A survey of formal software development methods[1]

Donald Sannella

Department of Artificial Intelligence and Department of Computer Science
University of Edinburgh

July 1988

1 Introduction

1.1 Scope

This paper is a survey of the current state of the art of research on methods for formal software development. The scope of this paper is necessarily restricted so as to avoid discussion of a great many approaches at a very superficial level. First, although some of the ideas discussed below could be (and have been) applied to hardware development as well as to software development, this topic will not be treated here. Second, the special problems involved in the development of concurrent systems will not be discussed here although again many of the approaches mentioned below could be applied in this context. Third, no attempt is made to treat programming methodologies such as Jackson's method and program development systems such as the MIT Programmer's Apprentice which are not formally based. Finally, this survey does not claim to be fully exhaustive although an attempt has been made to cover most of the main approaches. Many of the technical details of the different approaches discussed have been glossed over or simplified for the purposes of this presentation; full details may be found in the cited references.

1.2 Software development: from requirements to program

This section presents a general picture of the process by which a software system may be developed by formal methods from a specification of the requirements the system must fulfill. This overall picture will be useful in discussing the wide variety of formal program development approaches available, as the different approaches attack different aspects of the problem.

Let SP_0 be a specification of the requirements which the software system is expected to fulfill, expressed in some formal specification language SL. (The process, sometimes known as *requirements engineering*, by which such a precise formal specification is obtained starting from the informal and often vague requirements of the customer will not be discussed here although it is acknowledged that this problem is by no means a trivial one.) This specification constrains the input/output behaviour of the system in some fashion. It may

[1]This paper was written under contract to GEC Research as a part of the Alvey-sponsored ISF (Integrated Systems Factory) Study.

also place constraints on the time and space resources available, although most of the approaches to be discussed are unable to deal with constraints of this kind. The ultimate objective is a program P written in some given programming language PL which satisfies the requirements in SP_0.

The usual way to proceed is to construct P by whatever means are available, making informal reference to SP_0 in the process, and then verify in some way that P does indeed satisfy SP_0. The only practical verification method available at present is to test P, checking that in certain selected cases the input/output relation it computes satisfies the constraints imposed by SP_0. This has the obvious disadvantage that (except for trivial programs) correctness of P is never guaranteed by this process, even if the correct output is produced in all test cases. On the other hand, methods for automatically generating test cases which are likely to expose problems are available, see for example [BCFG 86].

An alternative to testing is a formal proof that the program P is correct with respect to the specification SP_0. However, after two decades of work on program verification it now seems to be more or less widely accepted that this will probably never be feasible for programs of realistic size. At the very least, initial hopes for a system capable of automatically generating proofs of program correctness are regarded as unrealizable.

Most recent work in this area has focused on methods for developing programs from specifications in such a way that the resulting program is guaranteed to be correct by construction. The main idea is to develop P from SP_0 via a series of small refinement steps, inspired by the programming discipline of stepwise refinement. Each refinement step captures a single design decision, for instance a choice between several algorithms which implement the same function or between several ways of efficiently representing a given data type. This yields the following diagram:

$$SP_0 \leadsto SP_1 \leadsto SP_2 \leadsto \cdots \leadsto P$$

If each individual refinement step ($SP_0 \leadsto SP_1$, $SP_1 \leadsto SP_2$ and so on) can be proved correct, then P itself is guaranteed to be correct. Each of these proofs is orders of magnitude easier than a proof that P itself is correct since each refinement step is small. In principle it would be possible to combine all the individual correctness proofs to yield a proof of the correctness of P with respect to SP_0, but in practice this would never be necessary.

When this approach is used to develop large and complex programs, the individual specifications SP_n become large and unwieldy. This is particularly true as n increases. As a consequence the proofs of correctness of refinement steps become difficult, even though the creative leap involved in a single step remains the same. The solution to this problem adopted by some formal program development approaches is to allow specifications to be decomposed into smaller units during the development process. These smaller specifications may then be refined independently of one another. A simple development involving only two decompositions and six refinement steps would then give the following diagram:

$$SP_0 \rightsquigarrow \left\{ \begin{array}{lll} SP_1 & \rightsquigarrow & SP_2 \rightsquigarrow P \\ \oplus & & \\ SP'_1 & \rightsquigarrow & \left\{ \begin{array}{l} SP'_2 \rightsquigarrow P' \\ \otimes \\ SP''_2 \rightsquigarrow P'' \end{array}\right. \end{array}\right.$$

Here $\oplus$ and $\otimes$ are intended to denote arbitrary specification-building operations, and P, P' and P'' are program modules. The program $P \oplus (P' \otimes P'')$ is guaranteed to be correct with respect to the specification SP_0 provided each of the individual refinement steps can be proved correct. This assumes that $\oplus$ and $\otimes$ can be used for combining program modules as well as specifications, or at least that operations corresponding to $\oplus$ and $\otimes$ exist on the level of program modules.

It is important to note that the neat and tidy diagrams above are not intended to suggest that the formal development of realistic programs proceeds without backtracking, mistakes and iteration. Formal program development approaches do not claim to remove the possibility of unwise design decisions. But once a program is obtained by means of some sequence of refinement steps then a diagram like the one above which omits all the blind alleys may be drawn. Then, provided all the required correctness proofs have been carried out, correctness of the resulting program is guaranteed. Most approaches do not require the correctness proofs to be carried out when the individual refinement steps are first proposed; the proof obligations can be recorded for later, to be discharged after it becomes clear if the refinement step will lead to (a module of) the final program.

There are a number of questions which the general view of formal program development presented above leaves completely open, and which any particular approach which fits into this mould must answer. These include:

- What is the specification language SL?
- What specification-building operations such as $\oplus$ and $\otimes$ above are available in SL, if any?
- What is the programming language PL?
- What is the relationship between PL and SL?
- What does "refinement" mean and under what circumstances is a refinement step correct?
- How is the transition between SL and PL made?
- What is the relationship between refinement and decomposition?
- Does refinement of programs $P \rightsquigarrow P'$ make sense? What is the relation between this and refinement of specifications $SP \rightsquigarrow SP'$?

- Does the refinement process itself have the status of a formal object subject to analysis and manipulation (see [SS 83])?

It is very important to the mathematical soundness of the particular program development approach under consideration that some of these questions (at least the first seven) be given very precise answers. For example, whatever specification language *SL* is used, it is important that it be given a complete formal semantics. Without such a semantics, specifications have no precise meaning and so no formal proofs can be undertaken. The same holds for the programming language *PL*. The notion of refinement must be given a precise mathematical definition and it must be shown that refinements preserve behaviour in an appropriate sense.

There are other questions which are important for determining the useability of an approach:[2]

- What methods are available for proving the correctness of refinement steps?
- Where do refinement steps come from?
- What tools are available for assisting with which aspects of the program development process?
- What level of sophistication is required by the user of such tools?
- Which aspects of the program development process as sketched above can be fully automated?
- Does the approach require specifications to be "complete" in any sense? Does it provide a way of checking completeness of a specification or identifying areas of incompleteness?
- Does the approach provide ways of deriving programs which are optimal (or at least adequate) with respect to some performance measure?
- Does the approach provide a formal way of comparing pros and cons of different implementations which meet a specification?
- What are the complexity properties of the approach; for example, what happens to the size of proofs of correctness as specifications grow in size?

Again, it is essential for soundness that any methods which are established for proving correctness of refinement steps etc. are shown to be sound with respect to the formal definition of refinement.

Even once all these questions have been given precise answers, there are additional conditions which the answers must satisfy. For example, whatever definition of refinement

[2] Thanks to Aaron Sloman for pointing out the importance of the last four of these.

is adopted, it is essential that refinement steps be composable as discussed at the beginning of section 3. It turns out that this in turn imposes constraints on the semantics of specification-building operations.

Although these are important questions, it is not the purpose of this paper to answer all of them for all the approaches which will be mentioned below. Some of the answers may be found in the cited papers. Sometimes an approach has no clear answer to one or more of the most central questions; in this case I try to point this out.

1.3 Structure of this paper

The rest of this paper surveys different approaches to formal program development in the light of the discussion above. Section 2 surveys languages for writing specifications and programs. Section 3 examines the notions of refinement adopted in different approaches. Finally, section 4 discusses the issue of where individual refinement steps come from.

2 Specification and programming languages

This section survey languages which are used in different formal program development approaches to write specifications and programs. As suggested by the gradual evolution from high-level specification to program described above, in many approaches the distinction between specifications and programs is blurred or even non-existent.

Some approaches described below adopt a single so-called *wide spectrum* language which can be used to write high-level specifications, efficient programs, and everything which arises in the transition from the former to the latter during the program development process. In these intermediate steps it is natural for specification constructs to be mixed freely with programming constructs because of the way that high-level specifications are gradually refined to programs. This also avoids various problems which arise when separate specification and programming languages are used: there is no essential difference between refinement of programs and refinement of specifications; the same modularization constructs can be used to structure specifications as well as programs; there is no sudden leap from one notation to another but rather a gradual transition from high-level specification to efficient program.

For the purposes of this paper, "high-level specifications" are descriptions which give details of *what* is required. This is contrasted with "programs" which suggest *how* the desired result is to be computed (which amounts to an algorithm of some kind). The word "specification" shall be used to refer to any description of the input/output behaviour of a system, whether algorithmic or not; thus a program is a specification which is executable. This is consistent with the terminology used in wide spectrum languages where a program is a specification which happens to use only the executable subset of the language.

In some approaches it is argued that the initial high-level specification of requirements must be executable. This is thought to be necessary in order to ensure that this formal specification accurately reflects the customer's intentions; with an executable specification this can be ascertained by testing. We agree that it is necessary to start from a specification of requirements which is correct with respect to our intentions, and that bug-free formal specifications are difficult to construct. But requiring the initial specification to be executable means that a major part (the most difficult part, in our view) of the program development process is not formalized. For one thing, in constructing an executable specification it is necessary to make many decisions which could be left open in a non-executable specification. This means that a whole range of perfectly acceptable alternative implementations is unnecessarily eliminated from consideration from the very beginning. Writing specifications in an executable specification language is just a form of programming, and developing efficient programs from such specifications is just program optimization. Aiming for an initial specification which is as abstract and non-algorithmic as possible often leads to useful simplifications and generalizations which would not be discovered otherwise. Such a specification can be "tested" and shown to accurately reflect our intentions using a theorem prover; instead of evaluating an expression e and checking that the resulting value is v as expected, we try to prove that $e = v$ is impled by our specification.

A number of papers discussing these issues and others can be found in [GMc 86]; unfortunately most of the papers in this collection were originally published in the period 1977-1982 and so recent developments are not discussed.

2.1 VDM

VDM (the **V**ienna **D**evelopment **M**ethod) is a method for *rigorous* (not formal) program development. The objective is to produce programs by a process similar to the one sketched in section 1.2 where the individual refinement steps are shown to be correct using arguments which are formalizable rather than formal, thus approximating the level of rigour used in mathematics. This is supposed to yield most of the advantages of formal program development by ensuring that sloppiness is avoided without the foundational and notational overhead of full formality. VDM is presented in [Jon 80] and [BJ 82]; a short introduction to the approach is in [Jon 86]. VDM is the most widely accepted approach to systematic program development available to date.

VDM uses a model-oriented approach to describing data types. Models are built using functions, relations and sets. A simple example from [Jon 80] is the following specification of dates:

$$\text{Date} \quad :: \quad \text{YEAR} : \text{Nat} \quad \text{MONTH} : \{\text{Jan},\text{Feb},\ldots,\text{Dec}\} \quad \text{DAY} : \{1:31\}$$

This models dates as triples, but does not require that dates be represented as triples in the final program. Not all of the values of type Date are valid; the legal ones are characterized by the following *data type invariant*:

$$
\begin{aligned}
&\text{inv-date}(\langle y, m, d\rangle) =_{\text{def}}\\
&\quad (m \in \{\text{Jan,Mar,May,Jul,Aug,Oct,Dec}\} \Rightarrow 1 \leq d \leq 31) \wedge\\
&\quad (m \in \{\text{Apr,Jun,Sep,Nov}\} \Rightarrow 1 \leq d \leq 30) \wedge\\
&\quad (m = \text{Feb} \wedge\neg\ \text{is-leap-year}(y) \Rightarrow 1 \leq d \leq 28) \wedge\\
&\quad (m = \text{Feb} \wedge \text{is-leap-year}(y) \Rightarrow 1 \leq d \leq 29)
\end{aligned}
$$

A problem with model-oriented specifications is that it is easy to overspecify a system, eliminating certain implementations from consideration from the beginning. In VDM a precise notion of overspecification has been studied. A model is called *biased* if it is not possible to define an equality test on the data values in the model in terms of the operators defined. Intuitively, a biased model contains unnecessary redundancy. An unbiased model is viewed as sufficiently abstract to be the initial high-level specification of a system. More concrete models are introduced during the process of refinement (section 3.1).

Pre- and post-conditions are used to specify procedures, which may have side effects. For example, a procedure called TIMEWARP which resets the date (part of the global state) to a point 100 years in the past provided it is not October may be specified as follows:

$$
\begin{aligned}
&\text{TIMEWARP}\\
&\text{states: DATE}\\
&\text{pre-TIMEWARP}(\langle y, m, d\rangle) =_{\text{def}} m \neq \text{Oct}\\
&\text{post-TIMEWARP}(\langle y, m, d\rangle, r) =_{\text{def}} r = \langle y - 100, m, d\rangle
\end{aligned}
$$

In the post-condition, r is the state which is produced by the procedure TIMEWARP. It is necessary to show that TIMEWARP preserves the invariant associated with Date to ensure that TIMEWARP cannot create an invalid date when given a valid date. Decomposition during the refinement process is a matter of breaking down procedures into individual statements which can themselves be specified using pre- and post-conditions. When this process has been carried out to completion the result is a program.

The ESPRIT project RAISE [BDMP 85] is attempting to provide VDM with a formal foundation and support tools. A similar aim is being pursued by the MetaSoft project at the Polish Academy of Sciences [Bli 87].

2.2 Z

Z is a specification language based on the principle that programs and data can be described using set theory just as all of mathematics can be built on a set-theoretic basis. Thus, Z is no more than a formal notation for ordinary naive set theory. The first version of Z [ASM 79] used a rather clumsy and verbose notation but the current version [Spi 85,87] adopts a more concise and elegant notation based on the idea of a *scheme* which generalizes the sort of thing behind mathematical notations like $\{x \mid x \leq 7\}$, $\lambda x.x + 1$, $\int 4x^3 dx$, $\forall x.p(x)$, $\exists x.p(x)$, all of which involve bound variables of some kind.

Data types are modelled in Z using set-theoretic constructions, just as mathematical "data types" like natural numbers, real numbers, ordered pairs and sequences are defined in set-theoretic terms in mathematics. For example, in specifying a display-oriented text editor [Suf 82] a document is described as an ordered pair consisting of the text before the cursor and the text after the cursor:

$$DOC \;\widehat{=}\; seq[CH] \times seq[CH]$$

(CH is the set of characters which may appear in documents.) Two DOC-transforming functions may then be specified as follows:

$$\begin{array}{l}
back : DOC \rightarrow DOC \\
ins : CH \rightarrow DOC \rightarrow DOC \\
\hline
dom\ back = \{l, r \mid l \neq \langle\rangle\} \\
(\forall (l, r) : DOC; ch : CH) \\
back(l * \langle ch \rangle, r) = (l, \langle ch \rangle * r); \\
ins\ ch(l, r) = (l * \langle ch \rangle, r);
\end{array}$$

(In this specification, $\langle\rangle$ is the empty sequence; $\langle ch \rangle$ is the sequence containing the single character ch; and $*$ is the append function on sequences.) The function *back* (move one character backwards) is partial; it is applicable only to documents having some text before the cursor. This restriction on its domain is given by the first axiom in the above specification. The set of documents whose cursor is positioned at the beginning of a word can be described as a subset of DOC as follows:

$$\begin{array}{l}
sp : CH \\
wordb : \mathcal{P}(DOC) \\
\hline
sp \neq nl \\
wordb = \{l, r \mid last(l) \in \{sp, nl\} \vee first(r) \notin \{sp, nl\}\}
\end{array}$$

This introduces two distinguished characters, sp (space) and nl (new line) which are required to be different, and then defines the set *wordb* as that subset of DOC satisfying the second axiom.

Pre- and post-conditions may be used to specify procedures with side effects]in a way similar to that used in VDM (section 2.2).

Z has been used with success in UK industry to specify real systems. [Hay 87] is a collection of case studies. However, there is still a great deal of work needed to turn Z into a complete formal program development method. For example, although some work has been done on theorem proving by the Z group at Oxford, this work is not yet integrated with the Z language. It seems that the design of the language will make this more difficult than its ease of use as a specification language would suggest [Far 87].

2.3 IOTA

The IOTA project at the University of Kyoto attempted to provide a formal basis and mechanizable verification method for modular program development [NY 83]. The IOTA language allows programs to be built by composing (possibly parameterized) modules containing Algol-like function definitions which are specified using a variant of predicate logic with equality. Although the programming language is imperative, programs with side effects are not permitted. Program development is supported by an integrated program development environment which includes an interactive theorem proving subsystem.

The specification of a program module is split into the *interface* part which gives the names of the types and functions introduced by the module and the *specification* part which gives the axioms which the functions are required to satisfy. There are three kinds of modules: *type* modules which introduce a new abstract data type, *procedure* modules which introduce new functions which operate on existing data types, and *sype* modules which are used to specify the parameters of parameterized type and procedure modules. Type and procedure modules have in addition a *realization* part which gives an implementation which has been proved to satisfy the axioms in the specification part. Sype modules are used to constrain the permissible actual parameters of a parameterized module: a parameterized type or procedure module can only be applied to an actual parameter (type) module if that type module can be proved to satisfy the axioms in the sype module.

Program development in IOTA is not stepwise: the implementation of a type or procedure module is developed from its specification in a single step rather than via a gradual refinement process. The intention of IOTA is to allow the programmer to concentrate on a single module at a time rather than to support the process of developing an implementation for that module. Thus the emphasis is on verification rather than formal development.

2.4 Rewrite rule based languages

During the past decade a number of very high-level programming languages which can be seen as executable specification languages have been developed. These are based on the idea that equations can be viewed as rewrite rules. That is, an equation $\forall X.t = t'$ can be viewed as a rewrite rule $t \Rightarrow t'$ (or $t' \Rightarrow t$) which says that any substitution instance of t in an expression can be replaced by the corresponding substitution instance of t'. Under certain conditions it is possible to "run" a set of such rules to compute the value of an expression.

These languages are related to logic programming languages like Prolog [CM 81] which has also been touted as an executable specification language. One view of the relationship between rewrite rule based programming and logic programming is given in [GM 86].

2.4.1 HOPE

HOPE [BMS 80] is a purely applicative programming language having a rich but secure type system which allows new data types to be defined by the user. For example, binary trees with integer labels on nodes and leaves may be defined as follows (we assume that the type *int* has already been defined):

```
data tree == empty ++ leaf(int) ++ node(tree,int,tree)
```

This defines a type called *tree* and three *constructors*, a constant $empty : tree$ and two functions $leaf : int \rightarrow tree$ and $node : tree \times int \times tree \rightarrow tree$. It is not necessary to provide a representation for trees in terms of more primitive types; a value of type *tree* is just an expression built from constructors such as $leaf(3)$ or $node(empty, 4, leaf(7))$. A function which produces the sum of all the labels in a tree is defined by cases as follows:

```
dec sum: tree → int
--- sum(empty) ⇐ 0
--- sum(leaf(n)) ⇐ n
--- sum(node(t1,n,t2)) ⇐ n + sum(t1) + sum(t2)
```

Equations are required to be of the form $f(pattern) \Leftarrow expression$ where *pattern* is an expression containing variables and constructors only. This syntactic restriction is what makes HOPE programs executable. Higher-order functions which take functions as arguments and/or return functions as results are permitted. Static binding is used rather than LISP-like dynamic binding.

HOPE has other features such as a simple program modularization facility and Milner-style *polymorphic types* [Mil 78] which will not be discussed here.

2.4.2 Standard ML

One of the innovative features of the LCF theorem proving system [GMW 79] was the use of a general-purpose programming language called ML as its metalanguage. ML soon took on a life of its own with a number of implementations, each of a different dialect, developed during the period 1980-1985. Standard ML [HMM 86] is an attempt to reconcile the features of all these dialects which was strongly influenced by the design of HOPE.

The main concepts of Standard ML are similar to those of HOPE. The examples above may be rewritten in Standard ML with only minor syntactic changes:

```
datatype tree = empty | leaf of int | node of tree * int * tree

fun sum(empty) = 0
  | sum(leaf(n)) = n
  | sum(node(t1,n,t2)) = n + sum(t1) + sum(t2)
```

Standard ML has many other features which will not be discussed here, for example a powerful exception mechanism, polymorphic types and some imperative constructs.

The main important advance of Standard ML with respect to HOPE is the powerful facilities it provides for program modularization. These provide for the separate definition of interfaces (*signatures* and their implementations (*structures*). Every structure has a signature which gives the names of the types and functions defined in the structure. Structures may be built on top of existing structures, so each one is actually a *hierarchy* of structures, and this is also reflected in its signature. It is possible, and sometimes necessary in order to allow interaction between different parts of a program, to declare that certain substructures in the hierarchy are identical or *shared*. *Functors* are like parameterized structures; applying a functor to a structure yields a structure. A functor has an input signature describing structures to which it may be applied, and an output signature describing the result of an application.

[Har 86] and [Wik 87] are readable introductions to Standard ML; the latter unfortunately does not mention Standard ML's modularization facilities.

2.4.3 OBJ2

OBJ2 [FGJM 85] is the most recent in a succession of programming languages which includes OBJ0, OBJT and OBJ1. The original motivation for this work was to allow algebraic specifications to be tested, although it is now advertised as a ultra high level programming language. An OBJ2 program (called an *object*) declares some new types and functions which are then defined by means of a set of equations. When viewed as rewrite rules, the equations are required to have the Church-Rosser and termination properties which guarantee that repeated rewriting using the rules will always terminate with a unique result. This allows equations which are not permitted in HOPE or Standard ML and does not require that constructors be distinguished from other functions. Functions may be declared as commutative, associative, idempotent and/or having an identity; this information is then used appropriately by the rewriting mechanism.

Objects may be parameterized, where the range of permissible actual parameter objects is described by a (non-executable) *requirement theory*. At application time it is necessary to check that the given actual parameter satisfies the axioms in the requirement theory, up to a renaming which is supplied by the user. The semantics of object application is the same as the semantics of parameterized specification application in CLEAR (section 2.5.1).

OBJ2 has other features which will not be discussed here, including a very flexible syntax and a notion of subtype. OBJ2 does not permit higher-order functions in contrast to HOPE and Standard ML. Some of the features of OBJ2 (for example, checking that a system of equations is Church-Rosser and terminating, and checking that an actual parameter object satisfies a requirement theory) require the use of a theorem prover. [FGJM 85] indicates that the integration of OBJ2 with the REVE theorem proving system [Les 83] is a topic for future work; it is not known what progress has been made in this direction since then.

2.5 Algebraic specification languages

A great deal of work has been devoted to methods of specification based on the idea that for specification purposes a functional program can be modelled as a *many-sorted algebra*, i.e. as a number of sets of data values (one set of values for each data type) together with a number of (total) functions on those sets corresponding to the functions in the program. This abstracts away from the algorithms used to compute the functions and how those algorithms are expressed in a given programming language, focusing instead on the representation of data and the input/output behaviour of functions. It is possible to extend this paradigm to handle imperative programs as well by modelling imperative programs as functional programs [GB 80b] or else by using a different notion of algebra, as will be discussed below. The original motivation for this work was to provide a formal basis for the use of data abstraction in program development.

The pioneering work in this area was [Zil 74], [Gut 75] and [GTW 78], of which the latter (the so-called *initial algebra approach*) is the most formal. A specification consists of a *signature* — a set of *sorts* (data type names) and a set of function names with their types — together with a set of equational axioms expressing constraints which the functions must satisfy. For example, here is a specification of an abstract data type of natural numbers:

$$
\begin{array}{lll}
\text{signature} & \text{sorts} & \text{nat} \\
 & \text{opns} & 0 : \to \text{nat} \\
 & & \text{succ} : \text{nat} \to \text{nat} \\
 & & + : \text{nat} \times \text{nat} \to \text{nat} \\
 & & \times : \text{nat} \times \text{nat} \to \text{nat} \\
 & \text{axioms} & \forall n : \text{nat}.\ 0 + n = n \\
 & & \forall m, n : \text{nat}.\ \text{succ}(m) + n = \text{succ}(m + n) \\
 & & \forall n : \text{nat}.\ 0 \times n = 0 \\
 & & \forall m, n : \text{nat}.\ \text{succ}(m) \times n = n + (m \times n)
\end{array}
$$

Notice how the constant 0 is viewed as a nullary function. This specification describes a certain class of algebras having the given signature and satisfying the given equations (the isomorphism class of so-called *initial models*, which includes the usual set-theoretic model of natural numbers). Another possibility is to view this specification as describing the isomorphism class of so-called *final models* [Kam 83] (see also [Gan 83]). Since this is a very simple specification it has only one sort but in general a specification may include many sorts. See [EM 85] for a detailed introduction to this style of specification.

2.5.1 CLEAR

Specifications such as those above are fine for specifying very simple data types such as natural numbers, booleans, stacks and queues. But specifying a large programs using this method would involve a list of hundreds or even thousands of axioms. Even if such a large specification could be constructed, it would be impossible to understand or use. The likelihood that a specification in this style accurately reflects the specifier's intention decreases dramatically with the size of the specification.

The specification language CLEAR [BG 77, 80, 81], [San 84] provides a small number of specification-building operations which allow large and complicated specifications to be built in a structured way from small, understandable and reuseable pieces. The operations provide ways of combining two specifications, of enriching a specification by some new sorts, functions and axioms, of renaming and/or forgetting some of the sorts and functions of a specification, and of constructing and applying *parameterized* specifications. In contrast to the simple approach sketched above it is possible to write *loose* specifications in CLEAR, i.e. specifications describing a range of non-isomorphic algebras. This allows decisions to be left deliberately open to be made later in the program development process. For example, it is possible to specify a function which takes the square root of a number without saying whether it produces the negative or positive square root.

The semantics of CLEAR allows it to be used with different kinds of axioms (not just equations) to specify different kinds of algebras. This allows appropriate treatment of exceptions, non-terminating functions and imperative programs, among other things. This point will be discussed at length in section 2.5.5 below.

2.5.2 CIP-L

The project CIP (**C**omputer-aided, **I**ntuition-guided **P**rogramming) at the Technische Universität München had as its aim the development of a methodology for formal program development by transformation (see section 4) and the implementation of a system to support the development process.

CIP-L [Bau 85] is the language on which the CIP project was based. CIP-L is a wide-spectrum language which includes constructs for writing high-level specifications (using predicate logic with equality, non-deterministic choice and set expressions), functional programs, imperative programs and unstructured programs with gotos. These constructs may be freely mixed in order to allow a gradual transition between the initial high-level specification of the problem to be solved and the final efficient program. Some advantages of a wide-spectrum language like CIP-L for formal program development were mentioned at the beginning of section 2. CIP-L includes different facilities for constructing (possibly parameterized) high-level specification modules and program modules in a hierarchical fashion; the relationship between these different kinds of modules is similar to the relation between the specification part and realization part of a module in IOTA (section 2.3) or between signatures and structures in Standard ML (section 2.4.2).

The semantics of the specification part of CIP-L is expressed in terms of classes of partial algebras (algebras in which partial functions may be modelled). In contrast to most earlier approaches in the algebraic tradition, the axioms used in specifications need not be equations. It is possible to write loose specifications in CIP-L as in CLEAR.

2.5.3 ACT ONE

The ideas incorporated in CLEAR and CIP-L diverge to some extent from the initial algebra approach to algebraic specifications in [GTW 78] outlined above. This earlier strand of theoretical work was continued by Thatcher, Wagner and Wright of IBM Yorktown Heights in collaboration with Ehrig and his colleagues at the Technische Universität Berlin. ACT ONE (described in [EM 85], chapters 9 and 10) is a specification language developed in Berlin which adheres more or less strictly to the initial algebra approach. ACT ONE includes specification-building operations similar to the ones in CLEAR except that the mechanism which supports parameterization of specifications is intentionally different. An important drawback of ACT ONE is that it does not permit loose specifications.

2.5.4 Larch

The Larch family of specification languages [GHW 82], [GH 83] was developed at MIT and Xerox PARC to support the productive use of formal specifications in programming. One of its goals is to support a variety of different programming languages, including imperative languages, while at the same time localizing programming language dependencies as much as possible. Each Larch language is composed of two components: the *interface language* which is specific to the particular programming language under consideration and the *shared language* which is common to all programming languages. The interface language is used to specify program modules using predicate logic with equality and constructs to deal with side effects, exception handling and other aspects of the given programming language. The shared language is an algebraic specification language used to describe programming-language independent abstractions using equational axioms which may be referred to by interface language specifications. The role of a specification in the shared language is to define the concepts in terms of which program modules may be specified. The shared language includes specification-building operations inspired by those in CLEAR, although these are viewed as purely syntactic operations on lists of axioms rather than as semantically non-trivial operations as in CLEAR.

2.5.5 Institutions

Any approach to algebraic specification must be based on some logical system. Typically many-sorted equational logic is used for this purpose. Nowadays, however, examples of logical systems in use include first-order logic (with and without equality), Horn-clause logic, higher-order logic, infinitary logic, temporal logic and many others. All these logical systems may be considered with or without predicates, admitting partial functions or not. This leads to different concepts of signature of algebra. There is no reason to view any of these logical systems as superior to the others; the choice must depend on the particular area of application and may also depend on personal taste.

The informal notion of a logical system has been formalised by Goguen and Burstall [GB 84], who introduced for this purpose the notion of *institution*. An institution consists of a collection of signatures together with for any signature Σ a set of well-formed Σ-sentences, a collection of Σ-algebras and a satisfaction relation between Σ-algebras and Σ-sentences. The signatures come with some notion of signature morphism $\sigma : \Sigma \to \Sigma'$ in order to provide for changing signatures, which induces the σ-translation of Σ-sentences to Σ'-sentences and of Σ'-algebras to Σ-algebras. When we change signatures, the induced translations of sentences and algebras are required to preserve the satisfaction relation. This condition expresses the intentional independence of the meaning of specifications from the actual notation. All the above logical systems (and many others) fit into this mould.

For purposes of generality, it is best to avoid choosing any particular logical system on which to base a specification approach. This leads to results and tools which can be reused in many different logical systems. The semantics of CLEAR (see section 2.5.1) is parameterized by an arbitrary institution in this fashion. This means that CLEAR accommodates a variety of logical systems and styles of specification without change. The other specification languages discussed up to this point do not share this feature. Thus, for example, CIP-L is solidly based on first-order logic and partial algebras and cannot be used to specify higher-order functions or non-deterministic functions without fundamental changes.

2.5.6 ASL

ASL [SWi 83], [Wir 86] is a kernel specification language intended primarily as a foundation for building other more high-level and user-friendly specification languages rather than for direct use in writing specifications. It comprises a small number of simple but powerful specification-building operations. It has a simple semantics in comparison with high-level specification languages like those described above. The semantics of the constructs of a high-level specification language built on top of ASL would be expressed by mapping these constructs into ASL expressions. Specification languages built on top of ASL in this fashion include Extended ML (section 2.5.7), PLUSS [Gau 83] and SMoLCS [AMRW 85] which has in turn been used to give a formal semantics of Ada [DDC 85].

One novel feature of ASL is a specification-building operation which can be used to *behaviourally abstract* from a specification, closing its collection of models under *behavioural equivalence* [GGM 76], [Rei 81], [ST 87]. This allows model-oriented specifications as in VDM (section 2.1) in which a desired behaviour is described by giving a simple concrete model which exhibits it. It is argued that such an operation is a necessary ingredient in an algebraic specification language since the specification of e.g. an abstract data type is supposed to describe a behaviour without regard to the particular representation used and therefore *all* algebras which realize the desired behaviour should be permitted. Furthermore, using algebraic specification languages which lack a behavioural abstraction operation, it is in general difficult (as in CLEAR) or impossible (as in ACT ONE) to describe collections of algebras which are closed under behavioural equivalence since such a collection may contain a wide range of non-isomorphic algebras.

The semantics of ASL is parameterized by an arbitrary institution [ST 88a] so like CLEAR it can be used with a wide variety of logical systems.

2.5.7 Extended ML

Extended ML [ST 85] is a wide-spectrum language obtained by extending Standard ML (section 2.4.2) to allow axioms to appear in signatures and in place of code in structure and functor definitions. Axioms in a signature place constraints on the permitted behaviour of the components of structures which match that signature. Axioms in a structure or functor are used to write "abstract programs" [Wirth 71], i.e. to define functions and data in a high-level way which is not necessarily executable. Some Extended ML specifications are executable, since Standard ML function definitions are just axioms of a certain special form. The goal of program development is to refine non-executable specifications until they contain axioms of this form only.

The semantics of Extended ML is defined by translation into ASL [ST 86] as described in section 2.5.6. Behavioural abstraction plays an important role in the semantics of signatures. Extended ML can be used with a wide variety of logical systems because of the way that its semantics is based on ASL. This also amounts to the independence of Extended ML from the programming language used to write code, since programs are just a form of axioms. Thus Extended ML can be used without major change to develop programs in languages other than Standard ML, for example Prolog (see also [SWa 87]).

3 Refinement of specifications

It turns out to be surprisingly difficult to give a precise definition which adequately captures the intuitively simple notion of refinement. The main problem is with the representation of data. During the process of refining an abstract specification to a concrete program, it is necessary to devise more and more concrete data representations. Ultimately all data must be represented using the primitive data types provided by the target programming language. Some of the issues which arise are illustrated by the following very simple example.

Consider a specification SP describing the function min which takes as input a set of numbers and produces as output the smallest number in the set. It is natural to consider representing sets as lists, but there are at least four different ways this may be done:

1. All of [1,2], [2,1], [2,1,2], [1,1,1,2] etc. represent the set $\{1,2\}$: order is insignificant and elements may be repeated.

2. All of [1,2], [1,2,2], [1,1,2,2,2] etc. represent the set $\{1,2\}$: elements may be repeated but order is significant.

3. Both [1,2] and [2,1] represent the set $\{1,2\}$: order is not significant but elements are not repeated.

4. The list [1,2] represents the set $\{1,2\}$: order is significant and elements may not be repeated.

Each of these representations is valid in the sense that the correctness of the resulting program will not depend on which one is chosen. Which representation is most useful depends on the functions which will be used to create and access sets and their relative frequency of use. Adding an element to a set using representation 1 is cheap since the new element may simply be added to the beginning of the list. Adding a new element using representation 3 involves checking if the element is already there, which may require searching the entire list. Adding an element using representation 2 or 4 involves finding the proper place in the list to deposit the new element. On the other hand, checking membership using representation 2 or 4 is cheap since the list is ordered; finding the minimum element is especially cheap. Finally, representations 3 and 4 are efficient in terms of space since set elements are not needlessly repeated.

This example demonstrates some of the degrees of freedom which are possible when refining the representing of data. Representations 1-3 show that it is possible to have several concrete representations of a single abstract value, and representations 2-4 show that sometimes there are concrete values (for example [2,1,2]) which are not used to represent abstract values. It is also possible to have several abstract values represented by the same concrete value which can happen if the original specification is biased, to use VDM terminology. Unless all of these degrees of freedom are captured by the formal notion of refinement adopted by a particular program development approach, there will correct programs which will not be obtainable using that approach.

Whatever formal notion of refinement is adopted, it is essential for the correctness of the development method that refinement steps be composable in two ways. First of all, two refinement steps $SP \rightsquigarrow SP'$ and $SP' \rightsquigarrow SP''$ should compose to give a correct refinement $SP \rightsquigarrow SP''$, for arbitrary specifications or programs SP, SP' and SP''. This is the property of refinement steps (known as *vertical composability* [GB 80a]) which guarantees the correctness of programs developed from specifications in a stepwise fashion. Secondly, if the program development approach under consideration allows specifications to be decomposed into smaller units during the development process, then the notion of refinement adopted must be compatible with the specification-building operations: given two refinement steps $SP_1 \rightsquigarrow SP'_1$ and $SP_2 \rightsquigarrow SP'_2$, it should be the case that $SP_1 \oplus SP_2 \rightsquigarrow SP'_1 \oplus SP'_2$ is a correct refinement for any specification-building operation $\oplus$. This is the property (known as *horizontal composability* [GB 80a]) which guarantees that separate development strands may proceed independently and then later be combined to yield a correct result. Finally, a formal program development method must provide some way of proving that refinement steps are correct with respect to the notion of refinement adopted.

Not all of the specification approaches presented in section 2 come with a corresponding notion of refinement. This concept has received most attention in connection with algebraic approaches.

3.1 VDM

VDM has already been introduced in section 2.1.

Suppose that we want to establish that one VDM specification SP' is a correct refinement of another VDM specification SP. The proof of correctness consists of the following steps:

- Define a "retrieve function" $retr$ which maps the data values specified in SP' to the data values specified in SP. This relates concrete data values to the abstract values they represent. Because of the direction of $retr$ there may be many concrete values which represent a single abstract value but not vice versa.

- Prove that $retr$ is total (on data values in SP' which satisfy the data type invariant) and surjective. This guarantees that every concrete data value represents some abstract value and that every abstract value has a representation. The data type invariant in SP' should restrict the domain of $retr$ to the concrete values which will be used as representations.

- Identify an operation f' in SP' corresponding to each operation f in SP. The operations on the concrete level are supposed to model those on the abstract level, as will be ensured by the final two steps.

- Prove that $\text{pre}-f(retr(v)) \Rightarrow \text{pre}-f'(v)$ for all concrete values v. This ensures that the pre-conditions of the concrete operations are not more restrictive than those of the corresponding abstract operations.

- Prove that $\text{pre}-f(retr(\overline{v})) \wedge \text{post}-f'(\overline{v}, v) \Rightarrow \text{post}-f(retr(\overline{v}), retr(v))$ for all concrete values $\overline{v}, v$. This guarantees that the results produced by the concrete operations mirror those produced by the abstract operations.

This notion of refinement is able to capture all four of the ways listed above of representing sets as lists. However, if SP is biased then this approach will exclude some refinements leading to programs which are correct from the point of view of the behaviour they display. This point is discussed in [Sch 86]. An attraction of the VDM approach is that the steps for verifying correctness are stated quite explicitly and are relatively easy to accomplish. This is often not the case in other approaches.

3.2 Z

Z was introduced in section 2.2. The Z approach to specification refinement as described in [Spi 87] is virtually identical to the VDM approach presented above except for differences in notation and terminology, leading to the same advantages and disadvantages. [Spi 87] claims that it is possible to handle situations in which several abstract values are represented by a single concrete value (as is sometimes required when SP is biased) using a slightly more complicated approach, but no details are provided.

3.3 Algebraic approaches

Some of the background to algebraic methods of program specification has already been given in section 2.5.

Work in this context on specification refinement has been inspired by the seminal work of Hoare [Hoa 72] on data refinement. According to [Hoa 72], an algebra A' is a refinement of an algebra A if there is some subalgebra A'' of A' with a surjective homomorphism $h: A'' \rightarrow A$. There is an intimate connection with the VDM approach described in section 3.1: the subalgebra A'' contains those data values which satisfy the data type invariant (the *representation invariant* in Hoare's terminology) and h is the retrieve function (*abstraction function*). Requiring h to be a homomorphism guarantees that the operations in A'' (and thus in A') behave the same as the operations in A.

This idea can be extended from algebras to specifications by regarding a specification SP' as a correct refinement of another specification SP if every model of SP' refines a model of SP in Hoare's sense. It is possible to modify Hoare's definition in various ways so as to take account of the possibility that SP' might contain operations having different names from those in SP and to allow an intermediate specification to be constructed based on SP'. There has been a great deal of work on this topic with more than twenty different algebraic definitions of refinement advocated in the literature, including definitions suitable for use with CLEAR (section 2.5.1) and CIP-L (section 2.5.2). Probably the most influential of these is the one given in [EKMP 82] which is suitable for use with ACT ONE (section 2.5.3). Although Hoare's refinement relation composes vertically (see section 3) most of the more elaborate refinement relations which have been proposed do not compose vertically except under conditions which are not easy to ensure. For many of these refinement relations the question of horizontal composability has not been investigated while for others it has been shown to be problematic.

A simpler approach is obtained by requiring specifications to describe *all* algebras which are to be regarded as acceptable realizations. Then SP' is a refinement of SP if all the models of SP' are also models of SP. This definition is only appropriate for use with a specification language which incorporates behavioural abstraction (such as ASL, discussed in section 2.5.6, and Extended ML, discussed in section 2.5.7) since otherwise it is not flexible enough to capture the kinds of refinements which are required in practice. It is easy to show that this refinement relation composes both vertically and horizontally.

A unifying and generalizing approach to the issue of specification refinement is [ST 88b]. All the other definitions of refinement (including the one used in VDM) are special cases of this one. Refinements compose vertically and horizontally and it is shown how previous problems in this area were the consequence of using notions of refinement which were not sufficiently flexible. This notion of refinement works in the context of an arbitrary institution and it is even possible to change institutions in the course of a refinement step, so for example refining an equational specification to yield a Hoare-logic specification of an imperative program.

4 Program transformation and program synthesis

Section 1.2 presented a general picture of formal program development in which programs were evolved from specifications in a gradual fashion via a series of refinement steps. The source of these refinement steps was left open. The program development approaches mentioned in section 3 assumed that refinement steps are supplied manually by the user and must be proved correct.

Some refinement steps are more or less routine. For example, there are certain standard concrete ways of representing common abstract data types like stacks and queues, and there are standard ways of converting recursive algorithms to iterative ones. If the refinement process is taken to include the process by which inefficient programs are transformed into efficient programs, then the techniques used by optimizing compilers (constant propagation, loop jamming, etc.) can be viewed as refinement steps as well. Such refinement steps can typically be described schematically; for example, replacement of a simple form of recursion by iteration is described as follows:

$$
\begin{array}{lll}
\text{Replace:} & f(x) = & \textbf{if } e_1(x) \textbf{ then } e_2(x) \textbf{ else } f(e_3(x)) \\
\text{by:} & f(x) = & \textbf{begin} \\
& & \quad \textbf{var } a; \\
& & \quad a := x; \\
& & \quad \textbf{while not } e_1(a) \textbf{ do } a := e_3(a); \\
& & \quad \textbf{return } e_2(a) \\
& & \textbf{end}
\end{array}
$$

Any refinement obtained by instantiating this transformation rule will be correct. Rather than proving correctness separately for each instantiation, the rule itself can be proved correct (with respect to a given notion of refinement) and then applied as desired without further proof. Sometimes such a rule will be correct only provided certain conditions are met by the program fragments matching the schematic variables or by the context in which the rule is applied; in this case the proof obligation is reduced to checking that these conditions are satisfied.

Research on program transformation aims at developing appropriate formalisms and notations, building computer-based systems for handling the bookkeeping involved in applying transformation rules, compiling libraries of useful transformation rules, and developing strategies for conducting the transformation process automatically or semi-automatically. See [PS 83] for a survey of this work. Two program transformation approaches are discussed in sections 4.1 and 4.2.

The entire program development process as presented in section 1.2 can be captured within the program transformation paradigm, given the following transformation rule:

$$
\begin{array}{ll}
\text{Replace:} & SP \\
\text{by:} & SP' \\
\text{provided:} & SP \rightsquigarrow SP' \text{ is a correct refinement}
\end{array}
$$

This transformation rule cannot be regarded as satisfactory because its schematic content is completely trivial; the work involved in using this rule is just the same as that involved in the direct use of the notion of refinement as described in section 1.2. In practice, most transformation rules lie somewhere between the extremes represented by the two transformation rules given above. Some invention on the part of the user is required before the schematic transformation can be applied and a proof that certain conditions hold is needed to ensure that the transformation is sound.

Schematic rules have mostly been applied at the level of programs rather than specifications, for transforming algorithms to increase efficiency rather than for developing an algorithm from scratch. Spectacular speedups far in excess of those achievable by an optimizing compiler are possible; an example in [PB 82] shows how a simple program running in exponential time can be transformed to one running in logarithmic time.

Transformation rules provide a way of generating correct refinement steps automatically. But it is still necessary for the person developing the program to select an appropriate rule to apply and in many cases considerable ingenuity is required to supply appropriate expressions and function definitions to substitute for the schematic variables in the rule. Program synthesis attempts to automate this process, thereby generating programs from specifications automatically. Two approaches to program synthesis are presented in sections 4.3 and 4.4.

4.1 Burstall, Darlington and Feather

The first work on program transformation was done by Burstall and Darlington in Edinburgh in the mid-1970's. In [BD 77] they describe a set of seven simple rules which can be used to transform programs written in a HOPE-like [BMS 80] language. These rules include *fold* (replace an expression $e(a)$ by a function call $f(a)$ where f is defined by $f(x) = e(x)$); *unfold* (dual to fold, expand a function call into the body of the function's definition); definition of new functions; and rewriting of expressions using the algebraic properties of primitive operations. Individual applications of these rules do not lead to improvements in efficiency, but certain sequences of rule applications can produce speedups. An example from [BD 77] is the following transformation of a simple program for generating Fibonacci numbers which runs in exponential time to a slightly more complicated linear-time program:

1.	$f(0)$	$= 1$	original program
2.	$f(1)$	$= 1$	original program
3.	$f(x+2)$	$= f(x+1) + f(x)$	original program
4.	$g(x)$	$= \langle f(x+1), f(x) \rangle$	definition
5.	$g(0)$	$= \langle f(1), f(0) \rangle$	instantiation of 4
		$= \langle 1, 1 \rangle$	unfold with 1 and 2
6.	$g(x+1)$	$= \langle f(x+2), f(x+1) \rangle$	instantiation of 4
		$= \langle f(x+1) + f(x), f(x+1) \rangle$	unfold with 3
		$= \langle u+v, u \rangle$ where $\langle u, v \rangle = \langle f(x+1), f(x) \rangle$	unfold with 3
		$= \langle u+v, u \rangle$ where $\langle u, v \rangle = g(x)$	fold with 4
7.	$f(x+2)$	$= u+v$ where $\langle u, v \rangle = \langle f(x+1), f(x) \rangle$	abstract 3
		$= u+v$ where $\langle u, v \rangle = g(x)$	fold with 4

Successful transformation sequences typically involve one or more creative so-called *eureka* steps in which a new function is invented or a critical rewriting is done. The eureka step in the above example is step 4 where the new function g is defined which proves crucial in transforming f in step 7.

Darlington and Burstall built a system which automatically transformed small programs by applying these transformation rules. Feather [Fea 82] built a system which allowed the user to supply a pattern indicating the overall form of the desired result. This allowed his system to handle much larger examples because of the resulting reduction in the search space.

Strictly speaking, the rules in [BD 77] do not preserve equivalence of programs under a call by value semantics; unfolding may lead to a program which terminates more often than the original program whereas folding has the opposite (and much less desirable) effect. Scherlis [Sch 81] proposes an alternative set of primitive transformation rules which do preserve equivalence.

4.2 CIP and PROSPECTRA

The CIP project and the CIP-L wide spectrum language have already been introduced in section 2.5.2.

The intention of the CIP project was to support program development by transformation all the way from the original high-level specification to a final efficient program. This typically includes the following stages of development [Par 86], [Möl 87]:

- Descriptive formal problem specification;
- Modified (still descriptive) specification;
- Non-deterministic implicitly recursive solution;
- Non-deterministic explicitly recursive solution;
- Deterministic tail-recursive solution;
- Further modified applicative program;
- Efficient imperative program.

A library of transformation rules has been developed to support all the stages of this process under the supervision of the system CIP-S [Bau 87]. For example, in transforming specifications the *embedding* rule is often useful (I simplify a little and use a different notation in all rules in this section):

$$
\begin{array}{ll}
\text{Replace:} & \textbf{function } f(x\colon t) = e_1 : t' \\
\text{by:} & \textbf{function } f(x\colon t) = \\
 & \qquad\qquad \textbf{function } g(y\colon t'') = e_2 : t''' \\
 & k(g(h(x)))\colon t' \\
\text{provided:} & h\colon t \rightarrow t'' \wedge k\colon t''' \rightarrow t' \wedge \forall y\colon t . k(g(h(y))) = f(y)
\end{array}
$$

Using this rule involves inventing h, k and e_2 in such a way that the condition is satisfied.

Burstall/Darlington-style folding and unfolding rules (modified to ensure that folding does not introduce non-termination) are used to derive theorems about the specification which can be read as recursion equations.

The recursive form is improved by applying rules which convert recursion equations to tail-recursive form, combine common sub-expressions, eliminate function composition, memo-ize functions, etc. For example, here is a rule which can be used to perform a general form of *strength reduction* in which recomputation of an expression on each recursive call is replaced by computing its value incrementally from one call to the next:

Replace: **function** $f(x{:}\,t) =$
  if e_1 **then** e_2 **else** $f(e_3)$
 $f(z)$
by: **function** $g(x{:}\,t, y{:}\,t') =$
  if $e_1[y/e]$ **then** $e_2[y/e]$ **else** $g(e_3[y/e], e[e_3/x][y/e])$
 $g(z, e)$
provided: determinate(e) $\wedge$ defined(e) $\wedge$ newvar(y)

($a[b/c]$ denotes the result of substituting b for every occurrence of c in a.) In applying this rule, e should be chosen so that the result is simpler to compute than the original form.

Finally, conversion from tail-recursive to imperative form is accomplished by transformation rules like the one at the beginning of section 4. For more general types of recursion (for example, mutual recursion) transformation rules are provided which yield programs containing gotos [BW 82].

These techniques are applicable mainly to developing the individual functions of a program. Another form of transformation is change of data structure via a form of refinement [BMPW 86] similar to the algebraic approaches discussed in section 3.3.

The ESPRIT project PROSPECTRA [Kri 87] is attempting to apply methods similar to those developed in the CIP project to the development of Ada programs by transformation from specifications written in the language Anna [LHKO 87]. Apart from the attempt to scale up the ideas to work in the context of Ada (which involves considering concurrency aspects, etc.), there is some work on an algebraic formalization of the transformation process itself so as to allow the transformation strategy itself to be developed by transformation [Kri 88].

4.3 Manna and Waldinger

The DEDALUS system [MW 79] was developed to synthesize LISP programs automatically from specifications written in a simple LISP-like notation. A goal-directed deductive approach is used whereby the reduction of a goal (to synthesize a program satisfying a given specification) to one or more subgoals by means of a transformation rule results in the generation of a program fragment which computes the desired result once it is completed with program fragments corresponding to the subgoal(s). So, for example, reducing a goal to two subgoals by means of a case analysis corresponds to the introduction of a conditional expression.

An idea corresponding to Burstall and Darlington's fold rule is used to introduce recursive function calls. This is done if a goal is produced which matches the original top-level goal, provided termination can be guaranteed. Auxiliary recursive functions are formed if a goal is encountered which matches some lower-level goal.

The system incorporates an automatic theorem prover and includes a number of strategies designed to direct it away from rule applications unlikely to lead to success. If a dead end is encountered then the system backtracks and tries another rule. This system has been used to generate a few simple list-processing functions such as the intersection of two lists and functions like the greatest common divisor of two numbers.

A later approach [MW 80] is based on a sequent-based system for theorem proving in first-order logic. In this system, a sequent consists of a number of assumptions and goals in first-order logic and a LISP-like expression attached to each goal. The meaning of a sequent is that if all the assumptions are true then some instance of some goal is true and is satisfied by the corresponding instance of the attached expression. Logical rules, resolution rules, transformation rules and rules like those in DEDALUS are used to operate on sequents. [MW 81] shows how this approach could be used to derive a unification algorithm but the example was not done automatically.

4.4 Nuprl

Nuprl [Con 86] is an interactive system for proving theorems in a constructive logic based on [Mar 82]. As a theorem-proving system it has many similarities with the LCF system [GMW 79], allowing the user to conduct proofs by constructing and applying goal-directed proof strategies (*tactics*) as programs in ML.

A consequence of the use of a constructive logic means that proofs embody constructions. A construction can be automatically extracted from a proof to yield a program. So for example, given a proof in Nuprl for the theorem

$$\forall x, y{:}\,\mathrm{int}.\exists q, r{:}\,\mathrm{int}.(y = (q * x + r) \wedge 0 \leq r < x)$$

one can extract a program for finding the quotient and remainder of two integers. In general, a conjecture (unproved theorem) of the form $\forall x.\exists y.R(x,y)$ may be viewed as a specification of a program which, given a value for x, computes a value for y such that $R(x,y)$ is true. Proving the theorem gives rise to a program which satisfies this specification. A proof which appeals to a lemma which has not yet been proved can be seen as a verified refinement step.

There are strong similarities with the work of Manna and Waldinger on program synthesis discussed above. The most important differences are that the logic underlying Nuprl is more expressive, including higher-order functions and dependent types, and that there is no real attempt to automate the synthesis process since programs are obtained directly from proofs which are performed interactively. But because of the flexible way that proof strategies may be added to the Nuprl system, the possibility of developing an automatic theorem prover based on Nuprl (which is then able to perform program synthesis automatically) is not excluded. Proof strategies for Nuprl based on the ones used in the

Boyer/Moore theorem prover [BM 79] are being studied in an Alvey project at Edinburgh [Bun 88].

Acknowledgements

I am grateful to Gavin Oddy, Aaron Sloman, Colin Tully and Lincoln Wallen for helpful criticism of a draft of this paper. This paper was finished during a visit to the C.S.I.C. Centre d'Estudis Avançats de Blanes funded by the Comité Conjunto Hispano-Norteamericano para le Cooperación Cultural y Educativa. I would like to thank the Comité for its financial support and the Centre and its staff (particularly Josep Puyol) for their help with computing facilities.

5 References

[Note: LNCS n = Springer Lecture Notes in Computer Science, Volume n]

[ASM 79] Abrial, J.R., Schuman, S.A. and Meyer, B. Specification Language Z. Massachusetts Computer Associates Inc., Boston (1979).

[AMRW 85] Astesiano, E., Mascari, G.F., Reggio, G. and Wirsing, M. On the parameterized specification of concurrent systems. *Proc. Joint Conf. on Theory and Practice of Software Development*, Berlin, Springer LNCS 185, pp. 342-358.

[Bau 85] Bauer, F.L. *et al* (the CIP Language Group). *The Wide-Spectrum Language CIP-L.* Springer LNCS 183 (1985).

[Bau 87] Bauer, F.L. *et al* (the CIP System Group). *The Program Transformation System CIP-S.* Springer LNCS 292 (1987).

[BW 82] Bauer, F.L. and Wössner, H. *Algorithmic Language and Program Development.* Springer (1982).

[BDMP 85] Bjørner, D., Denvir, T., Meiling, E. and Pedersen, J.S. The RAISE project: fundamental issues and requirements. Report RAISE/DDC/EM/1/V6, Dansk Datamatic Center (1985).

[BJ 82] Bjørner, D. and Jones, C.B. *Formal Specification and Software Development.* Prentice-Hall (1982).

[Bli 87] Blikle, A. *MetaSoft Primer.* Springer LNCS 288 (1987).

[BCFG 86] Bougé, L., Choquet, N., Fribourg, L. and Gaudel, M.-C. Test set generation from algebraic specifications using logic programming. *Journal of Systems and Software 6*, 343-360 (1986).

[BM 79] Boyer, R.S. and Moore, J.S. *A Computational Logic.* Academic Press (1979).

[**BMPW 86**] Broy, M., Möller, B., Pepper, P. and Wirsing, M. Algebraic implementations preserve program correctness. *Science of Computer Programming 7*, 35-53 (1986).

[**Bun 88**] Bundy, A. *et al.* Proving properties of logic programs: a progress report. *Proc. 1988 Alvey Conference*, to appear (1988).

[**BD 77**] Burstall, R.M. and Darlington, J. A transformation system for developing recursive programs. *Journal of the ACM 24*, 44-67 (1977).

[**BG 77**] Burstall, R.M. and Goguen, J.A. Putting theories together to make specifications. *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, 1045-1058 (1977).

[**BG 80**] Burstall, R.M. and Goguen, J.A. The semantics of CLEAR, a specification language. *Proc. 1979 Copenhagen Winter School on Abstract Software Specification*, Springer LNCS 86, 29-332 (1980).

[**BG 81**] Burstall, R.M. and Goguen, J.A. An informal introduction to specifications using CLEAR. *The Correctness Problem in Computer Science* (R.S. Boyer and J.S. Moore, eds.), pp. 185-213, Academic Press (1981).

[**BMS 80**] Burstall, R.M., MacQueen, D.B. and Sannella, D.T. HOPE: an experimental applicative language. *Proc. 1980 LISP Conference*, Stanford, 136-143 (1980).

[**CM 81**] Clocksin, W.F. and Mellish, C.S. *Programming in Prolog.* Springer (1981).

[**DDC 85**] Christensen, D. *et al* The draft formal definition of ANSI/MIL-STD 1815A Ada: dynamic semantics example Ada subset. Report Ada FD/DDC/02/v1.3, Dansk Datamatic Center (1985).

[**Con 86**] Constable, R.L. *et al* (the PRL Group) *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall (1986).

[**EKMP 82**] Ehrig, H., Kreowski, H.-J., Mahr, B. and Padawitz, P. Algebraic implementation of abstract data types. *Theoretical Computer Science 20*, 209-263 (1982).

[**EM 85**] Ehrig, H. and Mahr, B. *Fundamentals of Algebraic Specifications 1*, EATCS Monographs on Computer Science, Springer (1985).

[**Far 87**] Farrés, J. On the representation of Z in LF. Draft report, Univ. of Edinburgh (1987).

[**Fea 82**] Feather, M. A system for assisting program transformation. *ACM Trans. on Prog. Languages and Systems 4*, 1-20 (1982).

[**FGJM 85**] Futatsugi, K., Goguen, J.A., Jouannaud, J.-P. and Meseguer, J. Principles of OBJ2. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 52-66 (1985).

[Gau 83] Proposition pour un langage d'utilisation de spécifications structurées: PLUSS. C.G.E. Research Report (1983).

[Gan 83] Ganzinger, H. Parameterized specifications: parameter passing and implementation with respect to observability. *ACM Trans. on Prog. Languages and Systems 5*, 318-354 (1983).

[GMc 86] Gehani, N. and McGettrick, A.D. (eds.) *Software Specification Techniques*, Addison-Wesley (1986).

[GGM 76] Giarratana, V., Gimona, F. and Montanari, U. Observability concepts in abstract data type specification. *Proc. 5th Intl. Symp. on Mathematical Foundations of Computer Science*, Gdansk. Springer LNCS 45, pp. 576-587 (1976).

[GB 80a] Goguen, J.A. and Burstall, R.M. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report, SRI International (1980).

[GB 80b] Goguen, J.A. and Burstall, R.M. An ORDINARY design. Draft report, SRI International (1980).

[GB 84] Goguen, J.A. and Burstall, R.M. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. Springer LNCS 164, pp. 221-256 (1984).

[GM 86] Goguen, J.A. and Meseguer, J. Eqlog: equality, types and generic modules for logic programming. *Functional and Logic Programming* (D. DeGroot and G. Lindstrom, eds.), Prentice-Hall (1986).

[GTW 78] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. An initial algebra approach to the specification, correctness and implementation of abstract data types, *Current Trends in Programming Methodology, Vol. IV* (R.T. Yeh, ed.), pp. 80-149, Prentice-Hall (1978).

[GMW 79] Gordon, M.J., Milner, A.J.R. and Wadsworth, C.P. *Edinburgh LCF.* Springer LNCS 78 (1979).

[Gut 75] Guttag, J.V. The specification and application to programming of abstract data types. Ph.D. thesis, Univ. of Toronto (1975).

[GH 83] Guttag, J.V. and Horning, J.J. Preliminary report on the Larch Shared Language. Report CSL-83-6, Xerox PARC (1983).

[GHW 82] Guttag, J.V., Horning, J.J. and Wing, J.M. Some notes on putting formal specifications to productive use. *Science of Computer Programming 2*, 53-68 (1982).

[Har 86] Harper, R. Introduction to Standard ML. Report ECS-LFCS-86-14, University of Edinburgh (1986).

[HMM 86] Harper, R., MacQueen, D. and Milner, R. Standard ML. Report ECS-LFCS-86-2, University of Edinburgh (1986).

[Hay 87] Hayes, I.J. (ed.) *Specification Case Studies.* Prentice-Hall (1987).

[Hoa 72] Hoare, C.A.R. Proofs of correctness of data representations. *Acta Informatica 1*, 271-281 (1972).

[Jon 80] Jones, C.B. *Software Development: a Rigorous Approach.* Prentice-Hall (1980).

[Jon 86] Jones, C.B. Systematic program development. In [GMc 86], pp. 89-109.

[Kam 83] Kamin, S. Final data types and their specification. *ACM Trans. on Prog. Languages and Systems 5*, 97-123 (1983).

[Kri 88] Krieg-Brückner, B. Algebraic formalisation of program development by transformation. *Proc. European Symp. on Programming*, Nancy, France. Springer LNCS (1988).

[Kri 87] Krieg-Brückner, B. *et al* (the PROSPECTRA project) Program development by specification and transformation. *Proc. ESPRIT Conf. '86*, pp. 301-312. North-Holland (1987).

[Les 83] Lescanne, P. Computer experiments with the REVE term rewriting systems generator. *Proc. 10th ACM Symp. on Principles of Programming Languages*, (1983).

[LHKO 87] Luckham, D.C., von Henke, F.W., Krieg-Brückner, B. and Owe, O. *Anna, a Language for Annotating Ada Programs: Reference Manual.* Springer LNCS 260 (1987).

[MW 79] Manna, Z. and Waldinger, R. Synthesis: dreams $\rightarrow$ programs. *IEEE Trans. on Software Engineering SE-5*, 294-328 (1979).

[MW 80] Manna, Z. and Waldinger, R. A deductive approach to program synthesis. *ACM Trans. on Prog. Languages and Systems 2*, 90-121 (1980).

[MW 81] Manna, Z. and Waldinger, R. Deductive synthesis of the unification algorithm. *Science of Computer Programming 1*, 5-48 (1981).

[Mar 82] Martin-Löf, P. Constructive mathematics and computer programming. *Proc. 6th Intl. Congress for Logic, Methodology, and Philosophy of Science*, pp. 153-175. North-Holland (1982).

[Mil 78] Milner, R.M. A theory of type polymorphism in programming. *J. of Computer and System Sciences 17*, 348-375 (1978).

[Möl 87] Möller, B. A survey of the CIP methodology. Research report, Technische Universität München (1987).

[NY 83] Nakajima, R. and Yuasa, T. (eds.) *The IOTA Programming System.* Springer LNCS 160 (1983).

[Par 86] Partsch, H. Transformational program development in a particular problem domain. *Science of Computer Programming 7*, 99-241 (1986).

[PS 83] Partsch, H. and Steinbrüggen, R. Program transformation systems. *Computing Surveys 15*, 199-236 (1983).

[PB 82] Pettorossi, A. and Burstall, R.M. Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique. *Acta Informatica 18*, 181-206 (1982).

[Rei 81] Reichel, H. Behavioural equivalence – a unifying concept for initial and final specification methods. *Proc. 3rd Hungarian Computer Science Conference*, Budapest, pp. 27-39 (1981).

[San 84] Sannella, D.T. A set-theoretic semantics for CLEAR. *Acta Informatica 21*, 443-472 (1984).

[ST 85] Sannella, D.T. and Tarlecki, A. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 67-77 (1985).

[ST 86] Sannella, D.T. and Tarlecki, A. Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Computer Programming*, Guildford, Springer LNCS 240, pp. 364-389.

[ST 87] Sannella, D.T. and Tarlecki, A. On observational equivalence and algebraic specification. *J. of Computer and System Sciences 34*, pp. 150-178 (1987).

[ST 88a] Sannella, D.T. and Tarlecki, A. Specifications in an arbitrary institution. *Information and Computation 76*, pp. 165-210 (1988).

[ST 88b] Sannella, D.T. and Tarlecki, A. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica 25*, pp. 233-281 (1988).

[SWa 87] Sannella, D.T. and Wallen, L.A. A calculus for the construction of modular Prolog programs. *Proc. 1987 IEEE Symp. on Logic Programming*, San Francisco, pp. 368-378 (1987).

[SWi 83] Sannella, D.T. and Wirsing, M. A kernel language for algebraic specification and implementation. *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden. Springer LNCS 158, pp. 413-427 (1983).

[Sch 86] Schoett, O. Data abstraction and the correctness of modular programming. Ph.D. thesis, University of Edinburgh (1986).

[SS 83] Scherlis, W.L. and Scott, D.S. First steps towards inferential programming. *Information Processing '83*, pp. 199-212. North-Holland (1983).

[Spi 85] Spivey, J.M. Understanding Z: a specification language and its formal semantics. D.Phil. thesis, Oxford University (1985); Cambridge University Press (1987).

[Spi 87] Spivey, J.M. An introduction to Z and formal specifications. Research report, Oxford University (1987).

[Suf 82] Sufrin, B. Formal specification of a display-oriented text editor. *Science of Computer Programming 1*, 157-202 (1982).

[Wik 87] Wikström, Å. *Functional Programming Using Standard ML.* Prentice-Hall (1987).

[Wirth 71] Wirth, N. Program development by stepwise refinement. *Comm. of the ACM 14*, 221-227 (1971).

[Wir 86] Wirsing, M. Structured algebraic specifications: a kernel language. *Theoretical Computer Science 42*, 124-249 (1986).

[Zil 74] Zilles, S.N. Algebraic specification of data types. Computation Structures Group memo 119, Laboratory or Computer Science, MIT (1974).