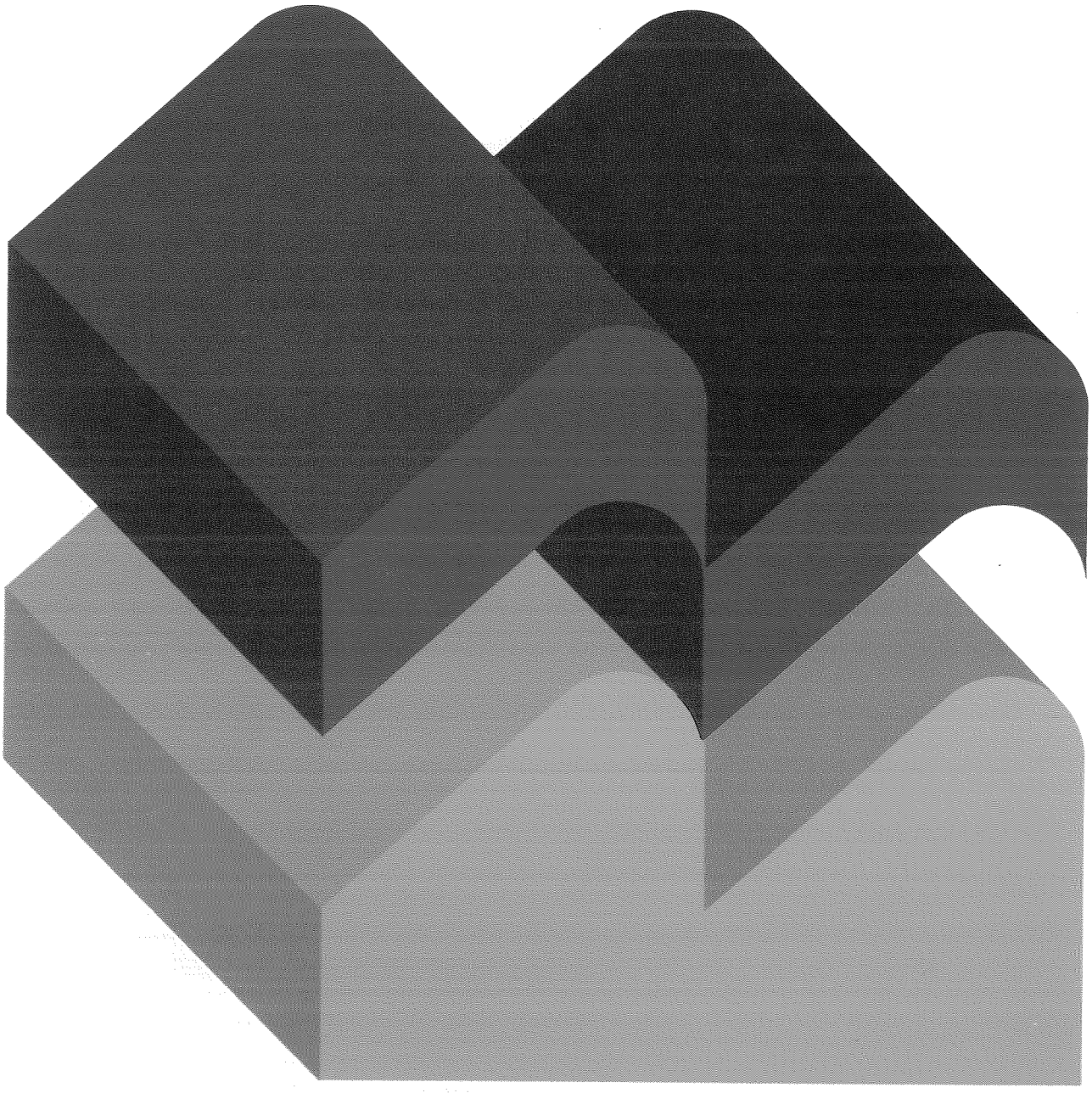
The Draft Formal Definition of Ada®

Commission of the European Communities: Multi-Annual Programme

Alessandro Fantechi, Stefania Gnesi, Paola Inverardi, Ugo Montanari

On the feasibility of the execution of the Ada formal definition

January 1987



®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

THE DRAFT FORMAL DEFINITION OF ADA®

ON THE FEASIBILITY OF THE EXECUTION OF THE ADA FORMAL DEFINITION

Authors: *Alessandro Fantechi (IEI)*
Stefania Gnesi (IEI)
Paola Inverardi (IEI)
Ugo Montanari (University of Pisa - CRAI)

Date: *January 1987*

Workpackage: *Q/S*

Status: *Final*

Distribution: *Free*

Doc. no. *AdaFD/IEI/Executability*
Annex to AdaFD/IEI/30 and AdaFD/IEI/33

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

1. Introduction

This report presents the results of a study on the execution of the formal definition of Ada, i.e. on the possibility of transforming it into an executable form. For this purpose, logic programming has been chosen as the executable form due to several reasons: for example, the translation of the Ada formal definition in Prolog is reasonably direct, thus guaranteeing a certain degree of correctness in the translation process; again, the use of metaprogramming techniques allows to control the execution of an Ada program, that is a meta-level control of the performed actions is possible. The resulting execution environment is shown to be attractive also with respect to its integration in the AdaFD tool set.

This report surveys all the problems encountered in this study, in particular it claims that the theoretical feasibility is possible once some conditions on the formal definition are verified.

In this report we are not addressing the static aspects of the formal definition of the language, namely the static semantics and the so called AS1-> AS2 transformation, whose executability is assumed to be feasible. Hence, only dynamic semantics aspects are considered.

2. Logic programming approach

The executable form of the Ada formal definition is realized translating in Prolog the denotational clauses and the concurrent algebra.

2.1. Translation of the denotational clauses

In order to make the first step executable, the translation of the denotational clauses in Prolog is performed.

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.

For example, a typical denotational semantic clause like:

S[[st1; st2]]ρθ = S [[st1]]ρ(S [[st2]]ρθ) , which means that the meaning of a sequence of statements in a given environment and with a given continuation is that of the first statement in the same environment and with the meaning of the second as continuation, 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.
Since all the semantic functions we are dealing with in the first step should be total [AR 86] the chosen evaluation order has no effect on the semantics of the first step. This corresponds to the intuition that the first step is a purely compilative one, translating from AS2 to the intermediate language of behaviours. This property of the first step is assured if certain rules has been followed in writing the denotational clauses (use of recursion only for structural induction, use only predicates on the syntactic domains as conditions in conditional clauses, and so on).

Another example of translation of denotational clauses, taken from the Ada Formal Definition [Den Clau - Aux.5<2>] is the following simple auxiliary clause:

.0 *Start-Erroneous-Exec*(li) ≙
.1 **choose** *Raise*(Program_Error)li
.2 **or** START-ERRONEOUS-EXECUTION Δ **exit** Start_Erroneous_Execution
.3 **end choose**

which is translated in:

```
starterroneousexec(LI,choose(Bh1,start-erroneous-execution-action^exit(Bh2)):-
    raise(program-error,LI,Bh1),
    start-erroneous-execution-bh(Bh2).
```

The whole first step is executed by querying the Prolog goal:
execProgram(PAda,ImpDepParms,P,InitInfo).
whose definition is obtained translating the definition of the corresponding semantic function *exec-Program* [Den Clau - 10.5<1>], and where PAda and ImpDepParms are bound to the AS2 program on which the first step is executed and to the implementation dependent parameters respectively, while P and InitInfo will return the associated set of behaviours and the initial Global Information for the second step.

2.2. Translation of the concurrent algebra

The concurrent algebra is defined by hierarchical algebraic specifications. The intermediate stages of this hierarchy express the simpler abstract data types, then the Task Transition Systems, the Global Information and the three steps of the SMoLCS methodology, to obtain the Program Concurrent System (shortly PCS).

The feasibility of the transformation of this algebraic specification in Prolog, maintaining a certain degree of correctness, requires that some theoretical conditions are verified. A first approach is to transform the equational - conditional axioms in rewrite rules using some known method [HO 80, Hus 85, Kap 84], to obtain a system of oriented equations. Now, the oriented equations can be expressed as Horn clauses, then directly translated in Prolog.

Another approach is to separate the equational axioms defining the basic abstract data types (local information, global information, etc..) from the conditional ones defining the transition rules.

The former are translated in Prolog following the algorithm shown in [vEY 86], which requires the noetherian and confluence conditions to be verified on the term rewriting system associated to the equations. The latter, due to their conditional nature, are directly expressed in Prolog.

The correctness and completeness of the two approaches with respect to the original algebraic specification should then be discussed.

Let us now explain in further detail what results from this transformation, recalling that the SMoLCS algebraic definitions of the transition system have also the intuitive meaning of rules of inference defining the transitions. In the Prolog translation of the concurrent algebra such rules of inference become Prolog clauses defining a predicate that gives the possible moves of a behaviour. For example, the following rule - one of the axioms in the specification BEHAVIOURS [Conc_Alg - Behaviour_Part] - 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:

$$\frac{bh1 \xrightarrow{a} bh3}{\textbf{choose } bh1 \textbf{ or } bh2 \textbf{ end choose} \xrightarrow{a} bh3}$$

becomes the Prolog clause:

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

The syntax of the behaviour needs to be modified in order to be acceptable by Prolog.

The more complex rules of inference that define the synchronization step of SMoLCS, which take in account the global information, are similarly translated. For example, the rule which describes the creation of a new task [Conc_Alg - Synchronization]:

$$\bullet \quad \frac{t_1 \xrightarrow{\text{CREATE-TASK(created-task, tgn, egn, mgn)}} t_3 \ \wedge \ t_2 \xrightarrow{\text{CREATED(created-task)}} t_4}{t_1 | t_2 \xrightarrow{\text{TAU}} t_3 | t_4}$$

cond: Is_New_Task_Global_Name (tgn, i)

transf: Add_Task (tgn, mgn, egn, i)

becomes the prolog clause about the predicate *synchr*:

```
synchr(par(T1,T2),Inf12,tau,par(T3,T4),Inf34):-
        trans(T1,create-task(Created-task,Tgn,Egn,Mgn),T3),
        trans(T2,created(Created-task),T4),
        isNewTaskGlobalName(Tgn, Inf12),
        addTask (Tgn, Mgn,Egn, Inf12,Inf34).
```

The translation of the other rules defining the parallelism and monitoring steps provide at the end a predicate (of the form *moves(PCSstate1, L, PCSstate2)*), which is demonstrable true if a move with label L from a state of the PCS to another state of the PCS can be derived by the axioms of the algebraic specification.

2.3. An intepreter

The model of an Ada program, a PCS, can be seen as a (possibly infinite) labelled tree in which labels identify the program interactions with the external world (represented mainly by the I/O operations of the program). Branches in the tree model the possible nondeterministic alternatives of the program execution, related for example to the internal nondeterminism implied by the concurrency of tasks or to the external nondeterminism implied by the incomplete definition of the I/O system.

In this way the semantics of an Ada program is defined taking into account all the theoretically possible executions of the program itself.
The predicate *moves* obtained by the above translation hence tells whether a move identified by a label L is present on the arc which links the two nodes corresponding to the states PCSstate1 and PCSstate2. Hence a recursive predicate *run* like the following, which at any step invokes the *moves* predicate, follows one of the branches of a tree corresponding to a set of behaviours P associated to an Ada program PAda, hence obtaining one of its possible executions, starting from an initial global information Info0:

```
run(P,Info0):-moves((P,Info0),Act,(P1,Info1)),write(Act),run(P1,Info1).
run(nil, Infofin).
```

The addition of the *run* predicate transforms the translation of the concurrent algebra in an interpreter for the set of behaviours. Adding another predicate definition:

```
interpret(PAda,ImpDepParms):-execProgram(PAda,ImpDepParms,P,Info0),run(P,Info0).
```

we obtain by using the translation of the denotational clauses and the translation of the concurrent algebra in an interpreter for the Ada program (more precisely, an AS2 program).

3. A proposal of execution environment

In this section we present a proposal for an environment for the execution of the formal definition, based on the logic programming approach described above. The items discussed are the use of advanced techniques like metaprogramming, the functionalities offered by the environment, and the integration in the AdaFD tool set.

3.1 Metaprogramming Approach

The use of logic programming allows to experiment the use of metaprogramming techniques in order to control 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 it 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 [SS 86], 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.

3.2. Functionalities offered to the user

The way the user interacts with the execution environment is realized having in mind the goals of the executability of the Ada formal definition, among which we list the following:

i) to be able to learn which are the critical points of the execution of an Ada program following the formal definition of the language; in particular to see which are the possible actions that can be performed by the program, for example in order to see if some of them were not considered in the development of the program, or if some unforeseen side effect is possible, and so on. The possibility to "zoom" on a particular construct of the program to study its effect as defined by the Ada formal definition should also be provided .

ii) to execute the ACVC tests in order to measure the relative conformance to the Ada formal definition [FGIM 87];

iii) to help the comprehension of the formal definition itself, by working on small program examples.

The above requirements concern mostly the execution of the set of behaviours corresponding to an Ada program; moreover, the user should have also the possibility to operate only the first step translation on a program or on a single construct, in order to understand which is their semantics along the formal definition. This is automatically provided by direct queries to the Prolog predicates which correspond to the first step. Once the behaviour set corresponding to a program is obtained, it should be possible for the user to retrieve automatically from the formal definition text

information explaining the resulting behaviour.
Three typical modalities of execution of a (piece of an) Ada program following the formal definition are the following:
i) Direct execution:
running a simple interpreter as *run* given at the end of section 2.3 allows to directly execute the set of behaviours associated to a program by the first step, internally choosing one of the possible paths of execution, and printing the corresponding sequence of observable actions.
ii) Step by step execution:
the user can execute a step of the program, which corresponds to perform one of the possible (synchronized) actions. The list of the next possible actions can be shown to the user to let him choose from the list. For example the user can issue a command which calls the Prolog goal:
externalactions(behexp,info0).
where behexp is again the result of the *execProgram* goal and info0 is a suitable default initial global information, and which is solved by the clause
externalactions(B,Info):-setof(Act,moves((B,Info),Act,(B1,Info1)),S),write(S).
This command shows a list of the next possible external actions, and hence 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:
moves((behekp,info0),chosenaction,(B,Info)).
The resulting B is the continuation, the resulting Info is the global information transformed by chosen action; to them the user can apply again the externalaction query, for a step by step execution. What we have obtained is thus an execution that stops at every point of interaction with the external world.
iii) Task monitoring:
the user is given the possibility to monitor the evolution of the tasks present in the original Ada program. For this purpose, it is possible to retrieve the identity of a task that has executed a particular action, it is possible to tell the system which subset of the tasks of the program should evolve, and so on - in practice, any useful "debugging" facility.
It should be noticed that this kind of facility does not require to interact with the program only at the level of the visible external actions, but needs to interact more closely, at the level of the single internal actions and the values of local and global information.
This granularity is easy to be achieved with the proposed approach, via direct queries to the predicates which are the translation of the relative algebraic definitions; it would be not possible if the translation of the concurrent algebra constituted a monolithic block, providing just the execution of whole programs.

Another aspect of the user view of the execution, aimed to a better comprehension of the formal definition itself, is to offer to the user the possibility to inspect the internal decisions taken by the behaviour interpreter in order to assert that a particular transition is possible for a given behaviour. Also here metaprogramming can show its benefits: if the Prolog goals has been solved by a meta-interpreter which records the history of the proof, such history can be backward analyzed to see why the execution of a program has chosen some path and not other ones.

3.3. Overall architecture

As we have seen the possible interactions of the user with the system discussed above can be implemented mostly by invoking proper queries to the Prolog clauses of the translation of the denotational clauses, and the translation of the concurrent algebra, either directly or via proper metainterpreters. The queries or the metainterpreters are invoked via a command environment which interpretes the commands given by the user.

The command interpreter plays an important role also in the integration of the execution environment in the toolset being developed for the Ada formal definition [GMP 87, GGMP 86], which is centered around the metalanguage in which the definition itself is written [AGIMRZ 86]. The toolset includes a database in which the whole formal definition is stored; the definition clauses are stored in an internal (tree-like) representation which is used by the syntax-directed editor and prettyprinter for the definition metalanguage.

The command interpreter interacts with the editor in order to translate the functional (prefix) notation of the behaviours handled by the Prolog programs into the notation used by the formal definition text, and vice-versa. Moreover, the command interpreter interacts with the database, for example in order to cross-reference the text produced by the first step transformation with respect to the formal definition text.

The internal form which is handled by the metalanguage tools helps also to automatize the translation in Prolog of the denotational clauses, when building up the system.

The overall architecture is summarized in fig.1. The user interface may provide aids to the user interaction, such as windowing, mouse interaction, etc.

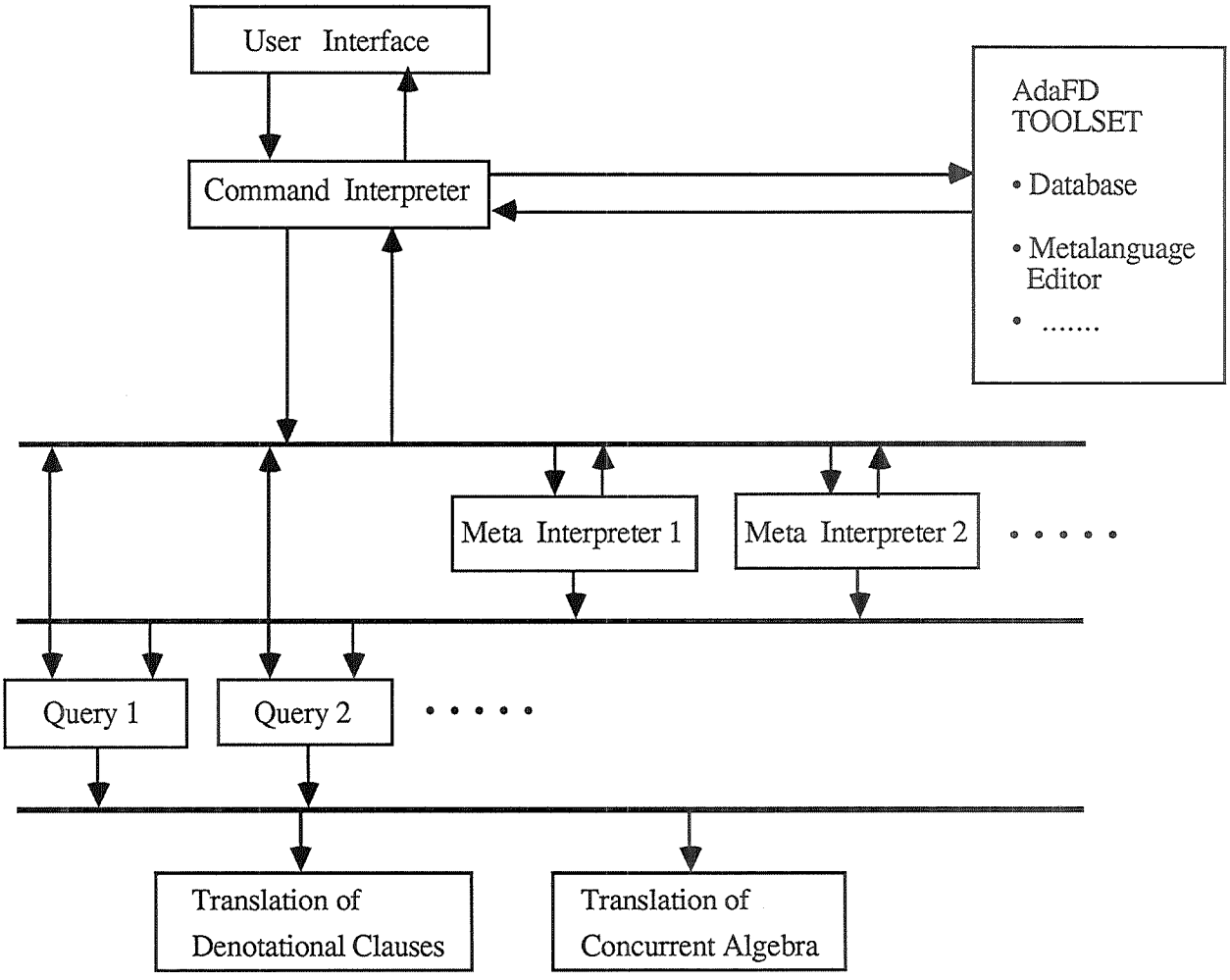


Figure 1.

4. Concluding remarks

In this report we have presented a logic programming approach to the executability of the Formal Definition, and a proposal of an execution environment integrated with the AdaFD tool set. Several problems related to this approach has been discussed and will be topics of further work.

In particular, in order to evaluate the method, we have partially implemented the approach with reference to a small example language (EL, see [AR 86]) on the VAX/ Unix BSD4.2 in C-prolog. This example implementation has shown the feasibility of the approach but does not give us enough input in order to predict the efficiency of the method when applied to AdaFD. Future work in this direction will be a complete EL implementation on Sun workstations using Quintus Prolog.

REFERENCES

[AGIMRZ 86] Astesiano, E., Giovini, A., Inverardi, P., Mazzanti, F., Reggio, G. and Zucca, E. "Final Specification Language", Deliverable 9 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada.

[AGMRZ 86] 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).

[AR 86] Astesiano, E. and Reggio, G. "A Syntax-Directed Approach to the Semantics of Concurrent Languages". Proc. 10th IFIP Wold Congress, Dublin, (1986), pp.571-576.

[FGIM 87] Fantechi, A., Gnesi, S., Inverardi, P., Mazzanti, F., "Feasibility of ACVC validation with respect to the Ada Formal Definition", Deliverable 33 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada.

[GMP 86] Gallo, T., Manfredi, F., Papa, M.P., "Requirements for an Ada FD Tool set", Deliverable 19 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada.

[GGMP 86] Gallo, T., Giovini, A., Manfredi, F., Papa, M.P., "Ada FD Tool set: Architecture and Preliminary Design", Deliverable 20 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada.

[HO 80] Heut, G., Oppen, D.C., "Equations and Rewrite Rules - A survey" in Book, R.V. ed., "Formal Language Theory", Academic Press, New York 1980.

[Hus 85] Hussmann, H., "Unification in Conditional-Equational Theories", EUROCAL'85, LNCS 204, 543-553.

[Kap 84] Kaplan, S., "Conditional rewrite rules", Theoretical Computer Science, 33, 2-3,

(1984), 175-193

[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) pp.271-278.

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

[vEY 86] van Emden, M. H., Yukawa, K., "Equational logic programming", Dept. of Comp. Sci., University of Waterloo, Canada, Technical Report CS-86-05, March 1986.

The other references are inside the Dynamic Semantics documents, Deliverables 15 and 16 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada.