SUGGESTIONS FOR COMPOSING AND SPECIFYING PROGRAM DESIGN DECISIONS

M. Sintzoff(*)

Abstract : It is proposed to express program designs by hierarchical specifications of design decisions. A case study of program construction is presented to substantiate the proposal. Composition rules based on logic are given for these hierarchical specifications. Advantages and disadvantages of the suggestions are assessed

(*) Philips Research Lab., av. Van Becelaere 2, Bte 8, B-1170 Bruxelles, Belgique.

1. INTRODUCTION AND MOTIVATIONS.

We view program design as a constructive intellectual activity. Such an activity should be organized and expressed using the successful principles laid down for composing and communicating programs.

Many styles are used to express software design : sequences of successive versions of specifications or programs are formulated in detail ; general and intuitive guidelines are presented to define methods of development ; single decisions are characterized as applications of formal deductive rules which are strongly associated with one specific technique of programming. As a consequence, the scientific study of the motivation, the essence and the composition of the decisions is made difficult.

We propose here to express design decisions by specifying their logical and qualitative effects precisely, but independently of their implementation on fully detailed program descriptions. Moreover, these specifications should be composed hierarchically : this would help to organize design methods by successive levels of importance, and to ensure the adequacy of each subdecision and of the interfaces between them. The effective realization of design decisions and the explicit result of their application should remain hidden, but one should be able to derive these implementations systematically, if and when needed. By abstracting from irrelevant details of program descriptions and by avoiding flat structure or vagueness, such an approach should help in establishing a common, well-defined framework for many useful software design methods, which are still formulated as general, intuitive advices or as successive manipulations of program descriptions. It would thus enhance the systematic composition and modification of program designs, and facilitate deeper understanding.

We shall first give an informal presentation of the main lines of the ideas. In order to suggest how they could be followed, we then develop a case study. After, we present a more precise framework based on logic. Related works, and pros and cons are discussed at the end.

2. SPECIFICATION AND COMPOSITION OF DECISIONS

We first give an informal development of the main lines of the suggestions. Let us assume that a given stage in a software design process is represented by a program description, which consists of (sub)problem specifications and (sub)program constructs. A design decision then defines a correspondance between two such design stages. We introduce below rules for specifying design decisions and for composing them. These rules are given precise meanings in Section 4.

SPECIFICATION

We specify a design decision by stating an applicability predicate (antecedent) and a predicate expressing the qualitative effect desired (consequent). Here is a too obvious illustration :

Dec : Replace recursion by iteration

Antec : the given program description is a linear recursive function

Conseq : the new version is equivalent to the given one and is an iteration using a fixed number of new auxiliary variables.

A specification of a decision defines a relation between program descriptions S and T : S verifies the antecedent, and S and T verify the consequent. The heart of the matter is to abstract relevant properties in the antecedent and resulting ones in the consequent.

We do not assume that the antecedent is a sufficient condition for the existence of a new program description verifying the consequent. To accommodate case analysis, a decision can also be specified by a set of pairs <antecedent, consequent > .

COMPOSITION

We merely lift at the level of design decisions the basic compositions rules successfully used at the algorithmic level. The specific notations used below for the various compositions can easily be modified, if so desired.

Serial composition : (Dec1; Dec2)

Case analysis : [Pred1 $\rightarrow$ Dec1 | Pred2 $\rightarrow$ Dec2]

The cases are distinguished by predicates on program descriptions.

Definition : Id = Dec spec

The identifier Id is a synonymous to a given decision specification which may well be a composed one.

Induction : D = F(D)

This defines decision D as the limit of a series of stronger and stronger ones :

$$D^0 = \text{Undefined decision}$$
$$D^{n+1} = F(D^n)$$

We assume F is a monotone mapping between decisions, for the set-inclusion ordering between decisions, viewed as relations.

Hereafter, in addition to the antecedent and consequent of each decision , we may add hints to indicate how the decision was conceived, and we mention its component subdecisions as the case may be .

3. CASE STUDY

The example has been selected for various reasons. Firstly, it is not usually considered as an illustration of program transformations but rather of program "formation". Secondly, the way the design decisions are organized, chosen or precised leaves room for improvement although its author already presents a quite helpful explicitation. Thirdly, the design has been invented already : we do not tackle here the problem of discovery.

MAXIMAL LENGTH OF ASCENDING SUBSEQUENCES

The considered problem is the third one in Dijkstra (1980). Our intent is to obtain hierarchical specifications of the very design decisions explained there. Since our main topic is composition and implementation independence, we content ourselves with semi-formal expressions, hopefully sufficiently precise.

The program description given initially consists of the two following assertions defining the problem.

Preassertion P(A) : A is a non-empty sequence.

Postassertion Q(A,k) : k is the maximal length of the upsequences in A ; u is an "upsequence in" v iff u is obtained from v by removing some or zero elements from v and by retaining the remaining elements in their order in v.

Design = (Dec I; Dec II)

Dec I : Derive the postassertion by iteration

Antec : P(A) and Q(A,k) are given

Conseq : A sequence $s=(s_1,\ldots,s_M)$, a predicate H, and terms U_1,U are found such that, writing H_n for $H(s_n,k_n)$, U_1 for $U_1(s_1)$, and U_{n+1} for $U(s_{n+1},k_n)$,

$$P(A) \Rightarrow H_1 \wedge (k_1=U_1), \quad H_M \wedge (k_M=k) \Rightarrow Q(A,k) \ ,$$
$$H_{n+1} \equiv H_n \wedge (k_{n+1} = U_{n+1})$$

Moreover, each s_n is defined in terms of A.

Compos : (Dec 11; Dec 12) ; Dec 2

Dec 11 : Choose a sequence

Antec : Antec of Dec 1, which includes the data A

Hint : Follow the sequence A from left to right, viz. reconstruct it naturally.

Conseq : s is the sequence $(s_1,\ldots,s_N)$ where $s_i=A(:i)=(A_1,\ldots,A_i)$

Dec 12 : First version of the iterative predicate

Antec : Dec 11. (This means the antecedent and the consequent of Dec 11 are known).

Hint : Since $H(A(:N),k) \Rightarrow Q(A(:N),k)$, weaken $Q(A(:N),k)$ to $Q(A(:n),k_n)$, as a tentative version I of H.

Conseq : $I_n \equiv I(A(:n),k_n)$
$\equiv$ (k_n is max. lgth of upseq. in A(:n))

Dec 2 : Find an inductive definition for I

Antec : Dec 12

Conseq : U_1,U s.th. $P(A) \Rightarrow I_1 \wedge (k_1=U_1)$
and $I_{n+1} \equiv I_n \wedge (k_{n+1}=U_{n+1})$

where $U_{n+1}=U(A(:n+1),k_n)$ and $U_1=U_1(A(:1))$.

Compos : (Dec 21; Dec 22; Dec 23); D3

Dec 21 : Find U_1

Antec : Antec of Dec 2

Hint : The max. lgth of any upsequence of a sequence of one element is one.

Conseq : $U_1=1$.

Dec 22 : First version of U_{n+1}

Antec : Antec of Dec 2

Hint : Since k_n and k_{n+1} are the max. lgths of upseq. in A(:n) and A(:n+1) respectively, we have $k_n \leqslant k_{n+1} \leqslant k_n+1$.

Conseq : $U_{n+1}=(C \rightarrow k_n+1 \mid \neg C \rightarrow k_n)$, where C is an unknown condition on A(:n+1) and k_n. (We use conditional expressions with their usual meaning).

Dec 23 : First version of C

Antec : Dec 22

Hint : Some longest upsequence in A(:n) can be extended in A(:n+1) iff A_{n+1} is not smaller than the minimum m_n of the elements ending the upsequences of length k_n in A(:n).

Conseq : $C \equiv m_n \leqslant A_{n+1}$, given an additional predicate

$$J_n \equiv J(A(:n),k_n,m_n)$$
$$\equiv (m_n \text{ is min. last el. of upseq. of lgth } k_n \text{ in } A(:n))$$

Dec 3 : Find an inductive definition of J

Antec : Dec 23

Conseq : V_1,V s.th. $J_1 \equiv I_1 \wedge (m_1=V_1)$

$$J_{n+1} \equiv I_{n+1} \wedge J_n \wedge (m_{n+1}=V_{n+1})$$

where $V_{n+1}=V(A(:n+1),k_{n+1},m_n)$ and $V_1=V_1(A(:1),k_1)$

Compos : (Dec 31; Dec 32; Dec 33); Dec 4

Dec 31 : Find V_1

Antec : Antec of Dec 3

Hint : The min. last el. of longest upseq. in a sequence of one element is this element.

Conseq : $V_1=A_1$

Dec 32 : First version of V_{n+1}

Antec : Antec of Dec 3

Hint : m_{n+1} is either A_{n+1} or m_n. It equals m_n iff A_{n+1} is strictly smaller than the minimum m'_n of the elements ending the upsequences of length (k_n-1) in A(:n).

Conseq : $V_{n+1} = ($ $m_n \leqslant A_{n+1} \rightarrow A_{n+1}$

$|k_n=1 \wedge A_{n+1}<m_n \rightarrow A_{n+1}$

$|k_n>1 \wedge (m'_n \leqslant A_{n+1} < m_n \rightarrow A_{n+1}$

$|A_{n+1} < m'_n \rightarrow m_n$))

given another additional predicate, defined for $k_n > 1$, by

$$J'_n \equiv J'(A(:n),k_n,m'_n)$$

$\equiv$ (m'_n is min. last el. of upseq. of lgth k_n-1 in A(:n))

$$\equiv J_n(A(:n),k_n-1,m'_n) \equiv f(J_n)$$

Dec 33 : Final version of the iterative predicate

Antec : Conseq. of Dec 32

Hint : To define J_n inductively, we needed J' if $k_n > 1$. Now, $J'_n=f(J_n)$ for some substitution f. Hence, to define J'_n, we need $J''_n=f2(J_n)=J_n(A(:n)$, k_n-2, $m''_n)$ when $k_n > 2$. Thus, we need $J_n(A(:n),k_n-p,m_n^{(p)})$ for $p=0,\ldots,k_n-1$.

Conseq: $H_n=I_n \wedge \forall_p [0 \leqslant p \leqslant k_n-1 \Rightarrow J_n(A(:n),k_n-p,\ m_n^{(p)})]$

Dec 4 : Find an inductive definition of H

Antec : Conseq. of Dec 33

Hint : Omitted in Dijkstra (1980). Generalize V_{n+1}, from Conseq of Dec 32, according to the hint of Dec 33.

Conseq : Omitted here. The interesting part is

$$m_n^{(p+1)} \leqslant A_{n+1} < m_n^{(p)} \Rightarrow m_{n+1}^{(p)} = A_{n+1}$$

Dec II : Implement a search efficiently

Antec : Conseq. of Dec I, viz. of Dec 4, implies a search : in the increasing sequence $(\ldots,m_n^{(p)},\ldots,m'_n,\ m_n)$, to find the smallest $m_n^{(p)}$ such that $A_{n+1}<m_n^{(p)}$.

Conseq : This search is implemented by dichotomy, which is of logarithmic complexity.

Reflections

Two surprises must be reported. First, the main structure of the original design in Dijkstra (1980) boils down here to two main steps, Dec I and Dec II, with just one crucial subdecision, Dec 33. Second, all the decisions and subdecisions, once clearly identified and composed, appear to be rather straightforward and easy: the above development could even seem boring because of the obviousness, the weakness or the repetitiousness of decisions. To contrast, the first reading of Dijkstra (1980) gave the impression that decisions were tricky, or inspired, as well as varied.

The original design description mixes three levels, namely the program descriptions, the definition of the decisions, and their discovery. Moreover, it gives priority to the clarity and the structure of the program descriptions, viz. assertions and algorithmic constructs, whereas the two higher levels are described *operationally*. Here, we have separated the concern about program descriptions from the one about decisions, and concentrated on the latter Yet, we still mixed the specification of the final, complete decisions with their derivation by successive approximations. To separate these two latter aspects would further clarify the design . For instance, instead of obtaining the complete predicate H by induction on the structure of the predicates J,J' in Dec 33, we could derive Dec 4 by *induction on the decision* Dec 32 itself ; as a by-product, the hint for Dec 4 would become redundant. Also, the wondrous accident thanks to which Dec I can be followed by the optimization Dec II, would be better analyzed and anticipated.

The design specification above uses definitions in the program descriptions, instead of, a.o., statements and assignments. The latter ones, although they do have well-defined semantics, have been abandoned here : using them, we were unable to specify subdecisions independently of one another, because the assignment requires carrying a global logical state; see Dec 21,22.

An alternative design for the same problem is given by Broy et al. (1979), in terms of successive transformations of specifications and algorithms. The two designs are reconstructed in Finance (1979), § 2.1.2, by use of logical deductions. Both works present a lucid analysis of the critical choices made for the sequence guiding the iteration, here Dec 11.

ABSTRACTING A METHOD FROM EXAMPLES

Examples are often used to suggest the nature of an underlying method. It is better, in this case, to specify the method directly, by parametrizing the program descriptions. This would make the method available for other applications, and would allow to separate the specification of the method from that of its application on a given problem. As an illustration, here is a sketch of the method behind the example above.

Dec A : Derive the result by iteration

Antec : $P(X)$ and $Q(X,Y)$ are respectively the initial and final assertions defining a problem ; the type of initial informations X contains that of sequences.

Conseq : A sequence $s=(s_1,\dots,s_M)$ and a predicate H are obtained, such that each s_n is defined from X and that

$$P(X) \Rightarrow H_1 \wedge (r_1=U_1),\ H_M \wedge (r_M=Y) \Rightarrow Q(X,Y)$$

$$H_{n+1} \equiv H_n \wedge (r_{n+1}=U_{n+1})$$

where H_n, U_1 and U_{n+1} denote $H(s_n,r_n)$, $U_1(s_1)$ and $U(s_{n+1},r_n)$.

Compos : (Dec A1; Dec A2; Dec A3)

Dec A1 : Choose a sequence

Antec : Antec of Dec A, with data X .

Conseq : $s=(s_1,\dots,s_M)$ such that $s_{n+1}=\text{concat}(s_n,g(n+1,X))$ and $s_1=g(1,X)$

Dec A2 : Choose an iterative predicate

Antec : Dec A1

Conseq : H s.th. $P(X) \Rightarrow \exists r_1 : H_1$, and $H(s_M,Y) \Rightarrow Q(X,Y)$

Dec A3 : Find an inductive definition

Antec : Dec A2

Conseq : E_1,E s.th. $P(X) \Rightarrow H_1 \wedge (r_1=E_1(s_1))$

and $H_{n+1} \equiv H_n \wedge (r_{n+1}=E(s_{n+1},r_n))$

In the problem of upsequences, Dec 2, Dec 3 and Dec 4 are all applications of Dec A3 , and they correspond to the successive approximations Dec 12, Dec 23, and Dec 33, respectively, of Dec A2.

OTHER EXERCISES

The example and the method above have been chosen also because they can be discussed in a short space. Yet, other case studies have been looked at too, viz. : programming by stepwise refinement, design guided by input- and output-structures, and methods to derive parallel or distributed programs. And there are of course many other useful methods and examples to consider.

4. TOWARDS AN ALGEBRA OF PROGRAMMING

We present a small logical system as a precise framework for the informal proposals made in Section 2.

Basic components

We assume that program descriptions S,T,..., are expressed by first-order predicates, following Manna (1969) for example; if second-order predicates are needed for this, then increase all orders by one. Antecedents $A,A_i\ldots$, and consequents $C,C_i\ldots$, define properties of program descriptions and thus are expressed by second-order predicates ; A and C have respectively one and two free variables. Now a pair $<A,C>$ of antecedent A and consequent C defines as follows a relation between program descriptions :

$$S <A,C> T \text{ iff } A(S) \wedge C(S,T)$$

A decision is expressed by a finite set of such pairs :

$$D = \sum_i \langle A_i, C_i \rangle \quad \text{with } A_i \wedge A_j = \underline{\text{false}} \ (i \neq j)$$

It defines the relation

$$SDT \equiv D(S,T) \equiv \bigvee_i S \langle A_i, C_i \rangle T \equiv \bigvee_i A_i(S) \wedge C_i(S,T)$$

Serial composition

$$(\sum_i \langle A_i, C_i \rangle \ ; \ \sum_j \langle A'_j, C'_j \rangle) = \sum_i \sum_j (\langle A_i, C_i \rangle \ ; \ \langle A'_j, C'_j \rangle)$$

$$(\langle A,C \rangle \ ; \ \langle A',C' \rangle) = \langle A'',C'' \rangle$$

$$\text{where } A''(S) \equiv A(S) \wedge \forall R \, [C(S,R) \Rightarrow A'(R)]$$

$$C''(S,T) \equiv \exists R \, [C(S,R) \wedge C'(R,T)]$$

Case analysis B, B_i..., are predicates on program descriptions.

$$[B1 \rightarrow D1 \mid B2 \rightarrow D2] = [B1 \rightarrow D1] + [B2 \rightarrow D2]$$

$$[B \rightarrow \langle A,C \rangle] = \langle B \wedge A, C \rangle$$

$$\text{hence} \quad [B \rightarrow [B' \rightarrow D]] = [B \wedge B' \rightarrow D]$$

The generalization to any number of alternatives is obvious. A non-deterministic decision is made deterministic by applying the following rule iteratively :

$$\langle A,C \rangle + \langle A',C' \rangle = \langle A \wedge \neg A', C \rangle + \langle \neg A \wedge A', C' \rangle + \langle A \wedge A', C \vee C' \rangle$$

Inductive definitions

Assume a decision D is defined as the smallest relation, or set, X verifying

$$\forall \, S,T \, [C((S,T),X) \Rightarrow X(S,T)]$$

The predicate C expresses a relation between pairs (S,T) of program descriptions and decisions X, such that, for all S,T,X,Y,

$$(X \subseteq Y) \wedge C((S,T),X) \Rightarrow C((S,T),Y)$$

Following Feferman (1977), p. 923, D can be defined using ordinals α and approximations D_α :

$$D = D_{\alpha 0} \text{ where } D_{\alpha 0} = D_{\alpha 0+1}$$

$$D_\alpha(S,T) \equiv C((S,T), \bigcup_{\beta<\alpha} D_\beta)$$

As illustration, take

$$C((S,T),X) \equiv B(S) \wedge G(S,T) \vee \neg B(S) \wedge \exists X' : R(X,X') \wedge X'(S,T)$$

for a given decision G and a given relation R between decisions. Then

$$D_\alpha(S,T) \equiv B(S) \wedge G(S,T) \vee \neg B(S) \wedge \exists X' : R(\bigcup_{\beta<\alpha} D_\beta, X') \wedge X'(S,T)$$

For a sufficiently simple R, this can be expanded linearly.

Comments.

In the algebra of programs presented by Backus (1978), the functionals are constructive maps between algorithms ; here, decisions are taken as specifications of relations between program specifications. The formalization above has not yet been strictly followed for any of the examples or methods mentioned before ; the resulting design descriptions could be disappointing. Remember, D. Scott wrote that formalization is an experimental science, and Lakatos (1976) warns against sterile formalizations.

We have used logic as foundation, and ignored the functional aspects too much : we should include some functional calculus. Also, we used a straightforward system of finite types, thereby ensuring a safe stratification. But this hinders the use of common composition rules for decisions to obtain decisions as well as for the obtained decisions ; a self-applicable, type-free system could be preferable.

5. DISCUSSION AND RELATED WORK

There exist numerous related works and viewpoints, although nowhere is the attention concentrated on precise rules for the specification and the composition of design decisions.

The principle of hierarchical design by use of abstract machines, a.o. Zurcher and Randell (1968) and Dijkstra (1972), has yielded "system design methods" expressed by informal know-how, techniques for module specification, or "program design languages" in which one may express successive program descriptions or manipulate modules. Yet, one needs to precisely communicate high-level programming decisions ; drawings, hints or rewriting rules are not enough. Manna and Waldinger (1979) do study the nature of important programming decisions, in the context of heuristic automation, but give priority to the specific definition of particular strategies. Jette (1979) presents suggestions similar to ours, but describes decisions by combinations of abstract and concrete specifications ; this facilitates implementation and hampers a common analysis of different methods.

A large "metacurrent" has appeared during the last years. In Feather (1979), metaprograms are hierarchical compositions of Darlington-Burstall transformation decisions ; they use patterns and rewriting rules on recursive functions. Balzer (1979) introduces "development plans" with a similar intent. Schwartz (1979) and Blikle (1979) propose to express their methods by transformations of specifications mixed with program components, and they study the logical effects of these specific transformations. In other works on program transformation systems, such as Standish et al. (1976), Bauer et al. (1979), and Arsac (1979), the initial effort was put on the building of basic and effective transformation rules, whereas growing attention is now paid to their logical interpendence and structure. Metalevels have been introduced in the context of logical proof systems : see the strategies and tactics in Gordon, Milner et al. (1978), the metatheories in Weyrhauch (1978), and the metafunctions in Boyer and Moore (1979). These concepts help to specify and structure proof steps, but are tied to specific proof systems and do not directly concern programming decisions. In a similar vein, let us recall the growing rôle of "metaknowledge" for heuristic automation ; see Feigenbaum (1977). In particular, Green and Barstow (1978) represent programming knowledge by sets of production rules for use in a synthesis system.

The formalization of design decisions has been studied mainly for individual, concrete transformation rules : see, a.o., Cooper (1969) and Huet and Lang(1979). However, Bentley and Shaw (1979) do abstractly specify a decision to efficiently represent data structures. Finance (1979) proposes formalized compositions of abstract design decisions, albeit for specific methods.

The main problem in the approach we suggest is to discover adequate hierarchical specifications of design methods and of their application. The following tasks are indeed difficult : to abstract the essential specifications of a given decision ; to choose its component subdecisions ; to control and to express successive approximations or backtracking when applying a given method to a given problem ; to ensure or to prove the correctness of a proposed design ; to decide on a sufficiently lucid style of program description. The later issue brings in the problem of how to implement design decisions on the basis of their specifications, in other words how to derive metaprograms for program design by human beings or automata. One way is to map the abstract specification and composition rules into implementable ones, but this amounts to solving a "metadesign" problem.

Our proposals can thus be seen as too general and overambitious. Yet, the following benefits have already been felt in a number of case studies, including our own previous work. The structure and the expression of design methods can be improved significantly. Their application on specific problems can be better thought out. The deep structures of methods become more transparent and may be compared more objectively. Program development does not have anymore to be presented by chains of successive versions interspersed with more or less well-founded hints, or by successive manipulations of formal texts. Designs can be modified more systematically, and libraries or families of designs become less unreasonable.

Acknowledgements. The suggestions presented here have been sharpened by discussions at meetings of the French working party Anna Gram (analyse et programmation), IFIP WG2.3, IFIP WG2.1, and the workshop on program transformation systems.

REFERENCES

Arsac, J.J., Syntactic source to source transforms and program manipulation, Comm. ACM 22, 43-53(1979).

Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Comm. ACM 21, 613-641 (1978).

Balzer, R., Transformational implementation : an example, Information Sci. Inst., Univ. South. Calif., 1979.

Bauer, F. L. et al., Systematics of transformation rules, Proc. Summer School Program Construction 78, LNCS 69, Springer, Berlin, 1979.

Bentley, T. L., and M. Shaw, An Alphard specification of a correct and efficient transformation on data structures, Proc. Conf. Specifications of Reliable Software, IEEE, 1979, 222-237.

Blikle, A., On the development of correct programs with the documentation, M79/25, Electronics Res. Lab., Univ. Calif., Berkeley, 1979.

Boyer, R. S., and J. S. Moore, Metafunctions : proving them correct and using them efficiently as new proof procedures, Computer Sci. Lab., SRI, Menlo Park, and Systèmes et Automatique , Univ. Liège, 1979.

Broy, M., et al. Methodical solution of the problem of ascending subsequences of maximum length within a given sequence, Information Processing Letters 8, 224-229(1979).

Cooper, D. C., Program scheme equivalences and second-order logic, in : B. Meltzer and D. Michie (eds.), Machine Intelligence, vol. 4, pp. 3-15, University Press, Edinburgh, 1969.

Dijkstra, E. W., Notes on structured programming, in : O. J. Dahl, E. W. Dijkstra and C.A.R. Hoare, Structured Programming, Academic Press, London, 1972, pp. 1-82.

Dijkstra, E. W., Some beautiful arguments using mathematical induction, Acta Informatica 13, 1-8(1980).

Feather, M. S., A system for program transformation, Working Paper, Univ. of Edinburgh, 1979.

Feferman, S., Theories of finite type related to mathematical practice, in : J. Barwise (Ed.), Handbook of Mathematical Logic, North-Holland, Amsterdam, 1977, pp. 913-971.

Feigenbaum, E. A., The art of artificial intelligence, Proc. 5th Int. Jt. Conf. Artif. Intell., Carnegie-Mellon Univ., Pittsburgh, 1977, pp. 1014-1029.

Finance, J. P., Etude de la construction des programmes : méthodes et langages de spécification et de résolution de problèmes, Thèse Doct. Sci., Univ. Nancy I, 1979.

Gordon, M., R. Milner et al., A meta-language for interactive proof in LCF, Proc. Conf. 5th Symp. Principles of Programming Languages, 1978, 119-130.

Green, C., and D. Barstow, On program synthesis knowledge, Artificial Intelligence 10 (1978) 241-280.

Huet, G., and B. Lang, Proving and applying program transformations expressed with second-order patterns, Acta Informatica 11, 31-55 (1978).

Jette, C. J., Heuristic control of design-directed program transformations, Proc. AFIPS Natl. Conf., Vol. 48 (1979), 1071-1077.

Lakatos, I., Proofs and Refutations : The Logic of Mathematical Discovery, University Press, Cambridge, 1976.

Manna, Z., Properties of programs and first-order predicate calculus, J. ACM, 16, 244-255 (1969).

Manna, Z., and R. Waldinger, Synthesis : dreams $\Rightarrow$ programs, IEEE Trans. Software Eng. SE-5(1979) 294-328.

Schwartz, J. T., Correct-program technology, Proc. School 77 Fondements de la Programmation, IRIA, Le Chesnay, 1979, 229-269.

Standish, T. A., et al., Improving and refining programs by program manipulation, Proc. ACM Natl. Conf. 1976, 509-516.

Weyrauch R., Prolegomena to a theory of formal reasoning, Report CS-78-687, Computer Sci. Dept., Stanford Univ., 1978.

Zurcher, F. W., and B. Randell, Iterative multi-level modelling : a methodology for computer system design, Proc. IFIP Congress 68, North-Holland, 1969, pp. 867-871.