Yugoslav Journal of Operations Research
7 (1997), Number 1, 79-96

FORMAL SPECIFICATIONS IN SOFTWARE DEVELOPMENT: AN OVERVIEW

Vojislav B. MIŠIĆ, Dušan M. VELAŠEVIĆ
School of Electrical Engineering
University of Belgrade
Belgrade, Yugoslavia

Abstract: Formal methods find increasing usage for system and software specifications. In this paper, we discuss some benefits resulting from the use of such methods, together with some properties shared by most of them. Some possible criteria for classification are also presented, and a tabular overview is given of some of the most well-known methods. A number of known formal methods are reviewed, and their similarities and differences discussed.

Keywords: Formal specification methods, software specification, formal specification languages.

1. INTRODUCTION

Software development is generally understood to consist of several distinct, yet interconnected phases: requirements analisys, specification, conceptual and detailed design, implementation, verification and validation, evaluation and maintenance. Most, if not all, of these phases may benefit from the disciplined and knowledgeable use of formal methods. However, the term *formal methods* is often used in a rather narrow sense to designate a number of mathematically based techniques for system specification, i.e., for describing the structural and behavioral properties of software systems. (The same holds for computer hardware, and computer systems in general, as well as other systems.)

Although sometimes considered to be difficult to construct, validate, and use, formal specification methods are beginning to find their proper place in the software design process. The size and complexity of modern software systems has made it painfully clear that software cannot be successfully designed, implemented and verified without solid, systematic methodology. The lack of such methods results in what is generally known as the *software crisis*, and the use of appropriate formal methods for system specifications may help designers to overcome at least some of these problems [13].

In this paper we present an extended overview of formal methods and their use for the specification of software systems. A number of similar surveys has been published, including tutorial papers (e.g., [43]), historical surveys, and books (e.g., [8, 16]).

The rest of the paper is organised as follows: Section 2 discusses why formal methods should be used, and what benefits may be expected from their use. The subsequent section briefly presents some of the properties of formal methods regarding mathematical foundation, executability, tool support, and other issues. Some classification criteria are discussed in Section 4, and an informal treatment of some of the well-known techniques in use today is given in the next section. Although some of these are not being used for new development, their basic ideas were considered to be interesting enough to justify their inclusion in this discussion. Finally, an informal comparison of the methods presented is given in tabular form, and some directions for future developments are outlined.

2. BENEFITS OF USING FORMAL METHODS

As defined above, the term formal technique denotes a mathematically based technique for system specifications, i.e., for describing the structural and behavioral properties of software systems. The description is most often expressed with a special notation, usually a complete formal specification language. The syntax of such a language (as is the case with programming languages) defines a set of symbols and a set of rules for combining those symbols into correct sentence. Symbols correspond to the so-called specificand objects, the universe of which is sometimes called the semantic domain of a formal specification language. The interpretation of syntactic elements is provided (implemented) by the so-called satisfies relation [43]. A clear destinction should be made, however, between a formal notation (which is used to describe the system being specified) and a formal method, which is a development method using some particular formal notation (and other tools as well). Some of the techniques presented are merely formal notations with no associated development method, while other techniques constitute part of a mature method; this will be noted whereever appropriate.

Formal specifications let designers use abstractions, thus reducing the conceptual complexity of the system being designed. They enable the precise, consistent, and complete definition of system properties, and facilitate system decomposition into modules [29]. Precise mathematical definitions permit machine analysis and manipulation. By analysis, we mean that the consistency and correctness of specifications may be checked, thus revealing certain design errors and inconsistencies which are have to find in later phases; some clases of errors may be totally eliminated. By manipulation, we mean that one specification may be mechanically transformed into another one, more detailed or more complex than its predecessor, and (eventually) into an executable program. Note that this is exactly what any assembler, or language compiler, does.

In addition, since the transformation is performed according to a set of well-defined rules (otherwise it would not be possible to mechanize it), it may be proved to be correct. What is also susceptible to mechanical verification is the consistency, completeness, and correctness of the specifications. As most specifications have some kind of mathematical foundation, applicable inference rules may be used to reveal contradictions, missing definitions, and other flaws. Some of these errors may have a

significant impact on implementation, yet they may be very hard to find by classical testing and debugging.

In general, since the initial user requirements for the system being designed are always informal, it is not possible to prove that any specification satisfies user requirements - that is something only the user can say. At most, we can expect to be able to prove that one statement of user requirements is equivalent to another one, i.e., that the specification at one level is equivalent to the specification at another level.

Some authors claim that the use of formal methods can be beneficial even if no formal verification is attempted at all [22]. The mere fact that a rigorous specification is constructed forces the designer to do the job more thoroughly and reach a better understanding of the problem, which must lead to a better solution. As stated in [43]: "The greatest benefit in applying formal methods often comes from the process of formalizing, rather than from the end result."

The application of formal methods for system specifications redistributes the effort invested in system design phases: more effort (and the appropriate proportion of the total budget) is invested early in the design cycle, where errors have more impact. Since a lesser number of errors survive through the final implementation, testing and debugging may be shorter (and cheaper), yet more effective. (Although the overall project costs should be reduced, the redistribution of project costs is still somewhat unpopular among project managers.) Formal methods may also be used to advantage in designing testing methods and test suites, enabling more errors to be found and eradicated in a systematic fashion.

Note that many informal or semiformal approaches do exist, and some of them enjoy widespread usage. Examples of such methodologies include many variants of Structured Analysis and Structured Design, Jackson Structured Design, SREM, HIPO, and others. These methods give strong guidance to the designer in various stages of requirements analysis, specification and software design. In addition, their use of intuitive concepts and, in most cases, easy-to-grasp graphical formalisms, adds to their popularity.

However, the increased size and complexity of modern systems make them far less tractable by these methodologies, most of which evolved some ten or even twenty years ago. Still, their main deficiency is that they allow inconsistent and ambiguous specifications, either through the possibility of multiple interpretations of a concept, or through use of some informal mechanisms for specification, e.g., by a natural language text. Moreover, there is no way to prove that the specifications are satisfied by the design. This does not preclude their use, but none of the aforementioned benefits, inherent to formal methods, will result from it. Nevertheless, using an informal method is still better than not using any method at all.

3. SOME PROPERTIES OF FORMAL METHODS

A majority formal methods rely on some well-known, or cafully designed, mathematical foundations such as typed set theory (Z), logic of partial functions (VDM), many-sorted first-order logic (IOTA, OBJ), initial algebras (Larch) and the like. Other techniques adopt a less formal approach, their main objective being the possibility to use the specification language as some kind of high-level programming language which may be executed directly (i.e., after interpretation or compilation). A number of such languages have been proposed, some of them not even attempting formal verification.

Languages of this kind are often used for rapid prototyping in order to demonstrate the behavior of the specified system; such prototypes are usually discarded in favor of full-fledged applications [35]. This group of languages will not be covered in this paper.

From the mathematical point of view, specification(s) corresponds to language module(s), and executing the specification means that we are able to prove mechanically certain kinds of formulas of the theory defined by the specification [45]. In other words, one approach to executability follows the philosophy of stepwise refinement, and implements one specification with another, more detailed one. Each successive level of specification contains more design decisions made, until eventually a specification is obtained which is sufficiently detailed to be considered a program. An outstanding example of the refinement philosophy is the Ina Jo specification method.

More often, however, executability is taken in the sense that an algorithmic procedure exists for transforming specifications into executable programs, without intermediate refinement steps. This transformation may then be performed mechanically, using an interpreter or a compiler. Interpreters and/or compilers exist for a small number of methods only; often they implement only a subset of the language [18].

A hybrid approach is exemplified by the two-tiered Larch specifications, where specifications are first written in an implementation-independent, shared language. These specifications are the implemented with more detailed specifications written in an implementation-specific language, closely corresponding to the programming language which will be used for actual coding.

It should be noted that most specification languages emphasize brevity and clarity rather than executability. This facilitates both writing specifications and their mechanical verification, usually with the aid of a manual, user-assisted, or partially automated theorem prover (fully automated provers are not yet available).

The amount of information which must be maintained in today's medium-and large-scale software projects necessitates the use of automated assistence, e.g., syntax-directed editors, code management techniques, version control systems, etc. Note that the large amount of information is also the main reason for the inability of classical structured methodologies to deal successfully with such software projects. Although formal methods reduce the quantity of information to be handled, at least in the early design stages, they would certainly benefit from having some form of machine support.

As most formal methods utilize some nonstandard symbols, a dedicated documentation system may be considered a *conditio sine qua non*, although this need is easily fulfilled by modern word-processing software and typesetting systems. The aid of a syntax-directed editor is almost a necessity, and the next level of support could involve sophisticated syntactic and semantic checkers, in which knowledge of both aspects of the specification language used would be embodied. Theorem provers are used for verification, either manual or with some degree of user interactive guidance. Finally, code generators transform the specifications into executable code. At present, most formal methods have only limited machine assistance, up to the level of theorem proving, while only a few are supported by code generators. However, one may "benefit from the use of formal methods without the use of sophisticated support tools. The advantages of formality are too significant to await the development of a fully supported method" [10].

As always, the integration and interoperability of these tools play a crucial role in their effectiveness. If tools are designed separately, without proper communication mechanisms built-in right from the start, their usability will be significantly reduced.

An integrated design environment should be equipped with dedicated tools for performing various tasks. A simple collection of tools is not enough: for maximum effectiveness good coordination should exist among various tools, and the transition between tasks should be smooth. Only then will the designer be freed of most, if not all, housekeeping functions, so that she/he can concentrate on specification. An ideal design environment should support all activities in each phase of software design; an interesting attempt in that direction is described by Blum [11].

Support and coordination for team development is also a very important issue, given the size and complexity of even modest projects today. This includes mostly version control, and (to a lesser extent) some support for multiuser operation. Formal specifications are still designed by rather small teams, hence the usual problems of concurrency, recovery and the like are not too significant.

Reusability is one of the key issues in all modern software design methodologies, and formal methods are no exception. Libraries of predefined constructs, describing commonly used modules (whatever they may be called in any particular language) enable the specifier to concentrate on the particularities of the system being modeled - without having to reinvent the wheel each time. Moreover, as specifications found in these libraries are usually written by the authors of the language themselves, they are usually well designed and thoroughly tested. The specifier may use them as examples of good style, after which her/his own specifications may be patterned. This is often hard to find in classical software systems, since most of the code written by experienced programmers is copyrighted and unavailable, while textbooks contain mostly toy codes, unfit for real-size projects [42]. It should be noted that a number of quite good formal libraries exist, but the organizations which have these are unwilling to make them widely available because of commercial reasons.

4. GENERAL CLASSIFICATION OF FORMAL METHODS

Specification languages may be classified according to different characteristics. One of the classification criteria, and possibly the most important one, is based on the mathematical foundation of these methods. According to this criterion, we may discriminate between *model-* and *property-oriented* languages. Model-oriented languages provide a twofold description of system behavior: first, the data structures (such as strings, numbers, sets, tuples, relations, sequences, etc) that constitute the system state are described. Then, the operations that manipulate that state are defined using assertions in a notation similar to first-order predicate calculus. These languages are sometimes called *constructive*, and they include Z, VDM, Milner's CCS, Hoare's CSP, and others.

Property-oriented methods define the system in terms of properties that must be satisfied in some or all of the system states. The latter include the so-called axiomatic approaches, such as those embodied in specification languages IOTA, OBJ, Larch, and Anna, as well as the so-called algebraic approaches, e.g., the one used in LOTOS. However, the data structures are described here as well, the most common being sets, lists, sequences, and other structures with a straightforward mathematical interpretation. The desired set of properties is usually expressed as a set of equations. Equations may be interpreted as directional rewrite rules, which may be used to simplify a statement by reducing it to some canonical form. Term rewriting is one of the most important research directions in connection with formal methods, as witnessed by several conferences and special journal issues.

In another classification, the distinction is made between specifications that

concentrate on internal structure of specificand objects (structural specifications), and those that predominantly describe the observable behavior of these objects (behavioral specifications). However, structural specifications often include some notions of specificand behavior, and vice versa, thus blurring the distinction between the two kinds of specifications.

As specification methods deal with large systems and data structures of appropriate size and complexity, support for specification modularization and structuring is essential. Some rather popular formal methods provide no such support, or a limited amount only (as is the case with Z and VDM). This is often considered to be their main defficiency, and extensions aimed to correct it have been reported, e.g., in [38, 30]. Most other methods have some provisions for modularization, usually in the form of one or more of the so-called *specification-building operators*. In its simplest form, one specification can import another one as a subspecification, enriching it with additional sorts and operators. The entire model is organized then in the form of a hierarchy (or, more generally, a directed, acyclic graph) of specifications. This is somewhat similar to the inheritance mechanism of object-oriented languages. Other methods provide more sophisticated mechanisms for importing subspecifications, offering the possibility to redefine some of its sorts, or further constrain their behavior by redefining some of the applicable operators.

Another important mechanism is that of *parameterization*, "a process of encapsulating a piece of software and abstracting from some names occuring in it, in order to replace them in other contexts by different actual operators" [45]. Together with structuring and modularization support, parameterization frees the designer from reinventing standard specifications, letting her/him concentrate on the particulars of the problem at hand. As is the case with other software components, standard specifications may be distributed in the form of libraries, thus promoting reusability. Such libraries are still not available for most specification techniques presented here; instead, the specifier must resort to case studies and introductory or reference texts, whatever is available.

Most specification languages contain only declarations and equations; if imperative and/or applicative style statements are included as well, the language is considered to be a wide-spectrum one [7].

A number of languages concentrate on functional properties of the system being modeled, while dynamic concepts (e.g., concurrency, real-time behavior, reliability, security, performance), cannot be easily accounted for in these specifications. Other languages have been specifically designed for systems in which the dynamic component is crucial. Most of these are based on some extension of the finite state machine concept; among the most distinguished are the well-known CCS and CSP formalisms, as well as a class of languages specialized for telecommunication systems, such as LOTOS [2], Estelle [3], and SDL [1]. We should also mention Petri Nets, known since the late 1960s, and their numerous variations (e.g., [31]). Petri Nets have been a particularly successful formalism for real-time system specification and analysis, as exemplified in both research and industrial projects. They have been coupled with other formal notations in order to extend versatility to a wider class of systems (e.g., [41]).

Instead of designing a completely new language, some researchers have been extending languages of the former group to incorporate some dynamic concepts. Examples of such an approach (which we might call evolutive) are extensions of the Ina Jo language with temporal logic [44], and the Maude language, which contains OBJ3 as a functional sublanguage and incorporates some novel concurrency concepts [28]. In ot-

her words, these languages may be considered attempts to get the best of both worlds by augmenting already existing specification languages with the desired properties. These issues are still new and more experience needs to be gathered before some definitive conclusions can be made; it seems a promising avenue anyway, and important results are to be expected.

5. REVIEW OF SOME FORMAL SPECIFICATION TECHNIQUES

A brief description of the characteristics of some of the well-known specification languages will be given in the following. We make no claim about the exhaustiveness of our selection, as many other methodologies exist, some of which we did not even mention. However, we believe our selection to be representative, in the sense that most directions of past and current research have been given attention, and that some of the most important and informative developments are covered. A notable exception is methods for the specification of real-time systems, since they are deemed to be still in their infancy, and that significant developments are yet to appear. The languages selected are ordered by decreasing degree of formalism utilized, and increasing ease of transformation of specifications into executable code.

5.1. Z

Z is one of the most popular specification notations. It is based on typed set theory, and uses familiar mathematical concepts like sets, relations, functions, etc., structured in the form of schemas. A schema consists of a declaration part, and a predicate part. Variables are declared, in the former, and the latter defines predicates relating these variables. Schemas are a convenient way of structuring the specifications, since they may be combined using several simple operations, e.g., conjuction, disjunction, implication, component renaming, composition, piping, and others as well. Operations which combine the declaration parts, or predicate parts, or both, of two (or more) schemas facilitate the modular construction of Z specifications. All of these operations, forming together what is usually designated as the *schema calculus*, are defined in accordance with the well-known rules of propositional and predicate calculi.

Z is able to define both deterministic and nondeterministic functions, and Z specifications are capable of encapsulating the state of a system (a model of a system, to be precise). It has little or no provisions for incorporating time dependencies, although some attempts have been made in that direction. Another promising avenue which is currently being explored is to extend Z notation with the class concept, together with the powerful encapsulation and inheritance mechanisms commonly found in object-oriented systems. Important system aspects such as object integrity and interobject communication can be formaliy specified. A number of techniques which extend Z with object-oriented concepts are reported in [38].

Formal proofs are given for Z specifications only occasionally, since the clarity of Z schemas contribute to their simplicity and expressive power without compromising the necessary rigour. Instead of proving a formal relationship between specifications and their implementations, Z specifications are used to explore the properties of those specifications, often quite informally. It is found that even informal arguments may lead to the detection of serious implementation errors, as witnessed in [23, 37].

However, "if it is difficult to reason about some expected property, it is usually a sign that the specification is poorly structured, if not wrong" [19].

Although the Z notation is very popular, there is no single Z development method rather quite a number of them (e.g., [5, 38]).

Z is extensively used for the specification of various software systems, including specifications for a large part of the CICS transaction control system [23], specification of the UNIX file system, specifications for several oscilloscope subsystems (e.g., [19]), and even hardware specifications [6]. Several textbooks exist, among these a reference manual [36], a collection of case studies [23], and a style manual [5]. Furthermore, work is under way to establish an ISO standard for Z.

A number of both research and commercial tools support Z, and it is worth noting that Z was the first formal language to obtain an electronic distribution list, with others to follow (some of which are listed in [13]). Z-related information currently available on the Internet includes an extensive bibliography [12], a number of papers and research reports, and a syntax checker.

5.2. VDM

VDM is one of the best-known formal methodologies. It is a constructive, or model-based specification technique, based on propositional and predicate calculus and the logic of partial functions which is used to circumvent the undecidability problem of standard first-order logic.

A VDM specification consists basically of two components: a model of the state, with whatever invariants must hold for it, and operations over the abstract data type comprising the state. Operations are defined implicitly, i.e., an operation is given in terms of its signature, the necessary pre-condition, and the post-condition which must hold after the operation has been performed. Implicit functions are preferred to explicit ones, mostly for reasons of clarity and ease of verification; explicit definition, however, must be used in the implementation phase.

The post-condition is defined in terms of parameter values after (and before, if necessary) the operation. An operation can access (and change) some external variables which comprise the system state; the type of access must be noted for each variable (i.e., whether it is modified - written, or just read), in each operation definition. The use of external variables facilitates the distinction between parameters and variables which are accessed by a side-effect; the choice is usually pragmatic.

State descriptions are defined at successive levels of abstraction, linked by implementation steps. The implementation of an abstract state by means of a more concrete one describe either a data reification or an operation decomposition. In the former case, state variables of a more concrete state implement those of a more abstract one, while in the latter, the operations of a more abstract state are implemented by those of a more concrete one. A distinguishing feature of VDM is that state descriptions and implementations must be formally verified; VDM provides rules to systematically derive proof obligations from object descriptions. Proof obligations are required to verify both the correctness of a specification, and its implementability. Correctness is concerned with the transformation from an abstract to a concrete state, and implementability verifies transformations from implicit functions to those defined explicitly.

The data reification constructs allow the designer to use abstract data structures in the specification, without too much concern about their implementation;

final data structures may be determined later, after the specification phase has been terminated. Again, proof obligations permit verification to be performed in a systematic and rigorous fashion. Automated tools exist which are capable of generating these obligations from the specification text. Actual proofs may be carried out either manually or with the aid of a theorem prover, such as B [4], or the Boyer-Moore prover [14].

However, VDM is not without its deficiencies. Some implicitly predefined data types may have additional properties beyond those explicitly defined and even some unwanted properties and side affects; other languages require all properties to be explicitly stated. Although states and operations form a kind of hierarchy, VDM has no explicit provisions for modularization. Therefore, when a proof obligation is constructed for an object, it is often large, containing some subgoals which are trivial to prove, as well as some subgoals which were already proved. This is particularly apt to appear in cases when two arbitrary state descriptions are grouped, since all obligations of both objects must be re-proved in order to account for the presence of the other object. Some of the difficulties that arise would be easily filtered out by imposing some visibility rules on variables, something which is not available in the current definition of VDM. Existensions have been proposed to support modularity (e.g., [30]) and to interface VDM with some complementary methodologies, such as Object-Oriented Analysis and Structured Analysis (e.g., [40]), but no standard set of extensions has been universally accepted yet.

VDM was initially devised for the specification of a large subset of PL/I programming language, and subsequently used in the development of a number of software and hardware systems. It is an established methodology with extensive documentation including a number of textbooks, ranging from an introductory course [24] to a collection of a case studies [25], and it has even been proposed as an official British project, which includes both a full-fledget methodology and substantial tool support [33].

5.3. IOTA

The specification language IOTA [32], although not a modern development, presents an excellent example of modular programming with parameterization mechanisms. IOTA specifications may be built bottom-up, starting from built-in modules such as *bool* and *int*, and extending these declarations as necessary. Declarations are grouped into modules which are themselves organized into a strict hierarchy with violations and circular dependencies being detected by the IOTA model processor.

IOTA specifications are theories of a many-sorted first-order logic. Each sort is associated with a so-called basic structure on that sort, consisting of a finite set of functions and a finite set of axioms. These functions are defined in the interface part, while axioms which characterize their properties (i.e., behavior) are grouped in the specification part of the module. Functions thus defined are called primitive functions on the underlying sort.

A primitive function whose range is the sort itself is called the *induction rule* for that sort. A sort for which induction rules are defined is called a *type*. It is possible to define sorts without induction rules, in which case the sort is called a *sype*. There are other subtle differences between types and sypes, but they are of no concern to us. It

should be noted that within each sort the IOTA system automatically adds equality to the primitive functions, and equality axioms to the sort's basic axioms.

More functions may be added with a procedure module, which again has both interface and specification parts. A procedure module defines, in effect, a procedural abstraction and it may introduce several procedures. Functions defined in a procedure module are considered non-primitive.

Finally, a realization part is provided, defining the implementation of a module in terms of other modules. The implementation language is syntactically similat to CLU, but the semantics are different. In particular, recursive definitions are not allowed to span multiple modules, in part because of the strict hierarchy imposed by the IOTA system. However, recursive and mutually recursive functions may appear within a single module. In a sense, the interface and specification part define an *abstract* view of a function, while the realization part gives its *concrete* view.

IOTA modules represent theories, or parts of a theory defined by the entire program. Verification of an IOTA specification consists of proving that the theory of each module is satisfied by its realization. The Prover component of the IOTA system reduces a goal formula into smaller subgoals, until they become small enough to be effectivaly handled by an automatic simplifier and resolution prover. The proof system necessarily relies on interactive guidance by the user, since even small module definitions may result in long formulas to prove, and fully automated theorem proving would take too long to terminate. Proofs are further complicated by the modularity of IOTA specifications, which often leads to formulas with a large number of user-defined axioms. Fortunately, a well thought-out modularity often results in the so-called proof locality property: a number of proof steps tend to depend on axioms from only a few, or even a single module. Hence, the interactive proof process can be made more efficient by narrowing the selection of axioms at each step, thus facilitating the application of reduction and simplification rules on a module. This is known as *theory-focusing* strategy.

Functions may be defined as specifications only, without the associated implementation. Verification of these functions is not possible, therefore this practice is generally discouraged since it may create "gaps" in consistency proofs. On the other hand, the realization part may contain definitions of local functions which have no abstract counterparts. The verification process is connected with another aspect of modularization in IOTA: namely, sypes may be used to model some general pattern of behavior, and utilized (included) within many different types. If proof procedures are developed for such sypes, they are readily available for proving appropriate properties in all types based on these sypes.

The significance of IOTA lies mainly in the following:

- it is a complete specification methodology with facilities for specification at both abstract and implementation levels (something which is absent in many other approaches), and

- an integrated tool-set is available built-in order to facilitate the interactive development of specifications and their implementation in the form of executable programs.

Automated support is made available in all phases of the development process. Module text is entered via the Syntax-directed Editor, and various tools for syntactic and semantic analysis are provided, in order to detect errors as early as possible. Modules found to be correct are stored in a dedicated database (module base), and an Executor subsystem translates them to object code. The verification is managed by the Verifier, and the actual proofs are carried out by the Prover. The system supports multiple-member programming teams, using the module base as a sophisticated data dictionary. Modules have owners and visibility, which is controlled by the system under user guidance. Furthermore, versioning is provided at both module and function levels, in order to retain the consistency of declarations after modifications.

5.4. OBJ

OBJ is an algebraic specification language which resembles IOTA in many details, both in syntax and semantics, yet some significant differences exist. OBJ has been one of the most popular languages, with several variations developed over the years (OBJ-T, OBJ2, OBJ3, etc.) [17, 18, 39]. An OBJ specification is a collection of equationally specified sorts (which implement abstract data types), a collection of operators defined in terms of these sorts, and a collection of algorithms designed using these sorts and operators.

An object is declared by a textual unit, which defines the underlying data sorts and their associated operations. Each sort name denotes a set of values, called the carrier of the sort; sort names are available within the scope of the object in which they were introduced. Operations are defined in terms of their signatures (syntax), and equations describing their properties (semantics). The scoping rules for sorts apply to operators as well. Necessary variables must be decalred in advance; they are considered to be universally quantified over the whole equation in which they are used. Constants of any sort may also be declared, while the TRUTH object with distinguished constants T and F, is built-in; these constants may be accessed from any object defined by the user. Each sort is automatically augmented with a boolean equality operator, much like the IOTA approach. Generally, an OBJ object semantics is defined as the initial algebra on the signature denoted by the sort and operator declarations.

Objects in an OBJ specification are structured in an acyclic graph structure, as each object may be defined as a refinement (or extension) of an already existing one. As OBJ objects are similar to the class concept of Simula and other object-oriented programming languages, this process may indeed be viewed as inheritance. Several import mechamisms exist (using, protecting, and extending), with different restrictions on imported data sorts and operators, thus allowing precise tuning of specifications. Not all versions of OBJ have all of these mechanisms available: at least two of them provide only the simplest (using) import mechanism [18, 39]. Other modularization mechanisms available in OBJ (again, not all versions support all of them) are the ability to define subsorts of a data sort, parameter application, and renaming.

Several interpreters are available for OBJ, but they usually implement only a subset of the full language [18, 39]. Nevertheless, the availability of these and their associated provers makes the design of OBJ specifications easier and more comfortable.

OBJ has no provisions for specifying dynamics properties and/or temporal behavior. It has recently been used as the basis for the language Maude, which may be briefly (and somewhat incorrectly) described as the OBJ3 language with the addition of

concurrent rewriting concepts [28]. This area is gaining attention, and important results should follow.

5.5. Larch

The Larch specification language forms just part of a larger project seeking to build tools that facilitate the construction of formal specifications for modular programs.

Larch specifications follow a two-tiered approach, similar in spirit (if not in detail) to the IOTA approach. Abstract specifications are written in the Larch Shared Language and they are independent of any particular implementation. These specifications are transformed to another language, the Larch Interface Language, in order to describe program units (modules, functions, procedures, types, ...) used for implementation in the chosen programming language. Several Larch interface languages exist, each designed to make best use of the facilities available in the appropriate programming language.

Larch Shared Language specifications are modularized in *traits* which introduce operators and specify their properties. Traits often correspond to abstract data types; other traits capture some useful properties which may be shared by many other traits, not unlike the concept of sypes found in IOTA. The operator set is declared in terms of operator symbols and their signatures, while their properties are described a theory: a set of theorems which may be derived from properties defined in the trait, using axioms and inference rules of first-order predicate calculus.

Traits may be defined by importing (i.e. extending) the definition of one or more previously defined traits, renaming some terms if necessary. As in OBJ, these extensions come in various flavors; they may additionally constrain some of the previously defined sorts and operators, while others are just used on an "as is" basis, without further manipulation. This variety of extension mechanisms enables the specification to exploit reusability to the fullest, yet retain the precision which would otherwise be possible only through a dedicated specification, written from scratch. Hiding or export mechanisms are not available in the shared language, being better suited to interface specifications which are closer to the actual implementation language.

The Larch Prover, a mechanical theorem-proving tool, facilitates verification of the correctness of specifications. It is based on the same equational first-order logic as the Larch Shared Language, with a number of built-in inference rules used for rewriting. Additional proof mechanisms are included to help overcome certain difficulties with completeness in a rewriting system, and to prevent the generation of nonterminating rewriting sequences. An automatic mechanism for rule ordering is available, although machine-assisted partial ordering is also provided. Proofs are initiated by the user through an interactive dialogue and a number of inference techniques may be applied; in most cases, strong user guidance is requested. Thus, the Larch Prover may be used in the same way as an interactive debugger is used in classical programming environments [20].

Larch interface languages are used to specify the interfaces between program components in the form of information necessary to write a program unit and to use it. The interface languages provide communication and exception handling, iterators, side effects, and other mechanisms similar to those found in the implementation language. The similarity between interface and implementation languages has a twofold advanta-

ge: it makes specifications shorter and more precise and it makes actual implementation easier since the specification is closer, in form and function, to the final program code. Interface languages are currently available for CLU, Pascal, and Ada, as well as for other programming languages. Larch has been applied to various software and hardware projects and a variety of reports are available, together with a library of reusable specifications [21].

5.6. FDM and the specification language Ina Jo

As noted above, one of the possible approaches to system specifications is based on the process of successive (stepwise) refinement, where each formal specification is transformed to the next level by adding more and more functionality and/or implementation details. An example of such an approach is the Formal Development Method (FDM) and its associated specification language Ina Jo [9, 26].

The process starts with an informal statement of system requirements, which is gradually converted into a complete formal specification. Each formal specification is named a *level of refinement* in Ina Jo terminology. Transformation to an executable program is performed only when the final specification is obtained. This final specification (i.e. level of refinement) is the only one that is required to be functionally complete. The correctness of the transformation from one level of refinement to the next one is checked with the aid of a machine theorem prover (ITP).

Ina Jo is based on an extension of first-order predicate logic: it treats the system and its data as a state machine, with the internal data making up the state. A machine, as described through its specification, is formed by a set of variables, each capable of holding a value of some type. A type is a predefined set of values; it would be called the carrier of a sort in OBJ terminology. A hierarchy of specifications contain descriptions of the underlying state together with initial conditions, transforms, and assertions.

A transform specification defines a set of preconditions (reference conditions) and a postcondition (effect), much like in VDM. A distinction is made between deterministic and nondeterministic transforms, the former having a unique ending state for each starting state. Nondeterminism, which may be introduced through reference conditions and through an effects clause, further complicates the Ina Jo proof system. For simplicity, the implementation of each transform is assumed to halt except at the final (code) level where this property must be verified. Modularization is partially supported since transforms may use other, previously defined transforms.

Assertions state properties of various kinds. Axioms state global properties which must hold in all states of all models. An assertion which holds in each state is termed a *criterion*, while those that are valid for each pair of consecutive states in a computation are termed *constraints*. (A computation is a sequence of states.) There are also *initial* state assertions and named *define* assertions which can be used as macros. In general, assertions should be provable, given the machine specifications and its associated set of possible computations.

Like other specification techniques, Ina Jo suffers from a lack of dynamic concepts and temporal modelling facilities; some extensions in that direction have been reported in [44].

5.7. Anna

All the languages discussed so far have been designed without any reference to any particular programming language, with the exception of Larch interface languages. Thus, the designers had no need to worry about the idiosyncracies of any existing programming language, at least at the abstract level. A rather different approach was taken by the designers of the specification language Anna (ANNotated Ada) [27]. Anna is an evolutionary approach, an attempt to extend the definition of Ada83 [1] to support the following:

- to extend and ameliorate activities of explanation,
- to add some new language constructs, mainly in the area of exception handling, context clauses, and subprograms, and
- to add specification constructs, predominantly in package semantics, and composite and access types.

Only the last group of extensions is relevant to our discussion. Since other intended uses exist beyond specification, the Anna programmer is free to specify as much, or as little as she/he wants - there is no requirement of completeness of specifications. Nevertheless, axiomatic semantics may be defined by the Anna language to be used later to verify the correctness of Ada programs vs. their original Anna specifications. In other cases, specifications are used to generate additional code in the form of run-time checks which may be used for testing and debugging Anna programs.

Annotations may be added for constraining sets of observable states of a program or for constraining values of program variables within the scope of annotation. They may also be used for specifying subprogram units independently of bodies that implement them, e.g. by constraining the propagation of an exception condition.

A particularly interesting feature of Anna is the possibility to define package axioms. These axioms state some properties of visible entities in the package which are promised to hold within the scope of its declaration. They may also be used to constrain local entities in the hidden part of the package.

It should be noted that extensions defined by Anna are designed to be upward compatible with existing Ada syntax. Standard Ada compilers treat Anna constructs as ordinary comments, hence all valid Anna programs are valid ordinary Ada programs as well. Anna specifications are translated into excutable Ada code by special preprocessors, while other tools (analyzers, theorem provers, etc) can be used for verification and other purposes.

An interesting development is reported in [46] where Anna is used to bridge the gap between Z and Ada. Initial specifications are written in Z and implemented by Anna specifications; Anna specifications are in turn used to develop Ada programs from which executable code is produced. It is argued that the process of refinement from specification to implementation is eaiser when performed in two smaller steps rather than in just one step (i.e. if Z specifications were directly transformed to Ada). This research should also investigate the possibility of constructing automated tools to assist in the refinement process.

---

[1] To the best of authors' knowledge, there has been no attempt to provide a similar extension for the latest Ada standard, Ada9X.

5.8. A rough comparison of the techniques presented

Some pertinent properties of the formal specification techniques presented in this overview are conveniently summarized in Table 1. The Anna language is not included in the comparison as it does not easily fit in these categories, being of a somewhat different character.

Table 1: Comparative review of some specification languages.

property	Z	VDM	IOTA	OBJ	Larch	Ina Jo
mathematical orientation	model	model	theory	theory	theory	model
theoretical foundation	typed set theory	logic of partial functions	many-sorted first-order logic		initial algebras	first-order logic-based state-machine
structural/ behavioral	structure-inclined	both aspects supported				behaviour-inclined
modularization support	weak, but numerous extensions exist		fully modular			weak
object support	numerous extensions		weak	inherent in language		weak
inheritance/ import	weak		generic concepts (via sypes)	several variants		limited support
parameterized specifications	generic concept	none		fully supported		
standards	informal	official	none, or internal as multiple versions exist			
implementation	manual, or via automated tools		part of specification	interpreted	Larch interface languages	from specification
general tool support	numerous tools available		integrated environment	interpreters available		
verification tools	external theorem provers		integrated theorem provers			

The methods and techniques are compared with respect to the following properties which were discussed in more detail in Section 3:

- mathematical orientation, i.e. whether the technique is generally model- or property-oriented (although the distinction is not always clear);
- theoretical foundation upon which the technique is based;
- relative predominance of structural vs. behavioral concepts;
- modularization support provided by the technique;
- compatibility with modern object-oriented methods;
- support for reuse through inheritance and imports of other specifications;
- capability for writting parameterized specifications;
- existence of official or de facto standards (official standards exist, or will exist in near future, for Z and VDM);
- how the specifications are transformed to more detailed and/or executable form, i.e. either manually or with a specific tool;
- general level of automated tool support, as of this writing;
- availability of verification tools, both specialized and general.

6. CONCLUSION

In summary, we may safely conclude that formal specification methods are an indispensable tool to aid designers in the software development process. Their main objectives are to enhance the precision, consistency, and completeness of system specifications, and to enable machine-aided analysis and manipulation of these specifications. A representative, though far from exhaustive, list of methods and techniques has been presented in some detail, and their pertinent properties compared. However, these methods are not without problems: most of them lack serious dynamic modeling caapabilities. In most languages, a significant gap still exist between formal specifications and executable code. Finally, tool support is still inadequate and robust integrated environments are yet to be developed. On the positive side, these shortcomings are likely to be corrected in time, significant industrial experience has been accumulated, and standards for some of the methods have been established, or will be the near future. The use of formal methods offers benefits which cannot be overlooked, and a working knowledge of at least one of these techniques may already be considered as something that designers of modern software systems simply cannot do without.

Acknowledgments: The authors would like to thank the anonymous reviewers for valuable comments and suggestions.

REFERENCES

[1] CCITT/SGX/WP3-1, Specification and Description Language SDL, *CCITT Recommendations, Z.100-Z-104*, 1988.

[2] ISO, *Information Processing Systems, Open Systems Interconnection: LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, IS 8807, 1989.

[3] ISO, *Information Processing Systems, Open Systems Interconnection: Estelle - A Formal Description Technique Based on an Extended Finite State Transition Model*, IS 9074, 1989.

[4] Abrial, J.R., "The B tool", in: Bloomfield, R., Marshall, L., Jones, R. (editors), *VDM'88 - The Way Ahead*. Springer, Berlin, 1988.

[5] Barden, R., Stepney, S. Cooper, D., *Z in Practice*, Prentice Hall International, 1994.

[6] Barrett, G., "Formal methods applied to a floating point number system", *IEEE Transactions on Software Engineering*, 15 (1989) 611-621.

[7] Bauer, F.L, Möler, B., Partsch, H., Pepper, P., "Formal program construction by transformations - computer-aided, intuition-guided programing", *IEEE Transactions on Software Engineering*, 15 (1989) 165-180.

[8] Berg, H.K., Boebert, W.E., Franta, W.R., Moher, T.G., *Formal Methods of Program Verification and Specification*, Prentice Hall International, 1982.

[9] Berry, D.M., "Towards a formal basis for the formal development method and the Ina Jo specification language", *IEEE Transactions on Software Engineering*, 13 (1987) 184-201.

[10] Bloomfield, R., Froome, P.K.D., "The application of formal method to the assessment of high integrity software", *IEEE Transactions on Software Engineering*, 12 (1985) 988-993.

[11] Blum, B.I., "A paradigm for developing information systems", *IEEE Transactions on Software Engineering*, 13 (1987) 432-439.
[12] Bowen, J.P., "Z bibliography," Oxford University Computing Laboratory", 1990-1995.
[13] Bowen, J.P., Hinchley, M.G., "Seven more myths of formal methods", *IEEE Software*, 12 (1995) 34-41.
[14] Boyer, R., Moore, J., *A Computational Logic*, Academic Press, New York, 1979.
[15] BSI, *VDM Specification Language, Proto-Standard*, IST, 5/50, 1989.
[16] Cohen, B., Harwood, W.T., Jackson, M.I., *The Specification of Complex Systems*, Addison-Wesley, 1986.
[17] Futatsugi, K., Gougen, J.A., Jouannaud, J.P., Meseguer, J., "Principles of OBJ2", in: *Proceedings of ACM Symposium on Principle of Programming Languages*, 1985.
[18] Gallimore, R.M., Coleman, D., Stavridou, V., "UMIST OBJ: A language for executable program specifications", *The Computer Journal*, 32 (1989) 413-421.
[19] Garlan, D., "The role of formal reusable frameworks", in: M. Moriconi (editor), *Proceedings of ACM SIGOSOFT Workshop on Formal Methods in Software Development*, Napa, CA, 1990, 42-44.
[20] Garland, S.J., Guttag, J.V., Horning, J.J., "Debuging Larch shared language specifications", *IEEE Transactions on Software Engineering*,16 (1990) 1044-1057.
[21] Guttag, J.V., Horning, J.J., Wing, J.M., "The Larch family of specification languages", *IEEE Software*, 2 (1985) 24-36.
[22] Hall, A., "Seven myths of formal methods", *IEEE Software*, 7 (1990) 11-19.
[23] Hayes, I. (editor), *Specification Case Studies*, Prentice Hall, Hemel Hempstead, UK, 2nd edition, 1993.
[24] Jones, C.B., *Systematic Software Development Using VDM*, Prentice Hall, Hemel Hempstead, UK, 2nd edition, 1990.
[25] Jones, C.B., Shaw, C.F., *Case Studies in Systematic Software Development*, Prentice Hall, Hemel Hempstead, UK, 2nd edition, 1990.
[26] Kemmerer, R.A., "Integrating formal methods into the development process", *IEEE Software*, 7 (1990) 37-50.
[27] Luckham, D.C., von Henke, F.W., "An overview of Anna, a specification language for Ada", *IEEE Software*, 2 (1985) 9-22.
[28] Meseguer, J., "A logical theory of concurrent objects", in: N. Meyrowitz (editor), *Proceedings of OOPSLA/ECOOP Conference*, Ottawa, Canada, 1990, 101-115.
[29] Meyer, B., "On formalism in specifications", *IEEE Software*, 2 (1985) 6-26.
[30} Middleburg, C.A., "VVSL: A language for structured VDM specifications", *Formal Aspects of Computing*, 1 (1989).
[31] Murata, T., Petri Nets: "Properties, analysis and applications", *Proceeding of the IEEE*, 77 (1989) 541-580.
[32] Nakajima, R., Yuasa, T. (editors), *The IOTA Programming System*, Springer, Berlin, 1983.
[33] The RAISE Method Group, *The RAISE Development Method*, Prentice Hall International, London, 1994.
[34] Semmens, L.T., France, R.B., Docker, T.W., "Integrated structured analysis and formal specification technique", *The Computer Journal*, 35 (1992) 600-610.
[35] Sluizer, S., Lee, S., "Applying entity-relationship concepts to executable specifications", in: S. Spaccapietra (editor), *Porceedings of the Fifth International Conference on Entity-Relationship Approach*, Dijon, France, 183-194.
[36] Spivey, J.M., *The Z Notation: A Reference Manual*, Prentice Hall, Hemel Hempstead, UK, 1989.
[37] Soivey, J.M., "Specifying a real-time kernel", *IEEE Software*, 7 (1990) 21-28.
[38] Stepney, S., Barden, R., Cooper, D. (editor), *Object Orientation in Z*, Workshops in Computing, Springer-Verlag, 1992.
[39] Tardo, J.J., "The design, specification, and implementation of OBJ-T: A lnguage for writing and testing abstract algebraic program specifications", PhD Thesis, UCLA, Los Angeles, 1981.

[40] Toetenel, H., van Katwijk, J., Plat, N., "Structured analysis - formal design", in: M. Moriconi (editor), *Proceedings of ACM SIGSOFT Workshop on Formal Methods in Software Development*, Napa, CA, 1990, 118-127..

[41] van Hee, K.M., Semmens, L.J., Voorhoeve, M., "Z and high level Petri nets", in: S. Prehn and, W.J. Toetnel (editors), *VDM'91: Formal Software Development Methods*, Vol. 551 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991, 204-219.

[42] Weiser, M., "Source code", *Computer*, 20 (1987) 66-73.

[43] Wing, J.M., "A specifier's introduction to formal methods", *Computer*, 23 (1990) 8-24.

[44] Wing, J.M., Nixon, M.R., "Extending Ina Jo with temporal logic", *IEEE Transactions on Software Engineering*, 15 (1989) 181-197.

[45] Wirsig, M., "Algebraic specifications", in: J. van Leeuwen (editor), *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, Elsevier, Amsterdam, 1990, 675-788.

[46] Wood, W., "Application of formal methods to system and software specifications", in: M. Moriconi (editor), *Proceedings of ACM SIGSOFT Workshop on Formal Methods in Software Development*, Napa, CA, 1990, 144-146.