# Application of Formal Methods in Development of Information Systems

**Željko Dobrović, Alen Lovrenčić**
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
{zeljko.dobrovic, alen.lovrencic}@foi.hr

**Abstract**. *During the 1960s IS community faced the failure of unsuccessful development of complex information systems, in spite of heaving large computers and higher programming languages available. This situation is known as "software crisis" and solution is recommended at conferences sponsored by NATO in 1968. and 1969. After participants have come up with the conclusion that more engineer-like discipline is needed in IS development, the term "software engineering" was introduced. Software engineering was based upon some formal methods that should be used in software development process. Since then philosophy that underpins the formal methods hasn't changed. Numerous methods and methodologies have been developed for supporting the IS development in last three decades. Majority, if not all of them, are based on common foundations provided by formal methods. However, the importance of formal methods decreased as development of structured methods shifted from programming to the analysis of IS, because analysis doesn't look so "formal". Great number of IS developers nowadays use contemporary IS development methods without even being aware of formalism that lay inside these methods. The authors in this paper elaborate the formal methods and propose the possible area of their usage in information system development.*

**Keywords.** formal methods, information systems, methodology

## 1 Introduction

In the 1960s, the information community was characterised by two significant trends. On the one hand, powerful computers (hardware) appeared, on the other higher programming languages were available. This resulted in the use of computers in the construction of information systems for various types of organisations. A large number of these information systems was unsuccessful, running over-time and running over-budget. These problems led to the situation widely known as the "software crisis" [15]. The crisis showed that a methodological approach is needed in the IS software development. In 1968 and 1969 NATO sponsored conferences at which this problem was clearly defined and initial steps were determined [16], [17]. The term "software engineering" appeared, based on the idea that an engineering-like approach should be applied to software development. Although research in the field of software engineering was carried out in the 1960s and the 1970s, these had a moderate impact on practical software development. However, the most important concepts in the field were developed, such as top-down formation, step-by-step improvement, modularity and structured programming. These concepts grew into methods and represented a turn in the software development approach and the development of IS in general. At the basis of the mentioned software development methods (as well as the ones created at a later point) lies a group of key ideas related to formal methods. Formalism ensures a unique philosophy in the creation of IS development methods which has not changed over the last thirty or so years. The first information systems were characterised by a program code (software) which was full of faults, since it was created without the use of formal methods. Various testing techniques were suggested in order to locate and eliminate the faults. However, testing was not the best way to create quality programmes. It was realised that it is the job of the software engineer to develop several models or real system descriptions with appertaining evidence that the models on lower abstraction levels correctly implement higher abstraction level models. Only this design process can ensure high quality software, and not testing. Dijkstra, the famous advocate of formal methods made his famous statement that "testing shows the presence, not the absence of bugs" [Dijk 76]. In other words, only the application of formal methods can ensure the quality of software, not the testing thereof. Numerous programmers apply the top-

down development and structured methods without being aware of the underlying formal apparatus. Critics of formal methods, on the other hand, point out the problem of their applicability to large systems and the impracticability of formalising naturally complex characteristics of large systems [15]. Although the application of software engineering is unavoidable, it does not cover all the activities in information systems development [8].

# 2 Definition of formal methods

All papers Various definitions of formal methods exist, depending on how widely their application is observed. One of the wider definitions is as follows (Nancy Leveson):

*"A wide view on formal methods includes all applications of (primarily) discrete mathematics to software engineering problems. This application usually includes modelling and analysis, where models and analysis procedures are performed or defined on mathematically precise bases."*

Narrower definitions of formal methods usually come down to the use of formal languages. Such is the definition by Jeannette M. Wing:

***"****Formal method in software development is a method ensuring a formal language for the description of software knowledge (eg. specifications, models, source codes) in such a way as to enable evidence of software knowledge features expressed in a formal language."*

This definition shows two important components:

- Formal method implies the use of formal languages. Formal language is a character set from a well defined alphabet. The rules (productions) for the distinction of sets (words) defined over the alphabet, belonging to a language, from the sets which do not are given.
- Formal methods in software development support formal thinking which can be formally verified. The verification begins with a set of axioms which are supposed to be true. The rules of deduction say that, if a specific formula (premise) is deductible from the axiom, then the second formula (consequent) is deductible from the axiom as well. The set of rules of deduction must be specified for every formal method. The verification consists of a set of well-defined formulas from a language in which each formula or axiom has been deducted from the previous formula in the set.

Formal methods have their root in specific axiomatic trends in mathematics of the 19th and the 20th century. Through formal methods, these trends have been adopted in software engineering. Edsgar Dijkstra, the advocate of formal methods, stated that computer sciences should be renamed into "Very Large Scale Application of Logic". In order to master formal methods in software engineering, it is necessary to understand the mathematic background. This background includes formal logic (propositional calculus and predicate calculus), set theory, formal languages and final automation [4].

# 3 Application of formal methods

Taking a good look at the development stages of structured methods shows that these move from programming, over design and analysis to automated techniques trying to provide computer support to a complete life-cycle of IS development. The application of formal methods in programming and partly in design is clear, as no one doubts that formal methods can be directly applied to these two stages of the IS development life-cycle. Finally, formal methods have had a direct influence on the development and standardisation of a large number of programming languages which provide a basic tool for programmers [9]. However, in the analysis stage, and especially in IS strategic planning, where the organisation is analysed as a whole and we have moved away from the programming code and action diagrams, the application of formal methods becomes less clear. The organisation analysis cannot avoid its social and sociological features, which cannot be easily formalised and modelled. This helps the critics of formal methods who claim that the naturally complex features of organisation systems cannot be formalised. In other words, formal methods as methods used in natural and technical sciences are not applicable to organisation, as it is, among other things, a social category (non-automated organisations in which people work are considered). However, successful attempts of describing social phenomena using the methods of natural and technical sciences are not unheard of. One of the more distinct examples is *cybernetics*, the science of transforming information for the purpose of managing complex systems [7]. Cybernetics has brought together several mathematical fields (information theory, game theory, operation research, mathematical logic), while its principles can be equally applied to mutually very different science fields – biology, chemistry mathematics, medicine, linguistics, pedagogy, economics, law, organisation sciences etc. In addition, scientific circles increasingly speak of *scientism* [11]. Webster's Dictionary defines scientism as:

"*trust that the assumptions, methods and research of natural sciences are equally appropriate and relevant to all other disciplines, including humanistic and social sciences*".

Another field with an upward trend, important for the understanding of the need for formal methods is metamodeling. In short, metamodeling is the modelling of reality with a specific purpose. One of

the uses of metamodeling is for a unique description of data generated by certain methodologies. One of the preconditions of good IS development methodology is the existence of a unique methodology metamodel encompassing data retrieved from all methods applied in the methodology [3]. As the metamodel of the IS development methodology contains the description of the data generated during IS analysis, where social characteristics of the organisation are emphasised, it can be said that metamodeling is another attempt of applying formal methods in non-technical field.

In addition to the aforementioned three topics (cybernetics, scientism and metamodeling), which prove the application of methods of technical or natural sciences to other fields, it should be emphasised that CASE tool for support to the entire life-cycle of the IS development is being developed. The basis of the CASE tool is a metamodel (data dictionary) which represents the scheme of the data base in which all data generated during the IS development are kept. Efforts are made in the formalisation of the aforementioned life-cycle stages (strategic planning, analysis) in order to facilitate and, in a way, standardise access to IS development. Problems in the application of formal methods to the aforementioned life-cycle stages are not accidental and can be categorised in the following way:

- Strategic planning and requests (specifications) analysis are related to the top management of an organisation which deals with strategic business planning. The nature of the strategic business planning [10] is uncertainty and the influence of numerous informal elements. Taking into account the fact that a large part of organisations does not perform quality strategic planning, it is clear that the application of formal methods to activities this informal is extremely difficult (figure 1).

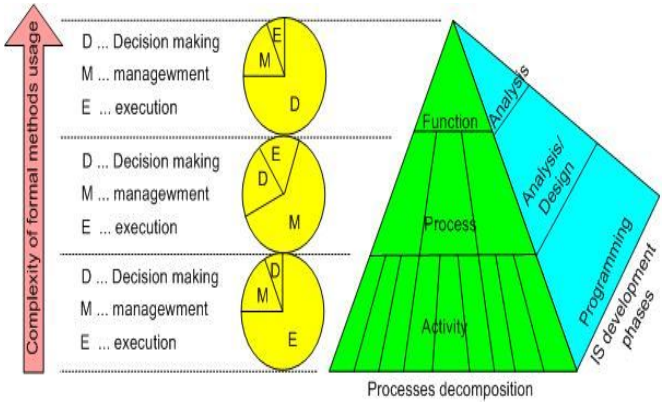


Figure 1. Complexity of applying formal methods in the IS development (authors)

- Observing the organisation through the decomposition of its processes (function, process, activity), it can be determined that the reasons making it difficult to introduce formal methods in the analysis of business functions, are the same reasons making it easy to introduce formal methods in programming activities (hierarchically the lowest processes which are highly structured).
- If we observe the type of organisation processes (decision-making, management and execution) [1], the execution level is predominated by operation processes which can easily be formally described. Moving over the management level to the decision-making level, more complex processes become predominant, with an increasing amount of uncertainty and various informal influences, making the application of formal methods more complex. On every level of decomposition of organisation processes (function, process, activity) all three types of processes are present, to various extents. Speaking of business functions, the decision-making process is represented in the highest amount, the execution process in the lowest amount. On the other hand, speaking of activities, the execution process is represented in the highest amount, the decision-making process in the lowest amount (figure 1).

Regardless of the difficulties in the application of formal methods, it should be mentioned that they are significant in software engineering and thus in the IS development. They are applicable, up to the certain degree, to all life-cycle stages of IS development: requests analysis, design and programming, and have significant influence on testing and maintenance of the IS. They also have significant influence on current research which could change the present practice in the IS development, particularly in the non-researched fields of requests analysis (specifications) and design. They are built into life-cycle models which may represent an alternative to the traditional "waterfall" model, eg. rapid prototyping, Cleanroom method or transformation paradigm.

# 4 Areas of usage of formal specification

Formal methods enable a precise and strict specification of those IS features which can be expressed in a certain specification language. Defining what the system needs to do and understanding the consequences of this definition represent the most difficult problems in software engineering [15], so the use of formal methods is highly advantageous. Generally speaking, the practitioners of formal methods often use them for the descriptions of precise specifications of the system and not for formal verification, which is a mentally more demanding process.

The functionality of the system (organisation) being described is the most common subject for the use of

formal methods, so well-known formal methods, adjusted to the IS development, contain a specification language for the description of functionality (Z, Vienna Development Method (VDM), Formal Development Methodology (FDM).
Although the most common, the functionality of the system is not the only subject of use of formal methods. IS security and protection are examples of other fields in which formal methods are occasionally applied. The profit of proving that IS will not transfer into an unwanted state, or that IS security will not be jeopardised may justify the price of the complete formal verification of specific IS elements (software), should the organisation the IS belongs to find IS security and protection important.
Formal methods may include graphic languages. Data Flow Diagrams (DFDs) represent the best-known graphic technique for the specification of system functions (figure 1). Although these diagrams may be considered semi-formal methods, various techniques for their treatment in a fully formal way have been examined.

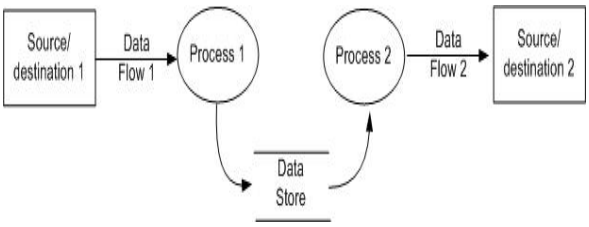


Figure 2. Concepts of Data Flow Diagram (authors)

Data Flow Diagrams are significant in the IS development for various reasons:

- enable IS designers a simple accrual of knowledge and a better understanding of the processes in an organisation,
- present a simple communication mechanism between users and designers of the IS,
- process decomposition, as a logical result of their application, suggests the structure of the future application,
- the lowest level of the Data Flow Diagram defines processes which are candidates for program modules,
- the process model obtained by the drawing up of Data Flow Diagrams can be used as basis for the application of other models, such as ABC (Activity Based Costing),
- Data Flow Diagrams identify all important documents in an organisation, later used to create data models,
- a number of CASE tools have been developed, supporting the Data Flow Diagram method.

In addition to Data Flow Diagrams, a number of formal methods with graphic languages exist, such as Petri nets and final automaton, two fully formal methods.
Software engineering practitioners create models and define characteristics of the organisation system on several levels of abstraction. The *specification (analysis)* level should describe what the organisation should do, but not how to do it. The *design* level provides more detail, and most details are entailed in the *source code*.
Summarizing the aforementioned, we reach the well-known taxonomy of using formal methods in software engineering:

1. **Specification –** With this form of use, the method is used to define a model which is then informally or formally translated into a system using other formal methods.
2. **Verification** – The use of formal methods to verify the correctness of the designed program solution.
3. **Implementation** – Formal methods can be used in the creation of a program product from predefined specifications.

# 5 Tools and Methodology

The development of technology for the formalisation of software solutions has been accompanied by the development of support tools. The basic idea is that the final product is not merely an acting system (application). Specifications and evidence that the application will meet the specification requirements are equally important. Evidence is difficult to develop after the application been finished. Therefore, evidence and programmes should be developed simultaneously, with strong mutual bonds during the development of the application. Since program correctness needs to be proved, only those program constructs which can be fully understood should be used. This was the basic motivation for the use of structured programming.
However, the first challenge was the application of these ideas to large-scale projects. The application of formal specification can be widened much more easily than the application of formal verification. Despite that, ideas related to formal verification are applicable to projects of any scale, particularly if the level of formalism may vary. David Gries suggests the application of analysis and design methods which entail a certain amount of heuristics, that is, he encourages the application of methods of researching new knowledge. The design results achieved this way will be more reliable and more easily proved. In accordance, Harlan Mills has developed the Cleanroom approach. It is the IS development life-cycle where formal methods, inspections and reliability modelling have been integrated into the

social process of software production, thus opening the way for formal methods into higher levels of the IS development life-cycle – analysis and design.
Formal methods have been an inspiration for the creation of numerous program tools. On the other hand, these tools have contributed to a wider use of formal methods. Some of the tools animate system specifications, thus converting formal specifications into an executive IS prototype. Other tools derive programmes from specifications through various automated transformations. According to some approaches, programmes are solutions to equations set in a formal language. Transformation implementation, as one of the modern approaches, suggests a future where numerous software systems would be developed without programmers or at least with a far larger proportion of automatism, high productivity and less human work.

The first examples of using formal methods in software engineering are related to the verification of program products. In the 1960s, E. Dijkstra developed a verification method of program correctness which converted the program code into algebraic equations, exchanging the semantics of the program language for algebraic model. Numerous formal methods and tools for code verification were developed later, based on algebraic equations or predicate calculus [13]. Today a variety of tools exist which automatically verify code correctness (Frama-C, PolySpace). The Dijkstra method, as well as the mentioned tools, primarily deals with syntax analysis and code verification. Semantic code verification is much more complex, making the methods and tools allowing serious semantic analysis much rarer. The best developed method of semantic code verification is Semantic and Description Language (SDL) and its implementation in a tool set called SITE. This method steps out of the field of code verification and covers, in addition to verification, semantic specification. It can thus be used as automated code generator from formal specification, as well as the verifier of its semantics. But formal methods dealing in verification are not restricted solely to program code verification and testing, but formal methods for verification exist as well.

The methods for IS specification are the most common and most developed. One of the best-known method for specification of a part of the IS is the relational model, the best prevailing formal method in computing in general. The relational model was defined by F.E.Codd in 1971. There are methods using for the specification of the dynamic (process) part of the IS, but these cannot be called formal, as they have no strictly defined declarative and operational semantics. Naturally, the methods used in CASE tools must be formalised, but it is only handy formalisation, not defined on the method level, but used only to allow the implementation of the method on a computer. Methods defined in the IDEF methodology [5] are closest to formalisation. The IDEF methodology strictly defines the semantics of each symbol used. There are, however, formal methods for system specification. One of the best known formal methods for program code specification is the Abstract State Machines method. One of the main contributors, particularly in the field of connecting Evolving algebra with the Warren Abstract Machine (WAM) and Prologue was Dean Rozenzweig, a scientist from Zagreb, Croatia. Today, the ASM is most commonly used as formal specification of program code. This method, however, is not as popular as the less formal UML, most commonly used today. Another method used for formal specification of program code is the Z notation [12]. It enables formal specification of data and processes in the system, making it suitable for complete IS specification. The problem with the Z notation is that it is oriented at generating the program code and not the system. Similar to the Z notation is the B method which operationalises the ASM taking a Z-like notation. The B method is implemented in the B-Toolkit of several tools, such as ABTOOLS, B-Core, B4free and Rodin.

All of the aforementioned methods, with the exception of the relational model, are oriented relatively low, on formal specification and verification of computer programmes. Although very useful, they do not satisfy the needs of IS designers. For this, methods defining data classes and processes in the IS are needed. Statistical (data) part of the IS is formalised through relational model, but the formalisation of the active (process) part of the IS has not been formalised to a satisfying extent. There are methods formalising this part, but they are not nearly as wide-spread or accepted as the relational model. A method called process algebra or process calculus ((π-calculus) should be mentioned here. Process algebra is a mathematical apparatus for defining processes in a system. It has been through several concretisations appropriate for designing the active part of the IS. C.A.R. Hoare developed a method called Communicating Sequential Processes (CSP) in 1978, enabling the modelling of systems with sequential processes [6]. R. Milner defined the Calculus of Communicating Systems (CCS) in 1980 and J.W. Klop the Algebra of Communicating Systems (ACS) in 1982. In 1990, the International Organization for Standardization standardised LOTOS (Language of Temporal Ordering Specification), which brings the π-calculus closer to practical users, providing simpler syntax, more appropriate for work on computer. Lately, as concurrent systems have gained a more significant position in the IS development, ambient calculus was developed (L. Cardelli i A.D. Gordon, 1998), π-calculus adapted to concurrent systems. Lately, repeated requests are heard for defining methods for IS design specification and verification

based on object orientation, including both the static and dynamic part of the IS. However, due to the complexity of object design formal methods for the specification and verification of the system design based on object orientation have not been defined.

Lately, the application of formal methods is spreading even to earlier IS development stages, all the way to the IS planning. Formal methods for the specification and verification of the static part of the IS are again developing much faster than the ones specifying the process part. Research in the field of system ontology is being conducted, planning relationships and connections between objects, thus formally specifying, as well as verifying early specifications of the IS. Several ontology development approaches have been developed, based on the description logic and OWL, as well as Frame Based Logic and Flora. Tools for the verification of the developed ontology have been developed, such as CELL, SHER and Pellet, enabling the verification of the developed ontology. UML, used for specification in later stages of the IS development, has been expanded to Object Constraint Language (OCL), a declarative language enabling the definition of ontologies within the UML specification. The development of the active part of the IS in earlier stages is far more modest, as is the case with the formal methods to accompany it. Currently no wide-spread formal method for the specification and verification of IS processes in the early stages exists, although some methods are starting to develop. But these are not nearly as wide-spread or formalised.

# 6 Example of formal method: Edsger Dijkstra and the development of correct programmes

Edsger Dijkstra, one of the pioneers and advocates of formal methods, developed a special approach to writing correct programmes [2]. His formal apparatus, with the use of the *diktran* formal language, enables writing correct programmes or programmes which will surely correctly finish when started and once they finish, will determine the truthfulness of the conclusion they were designed for. With the use of the predicate calculus and a number of logical transformations, the mechanism (programme) code is created. Programmes created in this way need not be tested – they are designed to be correct. Dijkstra is one of the pioneers of software engineering (an active participant in the aforementioned NATO conferences). His statement that „testing shows the presence, not the absence of bugs" is well-known.
The most significant concept used in the writing of correct programmes is the *weakest precondition for mechanism (programme) S to determine the truthfulness of the conclusion R.* This concept is a logical predicate (formula), marked with *wp (S, R)* and represents a set of all the states from which mechanism S, if started, will correctly finish and determine the truthfulness of the conclusion R.

*Example 1.* Let mechanism S denote a value assignment statement a:= b+5 and let conclusion R a>b be set. The states that mechanism S can be found in are defined by the values of variables a and b. From which states will mechanism S, if started, determine the truthfulness of the conclusion R? The answer to the question shall be gained in the following way:

$$wp(S,R)=wp("a:=b+5", a>b)=\left\{a>b\right\}^{a}_{b+5}=\left\{b+5>b\right\}=\left\{5>0\right\}=T$$

The result is the logical constant T (true), meaning that the statement a:= b+5 will determine the truthfulness of the conclusion a>b regardless of which state it is started from. In other words, regardless of the current values of variables a and b, the assignment statement will determine, after being executed, that a>b. In cases as simple as this one, where the observed mechanism (program) is a single value assignment statement, and the conclusion a simple judgement, the weakest precondition calculus is easily executed. A much more complex example is the case where it is necessary to define mechanism (program) S which will, once started, determine the truthfulness of the conclusion R.

*Example 2.* For the set N1 (N1>0) and N2 (N2>0), a mechanism (program) S needs to be written, which will, once carried out, determine the truthfulness of the conclusion R:

$$R: \quad product = \prod_{l=1}^{N2} \sum_{i=N1}^{N1+l} i^2 \qquad (1)$$

In other words, a program needs to be written which will calculate the stated sum product for the set values N1 and N2. Intuitively, most programmers would think the following: the program needs to read two values N1 and N2 and contain two loops (iteration mechanisms) – internal for the sum and external for the product. However, the formal method suggested by Dijkstra begins with the conclusion the program needs to determine (1). From the conclusion the so called invariant is defined, or the predicate which must be true during the entire implementation of the program. This is followed by a number of transformations from which program instructions result. The weakest precondition is calculated for statements to determine the truthfulness of the invariant. What is most significant for the application of this method is the fact that the final program significantly depends on the way the conclusion was defined. Without going further into formalism, we will examine two ways of defining conclusions (a) and the resulting two different programmes.

The conclusion (1) can be written in the following way:

$$R: \quad product = \prod_{l=1}^{N2} f(i) \quad and \quad f(i) = \sum_{i=N1}^{N1+l} i^2 \qquad (2)$$

If the sum index i=N1 is exchanged with the statement i=N1+0 in the second part of statement (2), and the constant 0 is exchanged with the variable k, the conclusion R will have the following form:

$$R: \quad product = \prod_{l=1}^{N2} f(i) \quad and \quad f(i) = \sum_{i=N1+k}^{N1+l} i^2 \quad and \quad k=0 \qquad (3)$$

Starting from this conclusion, Dijkstra's formal method, with certain optimisation, will lead to the following mechanism (program):

```
if N1>0 and N2>0 →
        j, product:= 0,1;
        do j<>N2 →      k, f := j+2, 0;
        do k<>0 → k := k-1; fakt := N1+k;
                  f := f+fakt*fakt od;
        product := product*f;
        j := j+1
        od
fi
```

Once the resulting program finishes, it will determine the truthfulness of conclusion (3) and the truthfulness of conclusion (1) accordingly.
Let us observe the conclusion (1) written in the following form:

$$R: \quad product = \prod_{l=1}^{k} \sum_{i=N1}^{N1+l} i^2 \quad and \quad 0 \le k \le N2 \qquad (4)$$

Applying Dijkstra's formal apparatus to conclusion (4), we come to the following program:

```
if N1>0 and N2>0 →
        k, product, sum := 0, 1, N1*N1;
        do k<>N2 →      k := k+1;
                sum:= sum + (N1+k)*(N1+k);
                product := product*sum;
        od
fi
```

The gained program only has one *do-loop* (iteration), unlike the previous one, it is faster and simpler.
The example shows that the final look and simplicity of the program resulting from this formal method depends on the manner the conclusion that the program needs to confirm has been defined. The conclusion represents the essence of the future program and program instructions are gained from it through specific transformations. Programmes created in this way need not be tested – they are conceptually designed in such a way not to have errors.
The main argument against the application of such formal methods is that applications today are developed with the help from the application generator. However, there will always be a need for writing specific program codes which can be automatically generated.

# 7 Conclusion

This paper provides a short overview of formal methods, their basics and applicability to certain stages of the IS life-cycle development. Although primarily used in lower stages of IS development (code formation and partly design), it can be stated that the idea of their application in the entire life-cycle is nothing new. Generally, formal methods enable:

- More precise system specifications,
- Better internal designer – user communication,
- Possibility of verifying the design prior to code execution,
- Higher IS quality and productivity.

These advantages do not come without costs related to the training and use of formal methods. There are no strict and quickly applicable rules on the correct choice and amount of formalism in the IS development projects, or defined manners of the introduction of formal methods into specific organisations. Their application is more a question of enthusiasm of those employees in organisations aware of the possibilities of formal methods in solving specific problems. The fact that the integration of formal methods into the life-cycle of the IS development is being seriously considered is proved, among other, by the Cleanroom methodology, developed by Harlan Mills. This approach combines formal methods and structured programming with statistical process control, spiral life-cycle, inspection and modelling software reliability.

In order to be used to their full extent, formal methods need to be incorporated into standard procedures of organisations dealing with software production. Software development is also a social process, so the applied techniques must support this process. How to fully incorporate formal methods into the life-cycle of IS development is not entirely clear. Perhaps no universal answer exists, only different solutions by individual organisations.

The most evident obstacle of a stronger break of formal methods in the practice of software engineering is the gap between theoreticians developing formal methods and practitioners who are supposed to use them. Theoreticians develop mathematical models which are complicated and difficult to understand for practitioners. More often than not, a language adapted to work on computer is

defined on the basis of a developed formal method, but even with the developed language the formal method remains unintelligible and too difficult to be used in practice. A clear example of this statement is the development of the ASM formal method, which formalises the specification of the program code and the UML language, which does the same in a practical way. Although much more concise than the UMPL in its concepts, the ASM is not used in practice as it holds heavy mathematical notation and is not understandable enough for people who are not mathematicians.

This is also one of the directions that further development of formal methods should take in software engineering. Another direction, repeatedly mentioned in the paper, is the strengthening of the application of formal methods applicable to the specification and verification of the active part of the IS, particularly in the earlier stages of the development, as well as the development of the object oriented formal methods which would combine the specification and verification of the active and passive part of the IS, thus providing a better insight into the complete IS specification, as well as verification of the IS features including aspects of its operational, as well as data part.

# References


[1] Brumec, J. Contribution to General Taxonomy of Information Systems. In *Proceedings of 7th International Symposium on Information Systems*, Varaždin, Croatia, 1996.

[2] Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, 1976.

[3] Dobrovic, Z. *Information System's Design and Development Methods for Object Systems with Time Changeable Functions*. Doctoral Dissertation, Varaždin: Faculty of Organization and Informatics, Republic of Croatia, 1998.

[4] Gries, D. *The Science of Programming*. Springer-Verlag New York Inc., 1983.

[5] Hanrahan, R.P. *The IDEF Process Modeling Methodology*. Software Technology Support Center, 2002.

[6] Hoare C.A.R. Communicating Sequential Processes. *Communications of the ACM*, 21(8), pp 666-677, 1991.

[7] Kalužnin, L.A. *What is Mathematical Logic*. Školska knjiga, Zagreb, 1971.

[8] Kopella, M.; Mursu, A.; Soriyan, H.A. Information Systems Development as an Activity. *Computer Supported Cooperative Work,* 2002.

[9] Manna, Z. *Mathematical Theory of Computation*. McGraw Hill, 1974.

[10] Martin, J.Leben, J. *Strategic Information Planning Methodologies.* Prentice-Hall, 1989.

[11] Menton, D.N. *Carl Sagan: Prophet of Scientism.* Missouri Association for Creation, 1997.

[12] Spivey J.M. *The Z Notation: A Reference Manual,* University of Oxford, Programming Research Group

[13] Tucker; Noonan, R. *Programming Languages: Principles and Paradigms*, 2nd ed., McGraw-Hill, 2002

[14] Vienneau, R.L. *A Review of Formal Methods.* Department of Defense, Data & Analysis Center for Software, 1996.

[15] U.S. DoD (Department of Defense), IAC (Information Analysis Center). *A State of the Art Report: Software Design Methods.* DoD DACS (Data & Analysis Center for Software), 1997.

[16] NATO Science Committee. *Software Engineering.* Report on a Conference, Garmisch, Germany, October 1968.

[17] NATO Science Committee. *Software Engineering Techniques*. Report on a Conference, Rome, Italy, October 1969.