DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

ONDERZOEKSRAPPORT NR 9744

COMPLEXITY MEASURES FOR OBJECT-ORIENTED CONCEPTUAL MODELS OF AN APPLICATION DOMAIN

by

Geert Poels

Guido Dedene

Katholieke Universiteit Leuven

Naamsestraat 69, B-3000 Leuven

ONDERZOEKSRAPPORT NR 9744

COMPLEXITY MEASURES FOR OBJECT-ORIENTED CONCEPTUAL MODELS OF AN APPLICATION DOMAIN

by

Geert Poels

Guido Dedene

D/1997/2376/46

Complexity measures for object-oriented conceptual models of an application domain

Geert Poels* [*]
Guido Dedene

Department of Applied Economic Sciences
Katholieke Universiteit Leuven
Naamsestraat 69, B-3000 Leuven, Belgium
tel: +32 16 32 68 91
fax: +32 16 32 67 32
e-mail: {geert.poels,guido.dedene}@econ.kuleuven.ac.be

Abstract

According to Norman Fenton few work has been done on measuring the complexity of the problems underlying software development. Nonetheless, it is believed that this attribute has a significant impact on software quality and development effort. A substantial portion of the underlying problems are captured in the conceptual model of the application domain. Based on previous work on conceptual modelling of application domains, the attribute 'complexity of a conceptual model' is formally defined in this paper using elementary concepts from Measure Theory. Moreover, a number of complexity measures are defined and validated against this complexity definition. It is argued and demonstrated that these problem domain measures are part of a solution to the problem outlined by Norman Fenton.

Keywords

conceptual modelling, formal specifications, Measure Theory, complexity measures, measure validity

[*] Research Assistant of the Fund for Scientific Research - Flanders

1. Introduction

A recurrent theme in software engineering research is the validation of the hypothesised relationship between software product abstraction attributes and software quality attributes [2], [4], [9], [10], [14]. Software product abstractions [1] do not merely include the source code of the software, but also various types of abstractions used in all phases of software development, e.g., flow graphs, inheritance trees, formal specifications, etc. When a significant relationship between quality attributes like correctness, reusability, adaptability or maintainability and attributes of early software product abstractions can be shown to exist, then a theoretical basis for quality prediction and control has been established. However, before such relationships can be validated, measures must be defined for the attributes of the early software product abstractions.

According to Norman Fenton one of the attributes that is potentially related to software quality, but also to software process attributes such as development time and costs, is the complexity of the problems underlying software development [7]. A problem qualifies as a software product if it can be stated as a list of requirements or a specification. The complexity of the underlying problem is in fact the same as the complexity of the requirements. Some problems are inherently more complex than others since they are more difficult to solve, implying that in a software engineering context they require more development effort. Also, more complex problems lead to more complex solutions resulting in software that is less understandable, less maintainable, etc.

Apart from studies on computational complexity, not much work has been done on measuring the complexity of the underlying problem [7]. To the best of our knowledge the complexity of software specifications or requirements has not been adequately measured. The aim of this paper is to present measures for the complexity of the underlying problem such as captured in a conceptual model of an application domain. Current methods for conceptual modelling offer a bundle of specification techniques to describe different views, i.e., static, dynamic and interaction views, on the same business reality. Few methods include a formal procedure for checking the consistency and correctness of these complementary views [15]. The approach to conceptual modelling taken here is the M.E.R.O.DE. process algebra [5], [15]. It is an object-oriented specification technique that guarantees model consistency and correctness. Since syntax and semantics of the technique have been defined, it is particularly suited to be supported by CASE-tools. Moreover, its formal definition allows to rigorously define specification measures. The M.E.R.O.DE. process algebra is briefly presented in section 2.

In section 3 the complexity measure definition approach is presented. According to Measurement Theory, measurement cannot proceed unless there is a clear understanding of the attribute [6], [13]. Although in general, software attributes such as complexity are badly understood [20], the approach presented here systematically defines the 'complexity of a conceptual model' using more elementary concepts having definitions that are universally agreed upon. Our definition of complexity also allows to distinguish this concept from other attributes of specifications such as length and structure. While this section presents a particular point of view on complexity, care has been taken in section 4 to define valid measures. If measures are proven to be valid, then the acceptance or rejection of a measure only depends on the viewpoint of the attribute.

It must be stressed right from the beginning that the goal of this research is to define and measure the complexity of the problems underlying software development, but not to demonstrate empirical relationships between this attribute and other attributes such as software quality or development effort. The software measure definition problem does not only precede empirical software engineering research. It is of crucial importance for the success of these research programs. Therefore it is believed that the problem of software measurement is interesting enough to be investigated on its own. Accordingly in section 5 our approach is evaluated mainly in terms of scientific validity (i.e., do the measures measure what they are supposed to measure) and completeness (i.e., which aspects of the complexity of the underlying problem have been measured). The usefulness of the measurements (i.e., the significance of attribute relationships [8], the construction of prediction models [8], etc.) is not assessed here and is left as an open question for further research.

2. Conceptual modelling with business objects

Conceptual modelling refers to the identification of the elements of an application domain [16]. Two relevant types of elements are business object types and event types. Business object types have occurrences (i.e., business objects) that participate in events that are atomic, have no duration and can be observed in the application domain. Events are occurrences of event types. The following definitions of (business) object types, event types and conceptual models are taken from [5], [15] and [16]. The example is taken from [15].

Let A be the universe of event types associated with the application domain that is our universe of discourse. The power set of A is P(A). The alphabet of an object type is the set of event types participated in. An object type participates in an event type if occurrences of the object type participate in occurrences of the event type. For every object type in the conceptual model with alphabet α, it holds that $\alpha \in P(A)$.

A set of regular expressions over A can be built by the operators '.' (sequence), '+' (selection) and '*' (iteration). The set of regular expressions over A is R*(A). The sequence constraints of object types on participation in event types are defined by a regular expression over A. For every object type in the conceptual model with regular expression e, it holds that $e \in R^*(A)$.

Basically, object types are defined as tuples $<\alpha,e> \in <P(A), R^*(A)>$ such that e is not in deadlock and every event type in α occurs at least once as an operand in e. Also, every operand in e is an event type in α.

To select the alphabet and regular expression of an object type, the selector functions S_A and S_R are defined:

S_A: $<P(A), R^*(A)> \rightarrow P(A): P \rightarrow \alpha$

S_R: $<P(A), R^*(A)> \rightarrow R^*(A): P \rightarrow e$

It is further required that for each object type it must be possible to create an occurrence and end the life of an occurrence. Hence, for the object type P, the alphabet S_AP is partitioned into c(P), m(P) and e(P) where

$c(P) = \{a \in A \mid a \text{ creates an occurrence of type } P\}$

$m(P) = \{a \in A \mid a \text{ modifies an occurrence of type } P\}$

$e(P) = \{a \in A \mid a \text{ ends the life of an occurrence of type } P\}$

and c(P) and e(P) may not be empty.

Based on these three subsets the default sequence constraints[1] are given by $\Sigma c(P) . (\Sigma m(P))^* . \Sigma e(P)$. This default describes the trivial life cycle of an object type. The actual sequence constraints of an object type cannot be less deterministic than the trivial life cycle.

The object types in a conceptual model are related. The classification schema used in M.E.R.O.DE. is the existence dependency relation. Object type P is existent dependent of object type Q if the life of each occurrence p of type P is embedded in the life of one single and always the same occurrence q of type Q. The object p is the marsupial object. The object q is the mother object.

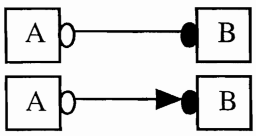
According to the M.E.R.O.DE. process algebra, if $P \leftarrow Q$ model consistency can be guaranteed by applying the following rules:

- Propagation rule: $S_A P \subseteq S_A Q$
  A marsupial cannot participate in an event without the mother having knowledge of this event.
- Type of involvement rule: $c(P) \subseteq c(Q) \cup m(Q)$ and $m(P) \subseteq m(Q)$ and $e(P) \subseteq m(Q) \cup e(Q)$
  A marsupial cannot be created before its mother exists nor can it exist after the life of its mother has ended.
- Restriction rule: $S_R P$ may not be less deterministic than $S_R Q$
  Any sequence of events in which a marsupial participates that is not acceptable from the point of view of the mother, must be rejected.

Let A be the universe of event types. A conceptual model is basically a set of object types $M \subseteq \langle P(A), R^*(A) \rangle$ on which an existence dependency relation is defined.

Example

A conceptual model for a hotel administration is presented. Fig. 1 is the existence dependency graph. For the cardinalities of the existence dependency relationships the Bachman notation is used.



An A is associated with zero or one existent dependent B's

An A is associated with zero, one or many existent dependent B's

Fig. 2 is the Object Event Table. It shows the alphabets of the object types and their partitioning into create event types (C), modify event types (M) and end event types (E).

The regular expressions of the object types are specified as follows:

CUSTOMER = create-customer . (reserve + confirm + cancel + no-show + first-check-in + next-check-in + invoice + dun + pay)* . file-customer
ROOMTYPE = create-room . (reserve + confirm + cancel + no-show + first-check-in + next-check-in + assign-roomtype)* . file-roomtype
RESERVATION = reserve . (cancel + confirm . (no-show + first-check-in + next-check-in))
ROOM = create-room . (first-check-in + next-check-in + consume + put-on-bill + invoice + dun + pay + assign-roomtype)* . file-room

[1] The symbol Σ must be read as an exclusive and exhaustive selection. For instance, $\Sigma c(P)$ means that object occurrences are created by one and only one create event belonging to an event type in c(P) [15, p. 69].

GUEST = first-check-in . (next-check-in + consume + put-on-bill + invoice + dun + pay)* . file-guest
STAY = (first-check-in + next-check-in) . (consume + put-on-bill)* . invoice . (dun)* . pay
CONSUMPTION = consume . put-on-bill
PAYMENT = invoice . (dun)* . pay

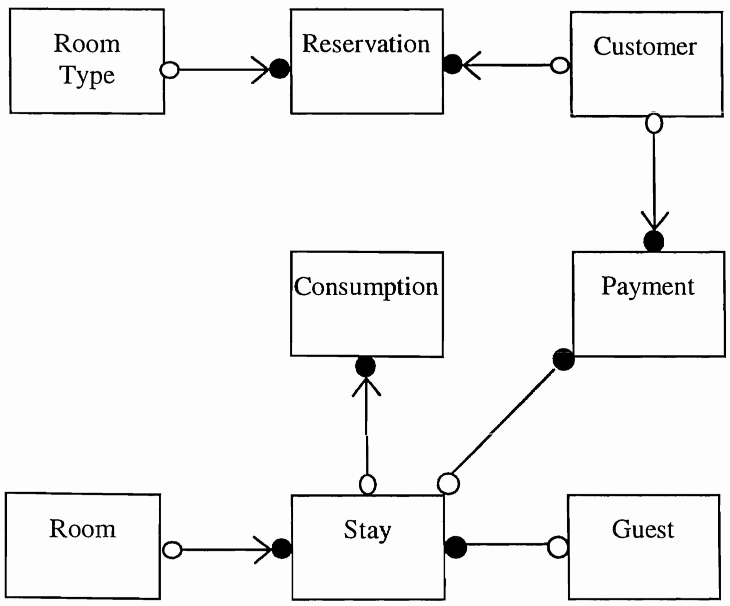


Figure 1: Existence Dependency Graph

	CUSTOM.	ROOMTP	RESERV.	ROOM	GUEST	STAY	CONSUM.	PAYM.
reserve	M	M	C					
confirm	M	M	M					
cancel	M	M	E					
no-show	M	M	E					
first-check-in	M	M	E	M	C	C		
next-check-in	M	M	E	M	M	C		
consume				M	M	M	C	
put-on-bill				M	M	M	E	
invoice	M			M	M	M		C
dun	M			M	M	M		M
pay	M			M	M	E		E
create-room				C				
file-room				E				
assign-roomtype		M		M				
create-roomtype		C						
file-roomtype		E						
create-customer	C							
file-customer	E							
file-guest					E			

Figure 2: Object Event Table

3. A systematic definition of the complexity of a conceptual model

Before the complexity of a conceptual model can be measured, it must be defined. It must be stressed that there does not exist a generally accepted definition of 'complexity' [7]. Therefore definitions of complexity are by their nature subjective. The aim of this section is to propose one such definition of complexity, in this case for the complexity of a conceptual model of an application domain. The definition is systematic in the sense that it captures the basic assumptions underlying our viewpoint of complexity.

Our specific definition of complexity is justified by three observations. *The first observation is that he complexity of a conceptual model must somehow be related to the complexity of its elements. The composing elements of a conceptual model are object types and existence dependency relationships between object types. According to [3] the complexity of a system only depends on the relationships between the elements of the system, while the elements themselves have no inherent complexity. On the one hand such a viewpoint is too simplistic and too abstract since object types are defined in terms of more atomic elements, i.e., event types (see previous section). On the other hand, it is clear that relationships between elements contribute to the complexity of the system. A definition that abstracts from the relationships in the model would qualify as a definition of the size of the system, but does not adequately capture its complexity. Therefore, the first observation leads to the following two assumptions:*

- ASSUMPTION 1. The complexity of a conceptual model is a function of the complexity of its object types;
- ASSUMPTION 2. The complexity of an object type is a function of the existence dependency relationships it participates in (i.e., both being existent dependent and having existent dependent object types).

The second observation is that whatever the definition of complexity is used, it must be possible to identify entities (object types or conceptual models) that are not complex, i.e., that have zero complexity. This is a crucial observation since it actually refers to the representational theory of measurement (e.g., [13]). If, prior to measurement, entities with zero complexity can be identified, then each entity can be classified as having either zero complexity or not zero complexity. So at least measurement in the sense of classification (i.e., measurement on a nominal scale) is possible. It may further be safely assumed that complexity is not negative. Hence, the second observation implies the assumptions that

- ASSUMPTION 3. The definition of complexity must allow the identification of object types and conceptual models with zero complexity;
- ASSUMPTION 4. When the complexity of an object type or conceptual model is not zero, then it is positive.

Assumptions 3 and 4 require a complexity definition that allows the classification of object types into a zero complexity class and a positive complexity class. The criteria for this classification express our viewpoint on complexity. Given assumption 2 these criteria depend on the position of the object type in the existence dependency graph. The same criteria can also be used to define a zero complexity object type for every position in the existence dependency graph.

The third observation pertains to a strategy for defining concepts. It is common to define concepts in terms of other concepts. For instance, Euclid defined geometrical figures in terms of elementary concepts such as point, line and plan. Since the concept of complexity in software engineering is badly understood [20] it is an acceptable strategy to define complexity in terms of concepts whose definitions are generally agreed upon. Given that for every position in the existence dependency graph a zero complexity object type can be *defined, complexity can be defined in* **terms** *of a simpler concept, i.e., difference which is mathematically equivalent to distance,* ***meaning*** *that difference is defined by the same postulates.*

Given these observations and assumptions **the** complexity of an object type is defined as ***the difference (i.e., distance) between its specifications and the specifications of the corresponding zero complexity object type.*** Object types correspond when they have the same position in the existence dependency graph.

This systematic definition of complexity must be further formalised. This is done next. Note that the definition of complexity as a distance implies a number of additional assumptions related to its representation and scale type. These assumptions are not discussed in detail here and must be investigated in subsequent research. Instead it is shown how assumptions 1 to 4 are applied in the definition process.

Assumption 3 necessitates a precise definition of an object type with zero complexity. Recall from the previous section that object types in a conceptual model are tuples $<\alpha,e> \in M \subseteq <P(A), R^*(A)>$ that satisfy a number of consistency and correctness constraints. Since these constraints must be satisfied for all object types in the model, it is our viewpoint that they do not contribute to the complexity of the object type. Merely satisfying the necessary constraints does not make the object type more complex than the other object types in the model. This conclusion leads to the following definition:

DEFINITION 1. Let A be the universe of event types, let $M \subseteq <P(A), R^*(A)>$ be a conceptual model. The object type $P = <\alpha,e> \in M$ has zero complexity if and only if

1. $\exists\, a \in A_{P,M} \cap c(P)$ and $\exists\, b \in A_{P,M} \cap e(P) \Rightarrow \alpha - A_{P,M} = \emptyset$
2. $\neg \exists\, a \in A_{P,M} \cap c(P)$ and $\exists\, b \in A_{P,M} \cap e(P) \Rightarrow \alpha - A_{P,M} = \{c_P\}$
3. $\exists\, a \in A_{P,M} \cap c(P)$ and $\neg \exists\, b \in A_{P,M} \cap e(P) \Rightarrow \alpha - A_{P,M} = \{e_P\}$
4. $\neg \exists\, a \in A_{P,M} \cap c(P)$ and $\neg \exists\, b \in A_{P,M} \cap e(P) \Rightarrow \alpha - A_{P,M} = \{c_P, e_P\}$
5. $e = \Sigma c(P)\ .\ (\Sigma m(P))^*\ .\ \Sigma e(P)$

where

$$A_{P,M} = \bigcup_{\substack{Q \leftarrow P \\ Q \in M}} S_A Q$$

is the set of event types propagated from the marsupial object types of P in M;
$c_P \in c(P)$ and $e_P \in e(P)$.

The alphabet of an object type with zero complexity is basically the union of alphabets of its marsupial object types. This satisfies the propagation rule. Only if this union does not contain an event type to create objects and/or to end the life of objects, then such (an) event type(s) may be added to the alphabet without making the object type complex. This rule does not contradict the type of involvement rule.

The regular expression of an object type with zero complexity is exactly the default life cycle on its alphabet. This guarantees that e is not more deterministic than the default life cycle. It also trivially satisfies the restriction rule.

Based on the criteria of definition 1 for every object type P in a conceptual model its corresponding zero complexity object type, hereafter denoted by min(P), can be defined. This is formalised in definition 2.

DEFINITION 2. Let A be the universe of event types, let $M \subseteq <P(A), R^*(A)>$ be a conceptual model and let $P = <\alpha,e> \in M$.
The object type $\min(P) = <\alpha',e'>$ is defined as
1. α' is partitioned into c(min(P)), m(min(P)) and e(min(P)) such that

a) $c(\min(P)) \subseteq c(P)$

b) $m(\min(P)) \subseteq m(P)$

c) $e(\min(P)) \subseteq e(P)$

d) $A_{\min(P),M} \subseteq \alpha'$

e) $\exists\, a \in A_{\min(P),M} \cap c(P)$ and $\exists\, b \in A_{\min(P),M} \cap e(P) \Rightarrow \alpha' - A_{\min(P),M} = \varnothing$

f) $\neg \exists\, a \in A_{\min(P),M} \cap c(P)$ and $\exists\, b \in A_{\min(P),M} \cap e(P) \Rightarrow \alpha' - A_{\min(P),M} = \{c_P\}$

g) $\exists\, a \in A_{\min(P),M} \cap c(P)$ and $\neg \exists\, b \in A_{\min(P),M} \cap e(P) \Rightarrow \alpha' - A_{\min(P),M} = \{e_P\}$

h) $\neg \exists\, a \in A_{\min(P),M} \cap c(P)$ and $\neg \exists\, b \in A_{\min(P),M} \cap e(P) \Rightarrow \alpha' - A_{\min(P),M} = \{c_P, e_P\}$

2. $e' = \Sigma c(\min(P)) . (\Sigma m(\min(P)))^* . \Sigma e(\min(P))$
where

$$A_{\min(P),M} = \bigcup_{\substack{Q \leftarrow P \\ Q \in M}} S_A \min(Q)$$

is the set of event types propagated from the marsupial object types of P in M;
$c_P \in c(P)$ and $e_P \in e(P)$.

This definition is consistent with the first definition. The main difference is that only those event types from the marsupial object types Q that belong to the alphabets of the min(Q) object types are propagated into α'. The definition is recursive in the sense that min(P) can only be defined if for all object types Q existent dependent of P, the corresponding zero complexity object types min(Q) are defined. As a consequence, when the zero complexity object types corresponding to the top[2] object types in the existence dependency graph are defined, then for all object types in the conceptual model the corresponding zero complexity object types are derived. Note that whenever it holds that an object type Q is existent dependent of an object type P, then min(Q) is existent dependent of min(P). Therefore, the set of zero complexity object types corresponding to the object types of a conceptual model M is also a valid conceptual model (hereafter denoted by min(M)). The model min(M) is the conceptual model with zero complexity that corresponds to M.

Example

A model min(HOTEL) corresponding to the model HOTEL is defined. The alphabets of the zero complexity object types are shown in fig. 3. If an entry in a cell is in bold then the event type in the row header belongs to the alphabet of the zero complexity object type that corresponds to the object type in the column header. The regular expressions of the zero complexity object types are easily derived from this table by defining a default life cycle on the alphabets.

Comments

- Consumption is the only zero complexity object type in the Hotel model. It satisfies all criteria of definition 1.
- The modify event type dun does not belong to the alphabet of min(Payment).
- The modify event type dun does not belong to the alphabet of min(Stay) since it does not belong to the alphabets of min(Consumption) and min(Payment). Since these alphabets contain no create event type for Stay, a choice has been made between first-check-in and next-check-in. Only one of these may be included in min(Stay).

[2] A top object types is not existent dependent of any other object type in the conceptual model [15, p. 95]

- The event types next-check-in, dun and assign-roomtype may not be included in the alphabet of min(Room).
- The event types next-check-in and dun are not contained in the alphabet of min(Guest).
- The event type confirm is a modify event type, not contained in a marsupial of Reservation. A choice has been made between the four end event types.
- A number of modify event types are not included in the alphabet of min(Roomtype) since they do not belong to the alphabet of min(Reservation).
- A number of modify event types are not included in the alphabet of min(Customer) since they do not belong to the alphabet of min(Reservation) and min(Payment).

	CUSTOM.	ROOMTP	RESERV.	ROOM	GUEST	STAY	CONSUM.	PAYM.
reserve	M	M	C					
confirm	M	M	M					
cancel	M	M	E					
no-show	M	M	E					
first-check-in	M	M	E	M	C	C		
next-check-in	M	M	E	M	M	C		
consume				M	M	M	C	
put-on-bill				M	M	M	E	
invoice	M			M	M	M		C
dun	M			M	M	M		M
pay	M			M	M	E		E
create-room				C				
file-room				E				
assign-roomtype		M		M				
create-roomtype		C						
file-roomtype		E						
create-customer	C							
file-customer	E							
file-guest					E			

Figure 3: Object Event Table for zero complexity model

Now the complexity of an object type and the complexity of a conceptual model can formally be defined.

DEFINITION 3. Let A be the universe of event types and let $M \subseteq$ <P(A), R*(A)> be a conceptual model.
a. The complexity of an object type $P = \langle\alpha,e\rangle \in M$ is the difference between $\langle\alpha,e\rangle$ and $\langle\alpha',e'\rangle$, where $\langle\alpha',e'\rangle$ is the object type min(P).
b. The complexity of M is the difference between M and min(M).

The concept of difference in definition 3 is mathematically equivalent to the concept of distance. Both are defined by the same postulates. As complexity is redefined in terms of distance, it cannot be negative. Hence, **assumption 4** is satisfied. The definition of the zero complexity object types in function of the existence dependency relationships their corresponding object types participate in, is in accordance with **assumption 2**.

The final decision to be made prior to measure definition concerns the modelling of the difference between object types and between conceptual schemes.

Let $P = \langle\alpha,e\rangle$ and $\min(P) = \langle\alpha',e'\rangle$. Since α and α' are sets, their difference can be modelled by their respective set differences. Moreover, since $\alpha' \subseteq \alpha$ the difference between α and α' is modelled by the set difference $\alpha - \alpha'$. To model the difference between the regular expressions e and e' an approach is taken similar to the solution to the tree-editing problem [11], [19]. First, a set of elementary transformations is defined. Each elementary transformation is an editing operation on regular expressions.

DEFINITION 4. Let A be the universe of event types. For e, e' $\in$ R*(A), x $\in$ A:
$t_i(e) = e'$
where $t_i(e)$ for subscript i = 0, 1, 2, ..., 9 is defined as:

$t_0(e) = e . x = e'$	(add right sequence event type)
$t_1(e) = x . e = e'$	(add left sequence event type)
$t_2(e) = e + x = e'$	(add right selection event type)
$t_3(e) = x + e = e'$	(add left selection event type)
$t_4(e) = (e)^* = e'$	(add iteration)
$t_5(e) = t_5(e' . x) = e'$	(delete right sequence event type)
$t_6(e) = t_6(x . e') = e'$	(delete left sequence event type)
$t_7(e) = t_7(e' + x) = e'$	(delete right selection event type)
$t_8(e) = t_8(x + e') = e'$	(delete left selection event type)
$t_9(e) = t_9((e')^*) = e'$	(delete iteration)

Given a regular expression e over A, all elementary transformations t_i may be applied to e or to any part of e that is a regular expression over A.
For e, e', e" $\in$ R*(A):
(i) $e = e' . e'' \Rightarrow t_i(e) = t_i(e' . e'')$ **or** $t_i(e') . e''$ **or** $e' . t_i(e'')$
(ii) $e = e' + e'' \Rightarrow t_i(e) = t_i(e' + e'')$ **or** $t_i(e') + e''$ **or** $e' + t_i(e'')$
(iii) $e = e'^* \Rightarrow t_i(e) = t_i((e')^*)$ **or** $(t_i(e'))^*$

Next it must be shown that a finite sequence of elementary transformations t_i can take every regular expression e $\in$ R*(A) to every other regular expression e' $\in$ R*(A). A proof can be found in [12]. The difference between e and e' is modelled as the **shortest T-derivation** from e to e' [19].

DEFINITION 5. Let T be a sequence of $t_{i1}, \ldots, t_{ik}$ elementary transformations.
A **T-derivation** from e $\in$ R*(A) to e' $\in$ R*(A) is a sequence of regular expressions $e_0, \ldots, e_k$ such that $e = e_0$, $e' = e_k$, and $t_{ij}(e_{j-1}) = e_j$ for $1 \leq j \leq k$. The **length of a T-derivation** is the number of transformations in T.

The differences between M and min(M) are modelled through the differences between their corresponding object types. This is in accordance with **assumption 1**.

Example

Let us illustrate definitions 4 and 5. Suppose we wish to model the difference between S_RSTAY = (first-check-in + next-check-in) . (consume + put-on-bill)* . invoice . (dun)* . pay and
$S_R$min(STAY) = first-check-in . (invoice + consume + put-on-bill)* . pay

A T-derivation from S_RSTAY to $S_R$min(STAY) is shown in fig. 4. Its length is 5.

T-derivation from S_RSTAY to $S_R$min(STAY) Regular Expression	Transformation used on previous regular expression
(first-check-in + next-check-in) . (consume + put-on-bill)* . invoice . (dun)* . pay	
first-check-in . (consume + put-on-bill)* . invoice . (dun)* . pay	t_7
first-check-in . (consume + put-on-bill)* . invoice . dun . pay	t_9
first-check-in . (consume + put-on-bill)* . invoice . pay	t_5
first-check-in . (consume + put-on-bill)* . pay	t_5
first-check-in . (invoice + consume + put-on-bill)* . pay	t_3

Figure 4: T-derivation from S_RSTAY to $S_R$min(STAY)

4. Complexity measures

Definition 3 defines the complexity of an object type P = <α,e> as the difference between <α,e> and <α',e'> = min(P), modelled as the set difference α - α' and the shortest T-derivation from e to e'. Such as mentioned in the previous section, this definition is based on a number of assumptions regarding representation and scale type that are not further discussed here. But even without a detailed discussion it is clear that these definitions require the complexity measures to be metrics in the sense of Measure Theory.

The set difference α - α' is a special case of the symmetric difference model that defines a metric distance between sets [17]. The symmetric difference between sets A and B (notation $A \Delta B$) is equal to $(A - B) \cup (B - A)$. It can be shown that for an additive function φ, $\delta(A,B) = \varphi(A - B) + \varphi(B - A)$ is a metric [17]. The most obvious function φ is the cardinality function [12]. Hence, if A and B are sets, then $\delta(A,B) = |A - B| + |B - A|$ is a metric. Note that if $B \subseteq A$ then $B - A = \emptyset$ and $\delta(A,B) = |A - B|$.

It is also proven that the length of the shortest T-derivation from $e \in R^*(A)$ to $e' \in R^*(A)$ is a metric. For a formal proof see appendix 2 in [12].

DEFINITION 6. Let A be the universe of event types and let $M \subseteq \langle P(A), R^*(A) \rangle$ be a conceptual model. The complexity of an object type $P = \langle \alpha,e \rangle \in M$ is measured by $\delta(P,\min(P)) = (\delta_{alph}(P,\min(P)), \delta_{seq}(P,\min(P)) = (|\alpha - \alpha'|$, length of the shortest T-derivation from e to e')

The complexity of a conceptual model is a function of the complexity of its object types. The model of the difference between conceptual models M and min(M) contains all set differences between the alphabets of the corresponding object types P and min(P), as well as all shortest T-derivations from the regular expressions of the P object types to the regular expressions of the corresponding min(P) object types. The difference between M and min(M) is defined as a distance if it is the sum of the distances from every object type P in M to its corresponding object type min(P) in min(M). Given this equality, a complexity measure for conceptual models can be defined as follows:

DEFINITION 7. Let A be the universe of event types and let $M \subseteq \langle P(A), R^*(A) \rangle$ be a conceptual model. The complexity of M is measured by

$$\sigma(M) = \sum_{P \in M} \delta(P,\min(P))$$

Example

The complexity measurements of the Hotel model are presented in fig. 5. The δ_{alph} values can easily be calculated using fig. 3. In each column the number of entries that are not in bold must be counted. Examples of shortest T-derivations for all object types in the model can be found in the appendix. The length of these T-derivations are the values of δ_{seq}.

Object type	δ_{alph}	δ_{seq}
Payment	1	**2**
Consumption	0	0
Stay	2	**5**
Room	3	3
Guest	2	2
Reservation	4	4
Roomtype	5	5
Customer	5	5
Model		
σ(Hotel) =	22	26

Figure 5: Complexity measurements

Note that the δ_{alph} values are lower bounds on the δ_{seq} values. However, the δ_{alph} values do not fully capture the complexity of object types. For instance, the object type Stay is complex because of the dun and next-check-in event types. But, it is also complex because of a number of sequence constraints imposed on its life cycle. If after the removal of the event types in S_AP - $S_A$min(P) from S_RP, the life cycle is trivial, then δ_{alph} and δ_{seq} values are equal. Otherwise, the δ_{seq} values capture additional complexity. If only one measure of complexity is needed, then we would choose δ_{seq}.

5. Evaluation

The first issue to evaluate is the validity of the complexity measures. From a Measure Theory point of view validity is guaranteed. The cardinality of the symmetric difference and the length of the shortest T-derivation are metrics on P(A) and R*(A) respectively. Since complexities are distances in P(A) and R*(A), the metrics can be used as complexity measures.

However, measure validity must also be evaluated from the viewpoint of Measurement Theory. This requires a detailed analysis of the representation, uniqueness and meaningfulness problems of measurement [6], [13]. Although these issues need to be addressed in the future, the results of this paper already allow a limited form of measurement theoretic validation.

The definitions of complexity allow to decide whether entities (object types and conceptual models) are complex or not. Recall that the postulates of distance dictate that, for all entities A and B:

- $A = B \Rightarrow$ the distance from A to B is zero;
- $A \neq B \Rightarrow$ the distance from A to B is positive.

Therefore, whenever P = min(P) and whenever M = min(M) the complexities of P and M must be zero. These requirements are satisfied as according to definitions 6 and 7 it holds that $\delta(P,\min(P)) = (0,0)$ and $\sigma(M) = (0,0)$.

On the other hand, whenever $P \neq min(P)$ and whenever $M \neq min(M)$ the complexities of P and M must be positive. In fact, if $P \neq min(P)$ then $\alpha - \alpha' \neq \emptyset$ and the length of the shortest T-derivation from e to e' is equal to or greater than 1 (i.e., at least one transformation is needed). Therefore, according to definition 6 it holds that $\delta_{alph}(P,min(P)) \geq 1$ and $\delta_{seq}(P,min(P)) \geq 1$. As a consequence $\sigma(M)$ is positive (cf. definition 7).

Now, define an empirical ordering relation $\angle$ on $M \subseteq \langle P(A), R^*(A)\rangle$ as

$P \angle Q \Leftrightarrow$ the complexity of P is zero and the complexity of Q is positive

This relation implies that empirically the complexity of an object type P can only be judged lower than the complexity of an object type Q if and only if P has zero complexity and Q has some positive complexity.

Note that M is a countable set of object types [13]. Since $\angle$ is asymmetric (i.e., $P \angle Q \Rightarrow \neg Q \angle P$) and negatively transitive (i.e., $P \angle Q \Rightarrow \forall R \in M$: $P \angle R$ or $R \angle Q$) it imposes a strict weak order on M. According to Cantor's theorem [13] when $\angle$ is a strict weak order on the countable set M, then there exists a real-valued function f on M such that

$P \angle Q \Leftrightarrow f(P) < f(Q)$

Moreover, the representation $((M, \angle), (Re, <), f)$ is an ordinal scale.

Clearly, the complexity measure δ does not satisfy as the function f since it is not a homomorphism from $(M, \angle)$ into $(Re, <)$. It holds that $P \angle Q \Rightarrow \delta(P,min(P)) < \delta(Q,min(Q))$, but it does not hold that $\delta(P,min(P)) < \delta(Q,min(Q)) \Rightarrow P \angle Q$. Therefore, we believe that the definition of complexity as a distance allows more complex representations than the mapping of the simple empirical relation $\angle$. This needs to be investigated in the future.

Note that it is possible to find a homomorphic function f that is a metric at the same time. For all $\alpha, \alpha' \in P(A)$ let $\delta_1(\alpha,\alpha') = 0$ if $\alpha = \alpha'$ and let $\delta_1(\alpha,\alpha') = c_1 > 0$ if $\alpha \neq \alpha'$. For all e, e' $\in R^*(A)$ let $\delta_2(e, e') = 0$ if the shortest T-derivation from e to e' has length zero and let $\delta_2(e, e') = c_2 > 0$ if the shortest T-derivation from e to e' has length not equal to zero. Since δ_1 and δ_2 satisfy the metric axioms, they are metrics on P(A) and R*(A) respectively. It now holds that $\forall P = \langle\alpha_P,e_P\rangle, Q = \langle\alpha_Q,e_Q\rangle \in M$: $P \angle Q \Leftrightarrow \delta_1(\alpha_P,\alpha_P') < \delta_1(\alpha_Q,\alpha_Q')$ and $\delta_2(e_P,e_P') < \delta_2(e_Q,e_Q')$. This example shows that our definition of complexity allows at least ordinal measurement.

Apart from their validity, it must be evaluated whether the complexity measures can be described as measures of the 'complexity of the problems underlying software development'. In M.E.R.O.DE. the conceptual model is that crucial part of the specifications that describes the business model, showing the exact functioning of the business in terms of entities, constraints and rules [15]. According to Zachman, the business model is an integral part of the system requirements [18]. Hence, it is a problem statement. As such it captures 'problems underlying software development'.

However, not all problems are modelled. For instance, apart from the process algebra, the M.E.R.O.DE. method includes a number of techniques to specify other types of requirements. Examples are business rules other than sequence constraints, information requirements of the users of a system, technology constraints and performance demands. A conceptual model is by definition an abstraction of the problem domain. It highlights some features, while it purposely omits other features. Of course, if a conceptual model is used as a measurement model, then only those features can be measured that are included in the model. Currently these features are the alphabet and the sequence constraints of object types. All other aspects that are not captured in the conceptual model, but that do contribute to the 'complexity of the underlying problem' are not measured. Further research on measuring 'the complexity of underlying problem' must focus on other problem domain abstractions.

References

[1] Baker A.L., J.M. Bieman, N. Fenton, D.A. Gustafson, A. Melton and R. Whitty, 'A Philosophy for Software Measurement', *Journal of Systems and Software*, Vol. 12, 1990, pp. 277-281.

[2] Basili V.R., L.C. Briand and W.L. Melo, 'A Validation of Object-Oriented Design Metrics as Quality Indicators', *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, 1996, pp. 751-761.

[3] Briand L.C., S. Morasca and V.R. Basili, 'Property-Based Software Engineering Measurement', *IEEE Transactions on Software Engineering*, Vol. 22, No. 1, 1996, pp. 68-86.

[4] Brito e Abreu F. and W. Melo, 'Evaluating the Impact of Object-Oriented Design on Software Quality', *IEEE Third International Software Metrics Symposium,* Berlin, March 1996.

[5] Dedene G. and M. Snoeck, 'Formal deadlock elimination in an object oriented conceptual schema', *Data and Knowledge Engineering*, Vol. 15, No. 1, 1995, pp. 1-30.

[6] Ellis B., *Basic Concepts of Measurement*, Cambridge Academic Press, 1968.

[7] Fenton N.E. and S.L. Pfleeger, *Software Metrics, A Rigorous and Practical Approach*, 2nd edition, International Thomson Computer Press, 1997.

[8] Kitchenham B., S.L. Pfleeger and N. Fenton, 'Towards a Framework for Software Measurement Validation', *IEEE Transactions on Software Engineering*, Vol. 21, No. 12, 1995, pp. 929-944.

[9] Lanubile F. and G. Visaggio, 'Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned', *Journal of Systems and Software*, Vol. 38, 1997, pp. 225-234.

[10] Li W. and S. Henry, 'Object-Oriented Metrics that Predict Maintainability', *Journal of Systems and Software*, Vol. 23, 1993, pp. 111-122.

[11] Oommen B.J., K. Zhang and W. Lee, 'Numerical Similarity and Dissimilarity Measures Between Two Trees', *IEEE Transactions on Computers*, Vol. 45, No. 12, 1996, pp. 1426-1434.

[12] Poels G. and G. Dedene, 'Formal Software Measurement for Object-Oriented Business Models', *DTEW research paper 9623*, Katholieke Universiteit Leuven, 1996.

[13] Roberts F.S., *Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences*, Addison-Wesley, 1979.

[14] Schneidewind N.F., 'Validating Metrics for Ensuring Space Shuttle Flight Software Quality', *IEEE Computer*, Vol. 27, No. 8, 1994, pp. 50-57.

[15] Snoeck M., *On a Process Algebra Approach for the Construction and Analysis of M.E.R.O.DE.-Based Conceptual Models*, Phd dissertation, department of computer science, Katholieke Universiteit Leuven, 1995.

[16] Snoeck M., 'Existence Dependency: Conceptual modelling by contract', *DTEW research paper 9640*, Katholieke Universiteit Leuven, 1996.

[17] Suppes P., D.H. Krantz, R.D. Luce and A. Tversky, *Foundations of Measurement. Volume II: Geometrical, Threshold and Probabilistic Representations*, Academic Press, Inc., 1989.

[18] Zachman J.A., 'A Framework for Information Architecture', *IBM Systems Journal*, Vol. 26, No. 3, 1987, pp. 276-292.
[19] Zhang K. and D. Shasha, 'Simple fast algorithms for the editing distance between trees and related problems', *Siam Journal on Computing*, Vol. 18, No. 6, 1989, pp. 1245-1262.
[20] Zuse H. and P. Bollmann, 'Software Metrics. Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics', *ACM Sigplan Notices*, Vol. 24, No. 8, 1989, pp. 23-33.

Appendix

T-derivation from S_RPAYMENT to $S_R$min(PAYMENT) **Regular Expression**	**Transformation used on previous regular expression**
invoice . (dun)* . pay	
invoice . dun . pay	t_9
invoice . pay	t_5

T-derivation from S_RCONSUMPTION to $S_R$min(CONSUMPT.) **Regular Expression**	**Transformation used on previous regular expression**
consume . put-on-bill	

T-derivation from S_RSTAY to $S_R$min(STAY) **Regular Expression**	**Transformation used on previous regular expression**
(first-check-in + next-check-in) . (consume + put-on-bill)* . invoice . (dun)* . pay	
first-check-in . (consume + put-on-bill)* . invoice . (dun)* . pay	t_7
first-check-in . (consume + put-on-bill)* . invoice . dun . pay	t_9
first-check-in . (consume + put-on-bill)* . invoice . pay	t_5
first-check-in . (consume + put-on-bill)* . pay	t_5
first-check-in . (invoice + consume + put-on-bill)* . pay	t_3

T-derivation from S_RROOM to $S_R$min(ROOM) **Regular Expression**	**Transformation used on previous regular expression**
create-room . (first-check-in + next-check-in + consume + put-on-bill + invoice + dun + pay + assign-roomtype)* . file-room	
create-room . (first-check-in + consume + put-on-bill + invoice + dun + pay + assign-roomtype)* . file-room	t_7
create-room . (first-check-in + consume + put-on-bill + invoice + pay + assign-roomtype)* . file-room	t_7
create-room . (first-check-in + consume + put-on-bill + invoice + pay)* . file-room	t_7

T-derivation from S_RGUEST to $S_R$min(GUEST) Regular Expression	Transformation used on previous regular expression
first-check-in . (next-check-in + consume + put-on-bill + invoice + dun + pay)* . file-guest	
first-check-in . (consume + put-on-bill + invoice + dun + pay)* . file-guest	t_8
first-check-in . (consume + put-on-bill + invoice + pay)* . file-guest	t_7

T-derivation from S_RRESERVATION to $S_R$min(RESERVAT.) Regular Expression	Transformation used on previous regular expression
reserve . (cancel + confirm . (no-show + first-check-in + next-check-in))	
reserve . (cancel + confirm . (no-show + first-check-in))	t_7
reserve . (cancel + confirm . first-check-in)	t_8
reserve . (cancel + first-check-in)	t_6
reserve . first-check-in	t_8

T-derivation from S_RROOMTYPE to $S_R$min(ROOMTYPE) Regular Expression	Transformation used on previous regular expression
create-room . (reserve + confirm + cancel + no-show + first-check-in + next-check-in + assign-roomtype)* . file-roomtype	
create-room . (reserve + cancel + no-show + first-check-in + next-check-in + assign-roomtype)* . file-roomtype	t_7
create-room . (reserve + no-show + first-check-in + next-check-in + assign-roomtype)* . file-roomtype	t_7
create-room . (reserve + first-check-in + next-check-in + assign-roomtype)* . file-roomtype	t_7
create-room . (reserve + first-check-in + assign-roomtype)* . file-roomtype	t_7
create-room . (reserve + first-check-in)* . file-roomtype	t_7

T-derivation from S_RCUSTOMER to $S_R$min(CUSTOMER) Regular Expression	Transformation used on previous regular expression
create-customer . (reserve + confirm + cancel + no-show + first-check-in + next-check-in + invoice + dun + pay)* . file-customer	
create-customer . (reserve + cancel + no-show + first-check-in + next-check-in + invoice + dun + pay)* . file-customer	t_7
create-customer . (reserve + no-show + first-check-in + next-check-in + invoice + dun + pay)* . file-customer	t_7
create-customer . (reserve + first-check-in + next-check-in + invoice + dun + pay)* . file-customer	t_7
create-customer . (reserve + first-check-in + invoice + dun + pay)* . file-customer	t_7
create-customer . (reserve + first-check-in + invoice + pay)* . file-customer	t_7