DOI: 10.20378/irb-57799

Interpretible Discrete Formalism in the Context of Modeling Information Systems

Walter Augsburger, Goran Mekinić

Abstract

Applying discrete formalism to computer systems development provides mathematically based techniques to describe system properties. They present an interesting framework for a systematic specification, development and verification of information systems. In this article we discuss the use of formal specifications in the context of modeling Business Information Systems and evolving theories in Business Informatics .

Keywords: Business Informatics, Information Systems, formal methods, syntax, semantics and pragmatics.

1 Theories in Business Informatics

Business Informatics is defined by the German Scientific Comission [21] a *real*, *formal* and *engineering* science. For that reason it aims to generate theories, methods and tools that have to be validated in practice. Here we identify three areas of interest: theory, tools, and practical applications (see Fig. 1).

This distinction is of interest because all of these areas are mutually depended and they place limitations on each other. Figure 1 illustrates that the development of tools, reference models and patterns presumes a theoretical foundation. Similarly practical applications presume tools and the development of new theories presumes an understanding of practical applications of theory. Each of these areas is interrelated to other scientific, technical or economic fields (Fig. 1).

Theories in Business Informatics can be described in two parts. On the one hand we have a set of consistent and powerful concepts, languages and descriptive techniques which are useful for precise and unambiguous descriptions of real systems and theoretical models in a compact and comprehensive manner. On the other hand we have a set of different analysis methods. The aim of these methods is to establish and prove ways to analyse properties of descriptions and models.

Tools, reference models and patterns as the end products of theories in Business Informatics build a bridge to practice. Tools have two roles in this framework. Firstly case tools, simulation and verification tools are necessary to support the application of analysis and descriptive methods developed in research. Secondly standard software tools developed from theoretical models enable the application and validation of theories in practice. The theoretical analysis of real information systems and requirements analysis deliver reference models and patterns. These are required in order to use the tools and develop IT-systems in practice. Additionaly reference models play an important role in customizing standard software systems for the special needs of the enterprises and whole branches [16].

The practical area of Business Informatics covers conception, development, service and use of systems which apply the computer supported information processing within enterprises [15]. These systems are often referred to as *Information and Communication Systems (ICS)* meaning sociotechnical systems containing automated and non-automated components [10], [14], [21], [17].

The concerns mentioned above imply that we need methods to provide frameworks within which people can specify, develop and verify systems and models in a systematic manner. We need

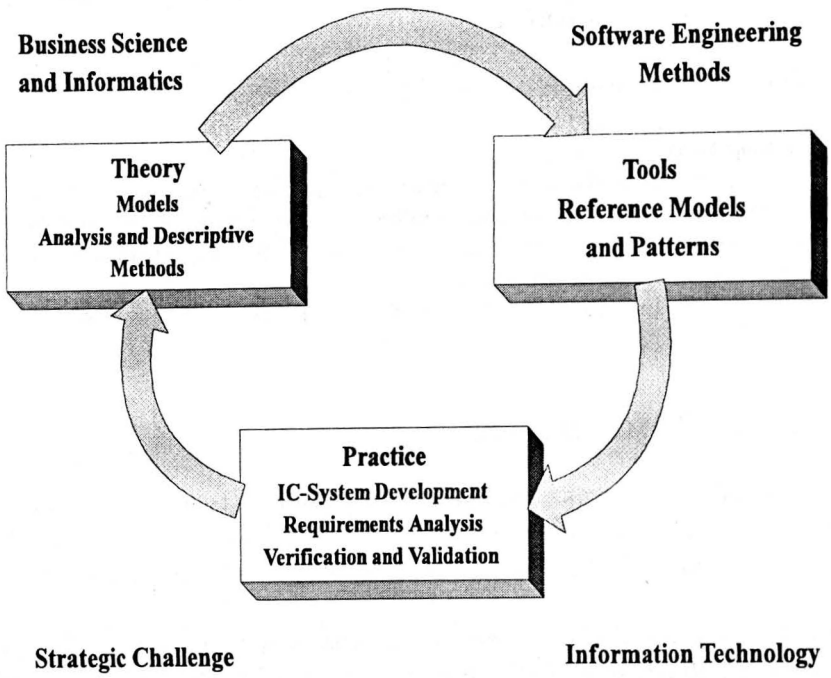

Figure 1: Theory and Practice in Business Informatics

descriptive techniques and concepts which reveal ambiguity, incompleteness and inconsistency in a system. On one side these techniques have to support the communication among customers, developers, programmers and clients and on the other side enable the use of analysis methods. Every engineering discipline, as Business Informatics partly is, develops therefore a body of mathematical techniques that is particularly appropriate for modeling, analyzing and predicting the phenomena relevant to its field. In many cases, the relevant applied mathematics uses partial differential equations to model the variations in continous physical quantities over time or space. For software, however, the familiar methods of calculus and differential equations are inapplicable because for modeling business information systems discrete, rather than continuous quantities are concerned. Instead of differential equations the properties and behavior of the systems considered are best described in terms of concepts from discrete mathematics: "sets", "graphs", "finite-state machines", etc. and computer-readable languages. Furthermore the "Calculation" in these finite domains is based on the methods of formal (or mathematical) logic rather than numerical computation.

2 Discrete formal methods

2.1 What is a formal specification language?

A formal specification language provides the mathematical basis for a formal method. There are standard definitions of formal languages and their properties that following from [5].

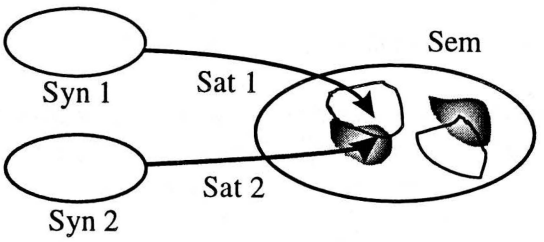

Figure 2: A formal specification language

Definition 2.1 *A formal specification language is a triple* $< Syn, Sem, Sat >$*, where* Syn *and* Sem *are sets and* $Sat \subseteq Syn \times Sem$ *is a relation between them.* Syn *is called the language's syntactic domain;* Sem*, its semantic domain; and* Sat*, its satisfies relation or meaning.*
In the picture below a syntactic universe Syn is linked to a semantic universe Sem by an arrow representing a satisfies relation Sat or meaning. As the figure 2 shows two formal languages can differ in their syntactic domain, in their semantic domain and in their meaning. Further, they are often incomplete in a logical sense. Both $Sat1$ and $Sat2$ specify only parts of the semantic domain Sem. In practice, we are usually dealing with incomplete specifications.

2.2 Syntactic domains

A specification language's snytactic domain is a set of symbols (for example constants, variables, logical connectives, function and predicate symbols) and a set of grammatical rules for combining these symbols into well–formed formulas. A syntactic domain need not to be restricted to textual elements. It can be given to graphical elements such as boxes, circles, lines, arrows, diagrams and icons a formal semantics just as precisely as textual ones. The examples for visual formal languages are Petri Nets (Exp. 2.3) and Harel's statecharts based on higraphs [7], [8]. A grammatical rule for such visual specifications might be that all arrows start and stop at boxes.

2.3 Semantic domains

Specification languages differ for the most part in the choice of their semantic domain. The mostly suggested classifications of formal specification languages are based on classification by semantic domain. These classifications are:

- Non-algebraic vs. Algebraic specification languages (Classification by entity types)
- Sequential vs. concurrent and distributed systems specification languages
- Programming languages vs. non-executable formal specification languages

Non-algebraic specification languages are based on set theory and logic. They only use mathematical entities: sets, functions and relations to model systems. In those languages data types are modeled using set-theoretic constructions, just as natural numbers, real numbers, ordered pairs or sequences are defined in set-theoretic terms in mathematics. These methods

like VDM, Z, statecharts, Petri Nets have been found useful for specifying large commercial systems.

Example 2.1 This example [20] illustrates a specification of a display-oriented text editor in Z [19]. Z is a formal method based on set-theory which can be used as analysis or descriptive method. A document is described as an ordered pair consisting of the text before and after the cursor:

$$DOC = seq[CH] \times seq[CH]$$

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

$$\begin{array}{l} back: \quad DOC \longrightarrow DOC \\ ins: \quad CH \times DOC \longrightarrow DOC \\ \hline domback = \{l, r \mid l \neq <>\} \\ (\forall\, (l, r) : DOC; ch : CH) \\ back(l * < ch >, r) = (l, < ch > * r); \\ ins(ch, (l, r)) = (l * < ch >, r); \end{array}$$

(In this specification, $<>$ is the empty sequence; $< ch >$ is the sequence containing the single character ch; and $*$ is the append function on sequences.) The function *back* (move one character backwards) is partial - it is applicable only to documents having some text before the cursor. This restriction on its domain is given by the first axiom $domback = \{l, r \mid l \neq <>\}$.
Pre- and post-conditions may be used to specify procedures with side effects in a way similar to that used in VDM.

Algebraic specification languages are based on the idea that for specification purposes an application system can be modeled as a many-sorted algebra. Data types are represented as sorts i.e. a sets of data values (one set of values for each data type). Functions (operations) in a program correspond to the specific total or partial functions on sorts. This abstracts away from the algorithms used to compute the functions and how those algorithms are expressed in a given programming language. Algebraic specification languages are focusing instead on the representation of data and the input/output behaviour of functions. Object classes might be defined in terms of relationships between the operations defined on those classes.

Example 2.2 As example we have choosen an algebraic specification of an abstract data type *Stack* [17]. We use a simplified version of the algebraic specification language SPECTRUM [2].
Signature:

$STACK$ = {
sorts (types): $Stack, ElementType, AnyType$;
operations: $NewStack : \emptyset \longrightarrow Stack$;
$Push : Stack \times ElementType \longrightarrow Stack$;
$Pop : Stack \longrightarrow Stack$;
$Top : Stack \longrightarrow ElementType$;
$IsNew : Stack \longrightarrow BOOLEAN$; }

Axioms:

$Push, Pop, IsNew$ *total*;
$\forall\, s \in Stack\ \forall\, e \in ElementType :$
$Pop(NewStack) = NewStack$;

$Pop(Push(s,e)) = s;$
$Top(Push(s,e)) = e;$
$IsNew(NewStack) = TRUE;$
$IsNew(Push(s,e)) = FALSE;$
endaxioms

Notice how the constant $NewStack$ is viewed as a nullary function. The line

$$Push, Pop, IsNew\ total;$$

is called a *totality axiom* and it demands totality of functions: $Push, Pop$ and $IsNew$ (Top is a partial function).
Algebraic specification languages provide the most abstract approaches to modeling and analyzing of systems. The specifications written in these languages may range over different semantic domains. Many approaches based on algebraic methods and formal language theory have been developed for complex discrete event dynamic systems.

Concurrent and distributed systems specification languages are used to specify state, event and transition sequences, streams, synchronization trees, partial orders and state machines. Concurrent and distributed systems can be specified in various styles. Some styles take several forms of communication as primitive and has programming-like features for sending and receiving values. These styles are often referred to as *process algebra* [1]. Another styles take shared variables as the primitive means of communication. They often use *temporal logic* to allow specification that a property should hold "henceforth" or "eventualy" on some or all execution paths. These styles have a property-oriented flavor.
One of the first and the most interesting formalisms to deal with concurrency, nondeterminism and causal connections between events are Petri Nets. According to [13] it was the first unified theory, with levels of abstraction, in which to describe and analyze all aspects of computer in the context of its enviroment. In contrast to most specification languages Petri nets are state and action oriented at the same time. Hence they provide an explicit description of both states and actions. This means that the specifier can choose whether he wants to concentrate on states or actions. Today we have a lot of different net models but most of the descriptions in one of the net models could be informaly translated to the other net models, and vice versa. The classic Petri Net model is a 5-tuple (P, T, I, O, M). P is a finite set of places (often drawn as circles or ellipses) representing states (or conditions). T is a finite set of transitions (often drawn as bars or rectangles) representing events. I and O are sets of input and output functions mapping transitions to bags of places (the incidence functions). M is a set of initial markings. Places may contain zero or more tokens. A marking of the Petri Nets is the distribution of tokens at a moment in time.
Example 2.3 The simple example of figure 3 illustrates an application of Coloured Petri Nets [11]. This example models a small distributed data base system. Similar models can be obtained for any other kind of loosly coupled distributed application systems in office automation (e. g. workflow systems or groupware).
The data base system in the example shown in the figure 3. has n different sites. Each site contains a copy of the entire data base and this copy is handeled by a local data base manager. When one manager updates the local copy of the data base on its site, it must send a message to the other managers - to ensure consistency between the local copies of the data base. Each

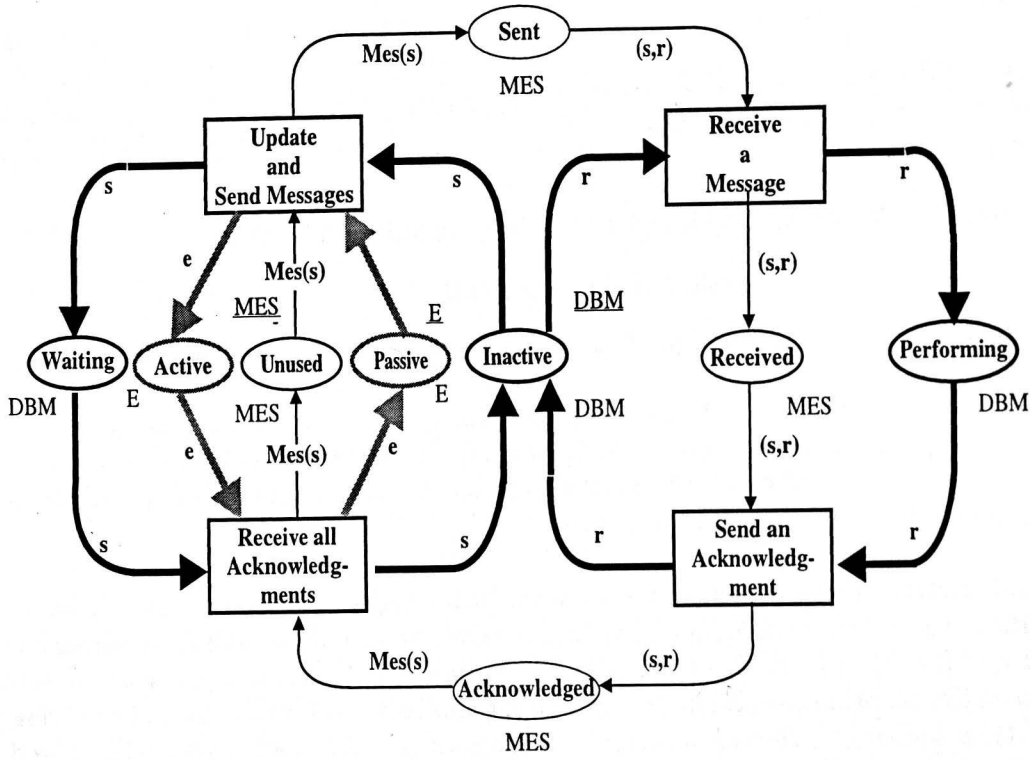

Figure 3: A small distributed data base system [11]

message is represented as a pair (s, r) where s identifies its sender and r the receiver of the message. There are nine different places (drawn as ellipses) in the net model. Each place has an associated *data type* determining the kind of tokens which the place may contain. There are three different data types for tokens: DBM (identifiers of data base managers), MES (messages) and E (activ/passiv). The three places: `Inactive`, `Waiting`, and `Performing` represent the three possible states of the data base managers. The tokens for those places are of the data type DBM. The four places: `Unused`, `Sent`, `Received` and `Acknowledged` represent the possible states of the messages among data base managers and the corresponding data type for their tokens is $MES \subset DBM \times DBM$. The two remaining places: `Active` and `Passive` may have a token of the type $E = \{e\}$ indicating whether an update is carried out or not.
The initial marking is specified by means of initialisation expressions which are written (see Fig. 3) by convention with an underline next to the place. The actions of Coloured Petri Nets are represented by means of transitions (which are drawn as rectangles). In the data base system there are four different transitions: `Update and Send Messeges`, `receive all Acknowledgements`, `Receive a Message`, and `Send an Acknowledgement`. $Mes(s)$ is a set of all messages which were sent by data base manager s.
The names of places, tokens and transitions *may not have any meaning.* They have just practical importance for the readability of a Petri Nets like the mnemonic names in the traditional programming. That is one of the key properties of formal methods.

Programming languages are used to specify input-output-functions, algorithms, predicate transformers, relations and machine instructions. Each programming language (with a well–

defined formal semantics) is a specification language, but the reverse is not true. In general specifications do not have to be executable on some machine whereas programs do.
The more abstract non-executable specification languages have also the advantage of not being restricted to express only computable functions [2], [6], [9], [22].

Some Examples of Formal Specification Methods	
Language	**Description**
VDM	non-algebraic, sequential
Z	non-algebraic, sequential
Larch	algebraic, sequential
Finite-State Maschines	non-algebraic, sequential
Petri Nets	non-algebraic, concurrent
Temporal Logic	non-algebraic, concurrent
CSP	non-algebraic, concurrent
CCS	algebraic concurrent
PAISLey	non-algebraic, concurrent and executable
SPECTRUM	algebraic, sequential
TROLL	non-algebraic, sequential

2.3.1 Properties of specifications

Every useful formal specification language must fulfil two requirements: unambiguity of any well-formed specification and consistency.
Informally, a specification is unambiguous if and only if it has exactly one meaning.
Definition 2.2 *Given a specification language* $< Syn, Sem, Sat >$ *a specification* syn *in* Syn *is unambiguous if and only if* Sat *is a (total or partial) function.*
Any specification language based on or incorporating a natural languages are inherently ambigous. Another desirable property of specifications is consistency:
Definition 2.3 *Given a specification languages* $< Syn, Sem, Sat >$ *a specification* syn *in* Syn *is consistent (or satisfiable) if and only if* Sat *is a function and maps* $syn \in Syn$ *to a non-empty subset* Mod *of* Sem. *We call* Mod *the underlying semantic model of* syn
The consistency of a specification implies that is impossible to derive anything logically contradictory from this specification. In terms of programming languages, consistency of a specification means there is some implementation that will satisfy the specification.

2.4 Pragmatics

Two salient pragmatic concerns about formal methods are their users and their uses.
Some users of formal methods produce formal specifications, we call them *specifiers* who may be scientists or developers. The other users who read only the specifications we call here *readers.* Specification readers are in practice for example customers, programmers, clients (people who use the specified application system) and verifiers. One type of specification can be more suitable to one type of specification user than to others. Every specification language has to lead to better communication among users, so it must be suitable for at least two types of users.
Formal methods can be applied, besides in scientific research, in all phases of system development: project planning, requirements analysis, system design, implementation, system verifcation and system documentation. It makes a great difference in Business Informatics whether

formal methods are used primarily for descriptive or for analytic purposes, which level of formality is employed or in which stage of the software development lifecycle formal methods are applied. The different uses have to be associated with different kinds of specification languages. It is important to recognize that different specification languages are often intended for very different purposes. Consequently they cannot be compared directly to one another. A common pitfall and a major source of misunderstanding is the failure to appreciate this point.

3 Conclusion

Formal methods provide frameworks within which people can specify, develop, research and verify systems in a systematic manner. Their further advantages are revealing ambiguity, incompleteness and inconsistency in a system, that otherwise might be discovered only during costly and late testing and debugging phases. When used after the implementation phase, they can help determine the correctness of a system implementation and the equivalence of different implementations. The greatest benefit of applying a formal method in theory and practice is often derived from the process of formalizing rather than from the end result: the specification document alone [18]. Formal methods provide a unified and concise view of the syntactic and semantic aspects of specifications, enable insights into and understanding of practice and theory in Business Informatics [18], [3], [4].

References

[1] P. Aczel, Final Universes of Processes, Tech. Report, Department of Mathematics and Computer Science, University Manchester, 1994.

[2] M. Broy, C. Facchi et al., The Requirement and Design Specifikation Language SPECTRUM, TUM–19311/2, Part I and II, Technische Universität München, München Mai 1994.

[3] D. Fensel, "Uber den Sinn formaler Spezifikationssprachen, Forschungsbericht 293, Institut für Angewandte Informatik und formale Beschreibungssprachen TH Karlsruhe, Karlsruhe, Februar 1994.

[4] Martin D. Fraser, Kuldeep Kumar, Vijay K. Vaishnavi, Strategies for Incorporating Formal Specifications in Software Development, in: Communications of the ACM, Vol. 37, No. 10, October 1994.

[5] J. V. Guttag, J. J. Horning and J. M. Wing, "Some Remarks on Putting Formal Specifications to Productive Use" in: Science of Computer Programming, North-Holland, Vol. 2, No. 1, Oct. 1982, 53-68.

[6] J. V. Guttag, J. J. Horning and J. M. Wing, "Larch in FiveEasy Pieces", Tech. Report 5, DEC Systems Research Center, July 1985.

[7] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and M. Trachtenbrot. Statemate: a working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering, 16:403-414, 1990.

[8] D. Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8:231-274, 1987.

[9] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, J. Kusch, Revised Version of the Modeling Language TROLL, Universität Braunschweig, Braunschweig 1994.

[10] L. J. Heinrich, F. Roithmayr, Wirtschaftsinformatik-Lexikon, 4. Auflage, R. Oldenbourg-Verlag, München–Wien 1992.

[11] K. Jensen, An Introduction to the Theoretical Aspects of Coloured Petri Netss in: J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): A Decade of Concurrency, Lecture Notes in Computer Science vol. 803, Springer-Verlag 1994, 230-272.

[12] C. B. Jones. Systematic Software Development using VDM. Hemel Hempsted. Prentice Hall, 1990

[13] R. Milner. Some directions in concurrency theory (panel statement). In Proceedings of the International Conference on Fifth Generation Computer Systems. ICOT, 1988.

[14] P. Mertens, Informatik-Magazin 4/1994, Springer-Verlag, Berlin, Heidelberg 1994.

[15] P. Mertens, F. Bodendorf, W. König, A. Picot, M. Schumann, Grundzüge der Wirtschaftsinformatik, Springer-Verlag, Berlin, Heidelberg 1991.

[16] A. W. Scheer, Wirtschaftsinformatik, Referenzmodelle für industrielle Geschäftsprozesse, Springer-Verlag, Berlin u.a. 1994, 4. erw. Aufl.

[17] E. J. Sinz, O. K. Ferstl, Grundlagen der Wirtschaftsinformatik, Bd. 1, Oldenbourg Verlag, München–Wien, 1993.

[18] I. Sommerville, Software engineering, Addison Wesley, Wokingham England u. a., 1992.

[19] J. M. Spivey, Introducing Z: A Specification Language and its Forml Semantics, Cambridge Univ. Press, 1988.

[20] B. Sufrin, Formal specifications of a display-oriented text editor. Scinece of Computer Programming 1, 1982, 157-202

[21] Wissenschaftlichen Kommission Wirtschaftsinformatik, Beschluss der Wissenschaftlichen Kommission Wirtschaftsinformatik im Verband der Hochschullehrer für Betriebswirtschaft e.V. vom 6.10.93, in: Wirtschaftsinformatik 36/1, Februar 1994, Mitteilungen der Wissenschaftlichen Kommission Wirtschaftsinformatik, Springer Verlag, Berlin, Heidelberg 1994.

[22] P. Zave, An Insider's Evaluation of PAISLey. In: LEEE Transactions on Software Engineering, Vol. 17, No. 3, March 1991.