DESCARTES : An Automatic Programming System for Algorithmically Simple Programs

Bruno Ginoux, Jean-Luc Dormoy, Claudia Jimenez-Dominguez,
Jean-Yves Lucas, Laurent Pierre
Electricité de France, Direction des Etudes et Recherches
1, Avenue du Général de Gaulle 92141 Clamart CEDEX France.
Email : René.Descartes@der.edfgdf.fr

Abstract

Most research work in the field of automatic programming has been focused on conceptually complex problems. However, although most of the programs we are generally faced with may be very big and manage large volumes of data, they are conceptually simple. Starting from this consideration, we have developed, since 1992, a system called DESCARTES which, fully automatically, generates programs written in conventional procedural languages, starting from program specifications expressed in a mathematical type formal specification language. The first operational version of the system which represents about one million of C lines has already been used in several industrial applications, and especially to specify and generate a nuclear power plant emergency shutdown system and a module scheduling control operations for a fuel power plant. In this paper, we describe the DESCARTES language and system as well as the results obtained and we comment our approach compared to traditional approaches in the field of automatic programming.

1. Introduction

Most automatic programming researchers have always been attracted by conceptually complex problems. In order to tackle these difficult problems while producing programs with reasonable complexities, automatic programming systems actually had to automatically invent sophisticated algorithms. They had therefore to make very intelligent choices of control and data structures. The obvious consequences of this approach were that successes remained limited and therefore automatic programming was considered to be too difficult or even impossible. The systems which are proposed today are, therefore, either semi-automatic or dedicated to very specific application domains such as man-machine interfaces, database management systems, scheduling problems, reactive systems, protocols, etc. However, most of the programs we are generally faced with are conceptually simple. This is true, not only for management applications, but also in industrial contexts. This remains true in scientific applications where, except for skilled optimized black box functions which actually perform the computations, the code consists of handling data in a very traditional way.

At the Research and Development center of Electricité de France, we have developed, since 1992, the DESCARTES system [5,6,7,8,9]. The DESCARTES language is a mathematical type formal specification language which is based on the set theory, the logic of first order predicates and the theory of recursive functions. The DESCARTES system is an automatic program generator which, fully automatically, generates programs written in conventional procedural languages.

In the next section, we present traditional approaches in the field of automatic programming. Then, in section 3, we describe our own approach. Section 4 presents the DESCARTES language and illustrates it in a simple example. In the following section, we describe the system, its architecture as well as the main transformations which have been implemented. Then, we present the main applications that have already been generated by using DESCARTES and the results we obtained. Finally we compare our language to others and we show the contribution of DESCARTES to software engineering.

2. Related work

There are basically two levels of automatic programming, both of which are called “automatic programming”, which are linked :

• The first level is based on a completely declarative specification expressed for example in a mathematical language. Starting from this kind of specification, i.e. a specification containing no indication about the way to solve the problem, it automatically generates a specification that we will describe as "pre-algorithmic" or "computational" or, alternatively, "constructive" because, while it does not constitute a program, it contains information that can be used to write a program.

• The second level is based on that "computational" specification, expressed for example in a recursive manner, and automatically generates an algorithm and, then, a program implementing that specification.

Typically, the following set definition, in which E is a given set, f a given function and Q a given predicate :

$$\{x \mid x \in E \text{ and } \forall y\,(y \in f(E, x) \Rightarrow \exists z\,(z \in E \text{ and } Q(x, y, z)))\}$$

is completely declarative as there is nothing specifying how to compute that set. In fact, the algorithm must be completely designed and there are many possibilities. On the other hand, the following specification :

$$f(x) = \text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x-1) \text{ endif}$$

is "computational" in the sense that, by using the definition itself, it is possible to compute the function (even if this is not the best way of doing it). This is, in fact, a constructive definition.

It is, of course, the first level that contains the real intelligence. For example, it is extremely difficult to start out with the completely declarative form of hcd :

```
hcd(a,b) =   ( d | d divides a and d divides b
                   and not ( ∃ d1 | d1 divides a
                                    and d1 divides b
                                    and d1 > d ) )
```

and automatically find out that a good computational specification is :

```
hcd(a,b) = if a = b then a
           else ( if a > b then hcd(a-b,b) else hcd(a,b-a)  endif )
           endif
```

Moreover, until recently, computers were still not powerful enough to carry out processing operations that were so demanding in terms of CPU time and memory space and the level of languages was not high enough to allow such processing operations to be expressed in a simple enough manner. It was, thus, materially impossible to build such systems.

It is, "however", essentially this level that kept automatic programming researchers busy for a long time, the best known being Manna and Waldinger at the beginning of the 1980s [13]. The consequence was that successes remained extremely limited and that this type of automatic programming was considered to be too difficult or, even, impossible and, therefore, it was somewhat neglected.

As the first level seemed to be out of reach, some researchers continued working on the second level, i.e. going automatically from a computational specification to a program. But here again, they concentrated mainly on the most complex aspects. It must be understood that, even though this second level of automatic programming is simpler than the first, it is far from being trivial. Indeed, a computational specification is still far from the program, even if it does contain information on the way to solve the problem algorithmically. It only suggests a possible implementation which, in fact, implicitly defines a class of potential implementations for the program. The best one must, therefore, be chosen (as, in general, it is not immediately suggested by the specification but, in most cases, has to be calculated from it) and efficiently implemented, i.e. all the successive transformations, refinements and optimizations required must be carried out [10].

Moreover, in complex cases, these transformations and refinements are difficult to select automatically. When one attempts to generate a sorting algorithm, for example, or, more generally, a program requiring a high level of efficiency such as optimization problems or combinatorial problems, the choice of the principle of the algorithm as well as the choice of data structures are crucial in order to avoid producing programs with exponential complexities. This led, once again, to results that did not measure up to what was hoped for. Some people therefore claimed that completely automatic programming was not possible and turned to semi-automatic systems, still at the second level, where the user can guide the machine in order to make crucial choices himself or herself [16,17].

However, apart from these complex cases that are difficult to process completely automatically, there are really simpler cases where automatic programming is perfectly possible and can be envisaged. These cases constitute, moreover, the great majority and it would be a shame, in our opinion, to forego the advantages of this approach on the pretext that there are cases which are beyond its capabilities. It should not be forgotten that we cannot compute everything on a computer but that does not prevent people from using them.

3. Our approach

DESCARTES is concerned with these simpler cases where automatic programming can be directly envisaged. More precisely, if we go along with our classification of automatic programming into two levels, DESCARTES is in a totally original position in that it is located on both the first and second levels. Indeed, a DESCARTES specification contains totally declarative elements such as set definitions, quantifiers, aggregates, etc., as well as computational elements which are, as it happens, recursive functions.

Our approach is, in fact, pragmatic. In a very schematic manner, we express everything that can be automatically re-expressed in computational form in a completely declarative manner, i.e. on the first level, and all the rest is expressed directly in computational form by means of recursive functions. We do not, therefore, have the same difficulties as those encountered by people working on the first level and seeking an exhaustive capability at that level. Furthermore, we do not have the problems encountered by those working on the second level as we do not tackle the same problems and, thus, we do not tackle the same programs. To explain this in a caricatural manner, we do not try to generate complex small programs to perform efficient sorting operations or rapid Fourier transforms but, rather, conceptually simple large programs handling large volumes of structured data. To be more precise, by « conceptually simple programs », we mean essentially programs which are not combinatorial. Indeed, although we can very easily specify scheduling or optimization problems, DESCARTES is not yet able to generate efficient programs for this kind of problems. Indeed, combinatorial problems are generally second-order problems which require specific knowledge, such as the design theories of KIDS [17], to reduce the searching space. Although the DESCARTES system can deal with some kind of second-order problems, due to a big amount of specialized knowledge on DESCARTES mathematical primitives, it remains, from a general point of view, limited to first-order logic with fixed point.

Therefore subtle choices of control structure and data structure seeking to reduce program complexity and requiring user intervention are then no longer required and can be replaced by "mean" choices that are reasonably efficient and, above all, completely automatic. It is largely for this reason that DESCARTES is completely automatic whereas the other general systems are not.

DESCARTES is therefore useful in the context of programs whose complexity resides mainly in their size and the volume of data manipulated and which remain conceptually simple enough not to require the intervention of a programmer. It should be noted that, in the industrial context in which we find ourselves, this limitation is not really a limitation at all as this type of program represents the great majority of programs to be generated (see in section 6, the examples the system already dealt with). In addition, when faced with a program that does not fall into the "right" category, the only risk run is that the program generated may be inefficient.

4. The DESCARTES language

4.1 Presentation

The DESCARTES language is a mathematical type formal specification language which is based on the set theory, the logic of first order predicates and the theory of recursive functions. It can be used to define and manipulate standard mathematical objects such as sets, vectors, lists and functions by using the usual quantifiers and mathematical operators : $\cup$, $\cap$, $\forall$, $\exists$, Σ, Π, MAX, MIN, Cardinal, and so on. In particular, lists should not be confused with programming language lists which are concrete data structures using addresses and memory locations and which are, thus, very close to the machine.

The mathematical aspect of the language is its prime feature, from which many of its strong points stem. We can mention the following :

• *Declarativity* : DESCARTES allows to express a problem with a high level of abstraction, i.e. independently of the machine.

• *Conciseness* : Mathematical language is much more concise than usual programming languages.

• *Readability and easy learning* : Mathematical language is well-known, at least as far as the basic concepts used in DESCARTES are concerned. In a manner of speaking, DESCARTES is not a new language. It therefore soon becomes easy to read and, for the same reasons, it is easy to learn.

• *Reusability* : Writing a DESCARTES specification consists in giving a set of function definitions where each function expresses an item of knowledge or a concept. As each of these definitions is written independently of any specific utilization, i.e. independently of any specific program, they can easily be reused. The level of reusability is, in fact, much better than for conventional languages in which items of knowledge are always expressed with respect to a specific need but, also, inextricably linked with a specific implementation.

A DESCARTES specification is made up of the following three parts :

• the conceptual schema of the data in the field of application. This schema is expressed in an Entity-Relationship model. It is common to all the applications corresponding to the field,

• a set of function definitions provided by the user. Each function expresses an item of knowledge on the field of application,

• a "COMPUTE" command which specifies the aim of the program, i.e. in fact, the quantity to be computed by the program. This "COMPUTE" command alone actually constitutes the specification of the program. It must be noted that the quantity to compute must be finite but this doesn't prevent the user from defining, and the system from handling, infinite sets during the process of construction of the program.

The conceptual data schema and, also, the definitions of functions are expressed independently of any specific program. In fact, they together represent what is generally called the *domain theory*.

4.2 Example

In order to give an idea of the DESCARTES language and although this problem is in essence a combinatorial problem, i.e . *not an algorithmically simple problem*, let us consider the DESCARTES specification of the domain theory of the meeting scheduler system

First, the conceptual data schema describes the objects and the relationships between them, that are handled in the application. A part of the conceptual data scheme of the meeting scheduler is given in Fig. 1. There are, what we call entity sets, which are : Meetings, Locations, Calendar Dates, Attendees and Equipments. There are also relationships between these entity sets : « Meeting location » which associates to a meeting its location, « Preferred location » which associates Meetings, Attendees and Locations, « Preferred dates » which associates Meetings, Attendees and Calendar Dates and so on. All the corresponding data may be stored for example in a database and may be modified as long as the meeting dates and location are not fixed. At a given state of the database, and for a given meeting, three relationships are to determine : « Meeting location », « Begin meeting date » and « End meeting date ».

4.2.1 Data-oriented functions. In the DESCARTES language, relationships are seen as sets of tuples. By directing these relationships we can define what we call « data-oriented functions » which become the basic vocabulary of the domain theory. For example, starting from the « Preferred dates » and « Impossible dates » relationships, we can define the two following functions (among others) :

LET PreferenceSet(m,a) : Meetings, Attendees → Set(CalendarDates) :
/ set of dates preferred by a given attendee for a given meeting. The tuple [m,a,d] must belong to the corresponding relationship*/*

{d | d ∈ CalendarDates and [m,a,d] ∈ PreferredDates}

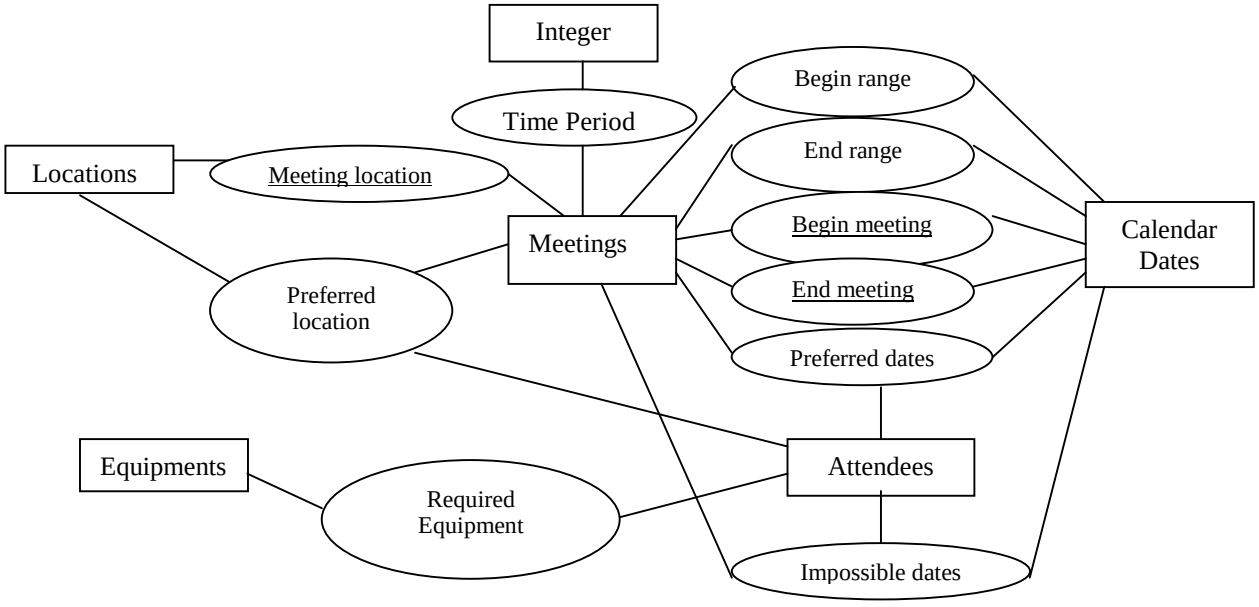

Fig. 1 Conceptual data scheme of the meeting scheduler

LET ExclusionSet(m,a) : Meetings, Attendees → Set(CalendarDates) :
/ set of dates which are excluded by a given attendee for a given meeting */*

{d | d ∈ CalendarDates and [m,a,d] ∈ ImpossibleDates}

In fact, these data-oriented functions are automatically derived. To be more precise, the user must specify the name, domain and codomain of the functions he desires and corresponding bodies are automatically determined by the system from the description of the corresponding relationships.

4.2.2 User-oriented functions. Once each required data-oriented function has been defined, the user can go on specifying the domain theory by defining other functions, called « user-defined functions » which represent more complex concepts of the domain. These functions are written by combining the basic data-oriented functions with DESCARTES primitives. For example, the following functions can be defined :

LET BelongsToRange(d,d1,d2): CalendarDates, CalendarDates, CalendarDates → BOOLEAN :
/ returns true if d belongs to the date range [d1,d2]. The functions « After » and « Before » are user-defined functions which state if a date is after or before an other date. They are not described here */*

After(d,d1) and Before(d,d2)

LET DateRange(d1,d2): CalendarDates, CalendarDates → Set(CalendarDates) :
/ returns the set of dates belonging to the range [d1,d2] */*

{ d | BelongsToRange(d,d1,d2) }

LET NonExcludedDates(m) : Meetings → Set(CalendarDates) :
/ the dates belonging to the meeting date range and belonging to none of the exclusion sets of all potential attendees. The functions « Attendees », « BeginRange » and « EndRange » come from the corresponding relationships of the conceptual data schema. They are data-oriented functions. The « with » statement allows to denote intermediary expressions */*

{d | BelongsToRange(d,db,de)
and not (∃ a (a ∈ Attendees(m)
and d ∈ ExclusionSet(m,a)))}

with (db = BeginRange(m), de = EndRange(m))

LET StrongConflict(m) : Meetings → BOOLEAN :
/ any date belonging to the meeting date range also belongs at least to the exclusion set of one potential attendee */*

NonExcludedDates(m) = ∅

LET WeakConflict(m) : Meetings → BOOLEAN :
/ the intersection of the non excluded dates and the intersection of the preference sets of all potential attendees is empty */*

NonExcludedDates(m) ∩ E = ∅

with (E = ∩ (PreferenceSet(m,a) | a ∈ Attendees(m)))

LET PotentialMeetingBeginDates(m) : Meetings → Set(CalendarDates) :
/ the set of dates such as none of the following dates (within the time period of the meeting) is excluded. « Time Period » is a data-oriented function. Succ(d,n) returns the n^{th} date following the date d. It is not described here */*

{db | ∀ d (d ∈ DateRange(db,Succ(db, TimePeriod(m)))
=> d ∈ NonExcludedDates(m)) }

LET NumberOfSatisfiedPreferenceSets(m,d) : Meetings, CalendarDates → INTEGER :
/ number of preference sets satisfied by a given date. « # » is the DESCARTES primitive for « Cardinal » */*

({a | a ∈ Attendees(m) and d ∈ PreferenceSet(m,a)})

LET ProposedMeetingBeginDates(m) : Meetings → Set(CalendarDates) :
/ the set of potential meeting begin dates which belong to as many preference sets as possible. We consider here that a meeting date, that is a couple [« begin date », « end date »], satisfies a preference set if its « begin date » satisfies it. We won't go into more details about the way to define a more general notion of satisfaction for an interval of date */*

{db | db ∈ PotentialMeetingBeginDates(m) and
not (∃ db' (db' ≠ db and
db' ∈ PotentialMeetingBeginDates(m) and
NumberOfSatisfiedPreferenceSets(m,db')
>
NumberOfSatisfiedPreferenceSets(m,db))}

We won't write here the functions corresponding to the problem of locations, equipments and so on. Anyway, our aim is not to have a complete specification of this example but rather to illustrate the DESCARTES language by using a known example.

4.2.3 Compute section. Once the domain theory has been specified, the user must give the COMPUTE statement which specifies the quantity to be computed by the program. For example, the user can write :

« COMPUTE ProposedMeetingBeginDates(m0) »

where m0 is a given meeting or :

« COMPUTE { ProposedMeetingBeginDates(m) | m ∈ M0 } »

where M0 is a given set of meetings or :

« COMPUTE WeakConflict(m0) »

and so on.

The DESCARTES system will then try to generate a program computing the desired quantity by using the knowledge expressed by the user-defined functions. The same item of knowledge (the same function definition) will generally be used differently depending on the program to be written.

This is not at all the case with programming languages (including functional languages) where functions are "frozen", i.e. can only be used for the "single" purpose they were defined for. It is therefore necessary to write as many different subroutines as there were "COMPUTE" commands although, all those subroutines would be based on the same item of knowledge. It is precisely that knowledge that constitutes the whole of the DESCARTES specification. In DESCARTES, the definitions of functions are, therefore, items of knowledge and not subroutines in the usual meaning of the word. In the generated program, the user-defined functions are adapted each time to the specific context in which they are used. This means much more than the abstraction mechanisms such as polymorphism that can be found for example in object-oriented languages. In our case, most of the time, functions are radically transformed, sometimes used "backwards" (inverted) or, even, completely left out if their calculation is useless in some particular context.

4.2.4 Replanning the meeting. Let us, in order to illustrate this very important feature, take the problem of dynamically replanning the meeting with as much flexibility as possible. Let us suppose that for a given meeting m0, the generated program has already obtained and stored in the database the set of potential beginning dates that we will call Y0. So, we have :

Y0 = ProposedMeetingBeginDates(m0)

We consider here that Y0 is our result : as previously, the problems of « meeting end date » and « meeting location » are not considered but it would be exactly the same. Now, we must take into account the fact that this result has been obtained for a given state B0 of the database, that is a given state of the entity sets and relationships of the data schema. In fact, the correct expression to define Y0 is :

Y0 = ProposedMeetingBeginDates(B0, m0)

Until now, we did not mention the B0 argument because, in many cases, the DESCARTES system can determine this kind of arguments by itself. They can therefore be omitted by the user. However, in order to retrieve referential transparency, they are determined and explicited by the system. Let us now, in order to simplify notations, call « f » the « ProposedMeetingBeginDates » function (so we have : Y0 = f(B0, m0)) and let dB0 be the variation of B0.

The solution consists in specifying the variation of « f » with regard to the variation of its argument. This takes the form of the following DESCARTES function, where Δ is the set symmetric difference operator, B a database and dB the variation of this database :

LET df(B,dB,m) : BASE, BASE, Meetings → Set(CalendarDates) :
/ returns the variation of « f » corresponding to the variation of B. (B Δ dB) represents in fact the new database*/*

f (B Δ dB, m) Δ f(B, m)

Then the COMPUTE statement just becomes :

COMPUTE Y0 Δ df(B0,dB0,m0)

All the function definitions we gave previously remain identical. But all these functions will be used differently if we add the « df » function definition to the specification and if we give the new COMPUTE statement. We have in fact specified a program which makes formal differentiation. The idea is that the sequence of unfolding, transformations and refinements which are going to apply on the initial definition of « df » will give a new expression of « df » which will probably avoid to compute the value of « f(B Δ dB, m) », that is the value of « f » for the new database. If this is the case, the generated program will only compute the variation of an already known set of potential beginning dates and will therefore be simpler than the program which computes the initial set of dates. As we will see later, formal differentiation is an optimization that we also uses inside the generator.

5. The DESCARTES system

The DESCARTES system is an automatic program generator which, fully automatically, generates programs written in conventional procedural languages. The first operational version of the system represents about one million of C lines. More than 80% of these lines have been generated by a dedicated code generator that we have built. The DESCARTES system has also been written in its own language in order to bootstrap the system, that is, to generate the system by using itself. This specification represents 50,000 lines. But the process of bootstrapping has not yet started.

DESCARTES is based on a transformational methodology. The system actually applies a long sequence of transformations, each transformation having the effect of "refining" the current state of the program being formed, i.e. making it increasingly concrete until the final program is obtained expressed in a target language. So, the system contains a large knowledge base of transformations, the aim of which being to reduce the program complexity. The computational complexity of the transformations themselves is once again first-order logic with fixed point.

This sequence of transformations has been structured by freezing the intermediate levels of abstraction of the program being formed and the three main levels are defined using intermediate languages COGITO, ERGO and SUM (which are internal and, therefore, transparent for the user). Each of these levels is represented in its own specific data model where specific integrity constraints are defined. Each of the subsystems used to move on from one level to another is structured in two parts. First, there are transformations that can be performed while remaining on the current level of abstraction (the aim being to work at the highest level of abstraction for as long as possible). Then, there are transformations which entail reducing the level of abstraction.

Thanks to this organization, the system architecture is modular and the system can therefore be written, debugged and maintained more easily. In particular, it is easier to improve the system by adding new transformations or optimizing existing transformations.

COGITO starts from the initial mathematical specification and deals with type verification, unfolding of functions, mathematical simplifications, normalizations and transformations, determination of loop domains, loop fusion and tupling, recursion removal of recursive systems and functions, tabulation and generation of the ERGO abstract algorithm. Some well-known program design tactics, such as formal differentiation apply as side effects of these transformations.

ERGO chooses concrete data structures to implement abstract ones, translates the operations on structures according to the choices it has made, optimizes the program by adding side effects, decides memory allocation and freeing, and generates the SUM program.

SUM deals with finite scalar types (coding, decoding and numbering), record implementation, complex assignment and comparisons, generates a dedicated dynamic memory allocation algorithm and translates the final program into a target language. Many target languages may be used. Indeed, the transformations are designed to remain independent from any target language for as long as possible. Practically all the SUM system has the purpose of obtaining a program expressed in an extremely poor, generic procedural programming language so as to be certain that all the control structures and data structures of conventional languages will be encompassed. It is the very last step in the SUM system that performs the task of translating this program expressed in the generic language into a program written in a specific target language. This translation is almost trivial as it consists in a mere syntactic process.

The system's main weak point is that, in its first operational version, it is somewhat limited compared with the language's possibilities of expression. In particular, it will only be able to deal with a limited second order (i.e., statically solvable) whereas the language can cope with order 2 in its full generality. For the same reasons, it will not be able to produce efficient programs for combinatorial problems, which are algorithmically difficult, and which, as we said previously, can very frequently be expressed as second order problems.

In the meeting scheduler problem for example, the system will probably have some difficulties with the « ProposedMeetingBeginDates » function because this function is in essence a combinatorial problem. In fact, in our specification we are looking for the set of potential beginning dates because it was sufficient in this particular case since the date of the end of the meeting is completely defined by the date of beginning and the time period which is constant. If it wasn't the case, the problem would be to find the set of intervals of dates which satisfy some constraints. So, instead of searching possible beginning dates inside a super-set of dates, which are first-order objects, the system would have to explore a super-set of intervals of dates, which would be huge and difficult to reduce. We are currently working on adding knowledge to reason on this kind of problem.

6. Applications and Results

DESCARTES has already been used in the context of an European ESPRIT project, to specify and generate a nuclear power plant emergency shutdown system called DARTS [15]. Starting from 600 DESCARTES specification lines, the system has produced a 12,000 line long code which passed official project tests.

We have also specified and generated a system called SACSO which schedules control operations for a fuel power plant. It advises the operator, in real time, about the best operating procedures to apply and observes the behaviour of physical processes in order to detect or anticipate failures. The main objectives of this system are to increase the safety of power plants by assisting control operators, to build on the know-how of control operators and to optimize the operation of the power plant. The DESCARTES specification is 2,500 lines long whereas the generated code is 43,000 C lines long. The total workload (design, development, test) for developing the SACSO application with DESCARTES has been 4 man-months whereas the cost of developing this application without using DESCARTES, i.e. by using more traditional approaches, has been estimated to 10 man-months. To the 6 man-months already saved must be added the saving which will be achieved on maintenance and modification which will certainly be much more important.

We have also specified, parts of the Electricité de France « Elaborate receipting », an application which deals with customer management, a program for planning core loading and unloading operations at nuclear power plants, a 0-order and a 1-order inference engines [8], a neurone network with learning, a bank automatic teller machine and also a bunch of small more « academic » examples in various fields. We are currently working in the fields of simulation, fault tree optimization and data validation and we go on exploring reactive systems.

In fact, DESCARTES is well suited to the manipulation of symbolic data. And it so happens that the ability to manipulate symbolic data is required in all fields. This symbolic data may concern nuclear power plants, customers, meters, neurone networks, uranium rods, inference rules, programs, bank automatic teller machines and so on, as applicable. DESCARTES can therefore be used just as well for management computing, industrial computing and scientific computing.

Obviously, DESCARTES may be much less suitable or, even, unsuitable for other types of use which may be required for the same application. In this case, the solution could be, for example, to interface codes: organize the cooperation of the code generated by DESCARTES on one part of the application with the code that is produced by other dedicated tools or hand-written.

7. Discussion

Programming languages can, in fact, be classified into a number of categories according to the “paradigm” they are based on. The purpose of the following considerations is to demonstrate the difference between DESCARTES and these various categories of languages and is not intended to be exhaustive.

Unlike procedural languages, there are, in DESCARTES, no variables, no pointers, no concrete data structures and, in fact, no reference to the computer’s memory. From the “processing” point of view, there are no assignments (and, therefore, no side effects), no loops and, indeed, no instructions (apart from the single “COMPUTE” instruction) and no time. In DESCARTES, there are abstract data structures (sets, lists and vectors), logical formulae of first order predicates and recursive functions. The level of abstraction is high : it is possible to perform reasoning on specifications in order to validate and/or generate programs automatically. Maintenance and modification are not restricted by data processing constraints (such as data structures).

Compared with procedural languages, functional languages unquestionably offer an important advantage: programs are shorter, more abstract and easier to understand. They make analysis and formal manipulation more convenient as they reside on a solid theoretical base (lambda calculus). They can also be implemented on parallel machines more immediately as there is no specific order in the definition of functions. In practice, there are two categories of functional languages. First of all, there are pure functional languages, i.e. with no side effect, such as FP [2], which really has all the advantages mentioned but “pays for them” with a certain degree of inefficiency on execution. Indeed, unlike DESCARTES, there is no reasoning or real prior transformation of programs. Some optimizations are carried out by the compiler but these optimizations are, essentially, based on abstract interpretation which means that the structure of programs is not called into question.

There are also impure functional languages (such as CAML) which contain procedural features (assignments) in order to make up for problems of inefficiency. Consequently, they lose their referential transparency (due to side effects) and thus find themselves on practically the same level as procedural languages with regard to readability and the possibility of reasoning on programs, on the one hand, as well as with regard to how easy it is to perform debugging, maintenance and modifications, on the other hand. Furthermore, these languages do not integrate the mathematical concepts of sets, vectors, quantifiers and set operators, etc., which are, as we have already seen,

very convenient and very practical to use when a problem is to be specified.

Finally, the functions defined consist in subroutines in the data processing meaning of the word and not, as in DESCARTES, in knowledge. This means that a given function can only be used for the purpose it was designed for and cannot in any way be used to derive functions suitable for each specific context.

Logical languages also provide many advantages compared with procedural languages: syntactic conciseness, powerful capability of semantic expression, management of nondeterministic cases, etc. In practice, for the sake of efficiency, procedural characteristics are often used, such as the "cut" instruction, to limit the search area (which, among other things, eliminates nondeterminism). Moreover, like those already mentioned, these languages do not integrate (at least, not really) the concepts of sets and set operations, etc., and cannot be used to write logical formulae in their full generality. (In Prolog, for example, there is restriction to Horn's clauses). Finally, whereas an algorithm dedicated to each specific context is automatically generated in DESCARTES, the logical languages underlying systems (interpreters or compilers) are generally limited to engines performing unification/resolution combinations.

Object-oriented languages offer a certain number of advantages such as data abstraction, encapsulation and inheritance. The modelling, based on the hierarchy of classes is often well suited to the description of data. However, the methods associated with these languages are, in most cases, written in procedural languages and therefore entail all the drawbacks of those languages.

It must be understood that whether these so-called "very high level" programming languages are functional, logical or object-oriented languages, they can be seen in two different ways and, to some extent, have two sides: one descriptive and the other operational. In functional languages, for example, a recursive function definition can be seen as an axiom and, also, as the means of computing the function. Similarly, in logical languages, a Prolog program, for example, can be considered either in terms of the logic of first order predicates or as a rewriting system. The first aspect corresponds to the specification aspect which is associated with declarative semantics defining all the valid formulae whereas the second corresponds to the resolution aspect which is associated with operational semantics describing the rules of calculation. For this second point of view, an order must be defined on the clauses.

All these languages share the common point that there is no prior reasoning and transformation of programs upstream of their compilation. Recursive functions remain recursive and logical programs are interpreted as bases for rewriting rules, etc. Whatever is written is executed and it is for this reason, in particular, that we refer to all these languages as programming languages.

On the contrary, in DESCARTES, what is written is not executed as a DESCARTES specification is not executable. The specification is not the program : as we said previously, it is a set of knowledge chunks from which the generator constructs a specific program that can be used to calculate the quantity specified in the "COMPUTE" section of the specification.

Specialized languages, such as man-machine interface languages, database query languages, reactive languages, etc. can, in their specific domain, reach a high level of declarativity since they can provide the concepts which are relevant to the field as primitives of the language. On the other hand, since they are dedicated, they generally have a limited expression power. For example, if the data-oriented functions we defined in pargraph 4.2 could quite easily be expressed by using a traditional query language such as SQL, it is not the case for user-defined functions which require, from a general point of view, a more powerful language and in particular the possibility to use recursion.

Finally, there are formal specification languages such as B [1] or VDM, which are, in most cases, like the DESCARTES language, i.e. of mathematical type, and which have been designed in the context of formal methods for program proving. In the B approach, as well as in the « algebraic specification » approach, the idea consists either in proving properties on the specification or in demonstrating that a refinement is satisfactory in the sense that it properly maintains the invariants of the specification. Proving is carried out using theorem provers which generally function interactively. But successive refinements are hand-written. There are no automatic generation. So, if the languages look quite similar, the aim is rather different and we think that although the problems of program generation and program proving are equivalent from a theoretical point of view, it is easier to automatically generate a program corresponding to a given specification than to prove that a given program constitutes a valid implementation of a given specification.

8. Conclusion

The results we obtained with DESCARTES show that it is possible, fully automatically, to generate efficient programs from high-level specifications, as long as the considered problems are conceptually simple. They also show that most of the problems we are generally faced with belong to this class of problems.

DESCARTES does not fundamentally change the V life cycle of software but, on the other hand, it places it at a higher level of abstraction. The stages of design, development, validation and maintenance become far less technical and this makes them much easier. Moreover this offers possibilities with regard to formal validation. Maintenance operations are also much easier since they are no longer frozen by data processing choices with regard to control structures and data structures. In short, programs are easier to write, shorter, clearer, more accurate (i.e. containing fewer bugs) and more open-ended. But the main advantage of DESCARTES is, of course, that development costs are drastically reduced.

Nevertheless, it must be noted that DESCARTES does more than just "improve" the current situation. The fact of raising the level of abstraction of the specification language means we can envisage writing programs which were previously not achievable. It is, however, easy to understand that, on the practical level, it is not possible to write absolutely any program in assembly language or, even, in C language. There is, in fact, a "practical" complexity threshold beyond which it becomes immeasurably difficult to write a program which, in any case, would be impossible to maintain.

References

1 Abrial J.R, 1996. The B Book. Cambridge University Press

2 Backus J.W., 1978. Can programming be liberated from the Von Neumann Style ? : A functional style of programming and its algebra of programs. Aout. ACM 21, 8.

3 Burstall R.M. et Darlington J., 1977. A transformation system for developing recursive programs. J; ACM 24, 1. Janvier. pp 44, 67.

4 Feather M-S., 1987. A Survey and Classification of some Program Transformation Approaches and Techniques. Program Specification and Transformation. L.G.L.T. Meertens (Ed.). Elsevier Science Publishers. North-Holland. IFIP.

5 Ginoux B. et Lagrange J-P. 1989a. An Expert System Approach To Program Synthesis, AAAI Spring Symposium, Séries 1989: Artificial Intelligence and Software Engineering, Standford, Ca. USA.

6 Ginoux B. et Lagrange J-P. 1989b. Synthesis of Simple Programs which handle Complex Data, IJCAI'89 Workshop on Automating Software Design, Detroit, Mich. USA.

7 Ginoux B., 1991. Génération automatique d'algorithmes par système expert à partir de spécifications déclaratives de haut niveau. Le système COGITO. Phd Thesis. Paris 9.

8 Ginoux B., 1994. How to Automatically Generate an Inference Engine from Declarative Specifications. In Proceedings of the 10th Canadian Conference on Artificial Intelligence. Banff. May 1994.

9 Ginoux B., 1997. DESCARTES in 32 questions, 32 answers and 32 pages. Collection de notes internes de la Direction des Etudes et Recherches d'EDF. Ref : 97NJ00027. Avril 1997.

10 Kant E. et Barstow D.R., 1981. The refinement paradigm : The interaction of coding and efficiency knowledge in program synthesis. IEEE Trans. Softw. Eng. 7, 458-471.

11 Kant E. et Newell A., 1983. An automatic algorithm designer : An initial implementation. In Proceedings of the National Conference on Artificial Intelligence. Août. pp 177, 181.

12 Lowry M-R, 1989. Algorithm Synthesis through Problem Reformulation. PhD Thesis. Stanford University.

13 Manna Z. et Waldinger R., 1978. DEDALUS : the DEDuctive Algorithm Ur-synthesizer. In Proceeding of National Computer Conference. vol 47 AFIPS Press, Reston, Va. pp 683, 690.

14 Partsch H. 1990. Specification and Transformation of Programs. Springer Verlag. 1990

15 Pierre L., Boudaoud A. et Ginoux B., 1993. An Expert System for Nuclear Plant Safety Control Software. Proceedings of the International Conference on Expert Systems Applications for the Electric Power Industry. Phoenix. Décembre 1993.

16 Rich C. et Waters R-C., 1986. Readings in Artificial Intelligence and Software Engineering. Morgan Kaufmann Publishers, Inc. Los Altos. Californie.

17 Smith D-R., 1990. KIDS: A Semiautomatic Program Development System. IEEE transactions on software engineering. Vol. 16. No. 9. Septembre