An Introduction to Semiotic-Conceptual Analysis with Formal Concept Analysis

Uta Priss

Zentrum für erfolgreiches Lehren und Lernen
Ostfalia University of Applied Sciences
Wolfenbüttel, Germany
www.upriss.org.uk

Abstract. This paper presents a formalisation of Peirce's notion of 'sign' using a triadic relation with a functional dependency. The three sign components are then modelled as concepts in lattices which are connected via a semiotic mapping. We call the study of relationships relating to semiotic systems modelled in this manner a *semiotic-conceptual analysis*. It is argued that semiotic-conceptual analyses allow for a variety of applications and serve as a unifying framework for a number of previously presented applications of FCA.

1 Introduction

The American philosopher C. S. Peirce was a major contributor to many fields with a particular interest in semiotics. The following quote shows one of his definitions for the relationships involved in using a sign:

> A REPRESENTAMEN is a subject of a triadic relation TO a second, called its OBJECT, FOR a third, called its INTERPRETANT, this triadic relation being such that the REPRESENTAMEN determines its interpretant to stand in the same triadic relation to the same object for some interpretant. (Peirce, CP 1.541)[1]

According to Peirce a sign consists of a physical form (representamen) which could, for example, be written, spoken or represented by neurons firing in a brain, a meaning (object) and another sign (interpretant) which mirrors the original sign, for example, in the mind of a person producing or observing a sign. It should be pointed out that the use of the term 'relation' by Peirce is not necessarily the same as in modern mathematics which distinguishes more clearly between a 'relation' and its 'instances'. Initially Peirce even referred to mathematical relations as 'relatives' (Maddux, 1991).

We have previously presented an attempt at mathematically formalising Peirce's definition (Priss, 2004). In our previous attempt we tried to presuppose as few assumptions about semiotic relations as possible which led to a fairly open structural description which, however, appeared to be limited with respect to usefulness in applications.

[1] It is customary among Peirce scholars to cite Peirce in this manner using an abbreviation of the publication series, volume number and paragraph or page numbers.

 Published in Sadok Ben Yahia, Jan Konecny (Eds.): CLA 2015, pp. 135–146, ISBN 978–2–9544948–0–7, Blaise Pascal University, LIMOS laboratory, Clermont-Ferrand, 2015.

Now we are presenting another formalisation which imposes a functional dependency on the triadic relation. The notions from Priss (2004) are translated into this new formalism which is in many ways simpler, more clearly defined and appears to be more useful for applications.

In order to avoid confusion with the notion of 'object' in FCA[2], we use the term 'denotation' instead of Peirce's 'object' in the remainder of this paper. We translate the first half of Peirce's definition into modern language as follows: "A representamen is a parameter of a function resulting in a second, called its denotation, where the third, the function instance, is called interpretant." – or in other words as a function of type 'third(first) = second'. In our modelling, a set of such functions together with their parameter/value pairs constitute a triadic semiotic relation. We use Peirce's notion of 'interpretant' for the function instances whereas the functions themselves are called 'interpretations'. A sign is then an instance of this triadic relation consisting of representamen, denotation and interpretation. This is more formally defined in the next section.

The second half of Peirce's definition refers to the mental image that a sign invokes in participants of communication acts. For Peirce, interpretants are mental images which can themselves be thought about and thus become representamens for other interpretants and so on. Because the focus of this paper is on formal, not natural languages, mental images are not important. We suggest that in formal languages, interpretants are not mental representations but instead other formal structures, for example, states in a computer program.

The data structures used in formal languages (such as programming languages, XML or UML) can contain a significant amount of complexity. A semiotic-conceptual analysis as proposed in this paper allows to investigate the components of such structures as signs with their representamens, denotations and interpretations and their relationships to each other. As a means of structuring the semiotic components we use FCA concept lattices.

It should be noted that there has recently been an increase of interest in triadic FCA (e.g., Gnatyshak et al. (2013), Belohlavek & Osicka (2012)) which could also be used to investigate triadic semiotic relations. But Peirce tends to see triadic relations as consisting of three components of increasing complexity:

> The First is that whose being is simply in itself, not referring to anything nor lying behind anything. The Second is that which is what it is by force of something to which it is second. The Third is that which is what it is owing to things between which it mediates and which it brings into relation to each other. (Peirce, EP 1:248; CP 1.356)

In our opinion this is better expressed by a function instance of type 'third(first) = second' than by an instance '(first, second, third)' of a triadic relation. Other researchers have already suggested formalisations of Peirce's philosophy. Interestingly, Marty (1992), Goguen (1999) and Zalamea (2010) all suggest using Category Theory

[2] Because Formal Concept Analysis (FCA) is the main topic of this conference, this paper does not provide an introduction to FCA. Information about FCA can be found, for example, on-line (http://www.fcahome.org.uk) and in the main FCA textbook by Ganter & Wille (1999).

for modelling Peirce's philosophy even though they appear to have worked independently of each other. Marty even connects Category Theory with FCA in his modelling. Goguen develops what he calls 'algebraic semiotics'. Zalamea is more focussed on Existential Graphs than semiotics (and unfortunately most of his papers are in Spanish). Nevertheless our formalisation is by far not as abstract as any of these existing formalisations which are therefore not further discussed in this paper.

This paper has five further sections. Section 2 presents the definitions of signs, semiotic relations and NULL-elements. Section 3 continues with defining concept lattices for semiotic relations. Section 4 explains degrees of equality among signs. Section 5 discusses mappings among the lattices from Section 3 and presents further examples. The paper ends with a concluding section.

2 Core definitions of a semiotic-conceptual analysis

The main purpose of this work is to extend Peirce's sign notion to formal languages such as computer programming languages and formal representations. Figure 1 displays a simple Python program which is called 'Example 1' in the remainder of this paper. The table underneath shows the values of the variables of Example 1 after an execution. The variables are representamens and their values are denotations. Because Peirce's definition of signs seems to indicate that there is a separate interpretant for each sign, there are at least eight different interpretants in column 3 of the table. It seems more interesting, however, to group interpretants than to consider them individually. We call such groupings of interpretants *interpretations*. In natural language examples, one could group all the interpretants that belong to a sentence or paragraph. In programming languages starting a loop or calling a subroutine might start a new interpretation. As a condition for interpretations we propose that each representamen must have a unique denotation in an interpretation, or in other words, interpretations are functions. There are many different possibilities for choosing sets of interpretations. Two possibilities, called I_A and I_B in the rest of the paper, are shown in the last two columns of the table. Each contains two elements which is in this case the minimum required number because some variables in Example 1 have two different values. In our modelling an interpretant corresponds to a pair of representamen and interpretation. For R and I_A there are ten interpretants (and therefore ten signs) whereas there are eight for R and I_B. The first three columns of the table can be automatically derived using a debugging tool. The interpretants are numbered in the sequence in which they are printed by the debugger.

Definition 1: A *semiotic relation* $\overline{S} \subseteq I \times R \times D$ is a relation between three sets (a set R of *representamens*, a set D of *denotations* and a set I of *interpretations*) with the condition that any $i \in I$ is a partial function $i : R \nrightarrow D$. A relation instance (i, r, d) with $i(r) = d$ is called a *sign*. In addition to $\overline{S}$, we define the *semiotic (partial) mapping* $S : I \times R \nrightarrow D$ with $S(i, r) = d$ iff $i(r) = d$. The pairs (i, r) for which d exists are called *interpretants*.

It follows that there are as many signs as there are interpretants. Example 1 shows two semiotic relations using either I_A or I_B for the interpretations. The interpretations firstLoop, secondLoop and firstValue are total functions. The interpretation second-

```
input_end = "no"
while input_end != "yes":
    input1 = raw_input("Please type something: ")
    input2 = raw_input("Please type something else: ")
    if (input1 == input2):
       counter = 1
       error = 1
       print "The two inputs should be different!"
    else:
       counter = 2
    input_end = raw_input("End this program? ")
```

representamens R (variables)	denotations D (values)	interpretants	interpretations I_A ($\sim$10 interpretants)	interpretations I_B ($\sim$ 8 interpretants)
input1	"Hello World"	j1	firstLoop	firstValue
input2	"Hello World"	j2	firstLoop	firstValue
counter	1	j3	firstLoop	firstValue
input_end	no	j4	firstLoop	firstValue
error	1	j5	firstLoop	firstValue
input1	"Hello World"	j6 (or j1)	secondLoop	firstValue
input2	"How are you"	j7	secondLoop	secondValue
counter	2	j8	secondLoop	secondValue
input_end	yes	j9	secondLoop	secondValue
error	1	j10 (or j5)	secondLoop	firstValue

Fig. 1. A Python program (called 'Example 1' in this paper)

Value is a partial function. Because for (i_1, r_1, d_1) and (i_2, r_2, d_2), $i_1 = i_2, r_1 = r_2 \Rightarrow d_1 = d_2$, it follows that all $r \in R$ with $r(i) = d$ are also partial functions $r : I \nrightarrow D$. The reason for having a relation $\overline{S}$ and a mapping S is because Peirce defines a relation but in applications a mapping might be more usable. In this paper the sets R, D and I are meant to be finite and not in any sense universal but built for an application. The assignment operation (written ':=' in mathematics or '=' in programming languages) is an example of $i(r) = d$ except that i is usually implied and not explicitly stated in that case.

Using the terminology from database theory, we call a semiotic relation *a triadic relation with functional dependency*. This is because, on the one hand, Peirce calls it not a mapping but a 'triadic relation', on the other hand, without this functional dependency it would not be possible to determine the meaning of a sign given its representamen and an interpretation. Some philosophers might object to Definition 1 because of the functional dependency. We argue that the added functional dependency yields an interesting structure which can be explored as shown in this paper.

The idea of using interpretations as a means of assigning meaning to symbols is already known from formal semantics and model theory. But this paper has a different focus by treating interpretations and representamens as dual structures. Furthermore in applications, $S(i, r)$ might be implemented as an algorithmic procedure which determines d for r based on information about i at runtime. A debugger as in Example 1

is not part of the original code but at a meta-level. Since the original code might request user input (as in Example 1), the relation instances (i, r, d) are only known while or after the code was executed. Thus the semiotic relation is dynamically generated in an application. This is in accordance with Peirce's ideas about how it is important for semiotics to consider how a sign is actually *used*. The mathematical modelling (as in Definition 1) which is conducted after a computer program finished running, ignores this and simply considers the semiotic relation to be statically presented.

Priss (2004) distinguishes between triadic signs and anonymous signs which are less complex. In the case of anonymous signs, the underlying semiotic relation can be reduced to a binary or unary relation because of additional constraints. Examples of anonymous signs are constants in programming languages and many variables used in mathematical expressions. For instance, the values of variables in the Pythagorean equation $a^2 + b^2 = c^2$ are all the values of all possibly existing right-angled triangles. But, on the one hand, if $a^2 + b^2 = c^2$ is used in a proof, it is fine to assume $|I| = 1$ because within the proof the variables do not change their values. On the other hand, if someone uses the formula for an existing triangle, one can assume $S(i, r) = r$ because in that case the denotations can be set equal to the representamens. Thus within a proof or within a mathematical calculation variables can be instances of binary or unary relations and thus anonymous signs. However, in the following Python program:

```
a = input("First side: ")
b = input("Second side: ")
print a*a + b*b
```

the values of the variables change depending on what is entered by a user. Here the signs a and b are triadic.

A sign is usually represented by its representamen. In a semiotic analysis it may be important to distinguish between 'sign' and 'representamen'. In natural language this is sometimes indicated by using quotes (e.g., the word 'word'). In Example 1, the variable 'input1' is a representamen whereas the variable 'input1' with a value 'Hello World' in the context of firstLoop is a sign. It can happen that a representamen is taken out of its context of use and loses its connection to an interpretation and a denotation. For example, one can encounter an old file which can no longer be read by any current program. But a sign always has three components (i, r, d). Thus just looking at the source code of a file creates an interpretant in that person's mind even though this new sign and the original sign may have nothing in common other than the representamen. Using the next definition, interpretations that are partial functions can be converted into total functions.

Definition 2: For a semiotic relation, a *NULL-element* $d_\perp$ is a special kind of denotation with the following conditions: (i) $i(r)$ undefined in $D \Rightarrow i(r) := d_\perp$ in $D \cup \{d_\perp\}$. (ii) $d_\perp \in D \Rightarrow$ all i are total functions.

Thus by enlarging D with one more element, one can convert all i into total functions. If all i are already total functions, then $d_\perp$ need not exist. The semiotic mapping S can be extended to a total function $S : I \times R \to D \cup \{d_\perp\}$. There can be different reasons for NULL-elements: caused by the selection of interpretations or by the code itself. Variables are successively added to a program and thus undefined for any interpretation that occurs before a variable is first defined. In Example 1, secondValue is a partial function because secondValue(input1) = secondValue(error) = $d_\perp$. But I_A shows

that all interpretations can be total functions. On the other hand, if a user always enters two different values, then the variable 'error' is undefined for all interpretations. This could be avoided by changing the code of Example 1. In more complex programs it may be more difficult to avoid $d_\perp$, for example if a call to an external routine returns an undefined value. Programming languages tend to allow operations with $d_\perp$, such as evaluating whether a variable is equal to $d_\perp$, in order to avoid run-time errors resulting from undefined values. Because the modelling in the next section would be more complex if conditions for $d_\perp$ were added we decided to mostly ignore $d_\perp$ in the remainder of this introductory paper.

3 Concept lattices of a semiotic relation

In order to explore relationships among signs we are suggesting to model the components of signs as concept lattices. The interpretations which are (partial) functions from R to D then give rise to mappings between the lattice for R and the lattice for D. Figure 2 shows an example of concept lattices for the semiotic relation from Example 1. The objects are the representamens, denotations and interpretations of Example 1. The attributes are selected for characterising the sets and depend on the purpose of an application. If the denotations are values of a programming language, then data types are a fairly natural choice for the attributes of a denotation lattice.

Attributes for representamens should focus on representational aspects. In Example 1, all input variables start with the letters 'input' because of a naming style used by the programmer of that program. In some languages certain variables start with upper or lowercase letters, use additional symbols (such as '@' for arrays) or are complex structures (such as 'root.find("file").attrib["size"]') which can be analysed in a representamen lattice. In strongly-typed languages, data types could be attributes of representamens but in languages where variables can change their type, data types do not belong into a representamen lattice. Rules for representamens also determine what is to be ignored. For example white space is ignored in many locations of a computer program. The font of written signs is often ignored but mathematicians might use Latin, Greek and Fraktur fonts for representamens of different types of denotations.

One way of deriving a lattice for interpretations is to create a partially ordered set using the ordering relation as to whether one interpretation precedes another one or whether they exist in parallel. A lattice is then generated using the Dedekind closure. In Figure 2 the attributes represent some scaling of the time points of the interpretations. Thus temporal sequences can be expressed but any other ordering can be used as well.

Definition 3: For a set R of representamens, a concept lattice $B(R, M_R, J_R)$ is defined where M_R is a set of attributes used to characterise representamens and J_R is a binary relation $J_R \subseteq R \times M_R$. $B(R, M_R, J_R)$ is called *representamen lattice*. $B(R, M_R, J_R)$ is *complete for a set of interpretations*[3] if for all $r \in R$: $\forall_{i \in I} : \gamma(r_1) = \gamma(r_2) \Rightarrow i(r_1) = i(r_2)$ and $\gamma(r_1) \neq \gamma(r_2) \Rightarrow \exists_{i \in I} : i(r_1) \neq i(r_2)$.

Definition 4: For a set I of interpretations, a concept lattice $B(I, M_I, J_I)$ is defined where M_I is a set of attributes used to characterise the interpretations and J_I is a binary

[3] For an object o its object concept $\gamma(o)$ is the smallest concept which has the object in its extension.

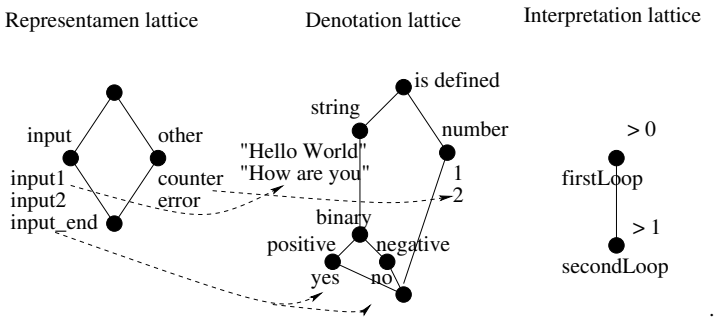

Fig. 2. Lattices for Example 1

relation $J_I \subseteq I \times M_I$. $B(I, M_I, J_I)$ is called *interpretation lattice*. $B(I, M_I, J_I)$ is *complete for a set of representamens* if for all $i \in I$: $\forall_{r \in R} : \gamma(i_1) = \gamma(i_2) \Rightarrow i_1(r) = i_2(r)$ and $\gamma(i_1) \neq \gamma(i_2) \Rightarrow \exists_{r \in R} : i_1(r) \neq i_2(r)$.

The representamen lattice in Figure 2 is not complete for I_A because, for example, 'input_end' and 'input1' have different denotations. The interpretation lattice is complete for R because no objects have the same object concept and firstLoop and secondLoop have different denotations, for example, for 'counter'. Completeness means that exactly those representamens or interpretations that share their object concepts can be used interchangeably without any impact on their relationship with the other sets. The dashed lines in Figure 2 are explained below in Section 5.

Definition 5: For a set $D \setminus \{d_\perp\}$ of denotations, a concept lattice $B(D, M_D, J_D)$ is defined where M_D is a set of attributes used to characterise the denotations and J_D is a binary relation $J_D \subseteq D \times M_D$. $B(D, M_D, J_D)$ is called *denotation lattice*.

4 Equality and other sign properties

Before continuing with the consequences of the definitions of the previous section, equality of signs should be discussed because there are different degrees of equality. Two signs, (i_1, r_1, d_1) and (i_2, r_2, d_2), are equal if all three components are equal. Because of the functional dependency this means that two signs are equal if $i_1 = i_2$ and $r_1 = r_2$. In normal mathematics the equal sign is used for denotational equality. For example, $x = 5$ means that x has the value of 5 although clearly the representamen x has nothing in common with the representamen 5. Since signs are usually represented by their representamens denotational equality needs to be distinguished from equality between signs. Denotational equality is called 'strong synonymy' in the definition below. Even strong synonymy is sometimes too much. For example natural language synonyms (such as 'car' and 'automobile') tend to always still have subtle differences in meaning. In programming languages, if a counter variable increases its value by 1, it is still thought of as the same variable. But if such a variable changes from '3' to 'Hello World' and then to '4', depending on the circumstances, it might indicate an

error. Therefore we are defining a tolerance relation[4] $T \subseteq D \times D$ to express that some denotations are close to each other in meaning. With respect to the denotation lattice, the relation T can be defined as the equivalence relation of having the same object concept or via a distance metric between concepts. The following definition is an adaptation from Priss (2004) that is adjusted to the formalisation in this paper.

Definition 6: For a semiotic relation with tolerance relations $T_D \subseteq D \times D$ and $T_I \subseteq I \times I$ the following are defined:

- i_1 and i_2 are *compatible* $\Leftrightarrow \forall_{r \in R, i_1(r) \neq d_\perp, i_2(r) \neq d_\perp} : (i_1(r), i_2(r)) \in T_D$
- i_1 and i_2 are *mergeable* $\Leftrightarrow \forall_{r \in R, i_1(r) \neq d_\perp, i_2(r) \neq d_\perp} : i_1(r) = i_2(r)$
- i_1 and i_2 are *T_I-mergeable* $\Leftrightarrow (i_1, i_2) \in T_I$ and i_1 and i_2 are mergeable
- (i_1, r_1, d_1) and (i_2, r_2, d_2) are *strong synonyms* $\Leftrightarrow r_1 \neq r_2$ and $d_1 = d_2$
- (i_1, r_1, d_1) and (i_2, r_2, d_2) are *synonyms* $\Leftrightarrow r_1 \neq r_2$ and $(d_1, d_2) \in T_D$
- (i_1, r_1, d_1) and (i_2, r_2, d_2) are *equinyms* $\Leftrightarrow r_1 = r_2$ and $d_1 = d_2$
- (i_1, r_1, d_1) and (i_2, r_2, d_2) are *polysemous* $\Leftrightarrow r_1 = r_2$ and $(d_1, d_2) \in T_D$
- (i_1, r_1, d_1) and (i_2, r_2, d_2) are *homographs* $\Leftrightarrow r_1 = r_2$ and $(d_1, d_2) \notin T_D$

It follows that if a representamen lattice is complete for a set of interpretations, representamens that share their object concepts are strong synonyms for all interpretations. In Example 1, if T_D corresponds to $\{$Hello World, How are you$\}$, $\{$yes, no$\}$, $\{1, 2\}$ then firstLoop and secondLoop are compatible. Essentially this means that variables do not radically change their meaning between firstLoop and secondLoop. Mergeable interpretations have the same denotation for each representamen and could be merged into one interpretation. In Example 1 the interpretations in I_A (or in I_B) are not mergeable. Using T_I-mergeability it can be ensured that only interpretations which have something in common (for example temporal adjacency) are merged. There are no examples of homographs in Example 1 but the following table shows some examples for the other notions of Definition 6.

strong synonyms	(firstLoop, input2, "Hello World")	(secondLoop, input1, "Hello World")
synonyms	(firstLoop, input1, "Hello World")	(secondLoop, input2, "How are you")
equinyms	(firstLoop, input1, "Hello World")	(secondLoop, input1, "Hello World")
polysemous	(firstLoop, input2, "Hello World")	(secondLoop, input2, "How are you")

Some programming languages use further types of synonymy-like relations, for example, variables can have the same value and but not the same data type or the same value but not be referring to the same object. An example of homographs in natural languages is presented by the verb 'lead' and the metal 'lead'. In programming languages, homographs are variables which have the same name but are used for totally different purposes. If this happens in separate subroutines of a program, it does not pose a problem. But if it involves global variables it might indicate an error in the code. Thus algorithms for *homograph detection* can be useful for checking the consistency of programs. Compatible interpretations are free of homographs.

Definition 7: A semiotic relation with concept lattices as defined in Definitions 3-5 is called a *semiotic system*. The study of semiotic systems is called a *semiotic-conceptual analysis*.

[4] A tolerance relation is reflexive and symmetric.

5 Mappings between the concept lattices

A next step is to investigate how (and whether) the interpretations as functions from R to D give rise to interesting mappings between the representamen and denotation lattice. For example, if the representamen lattice has an attribute 'starts with uppercase letter' and it is common practice in a programming language to use uppercase letters for names of classes and there is an attribute 'classes' in the denotation lattice, then one would want to investigate whether this information is preserved by the mapping amongst the lattices. The following definition describes a basic relationship:

Definition 8: For a semiotic relation, the power set P(R), subsets $I_1 \subseteq I$ and $R_1 \subseteq R$ we define: $I_1^\vee : P(R)\backslash\{\} \to B(D, M_D, J_D)$ with $I_1^\vee(R_1) := \bigvee_{i\in I_1} \bigvee_{r\in R_1} \gamma(i(r))$.

Because the join relation in a lattice is commutative and associative it does not matter whether one first iterates through interpretations or through representamens (i.e., $\bigvee_{i\in I_1} \bigvee_{r\in R_1}$ or $\bigvee_{r\in R_1} \bigvee_{i\in I_1}$). An analogous function can be defined for infima. One can also consider the inverse $(I_1^\vee)^{-1}$.

Definition 8 allows for different types of applications. One can look at the results for extensions (and thus concepts of the representamen lattice), one-element sets (corresponding to individual elements in R) or elements of a tolerance relation. The same holds for the subsets of I. The question that arises in each application is whether the mapping $I_1^\vee$ has some further properties, such as being order-preserving or whether it forms an 'infomorphism' in Barwise & Seligman's (1997) terminology (together with an inverse mapping). It may be of interest to find the subsets of R for which $(I_1^\vee)^{-1} I_1^\vee(R_1) = R_1$.

In the case of Figure 2, 'input_end' is mapped onto the concepts with attribute 'positive', 'negative' or 'binary' depending on which set of interpretations is used. The other representamens are always mapped onto the same concepts no matter which set of interpretations is used. For the extensions of the representamen lattice this leads to an order-preserving mapping. Thus overall the structures of the representamen and denotation lattice seem very compatible in this example. Other examples could produce mappings which change radically between different interpretations. In a worst case scenario, every representamen is mapped to the top concept of the denotation lattice as soon as more than one interpretation is involved.

In Figure 2 the interpretation lattice is depicted without any connection to the other two lattices. Furthermore even though a construction of $R_1^\vee$ in analogy to $I_1^\vee$ would be possible it would not be interesting for most applications because most elements would be mapped to the top element of the denotation lattice. Thus different strategies are needed for the representamen and interpretation lattices. One possibility of connecting the three lattices is to use a 'faceted display' similar to Priss (2000). The idea for Figure 3 is to use two facets: the denotation lattice which also contains the mapped representamens and the interpretation lattice. If a user 'clicks' on the upper concept in the interpretation lattice, the lattice on the left-hand side of Figure 3 is displayed. If a user clicks on the lower interpretation, the lattice on the right-hand side is displayed. Switching between the two interpretations would show the movement of 'input_end'. This is also reminiscent of the work by Wolff (2004) who uses 'animated' concept lattices which show the movement of 'complex objects' (in contrast to formal objects)

across the nodes of a concept lattice. In our semiotic-conceptual analysis the interpretations are not necessarily linearly-ordered (as Wolff's time units) but ordered according to a concept lattice.

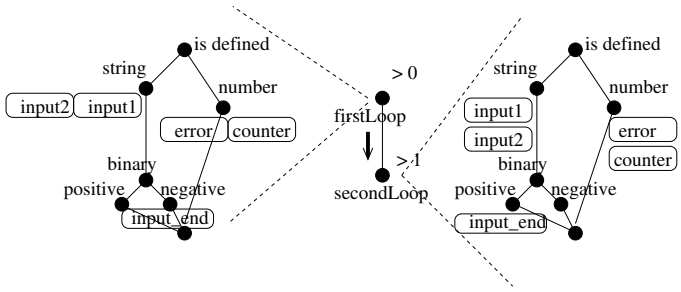

Fig. 3. Switching between interpretations

Instead of variables or strings, representamens can also be more complex structures, such as graphs, UML diagrams, Peirce's existential graphs, relations or other complex mathematical structures which are then analysed using interpretations. Figure 4 shows an example from Priss (1998) which was originally presented in terms of what Priss called 'relational concept analysis[5]'. The words in the figure are entries from the electronic lexical database WordNet[6]. The solid lines in Figure 4 are subconcept relation instances from a concept lattice although the lattice drawing is incomplete in the figure. The dashed lines are part-whole relation instances that are defined among the concepts. Using a semiotic-conceptual analysis, this figure can be generated by using representamens which are instances of a part-whole relation. Two interpretations are involved: one maps the first component of each relation instance into the denotation lattice, the other one maps the second component. Each dashed line corresponds to the mapping of one representamen. For each representamen, $I^{\vee}(r)$ is the whole and $I^{\wedge}(r)$ the part of the relation instance. Priss (1999) calculates bases for semantic relations which in this modelling as a semiotic-conceptual analysis correspond to searching for infima and suprema of such representamens as binary relations.

Figure 4 shows an example of a data error. The supremum of 'hand' and 'foot' should be the concept which is a part of 'limb'. There should be a part-whole relation from 'digit' to that 'extremity' concept. Although it would be possible to write an algorithm that checks for this error systematically, this is probably again an example of where a user can detect an error in the data more easily (because of the lack of symmetry) if the data is graphically presented. We argue that there are so many different ways of how semiotic-conceptual analyses can be used that it is not feasible to write

[5] These days the notion 'relational concept analysis' is used in a different meaning by other authors.

[6] https://wordnet.princeton.edu/

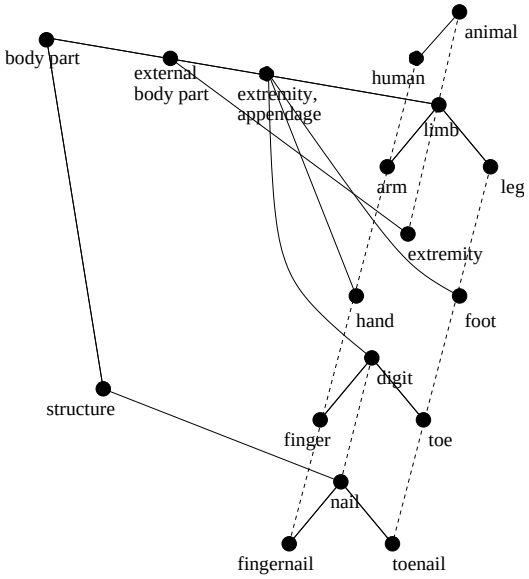

Fig. 4. Representamens showing a part-whole relation

algorithms for any possible situation. In many cases the data can be modelled for an application and then interactively investigated.

In Figure 4, the representamens are instances of a binary relation or pairs of denotations. Thus there are no intrinsic differences between what is a representamen, denotation or interpretation. Denotations are often represented by strings and thus are signs themselves (with respect to another semiotic relation). A computer program as a whole can also be a representamen. Since that is then a single representamen, the relation between the program output (its denotations) and the succession of states (its interpretations) is then a binary relation. Priss (2004) shows an example of a concept lattice for such a relation.

6 Conclusion and outlook

This paper presents a semiotic-conceptual analysis that models the three components of a Peircean semiotic relation as concept lattices which are connected via a semiotic mapping. The paper shows that the formalisation of such a semiotic-conceptual analysis provides a unified framework for a number of our previous FCA applications (Priss, 1998-2004). It also presents another view on Wolff's (2004) animated concept lattices.

But this paper only describes a starting point for this kind of modelling. Instead of considering one semiotic system with sets R, D, I, one could also consider several semiotic systems with sets R_1, D_1, I_1 and so on as subsets of larger sets R, D, I. Then one could investigate what happens if, for example, the signs from one semiotic system become the representamens, interpretations or denotations of another semiotic system.

For example, in the second half of Peirce's sign definition in Section 1 he suggests that for $i_1(r) = d$ there should be an i_2 with $i_2(i_1) = d$. Furthermore one could consider a denotation lattice as a channel between different representamen lattices in the terminology of Barwise & Seligman's (1997) information flow theory as briefly mentioned in Section 5 which also poses some other open questions.

There are connections with existing formalisms (for example model-theoretic semantics) that need further exploration. In some sense a semiotic-conceptual analysis subsumes syntactic relationships (among representamens), semantic relationships (among denotations) and pragmatic relationships (among interpretations) in one formalisation. Other forms of semiotic analyses which use the definitions from Section 2 and 4 but use other structures than concept lattices (as suggested in Section 3) are possible as well. Hopefully future research will address such questions and continue this work.

References

1. Barwise, Jon; Seligman, Jerry (1997). *Information Flow. The Logic of Distributed Systems.* Cambridge University Press.
2. Belohlavek, Radim; Osicka, Petr (2012). *Triadic concept lattices of data with graded attributes.* International Journal of General Systems 41.2, p. 93-108.
3. Ganter, Bernhard; Wille, Rudolf (1999). *Formal Concept Analysis. Mathematical Foundations.* Berlin-Heidelberg-New York: Springer.
4. Gnatyshak, Dmitry; Ignatov, Dmitry; Kuznetsov, Sergei O. (2013). *From Triadic FCA to Triclustering: Experimental Comparison of Some Triclustering Algorithms.* CLA. Vol. 1062.
5. Goguen, Joseph (1999). *An introduction to algebraic semiotics, with application to user interface design.* Computation for metaphors, analogy, and agents. Springer Berlin Heidelberg, p. 242-291.
6. Priss, Uta (1998). *The Formalization of WordNet by Methods of Relational Concept Analysis.* In: Fellbaum, Christiane (ed.), WordNet: An Electronic Lexical Database and Some of its Applications, MIT press, p. 179-196.
7. Priss, Uta (1999). *Efficient Implementation of Semantic Relations in Lexical Databases.* Computational Intelligence, Vol. 15, 1, p. 79-87.
8. Priss, Uta (2000). *Lattice-based Information Retrieval.* Knowledge Organization, Vol. 27, 3, p. 132-142.
9. Priss, Uta (2004). *Signs and Formal Concepts.* In: Eklund (ed.), Concept Lattices: Second International Conference on Formal Concept Analysis, Springer Verlag, LNCS 2961, 2004, p. 28-38.
10. Maddux, Roger D. (1991). *The origin of relation algebras in the development and axiomatization of the calculus of relations.* Studia Logica 50, 3-4, p. 421-455.
11. Marty, Robert (1992). *Foliated semantic networks: concepts, facts, qualities.* Computers & mathematics with applications 23.6, p. 679-696.
12. Wolff, Karl Erich (2004). *Towards a conceptual theory of indistinguishable objects.* Concept Lattices. Springer Berlin Heidelberg, p. 180-188.
13. Zalamea, Fernando (2010). *Towards a Complex Variable Interpretation of Peirces Existential Graphs.* In: Bergman, M., Paavola, S., Pietarinen, A.-V., & Rydenfelt, H. (Eds.). Ideas in Action: Proceedings of the Applying Peirce Conference, p. 277-287.