Signs and Formal Concepts*

Uta Priss

School of Computing, Napier University, u.priss@napier.ac.uk

1 Introduction

In this paper we propose a semiotic conceptual framework which is compatible with Peirce's definition of signs and uses formal concept analysis for its conceptual structures. The goal of our research is to improve the use of formal languages such as ontology languages and programming languages. Even though there exist a myriad of theories, models and implementations of formal languages, in practice it is often not clear which strategies to use. AI ontology language research is in danger of repeating mistakes that have already been studied in other disciplines (such as linguistics and library science) years ago.

Just to give an example of existing inefficiencies: Prechelt (2000) compares the implementations of the same program in different programming languages. In an experiment he asked programmers of different languages to write a program for a certain problem. All programmers of so-called scripting languages (Perl, Python, Tcl) used associative arrays as the main data structure for their solution, which resulted in very efficient code. C++ and Java programmers did not use associative arrays but instead manually designed suitable data structures, which in many cases were not very efficient. In scripting languages associative arrays are very commonly used and every student of such languages is usually taught how to use them. In Java and C++, associative arrays are available via the class hierarchy, but not many programmers know about them. Therefore scripting languages performed better in the experiment simply because programmers of Java and C++ were not able to find available, efficient data structures within the large class libraries of these languages. Of course, this does not imply that scripting languages always perform better, but in some cases apparently large class libraries can be a hindrance.

These kinds of problems indicate that the challenges of computing nowadays lie frequently in the area of information management. A semiotic-conceptual framework as proposed in this paper views formal languages within a system of information management tasks. More specifically, it identifies management tasks relating to names (namespaces), contexts and (object) identifiers as the three contributing factors. These three management tasks correspond to the three components of a sign: representation, context and denotation.

It has been shown in the area of software engineering that formal concept analysis can be used for such information management tasks (Snelting, 1995). Snelting uses formal concept analysis for managing the dependencies of variables within legacy code.

* This is a preprint of a paper published in Eklund (ed.), Concept Lattices: Second International Conference on Formal Concept Analysis, Springer Verlag, LNCS 2961, 2004, p. 28-38. ©Springer Verlag.

But we argue that it is not obvious how to connect the three different areas of management tasks to each other if considering only conceptual structures, because sign use involves semiotic aspects in addition to conceptual structures. The semiotic conceptual framework described in this paper facilitates a formal description of semiotic aspects of formal languages. It predicts the roles which conceptual and semiotic aspects play in formal languages. This is illustrated in a few examples in section 8 of this paper. It should be pointed out, however, that this research is still in its beginnings. We have not yet explored the full potential of applications of this semiotic conceptual framework.

2 The difference between signs and mathematical entities

A semiotic conceptual framework contrasts signs with mathematical entities. The variables in formal logic and mathematics are mathematical entities because they are fully described by rules, axioms and grammars. Programmers might think of mathematical entities as "strings", which have no other meaning apart from their functioning as place holders. Variables in declarative programming languages are richer entities than strings because they have a name, a data type and a value (or state) which depends on the time and context of the program when it is executed. These variables are modelled as signs in a semiotic conceptual framework.

One difference between mathematical entities and signs is the relevance of context. Mathematics employs global contexts. From a mathematical view, formal contexts in formal concept analysis are just mathematical entities. The socio-pragmatic context of an application of formal concept analysis involves signs but extends far beyond formal structures. On the other hand, computer programs are completely formal but their contexts always have a real-time spatial-temporal component, including the version of the underlying operating system and the programmer's intentions. Computer programs cannot exist without user judgements, whereas mathematical entities are fully defined independently of a specific user.

Many areas of computing require an explicit handling of contextual aspects of signs. For example, contextual aspects of databases include transaction logs, recovery routines, performance tuning, and user support. Programmers often classify these as "error" or "exception" handling procedures because they appear to distract from the elegant, logical and deterministic aspects of computer programs. But if one considers elements of computers as "signs", which exist in real world contexts, then maybe contextual aspects can be considered the norm whereas the existence of logical, deterministic, algorithmic aspects is a minor (although very convenient) factor.

3 A semiotic conceptual definition of signs

Peirce (1897) defines a sign as follows: "*A sign, or representamen, is something which stands to somebody for something in some respect or capacity. It addresses somebody, that is, creates in the mind of that person an equivalent sign, or perhaps a more developed sign. That sign which it creates I call the interpretant of the first sign. The sign stands for something, its object.*" Our semiotic conceptual framework is based on a formalisation of this definition, which is described below. To avoid confusion with the

modern meaning of “object” in programming languages, “denotation” is used instead of “object”.

A representamen is a physical form for communication purposes. Representamens of formal languages, for example variable names, are considered mathematical entities in this paper. All allowable operations among representamens are fully described by the representamen rules of a formal language. Two representamens are equal if their equality can be mathematically concluded from the representamen rules. For example, representamen rules could state that a string “4+1” is different from a string “5”, whereas for numbers: $4 + 1 = 5$.

In a semiotic conceptual framework, Peirce’s sign definition is formalised as follows: A sign is a triadic relation $(rmn(s), den(s), ipt(s))$ consisting of a representamen $rmn(s)$, a denotation $den(s)$ and an interpretant $ipt(s)$ where $den(s)$ and $ipt(s)$ are signs themselves (cf. figure 1). It is sometimes difficult to distinguish between a sign and its representamen, but $rmn(s)$ is used for the mathematical entity that refers to the sign and s for the sign itself.

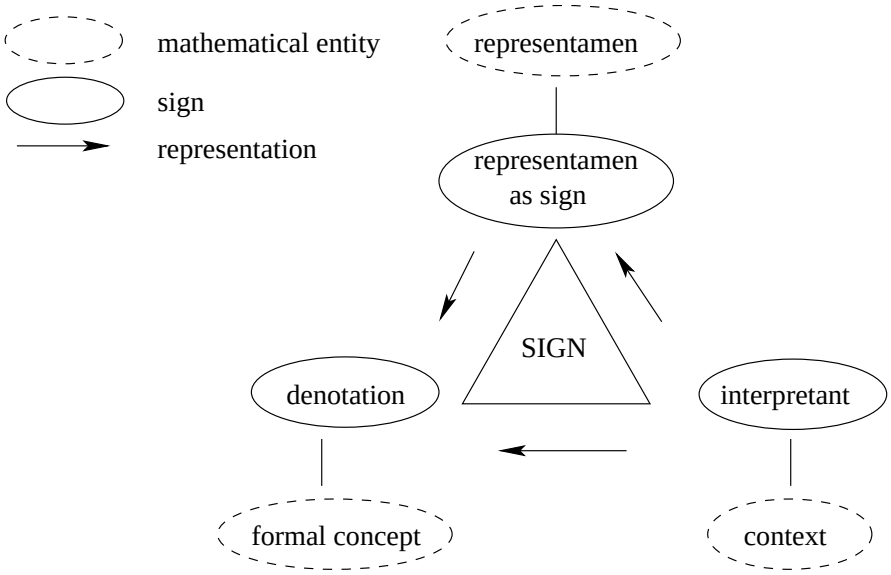

Fig. 1. The sign triad

Even though the three components of the sign are written as mappings, $rmn()$, $den()$, $ipt()$, these mappings only need to be defined with respect to the smallest possible interpretant which is the sign’s own interpretant at the moment when the sign is actually used. Further conditions for compatibility among interpretants must be provided (see below) before triadic sign relations can be considered across several or larger

interpretants. Such compatibility conditions must exist because otherwise signs would be completely isolated from each other and communication would not be possible.

Interpretants and denotations are signs themselves with respect to other interpretants. But they also relate to mathematical entities. Denotations relate to (formal) concepts. Interpretants relate to contexts, which are in this paper defined as the formalisable aspects of interpretants. Interpretants contain more information than contexts. According to Peirce, interpretants mediate between representamens and denotations. Therefore an interpretant or context only needs to contain as much information as is needed for understanding a sign. Because "a sign, or representamen, is something which stands to somebody for something in some respect or capacity" (Peirce, 1897), it follows that signs are not hypothetical but actually exist in the present or have existed in the past.

4 Synonymy and similar sign equivalences

The definition of signs in a semiotic conceptual framework does not guarantee that representamens are unambiguous and represent exactly one denotation with respect to one interpretant. Furthermore, the definition does not specify under which conditions a sign is equal to another sign or even to itself. Conditions for interpretants must be described which facilitate disambiguation and equality.

A set I of n interpretants, $i_1, i_2, ..., i_n$, is called overlapping iff

$$\forall_{a,\ 1\leq a\leq n} \exists_{b,\ 1\leq b\leq n, b\neq a} \exists_s :\ i_a \rightarrow s, i_b \rightarrow s \tag{1}$$

where s denotes a sign. The arrow relation "$\rightarrow$" is called "representation" and is the same as in figure 1. This relation is central to Peirce's definition of signs but shall not be further discussed in this paper.

With respect to a set I of overlapping interpretants, any equivalence relation can be called synonymy, denoted by $\equiv_I$, if the following necessary condition is fulfilled for all signs represented by interpretants in I:

$$\begin{gathered}(rmn(s_1), den(s_1), ipt(s_1)) \equiv_I (rmn(s_2), den(s_2), ipt(s_2)) \Longrightarrow \\ s_1 \rightarrow den(s_2),\ s_2 \rightarrow den(s_1),\ den(s_1) \equiv_I den(s_2)\end{gathered} \tag{2}$$

Only a necessary but not sufficient condition is provided for synonymy because it depends on user judgements. In a programming language, synonymy can be a form of assignment. If a programmer assigns a variable to be a pointer (or reference) to another variable's value, then these two variables are synonyms. Denotational equality is usually not required for synonymous variables because values can change over time. For example two variables, "counter := 5" and "age := 5", are not synonymous just because they have the same value. Because synonymy is an equivalence relation, signs are synonymous to themselves.

A further condition for interpretants ensures disambiguation of representamens: A set I of overlapping interpretants is called compatible iff

$$\forall_{i_1, i_2 \in I} \forall_{s_1, s_2} : (i_1 \rightarrow s_1, i_2 \rightarrow s_2,\ rmn(s_1) = rmn(s_2) \Longrightarrow s_1 \equiv_I s_2) \tag{3}$$

In the rest of this paper, all single interpretants are always assumed to be compatible with themselves. Compatibility between interpretants can always be achieved by renaming of signs, for example, by adding a prefix or suffix to a sign.

The following other equivalences are defined for signs in a set I of compatible interpretants:

$$\text{identity: } s_1 \asymp_I s_2 :\Longleftrightarrow \quad id(s_1) = id(s_2), \quad s_1 \equiv_I s_2 \tag{4}$$

$$\text{polysemy: } s_1 \doteq_I s_2 :\Longleftrightarrow \quad rmn(s_1) = rmn(s_2) \tag{5}$$

$$\text{equality: } s_1 =_I s_2 :\Longleftrightarrow den(s_1) =_I den(s_2),\ rmn(s_1) = rmn(s_2) \tag{6}$$

$$\text{equinymy: } s_1 \cong_I s_2 \Longrightarrow den(s_1) =_I den(s_2),\ s_1 \equiv_I s_2 \tag{7}$$

$$s_1 \cong_I s_2 \Longleftarrow \quad s_1 =_I s_2$$

Identity refers to what is called "object identifiers" in object-oriented languages whereas equinymy is a form of value equality. For example, in a program sequence, "age := 5, counter := 5, age := 6", the variables "age" and "counter" are initially equal. But "age" is identical to itself even though it changes its value. Identity is implemented via a set $\mathcal{I}$ of mathematical entities. The elements of $\mathcal{I}$ are called identifiers. A mapping $id()$ maps a sign onto an identifier or onto NULL if the sign does not have an identifier. It should be required that if two signs are equal and one of them has an identifier then both signs are also identical. The only operation or relation that is available for identifiers is "$=$". In contrast to synonymy which is asserted by users, object-oriented programming languages and databases have rules for when and how to create "object identifiers".

Because the relations in (4)-(7) are equivalence relations, signs are identical, polysemous, equinymous and equal to themselves. In (5) polysemy is defined with respect to equal representamens but only in compatible interpretants. Signs with equal representamens across non-compatible interpretants are often called "homographs". This definition of "polysemy" is different from the one in linguistics which does not usually imply synonymy.

The following statements summarise the implications among the relations in (4)-(7).

$$s_1 \asymp s_2 \text{ or } s_1 \doteq_I s_2 \text{ or } s_1 \cong_I s_2 \Longrightarrow s_1 \equiv_I s_2 \tag{8}$$

$$s_1 = s_2 \Longleftrightarrow s_1 \doteq_I s_2,\ s_1 \cong_I s_2 \tag{9}$$

5 Anonymous signs and mergeable interpretants

Two special cases are of interest: anonymous signs and mergeable interpretants. In programming languages, anonymous signs are constants. An anonymous sign with respect to compatible interpretants I is defined as a sign with

$$s =_I den(s) \tag{10}$$

An anonymous sign denotes itself and has no other representamen than the representamen of its denotation. Signs which are anonymous with respect to one interpretant need not be anonymous with respect to other interpretants.

The following equations and statements are true for anonymous signs s, s_1, s_2

$$s =_I den(s) \implies s =_I den(s) =_I den(den(s)) =_I \ldots \quad (11)$$

$$s =_I den(s) \implies rmn(s) = rmn(den(s)) = rmn(den(den(s)) = \ldots \quad (12)$$

$$s_1 =_I s_2 \iff den(s_1) =_I den(s_2) \quad (13)$$

$$s_1 =_I s_2 \iff s_1 \cong_I s_2 \quad (14)$$

Statement (14) is true because of $s_1 \cong s_2 \implies den(s_1) =_I den(s_2) \implies s_1 =_I s_2$. Thus for anonymous signs equality and equinymy coincide. It is of interest to consider interpretants in which equinymy and synonymy coincide. That means that synonyms have equal instead of just synonymous denotations. This leads to the next definition:

A set I of compatible interpretants is called mergeable iff for all signs in I

$$s_1 \equiv_I s_2 \implies s_1 \cong_I s_2 \quad (15)$$

which means that all of its synonyms are equinyms. If an interpretant is not mergeable with itself it can usually be split into several different interpretants which are mergeable with themselves. In mergeable interpretants, it follows that

$$s_1 =_I s_2 \iff s_1 \doteq_I s_2 \implies s_1 \cong_I s_2 \iff s_1 \equiv_I s_2 \quad (16)$$

$$s_1 =_I s_2 \iff rmn(s_1) = rmn(s_2) \quad (17)$$

because $rmn(s_1) = rmn(s_2) \implies dmn(s_1) \equiv_I dmn(s_2) \implies dmn(s_1) \doteq_I dmn(s_2)$.

From (14) and (16) it follows that for anonymous signs in mergeable interpretants, the four equivalences, synonymy, polysemy, equality and equinymy are all the same. For anonymous signs in a mergeable interpretant, the representamen rules alone determine synonymy. Because representamens are mathematical entities, it follows that anonymous signs in mergeable interpretants behave like mathematical entities.

6 Conceptual structures

While semiotic structures, such as synonymy, model the decisions a user makes with respect to a formal language, mathematical entities can be used to compute the consequences of such decisions. It is argued in this paper that the mathematical entities involved in signs (especially formal concepts and contexts) can be modelled as conceptual structures using formal concept analysis. Concept lattices can be used to show the consequences of the semiotic design decisions. Users can browse through concept lattices using a variety of existing software tools to explore the signs.

This insight is not new. In fact there are several papers, (for example, Snelting (1996)) which demonstrate the usefulness of formal concept analysis in software engineering. Our semiotic conceptual framework adds a layer of explanation to these studies by detailing how semiotic and conceptual aspects both contribute to formal languages. A semiotic perspective also adds modes of communication or “speech acts” to the conceptual framework. Examples are “assertion”, “query” and “question”. But these are not further discussed in this paper.

Formal concept analysis, description logics, Sowa's (1984) conceptual graphs, Barwise & Seligman's (1997) classifications and object-oriented formalisms each provide a slightly different definition of concepts. But they all model denotations as binary relations of types and instances/values. Thus denotations are signs of the form $[typ(s) : ins(s)]$ where $typ(s)$ is a sign called type and $ins(s)$ is a sign called instance (or value). A sign s with $den(s) \cong_I [typ(s) : ins(s)]$ is written as $s[typ(s) : ins(s)]$. A formal concept is an anonymous sign $c =_I (\{e_1, e_2, \ldots\}; \{i_1, i_2, \ldots\})$ where $e_1, e_2, \ldots$ are mathematical entities that form the extension and $i_1, i_2, \ldots$ are mathematical entities that form the intension of the formal sign.

For a fixed interpretant, denotations are mapped onto formal concepts as follows: the intension is the set of types that are inherited by the sign via a type hierarchy and the extension is the set of instances that share exactly those types. As a restriction it should be required that each sign is synonymous to at most one formal concept within the given interpretant. Within the framework of formal concepts, equality of denotations can be mathematically evaluated because formal concepts are anonymous signs. Formal concepts as defined in this paper form concept lattices as defined in formal concept analysis.

7 Contexts and meta-constructs

Several formalisms for contexts have been suggested by AI researchers (eg. McCarthy (1993)) but in general they are not integrated into reasoning applications as frequently and not as well understood as representamens and formal concepts. Figure 1 indicates that contexts should play an important role in the formalisation of signs besides representamens and formal concepts. We argue in this paper, that contexts are not as difficult to deal with as AI research suggests if they are modelled as formal concepts as well.

If contexts are modelled as formal concepts, relationships between contexts, such as containment, temporal and spatial adjacency or overlap can be modelled as conceptual relations. Contexts as formal concepts are denotations of other signs with respect to other interpretants. Peirce stresses the existence of infinite chains of interpretants (interpretants of interpretants of interpretants ...). But as formal concepts, contexts are not modelled as contexts of contexts of contexts. All contexts can be modelled as formal concepts with respect to one special meta-context of contexts because containment chains are just conceptual relations, not meta-relations.

An advantage of this approach is that apart from one meta-language which describes the semiotic conceptual framework, no other meta-languages are required. All other seemingly "meta"-languages are modelled via conceptual containment relations between their corresponding contexts. For example, all facts, rules and constructors of a programming language can be described in a single context. A program of that language is executed in a different context. Both contexts are related via a containment relation with respect to the meta-context of contexts. But the context of a programming language is not a meta-context of a program.

8 Examples

The condition of mergeability of interpretants states that synonymous signs must have equal denotations. With respect to programming languages this means that as soon as a variable changes its value, a new interpretant must be formed. It may be possible to bundle sequential changes of values. For example, if all values in an array are updated sequentially, it may be sufficient to assume one interpretant before the changes and one after the changes instead of forming a separate interpretant after each change. Some variables may also be ignored, such as counters in for-statements. This corresponds to the distinction between persistent and transient objects in object-oriented modelling. Transient variables do not initiate new interpretants.

The significance of the following examples is not that this is just another application of formal concept analysis but instead that this kind of analysis is suggested by the semiotic conceptual framework. The theory about mergeability of interpretants suggests that a number of different interpretants are invoked by any computer program depending on when certain variables change their values. It just happens that formal concept analysis can be used to analyse this data. We believe that careful consideration of the predictions made by the semiotic conceptual framework can potentially provide interesting insights. But we have not yet explored this further.

The example in figure 2 shows a piece of Python code and a corresponding concept lattice in figure 3. The contexts (or formalisable parts of interpretants) are initiated by the changes of the variables “counter” and “number”. Each context produces a different behaviour, i.e., a different print statement by the program. The lattice in figure 3 is modelled as follows: the objects are the observable behaviours of the program (i.e., the print statements). The attributes are the states of the variables which are relevant for the contexts. The formal concepts are contexts of the program. If the counter is smaller than 5 and the user guesses the number 5, the program prints “good guess”. If the number is not 5 but the counter is smaller than 5, the program prints “try again” and “please guess”, except in the first instance (counter =1) when it prints “please guess” after having printed “game starts”. If the counter is larger than 5 the game prints “game over”.

In contrast to flowcharts, the lattice representation does not represent the sequence of the statements. Wolff & Yameogo’s (2003) temporal concept analysis could be applied to the lattice to insert the temporal sequence. The lattice shows the relationships among the contexts. For example, it shows that the start-context (counter = 1) and the contexts in which the wrong number was guessed share behaviour. This is not necessarily clearly expressed in the code itself. In fact our experience with teaching scripting languages to non-programmers has shown that students often have a problem comprehending where the ’print “please guess”’ statement needs to be placed within the while loop so that it pertains to both the first iteration and to some of the later iterations. In the lattice this relationship is shown more clearly.

It should be noted that we have not yet tested whether students can read the lattices. But we are not suggesting that lattices must be used directly as a software engineering tool. The information contained in the lattice could be displayed in a different format, which would still need to be determined. We have also not yet determined in how far

```
counter = 1
print "game starts"
while counter <= 5:
    number = input("please guess the number")
    if number == 5:
        print "good guess"
        break
    else:
        print "try again"
    counter = counter +1
else:
    print "game over"
```

Fig. 2. A piece of Python code

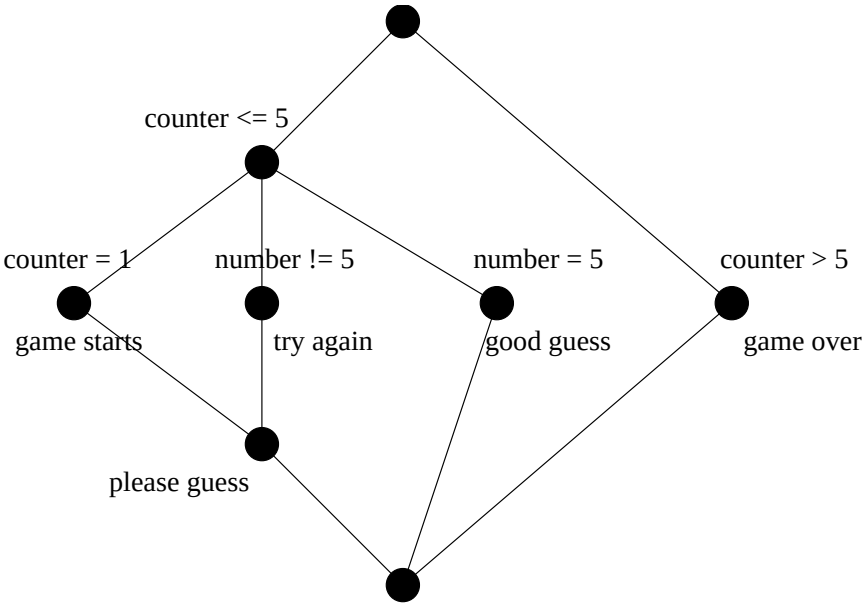

Fig. 3. A lattice of contexts for the code in figure 2

such lattices can be automatically generated from code. We intend to investigate this further in the near future.

The second example, which was taken from Ballentyne (1992) demonstrates the equivalence of Warnier diagrams (Orr, 1977) and concept lattices. The original data is shown in the upper left table in figure 4 and to be read as follows: action 1 has four crosses which correspond to the conditions $A, \neg B, \neg C$ or $A, \neg B, C$ or $A, B, \neg C$ or A, B, C. This is equivalent to condition A implying action 1. Therefore in the formal context on the left there is a cross for A and 1. Action 2 is conditioned by $\neg A$ and B which is visible both in the left table and in the formal context. After the whole context has been constructed in that manner, a second condition is to be considered which is that A and $\neg A$ and so on must exclude each other. This is true for A and B but $\neg C$ does not have any attributes and must be excluded from the lattice. A third condition is that any meet irreducible concepts in the lattice must not be labelled by an attribute because otherwise that attribute would be implied by other attributes without serving as a condition itself. For this reason, the temporary attribute t is inserted. The resulting lattice can be read in the same manner as the one in figure 3. The Warnier diagram corresponds to a set of paths from the top to the bottom of the lattice which cover all concepts. Obviously, there can be different Warnier diagrams corresponding to the same lattice.

9 Conclusion

A semiotic conceptual framework for formal languages combines conceptual reasoning and inference structures with semiotic modes, such as assertion, question and query. By considering the denotations of formal signs as formal concepts, structure is imposed. Because denotations are both signs and can be mapped to formal concepts which are mathematical entities, denotations serve as boundary objects (Star, 1989) between the mathematical world and the pragmatic world of signs. The role of contexts is often neglected. This is understandable in mathematical applications because mathematical entities exist in more global contexts. But in other formal languages, which employ richer signs, contexts are frequently changing and cannot be ignored. If contexts are modelled as formal concepts, it is not necessary to invent any new structures but instead the mechanisms of formal concept analysis can also be applied to contexts. Contexts provide a means for managing signs and sign relations. Programming languages and databases already fulfill these functions, but so far not much theory has been developed which explains the theoretical foundations of context management. A semiotic conceptual framework can provide such a theory.

References

1. Ballentyne, George (1992). Class notes. Unpublished manuscript.
2. Barwise, Jon; Seligman, Jerry (1997). *Information Flow.* The Logic of Distributed Systems. Cambridge University Press.
3. Ganter, B.; Wille, R. (1999). *Formal Concept Analysis.* Mathematical Foundations. Berlin-Heidelberg-New York: Springer, Berlin-Heidelberg.

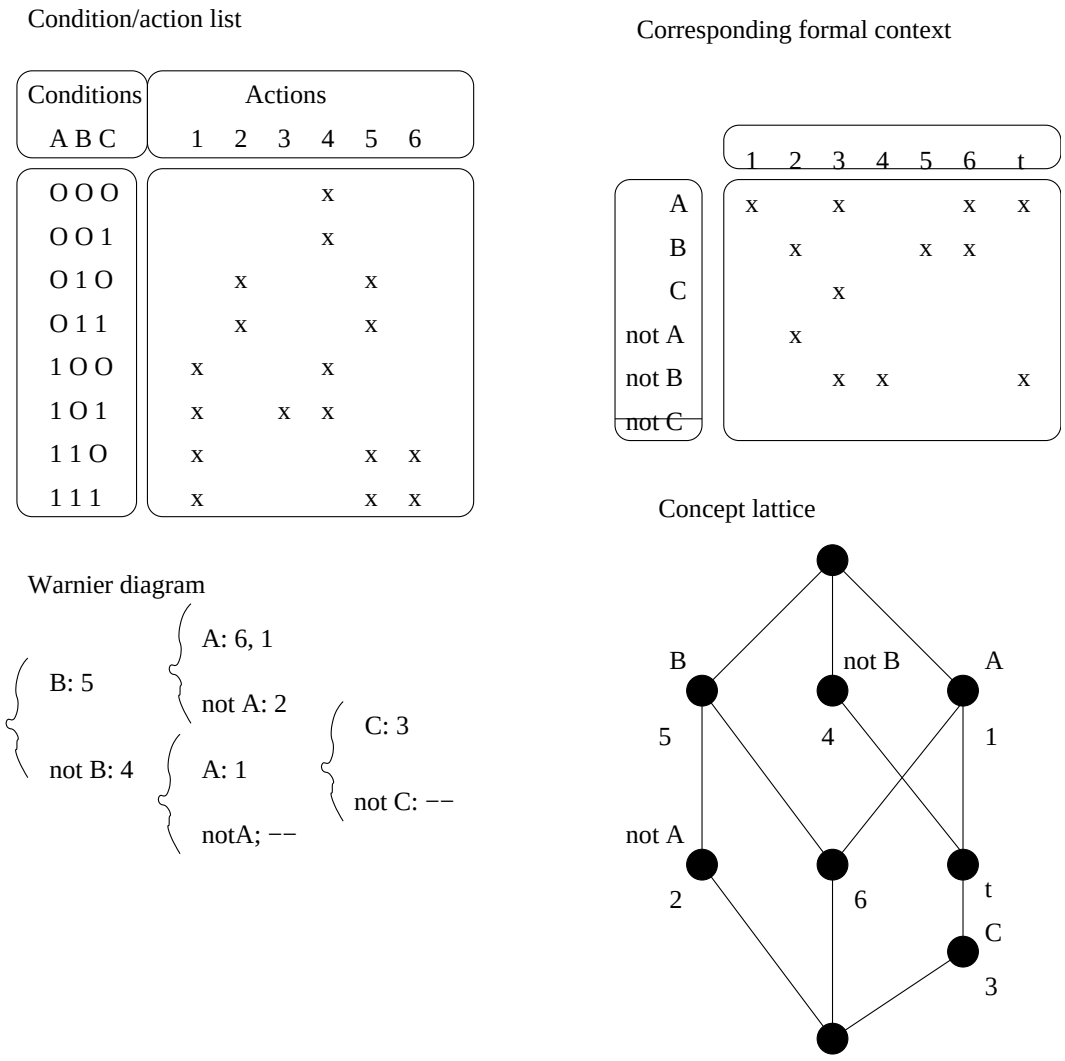

Fig. 4. Warnier diagrams and lattices

4. McCarthy, John (1993). *Notes on Formalizing Context.* Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambery, France. p. 555-560.
5. Orr, K. T. (1977). *Structured Systems Development.* Yourdon Press, New York.
6. Peirce, Charles (1897). *A Fragment.* CP 2.228. In: Collected papers. Hartshorne & Weiss. (Eds. Vol 1-6); Burks (Ed. Vol 7-8), Cambridge, Harvard University Press, 1958-1966.
7. Prechelt, Lutz (2000). *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and TCL.* IEEE Computer, 33(10), p. 23-29.
8. Snelting, Gregor (1995). *Reengineering of Configurations Based on Mathematical Concept Analysis.* ACM Transactions on Software Engineering and Methodology 5, 2, p. 99-110.
9. Sowa, J. (1984). *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, Reading, MA.
10. Star, Susan Leigh (1989). *The structure of ill-structured solutions: Boundary objects and heterogeneous problem-solving.* In: Gasser & Huhns (Eds). Distributed Artificial Intelligence, Vol 2, Pirman, p. 37-54.
11. Wolff, Karl Erich; Yameogo, Wendsomde (2003). *Time Dimension, Objects, and Life Tracks. A Conceptual Analysis.* In: de Moor; Lex; Ganter (Eds.). Conceptual Structures for Knowledge Creation and Communication. Lecture notes in Ai 2746. Springer Verlag.