Available online at www.sciencedirect.com

SCIENCE DIRECT®

The Journal of Systems and Software 79 (2006) 1341–1358

The Journal of Systems and Software

www.elsevier.com/locate/jss

A formal representation of functional size measurement methods

Marjan Heričko *, Ivan Rozman, Aleš Živkovič

Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, SI-2000 Maribor, Slovenia

Received 11 April 2005; accepted 5 November 2005
Available online 10 January 2006

Abstract

Estimating software size is a difficult task that requires a methodological approach. Many different methods that exist today use distinct abstractions to depict a software system. The gap between abstractions becomes even greater with object-oriented artifacts developed in unified modeling language (UML). In this paper, a formal foundation for the representation of functional size measurement (FSM) methods is presented. The generalized abstraction of the software system (GASS) is then used to formalize different functional measurement methods, namely the FPA, MK II FPA and COSMIC-FFP. The same model is also used for object-oriented projects where UML artifacts are mapped into the GASS form. The algorithms in symbolic code for those UML diagrams that are crucial for size estimation are also given. The mappings defined in this paper enable diverse FSM methods to be supported in estimation tools, the automation of counting steps and a higher-level of independence from the FSM method, since the software abstraction is written in a generalized form. Both improvements are crucial for the practical use of FSM methods.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Function points; Software size; Formal model; Object-oriented projects

1. Introduction

The FPA method (Albrecht, 1979; IFPUG, 2004) defines a software-size estimation procedure in a descriptive way that lacks a formal foundation (Fetcke, 1999a). Different methods that have updated the original version seem to ignore this very important issue. Consequently, the comparison of different methods (Jeffery et al., 1993) in an analytical way is impossible, as is determining the level of improvement introduced with any new method. The automation of the measurement procedure is likewise impossible. A formal mathematical model could help us overcome these problems. To define a universal mathematical model two abstractions are needed:

1. A universal data model that describes a software system in a form appropriate for size estimation.
2. A function that maps universal data elements into a numerical value.

Fetcke (1999a) defined a universal data model that enables the formal representation of any FSM method. Using this mathematical model, we have introduced a new model, supplemented with a formalized functions that calculate system size. The model is expressed with mathematical formulas and also written in UML (OMG, 2001) with developed data models for the FPA (IFPUG, 2004), MK II FPA (UKSMA, 1998) and COSMIC-FPP (COSMIC, 2003) methods. Further on, a uniform transformation of the universal data model into models of these methods will also be defined. The universal data model will then be supplemented with a function that maps elements of the data abstraction into a software size.

* Corresponding author. Tel.: +386 2 235 5112; fax: +386 2 235 5134.
E-mail address: marjan.hericko@uni-mb.si (M. Heričko).

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.11.568

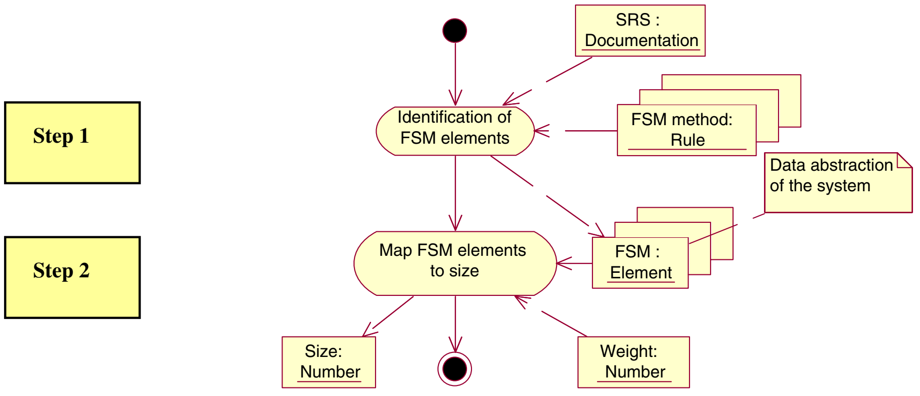


Fig. 1. The FSM abstraction steps.

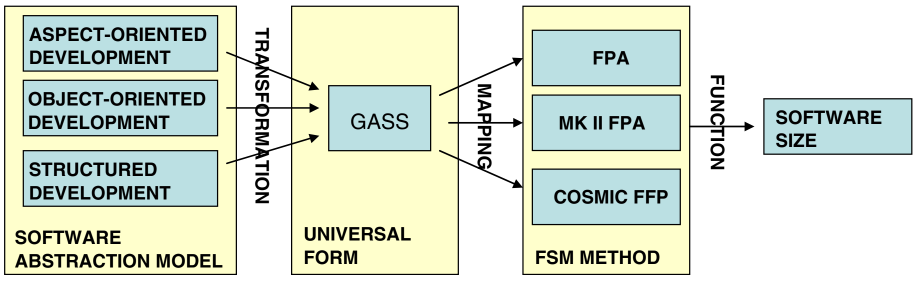


Fig. 2. The role of the GASS model in the size estimation process.

According to measurement theory, every measurement can be represented as a function that maps an empirical object into a numerical one (Lorenz and Kidd, 1994). The FSM method defines a function that maps the characteristics of a software system into a number (ISO, 1998, 2002a,b, 2003). That number represents the functional size of the software system in question. Since the FSM method tends to be independent of technology and its corresponding documentation, each FSM method introduces its own abstraction of the software system (Živkovič et al., 2003). Abstraction is usually data-oriented and takes two steps to transform elements of the software system into its size. The procedure is presented in Fig. 1.

The FSM elements are extracted from the documentation (for example, SRS: software requirements specification) using a set of rules. The rules, defined by most FSM methods, are given in a textual form. The result of the first step is the data-oriented abstraction of the software system to be built, which then serves as input for the second step in the estimation procedure, where element weights defined in a specific FSM method are also used in order to calculate software size. The element weight is a number defined by the FSM method that converts an FSM element into a software size. One FSM method may use several weights and transformation rules.

Fig. 2 shows the main idea of this paper. The formalization of the estimation procedures and the universal form for software abstraction (GASS) play a central role in the estimation process. A model of the software system to be built is first transformed into the GASS form, which is then mapped to the FSM method of one's choice in order to calculate software size.

This paper is divided into six sections. In the next section, a detailed review of the literature is given. In Section 3, the generalized abstraction of the software system (GASS) is introduced and then applied on several FSM methods in the fourth section. The mapping of the UML diagrams (OMG, 2001) into the GASS form is presented in Section 4 as well. Section 5 gives some instructions for the practical application of the GASS form, as defined in this research. The advantages are also discussed. The last section summarizes the results and improvements that were introduced in the field of size estimation and also discusses some directions for future work.

2. Related work

All FSM methods for estimating software size lack adequate formal foundations in their original descriptions. There were some attempts (Fetcke, 1999a; Diab et al., 2002) to add formalism to functional size measurement. Fetcke (1999a) realized that each FSM method measures software size using two steps. In the first step, software documentation is transformed to some abstract form defined by the FSM method, and in the second step this abstract form is used to calculate the software size using a function. Fetcke introduced data-oriented abstraction that is based on two main concepts—transactional types and data group types. Formalization is called generalized function point structure, where an application is represented as a vector that contains elements of both of the main concepts. Our research is based on the generalized function point structure; however, it introduces several improvements and novelties. The function that calculates software size from the abstract form used to represent a software system is formalized. The representation of different FSM methods with the generalized structure is mathematically expressed. The UML class diagrams are used to graphically show the differences between the FSM methods. Finally, a transformation between the UML diagrams and the FSM methods is given in a mathematical form together with algorithms in symbolic code.

The approach proposed by Diab et al. (2002) was expressly designed for COSMIC-FFP (COSMIC, 2003) using a formal modeling language named ROOM to formalize the COSMIC-FFP method. Although the ROOM language could be used in our approach, we decided to use a more universal, mathematical form and symbolic code that could be easily transformed to programming languages like Java, C++ and C#. In their future work section, Diab et al. (2002) expressed their intention of using UML diagrams as a source of information to identify COSMIC-FFP components. In our research, this idea was realized, with the additional feature of being applicable to any FSM method.

3. GASS model

3.1. Data-oriented abstraction

Different software-size estimation methods enumerated in the introduction use different names for data abstraction elements; the rules for element identification are different, and mapping functions also differ significantly. However, similarities exist that can be described by the following core concepts:

- The user concept covers the interaction between a user and the system. The user can be human or mechanical e.g. other systems.
- The application concept represents the whole system as an object of the measurement.
- The transaction concept is a logical representation of the system's functionality. Transaction is the smallest independent unit of interest.
- The data concept deals with the subject of change within the system. The data element is the smallest unit observed by a user.
- The type concept simplifies data handling via the abstraction of individual data elements.

On a higher-level of abstraction, an application is represented with data and transactional types. A data type is a set of data elements handled within the system. A transactional type is a sequence of logical activities. Fetcke defined seven classes of logical activities (Fetcke, 1999a):

- *Entry activity*. The user enters data into the application.
- *Exit activity*. Data is outputted to the user.
- *Control activity*. The user enters control information data.
- *Confirm activity*. Confirmation data is outputted to the user.
- *Read activity*. Data is read from a stored data group type.
- *Write activity*. Data is written to a stored data group type.
- *Calculate activity*. New data is calculated from existing data.

Fig. 3 shows a UML class diagram for data-oriented abstraction, which is later referred to as a GASS model. Based on the abstract presentation, the mapping for a specific method can be defined.

3.2. Abstraction of the software system

In this subsection, we summarize the formal representation of the concepts described in the previous section. The application closure H is defined as a vector of τ transactional types T and σ data group types F.

$$\mathrm{H} = (T_1, \ldots, T_\tau, F_1, \ldots, F_\sigma) \tag{E1}$$

The transactional type T_i is a vector of activities P:

$$T_i = (P_{i1}, \ldots, P_{in_i}) \tag{E2}$$

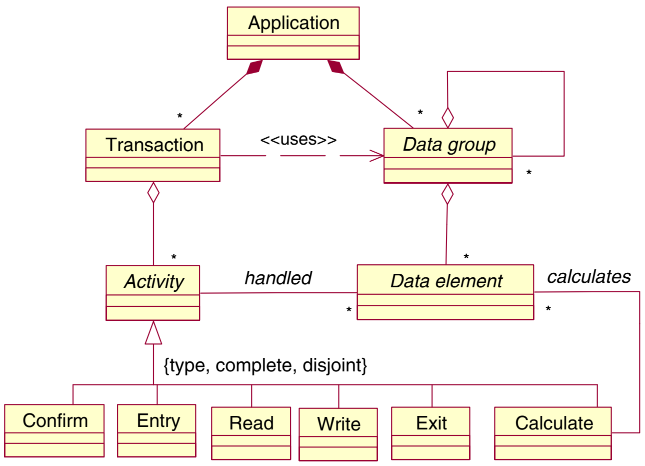


Fig. 3. Graphical representation of the GASS model.

An activity is further described by four attributes:

- its class $\theta_{ik} \in \{\text{Entry, Exit, Control, Confirm, Read, Write, Calculate}\}$,
- for read and write activities, the data group type referenced r_{ik},
- the set of data elements D_{ik} handled and
- for calculating activities, the set of data elements calculated C_{ik}.

In Eq. (E3), i can have a value from 1 to τ and represents the transaction activity it conforms to, while k runs from 1 to n, identifying activity within the transaction.

$$P_{ik} = (\Theta_{ik}, r_{ik}, D_{ik}, C_{ik}) \tag{E3}$$

The data group type F_j is a set:

$$F_j = \{(d_{j1}, g_{j1}), \ldots, (d_{jr_j}, g_{jr_j})\} \tag{E4}$$

where the d_{jk} are data elements and the g_{jk} the designate sub-groups. j can have a value from 1 to σ and represents the number of data types and k distinguishes between data elements and can have a value from 1 to r.

3.3. Representation of the mapping function

In the previous section, a formal representation of transactional and data types was introduced. Every software system is composed of different data and transactional types. The number of data and transactional types, and their attributes, contribute to the size of the software system. Some FSM methods also define the third component that has an influence on software size—the technical complexity of the solution. The universal function that maps application attributes into size is therefore:

$$\text{FPC}(a) = \left(\sum_i \text{FPC}_1(t_i) + \sum_j \text{FPC}_2(f_j)\right) * \text{FPC}_3(TC) \tag{E5}$$

where

FPC(a) is a function that maps attributes of the application a into the software size.
$\text{FPC}_1(t_i)$ is a function that maps transactional type t_i into size.
$\text{FPC}_2(f_j)$ is a function that maps data type f_j into size.
$\text{FPC}_3(TC)$ is the function that maps technical complexity of the anticipated solution for application a into a factor.

The total value for an application size is the sum of both parts multiplied by the factor of the solution's complexity. The factor can reduce or increase the overall size. However, it is not clear if the factor actually measures raw application size or is a characteristic of the implementation and should be a part of the function that maps size to effort (Lokan, 1999, 2000). For this reason, the function of FPC_3 is not examined in this research.

GASS can now be used to define different methods. First, we will use it to represent the original FPA method.

4. GASS model application

4.1. Mapping the FPA method

Fig. 4 shows the data model for the FPA method (Albrecht, 1979; IFPUG, 2004) conducted using the GASS model as defined in this paper (see Fig. 3). The data model shows that the FPA method introduces two types of data functions, namely internal logical files (ILF) and external interface files (ILF) and splits data elements into simple (DET) and complex (RET/FTR), while transactional functions use only a sub-set of activities defined in the GASS model. More than one activity from the general structure is mapped to a single FPA activity. The mapping is summarized in Table 1. In the table, activities that are allowed in the transaction type are marked with an ✓ sign.

Now the mapping can be used in a mathematical representation of the method. Data functions from the FPA method are equivalent to the data element type (F) while DET is equivalent to d and RET to g, as defined in the generalized representation. The FPA method distinguishes between internal and external data requirements; generalized representation, however, defines more activity types than the FPA method. Therefore, we define external interface files (EIFs) as a data type that cannot be used in write type activities.

$$
\begin{aligned}
&H = (T_1, \ldots, T_\tau, F_1, \ldots, F_\sigma)\\
&T_i = (P_{i1}, \ldots, P_{in})\\
&P_{ik} = (\Theta_{ik}, r_{ik}, D_{ik}, C_{ik})\\
&\Theta \in \{\mathrm{EI}, \mathrm{EO}, \mathrm{EQ}\}, \quad r_{ik} \in \{\mathrm{FTR}\},\ D_{ik} \in \{\mathrm{DET}\},\ C_{ik} \in \{\Phi\}\\
&F \in \{\mathrm{ILF}, \mathrm{EIF}\}\\
&F_j = \{(d_{j1}, g_{j1}), \ldots, (d_{jr}, g_{jr})\} = \{(\mathrm{DET}_{j1}, \mathrm{RET}_{j1}), \ldots, (\mathrm{DET}_{jr}, \mathrm{RET}_{jr})\}
\end{aligned}
$$

The FPC functions for the FPA method would look like this:

$$
\begin{aligned}
&\mathrm{FPC}_{1\mathrm{FPA}} = \sum_i W_T(D_i, r_i) = \sum_a W_{\mathrm{EI}}(N_{\mathrm{DET}}, N_{\mathrm{FTR}}) + \sum_b W_{\mathrm{EO}}(N_{\mathrm{DET}}, N_{\mathrm{FTR}}) + \sum_c W_{\mathrm{EQ}}(N_{\mathrm{DET}}, N_{\mathrm{FTR}})\\
&\mathrm{FPC}_{2\mathrm{FPA}} = \sum\nolimits_j W_F(d_j, g_j) = \sum_l W_{\mathrm{EIF}}(N_{\mathrm{DET}}, N_{\mathrm{RET}}) + \sum_m W_{\mathrm{EIF}}(N_{\mathrm{DET}}, N_{\mathrm{RET}})
\end{aligned} \tag{E6}
$$

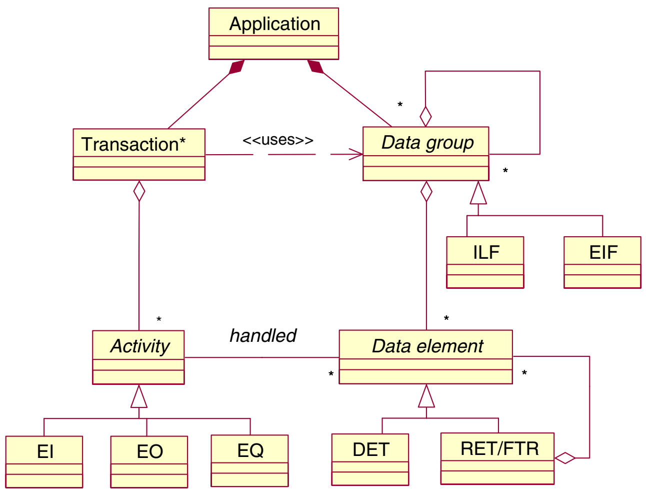


Fig. 4. The FPA data model.

Table 1
Mapping of transactions into activities

	GASS model					
	Entry	Exit	Write	Read	Confirm	Calculate
FPA						
EI	✓		✓	✓		
EQ						✓
EQ		✓			✓	

The W_T function prescribes the number of function points for a transactional function using the number of data elements and referenced elements. There are three types of function: W_{EI}, W_{EO}, W_{EQ}.

The W_T function prescribes the number of function points for the data function using the number of data elements and data groups.

The W_{ILF}, W_{EIF}, W_{EI}, W_{EO}, W_{EQ} functions define the weight in function points for each element identified in the measurement process. Function W has two parameters. For transactional functions, the parameters are the number of data element types (N_d) and number of file types referenced (N_r). For data functions, the parameter N_g is used instead of N_r, representing the number of record element types. W_x functions are the step functions represented by discrete values with the following range:

$$W_{\text{ILF}} = \{7, 10, 15\}$$
$$W_{\text{EIF}} = \{5, 7, 10\}$$
$$W_{\text{EI}} = W_{\text{EQ}} = \{3, 4, 6\}$$
$$W_{\text{EO}} = \{4, 5, 7\}$$

N_{DET}, N_{RET}, N_{FTR} are the number of elements according to the element type.

Given as an example, the function W_{ILF} is originally (IFPUG, 2004) defined as

$$W_{\text{ILF}}(N_d, N_g) = \begin{cases} & ((1 \leqslant N_d \leqslant 19) \wedge (1 \leqslant N_g \leqslant 5)) \vee \\ 7; & ((20 \leqslant N_d \leqslant 50) \wedge (N_g = 1)) \\ & ((1 \leqslant N_d \leqslant 19) \wedge (6 \leqslant N_g)) \vee \\ 10; & ((20 \leqslant N_d \leqslant 50) \wedge (2 \leqslant N_g \leqslant 5)) \vee \\ & ((51 \leqslant N_d) \wedge (N_g = 1)) \\ & ((1 \leqslant N_d \leqslant 19) \wedge (6 \leqslant N_g)) \vee \\ 15; & ((51 \leqslant N_d) \wedge (N_g \geqslant 2)) \end{cases} \tag{E7}$$

The mapping of the GASS form into the FPA elements and the algorithm that calculates software size according to the FPA method can be written in symbolic code. The symbolic code can then be easily transformed into a specific programming language that implements a software-size estimation tool.

Symbolic code for the FPA method (transactional types):
```
size := 0;
transactional_types := h.getTransactionalTypes();  //h is an instance of H - //aplication
for i = 1 to transactional_types.size() do
  t := transactional_types.getType(i);
  activities := t.getActivities();
  for ii = 1 to activities.size() do
    p := activities.getActivity(ii);
    numOfDET := p.getDataElements() .size ();  //Dik
    numOfFTR := p.getReferencedTypes().size();  //rik
    case p.getClass() of
      ENTRY, WRITE, READ: size := size + evaluate(numOfDET, numOffTR,EI);
      CALCULATE: size := size + evaluate(numOfDET, numOfFTR, EO);
      EXIT, CONFIRM: size := size + evaluate(numOfDET, numOfFTR, EQ);
    endcase;
  enddo
enddo
```

Symbolic code for the FPA method (data types):
```
size := 0;
data_types := h.getDataTypes(); //h is an instance of H - application
for i = 1 to data_types.size() do
  data_type := data_types.getDataType(i);
  if (data_type = SIMPLE_TYPE)
    numOfDET++;
  else
```

```
      numOfRET++;
    endif;
    if(data_type = ILF)
      size := size + evaluate (numOfDET, numOfRET, ILF);
    else
      size := size + evaluate (numOfDET, numOfRET, EIF);
    endif;
  enddo
```

4.2. Mapping for MKII FPA

Fig. 5 contains an instance of the GASS model showing the MK II FPA (UKSMA, 1998) data model. In the Mark II FPA method, data groups are called entity types and do not directly contribute to the functional size. Therefore, $FPC_2 = 0$ in all cases. Logical transactions are broken down into activities. There are only three types of activities in MK II FPA, namely input, processing and output. Table 2 shows mapping for activities defined in a generalized form. Notice that MK II FPA does not have an equivalent to the calculate activity, which is due to the fact that processing activity deals with existing entities.

As in the case of the FPA method, the mapping is formalized with the formula E8.

$$\begin{aligned}
&H = (T_1, \ldots, T_\tau)\\
&T_i = (P_{i1}, \ldots, P_{in})\\
&P_{ik} = (\Theta_{ik}, r_{ik}, D_{ik}, C_{ik})\\
&\Theta \in \{\text{Input},\ \text{Processing},\ \text{Output}\}, \quad C_{ik} \in \{\Phi\}\\
&F \in \{\Phi\}\\
&\text{FPC}_{MKII} = \sum_i \sum_k K(\Theta_{ik}) * N(D_{ik}, \Theta_{ik}, r_{ik}) = \sum_i k_{\text{input}} * N(D_{i1}) + k_{\text{output}} * N(D_{i2}) + k_{\text{processing}} * N(r_i)
\end{aligned} \tag{E8}$$

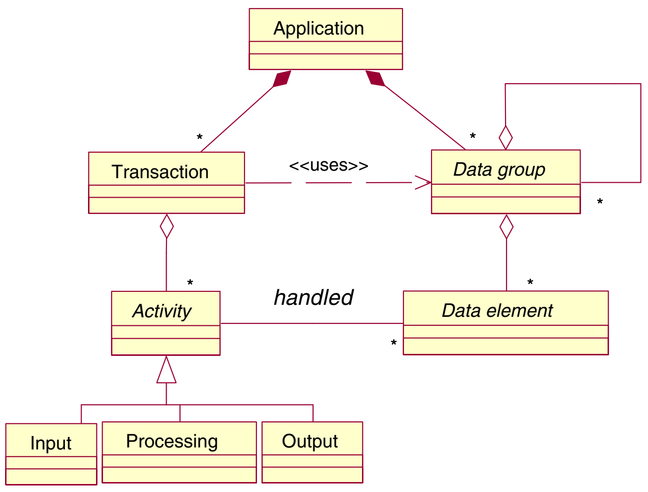


Fig. 5. The MK II FPA data model.

Table 2
Mapping for MK II FPA

	GASS model					
	Entry	Exit	Write	Read	Confirm	Calculate
MK II FPA						
Input	✓					
Output		✓			✓	
Processing			✓	✓		✓

K, function that defines the numerical weight according to the element type Θ,
N, function that returns the number of data elements d or the number of referenced elements r used in an activity of type Θ,
k_{input}, weight for the input elements (the statistically set value is 0.58),
K_{output}, weight for the output elements (the statistically set value is 0.26),
$K_{\text{processing}}$, weight for the processing elements (the statistically set value is 1.66),
D_{i1}, set of data elements used in the input activity of transaction T_i,
D_{i2}, set of data elements used in the output activity of transaction T_i.

Symbolic code for the MKII FPA method:
```
size := 0;
transactional_types := h.getTransactionalTypes(); //h is an instance of H
for i = 1 to transactional_types.size() do
  t := transactional_types.getType(i);
  activities := t.getActivities();
  for ii = 1 to activities.size() do
    p := activities.getActivity(ii);
    case p.getClass() of
      ENTRY: size := size + p.getDataElements().size() * 0.58;
      EXIT, CONFIRM: size := size + p.getDataElements().size() * 0.26;
      WRITE, READ, CALCULATE: size := size + p.getDataElements().size() * 1.66;
    endcase;
  enddo;
enddo;
```

4.3. Mapping for COSMIC-FFP

The COSMIC-FFP (COSMIC, 2003) method defines cfsu as a unit of measure and introduces a different approach to software sizing. The method counts data movements that can be either one of four types: entry, exit, read, and write. The method does not distinguish between data elements and data groups, thus it introduces the term “object of interest” that can represent both types. Fig. 6 shows these changes graphically. Please note that the data group always has only one data element; in other words the data element is equivalent to the data group.

$$\begin{aligned}
&H = (T_1, \ldots, T_\tau, F_1, \ldots, F_\sigma)\\
&T_i = (P_{i1}, \ldots, P_{in})\\
&P_{ik} = (\Theta_{ik}, r_{ik}, D_{ik}, C_{ik})\\
&\Theta \in \{\text{Entry},\ \text{Exit},\ \text{Read},\ \text{Write}\}, \quad C_{ik} \in \{\Phi\}\\
&F_j = g_j = d_j \Rightarrow r_{ik} = D_{ik}\\
&\text{FPC}_{\text{COSMIC}} = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} \sum_{k=1}^{4} P_{ik}(F)
\end{aligned} \tag{E9}$$

Eq. (E9) shows the formal representation of the method. Since the data groups (g) are equal to the data elements (d) also the referenced data groups (r) are equal to handled data elements (D). The sum across all identified transactions (T_i) is made in the first part of the calculation. In the second part, transactions are broken down into activities (P_{ik}), where k runs from 1 to 4, since the method has only four types of activities. With the F in brackets, we have revealed that activity depends on data types, since data is the object of movement. Again $\text{FPC}_2 = 0$ and only FPC_1 contributes to the application size. Table 3 shows the mapping of elements.

An algorithm is quite simple since every data movement/activity counts as 1 cfsu. The difficult part is the identification of all data movements.

Symbolic code for the COSMIC-FFP method:
```
size := 0;
transactional_types := h.getTransactionalTypes(); //h is an instance of H
for i = 1 to transactional_types.size() do
```

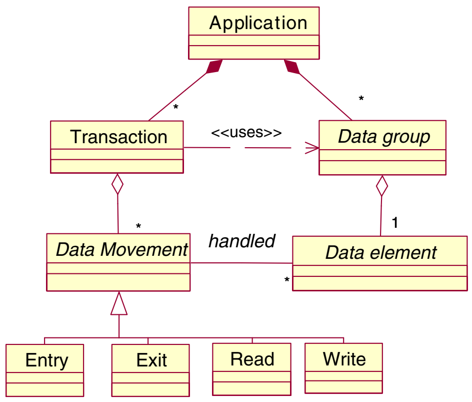


Fig. 6. The COSMIC-FFP data model.

Table 3
Mapping for COSMIC-FFP

	GASS model					
	Entry	Exit	Write	Read	Confirm	Calculate
COSMIC FFP						
Entry	✓					
Exit		✓			✓	
Read				✓		
Write			✓			✓

```
    t := transactional_types.getType(i);
    activities := t.getActivities(); //data movements
    for ii = 1 to activities.size() do
      size := size + 1;
    enddo;
  enddo;
```

5. Transformation of object-oriented concepts to GASS

Since the FPA method's family does not define an appropriate abstraction for object-oriented systems (Antoniol et al., 1999, 2003; Uemura et al., 1999, 2001; Živkovič et al., 2005a,b), our approach takes two steps:

1. A software system represented with UML models is transformed to generalized abstraction of the software system (GASS).
2. The GASS form is transformed to software size using the most suitable method (for example FPA, MK II FPA, COSMIC).

To be able to transform UML models into the GASS form it is important to analyze UML diagrams and define sources of valuable information for the size estimation process. In Table 4, a summary of this analysis for the FPA method can be found. The findings helped us formulate the transformation to the GASS form.

Fig. 7 shows a data model for estimating software size in object-oriented projects. The model is based on the GASS model discussed in more detail in Section 3. Transactions defined in the GASS model are use cases in object-oriented projects. Use cases can be described in more detail with activity diagrams and/or sequence diagrams. The transformation of activities is obvious. The type of an activity is defined by a stereotype. We defined six new stereotypes with the same names and purposes as defined in the GASS model. The realization of an activity during the design time is a method. Methods can be of five types that correspond to the classifications in the GASS model. *Constructors* represent entry activities, *get* methods are read activities, *set* methods are write activities and for the exit activities we defined a new type of method. These methods are labeled with *view* at the beginning of the method's name, as is the case with the *get* and *set* methods. All other methods fall into a *business* methods group and are classified as calculate activities. The data groups in the GASS model are classes in an object-oriented paradigm and data elements are attributes.

Table 4
UML diagrams as a source of information relevant to the size estimation process

UML diagram	Data functions	Transactional functions	Complexity		
			DET	RET	FTR
Use cases	x	x			
Class diagram	x	x	x	x	x
Sequence diagram		x	x		x
Collaboration diagram		x	x		x
Activity diagram	x	x	x		x
Statechart diagram			x	x	x

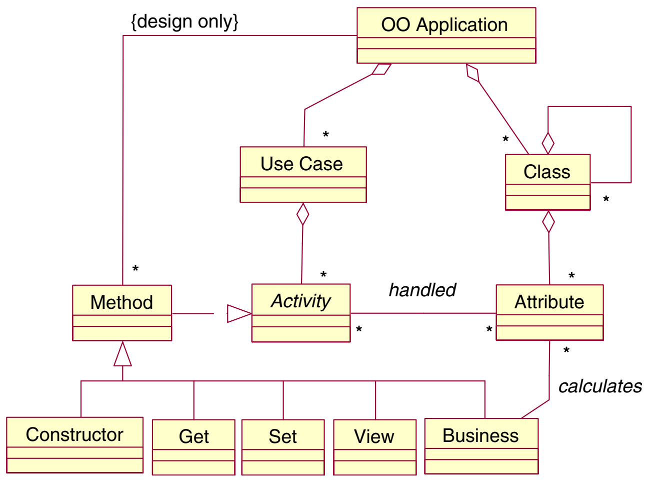


Fig. 7. Data model for estimating OO projects.

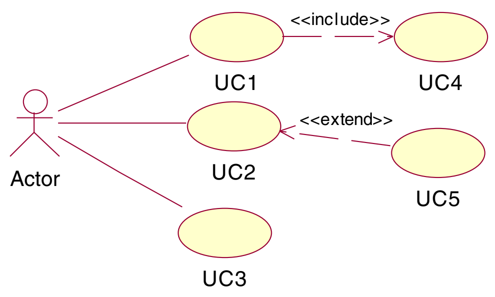
Table 5
Transformation of activities for OO

	GASS model					
	Entry	Exit	Write	Read	Confirm	Calculate
OO elements						
Constructor	✎					
Get method				✎		
Set method			✎			
View method		✎				
Business method						✎

Table 5 summarizes the transformation of activities from the GASS model and different types of methods in object-oriented systems.

5.1. Use case diagram

A use case diagram describes the functionality of the software system at a high level of abstraction. Nevertheless, the diagram can be used in size estimation. For each use case in the diagram its complexity is evaluated using use case description or experience. The description in a textual or more formal form contains the number of transactional functions. With the number of transactional functions and historical data, the use case size can be calculated (Živkovič et al., 2005b). Usually data from our own completed projects give the best results; however we can also use industrial repositories and average values instead. Fig. 8 shows an example of the transformation. The application H has five transactional types denoted as $T_{\mathrm{UC1}} - T_{\mathrm{UC5}}$. Referring only to the use case diagram in Fig. 8 the transactional types can not be further broken down into activities (elements P of the GASS form).



$H=(T_{UC1}, T_{UC2}, T_{UC3}, T_{UC4}, T_{UC5})$

Fig. 8. Transformation of a use case diagram into universal representation.

The algorithm in the symbolic code is:

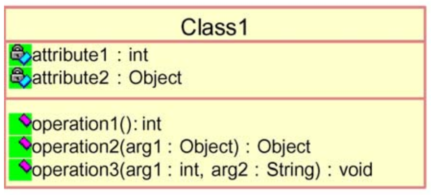
```
uc := UC_diagram.getAllUC ();
for i=1 to uc.size() do
  create new T;
enddo
```

5.2. Class diagram

The completed class diagrams contain all the information needed to calculate software size. A class is mapped to a data group (*F*) while its methods are mapped into transactional types (*T*). Attributes of a primitive type (whole numbers, decimal fractions, characters, character sequences and logical types) are transformed into data elements (*d*) while references to composite types (classes) are mapped into data sub-groups (*g*). Fig. 9 shows an example of a class and its mathematical representation. The class has three operations and two attributes.

The algorithm in the symbolic code is:

```
classes := class_diagram.getAllClasses ();
for i = 1 to classes.size() do
  create new F;
  class := classes.getClass(i);
  class_attributes := class.getAttributes();
  for ii = 1 to class_attributes.size() do
    attribute := class_attributes.getAttribute(ii);
    if (attribute.getType() = basic_type) then
      create new d;
    else
      create new g;
    endif
  enddo
  class_methods := class.getMethods();
  for iii = 1 to class_methods.size() do
    method := class_methods.getMethod(iii);
```



Class1
attribute1 : int
attribute2 : Object
operation1(): int
operation2(arg1 : Object) : Object
operation3(arg1 : int, arg2 : String) : void

$H=(T_{UC1}, F_{C1})$
$T_{UC1}=(P_{op1}, P_{op2}, P_{op3})$
$P_{op1}= (\Theta_{write}, \Phi, d_{simple}, \Phi)$
$P_{op2}= (\Theta_{calculate}, r_{object}, \Phi, c_{object})$
$P_{op3}= (\Theta_{read}, \Phi, D_1, \Phi)$
$D_1=(d_{arg1},d_{arg2})$
$F_{C1}=(d_{att1}, g_{att2})$

Fig. 9. Transformation of a class into the universal representation.

```
      if (method.getMethodType() = TYPES.CONSTRUCTOR) then
        create new P(Activity.ENTRY);
      endif
      if (method.getMethodType() = TYPES.GET) then
        create new P (Activity.READ);
      endif
      if (method.getMethodType() = TYPES.SET) then
        create new P (Activity.WRITE);
      endif
      if (method.getMethodType() = TYPES.VIEW) then
        create new P (Activity.EXIT);
      endif
      if (method.getMethodType() = TYPES.BUSINESS) then
        create new P (Activity.CALCULATE);
      endif
      method_parameters := method.getParameters();
      for a = 1 to method_parameters.size() do
        parameter = method_parameters.getParameter(a);
        if(parameter.getType = basic_type) then
          P.add(d);
        else
          P.add(r);
        endif
      enddo;
      if (method.getReturnType () <> void) then
        P.add(c);
      end if
    enddo;
  enddo;
```

5.3. Sequence diagram

From the approaches found in the literature, only Uemura's (Uemura et al., 1999, 2001) approach uses sequence diagrams as a source of information for size estimation. According to our tests, his approach using five interaction patterns does not give appropriate results compared to Antoniol et al.'s (1999, 2003) approach, which uses information from class diagrams. The advantage of Uemura's approach is that the type of the transactional function is determined without relying on additional information using the five interaction patterns. However there are also several disadvantages: (1) messages without any arguments are not counted, (2) it is not clear what happens when the same message is used in different sequence diagrams, (3) pattern 2 may be confusing since return messages are commonly used in the sequence diagrams to report the result back to the actor, (4) the number of arguments is mapped to DETs although the argument may be of a complex type and should be regarded as FTR and (5) the argument problem described under 4 also has consequences for patterns 2 and 3, which also influences patterns 4 and 5 since the return may be the data function (DF) itself and therefore it may be treated as returning all the attributes of the DF. Consequently Uemura's approach was abandoned and new rules were formulated that help us determine transactional functions from a sequence diagram.

The algorithm in the symbolic code is:

```
objects := sequence_diagram.getAllObjects();
for i = 1 to objects.size() do
  create new F;
  object := objects.getObject(i);
  class := object.getClass();
  class_methods := class.getMethods();
  for iii = 1 to class_methods.size() do
    method := class_methods.getMethod(iii);
    if (method.getMethodType() = TYPES.CONSTRUCTOR) then
  create new P (Activity.ENTRY);
```

```
      endif
      if (method.getMethodType() = TYPES.GET) then
    create new P (Activity.READ);
      endif
      if (method.getMethodType() = TYPES.SET) then
    create new P (Activity.WRITE);
      endif
      if (method.getMethodType() = TYPES.VIEW) then
    create new P (Activity.EXIT);
      endif
      if (method.getMethodType() = TYPES.BUSINESS) then
    create new P (Activity.CALCULATE);
      endif
      method_parameters := method.getParameters();
      for a = 1 to method_parameters.size() do
        parameter = method_parameters.getParameter(a);
        if (parameter.getType = basic_type) then
          P.add(d);
        else
      P.add(r);
    endif
      enddo;
      if (method.getReturnType () <>void) then
        P.add(c);
      end if
    enddo;
  enddo;
```

Fig. 10 shows an example of a transformation for a given sequence diagram. Since a message in the object-oriented system manifests itself as an operation call, the operations can be found in the example above. In the universal representation, it is assumed that a sequence diagram describes a use case UC1. UC1 is an example of the transactional type that is further described by the activities P. The operation *operation1* from the example is mapped to P_{opC1} and *operationC2(String)* into P_{opC2}. Activity type (Θ) is determined from the operation name or from its stereotype, if one is set. Since P_{opC1} does not have parameters but returns the value of an attribute, it is classified as a read activity. Other values are empty sets denoted as Φ. Activity P_{opC2} is of the type calculate. In its set of used data elements (D) only one data element (d_{arg1}) can be found. This activity also creates new data elements c_{object} that are of a complex type. Please note that *Class1* and *Class2* correspond to data groups F_{c1} and F_{C2}. From the sequence diagram, data groups cannot be described in more detail, therefore the values for F_{C1} and F_{C2} are labeled with η—a special symbol that in the universal form represents undefined value.

5.4. Activity diagram

In object development, the activity diagram is used to define procedures and algorithms. In the past, activity diagrams were not used as a source of information for software-size estimation. However, from Table 4 it can be concluded that the

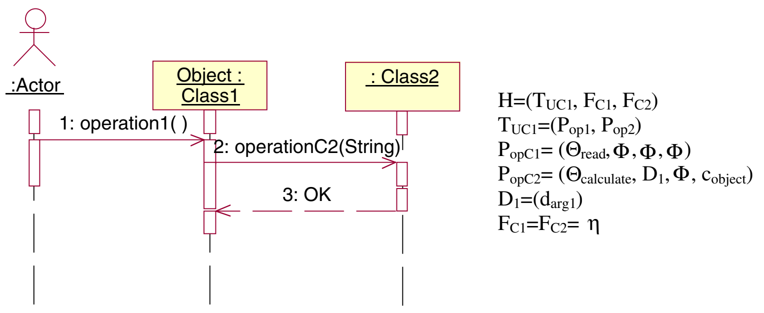

:Actor
Object : Class1
: Class2
1: operation1( )
2: operationC2(String)
3: OK
$H=(T_{UC1}, F_{C1}, F_{C2})$
$T_{UC1}=(P_{op1}, P_{op2})$
$P_{opC1}= (\Theta_{read}, \Phi, \Phi, \Phi)$
$P_{opC2}= (\Theta_{calculate}, D_1, \Phi, c_{object})$
$D_1=(d_{arg1})$
$F_{C1}=F_{C2}= \eta$

Fig. 10. Transformation of a sequence diagram into a universal representation.

quality of information is right behind that of the class diagram. The main purpose of activity diagrams is to document some procedure; therefore we used them to describe use cases in a semi-formal way.

The use of activity diagrams to describe use cases enables us to determine the number of transactional and data functions following these rules:

P1. All activities in the swimlane named "system" are treated as transactional functions.
P2. All objects that have a user-defined type are allocated to data groups (F).

The complexity of a transactional function (T) is determined according to the number of input and output object flows. If object flows are not shown at the diagram, all transactional functions are given an average complexity.

Fig. 11 shows the transformation of an activity diagram into universal representation. As in the case of sequence diagram, the activity diagram is made for UC1. The steps for UC1 are defined with activities Activity1 to Activity5. For mathematical representation, only activities within the swimlane named System (*Activity 1*, *Activity3* and *Activity4*) are significant. In the universal representation, they are labeled as P_{A1}, P_{A3} and P_{A4}. If the object flow is defined on the activity diagram, activities in the universal representation can be further described. In our simplified example, only activity P_{A3} has detailed information on data groups ($D1$). Since the type is "object reference" it is marked as g_1. Data group F_{C1}, shown in the activity diagram, cannot be further broken down. Therefore, its value is denoted as η.

The algorithm in the symbolic code is:

```
activities := activity_diagram.getAllActivities ();
for i = 1 to activities.size() do
  activity = activities.getActivity(i);
  if (activity.getSwimlane() = system) then
    create new T (activity.getStereotype());
    input_object_flows := activity.getInputObjectFlows();
    for ii = 1 to input_object_flows.size() do
      input_object_flow := input_object_flows.getObjectFlow(ii);
      if(input_object_flow.getType = new_type) then
        create new F;
        if (input_object_flow.getType = basic_type) then
          create new d;
```

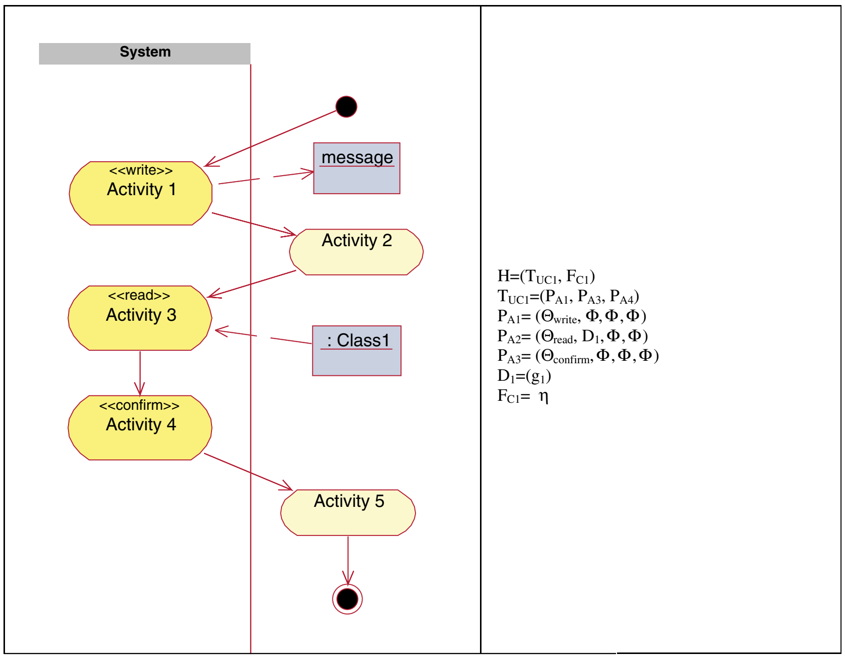

System
<<write>>
Activity 1
message
Activity 2
<<read>>
Activity 3
: Class1
<<confirm>>
Activity 4
Activity 5
$H=(T_{UC1}, F_{C1})$
$T_{UC1}=(P_{A1}, P_{A3}, P_{A4})$
$P_{A1}= (\Theta_{write}, \Phi, \Phi, \Phi)$
$P_{A2}= (\Theta_{read}, D_1, \Phi, \Phi)$
$P_{A3}= (\Theta_{confirm}, \Phi, \Phi, \Phi)$
$D_1=(g_1)$
$F_{C1}= \eta$

Fig. 11. Transformation of an activity diagram into universal representation.

```
          F.add(d);
        else
          create new g;
          F.add(g);
        endif
      endif
    enddo
  endif;
enddo;
```

With activity diagrams, it is possible to document a method's body. In that case, the complexity of the method can be evaluated; this turns out to be the complexity of a transactional function. However, this information is usually available late in the design.

5.5. Statechart diagram

The Statechart diagram defines all the states of one class. In the universal form, a class is represented as a data group. To determine the complexity of a data group the attributes of a class are used. The Statechart diagram does not deal with attributes and, from that perspective, are not valuable for estimation purposes. One could argue that complexity can be determined using the number of states in the diagram and the complexity of transitions. The rules can be defined according to an analysis of empirical data. However, statecharts are rarely used in business domains and have moderate value for evaluating the complexity of data groups. The complexity of transactional functions can be directly evaluated using methods defined on transitions and the rules for sequence diagrams. In practice, statecharts are not expected to be used in the size estimation process if the information for the evaluation of an element's complexity is available in other diagrams.

6. The application of the GASS model

In previous sections, a formal view of the FSM methods was given. First, the GASS model was defined and supported with a formal representation of abstract elements that help us depict any software system in a way that is convenient for software-size estimation. Then the GASS model was used to instantiate several predominant FSM methods in use today. Finally, the transformation between two abstractions used to define a software system was defined and equipped with algorithms expressed in symbolic code. The input abstraction used the UML diagrams and the output abstraction was an instance of the GASS model ready for use in the second step of the estimation process: software-size calculation. Now the question is: why do we need all this and how can a size estimator benefit from the defined GASS model? There are several application areas where the GASS model could be used.

1. *Definition of a new FSM method*—if the new FSM method is defined in a generalized form, it can easily be plugged into an FSM tool. The new method can be compared to existing FSM methods. Recalculation for past projects is easier as well.
2. *Comparison of different FSM methods*—with the formal model defined, a two-level comparison can be performed, namely a data model comparison and a comparison of the function that maps data elements into software size.
3. *The automation of the measurement steps*—if software models are prepared, the size can be calculated automatically using an FSM tool, since the estimation process is formally defined. An FSM expert's mediation is not needed. The software size is automatically calculated using built-in transformation rules.
4. *For FSM tool implementation*—the algorithms and functions are well defined; therefore the implementation of the tool should be straightforward.
5. *To keep data about the software system in the repository* such as ISBSG (2001)—if the data in the FSM repository are kept in a universal form, similar to the form presented in this research, the size estimation becomes method independent since the value for size can be easily recalculated.
6. *To minimize the measurement error*—the formal model enables the identification of data elements that have the biggest influence on error. In one of our previous studies, the error rate was determined for the FPA elements mismatch (Živković et al., 2005a).
7. *In the FSM method selection process*—some methods perform better in specific domains than others. To be able to select the method that suits one's needs the best, the estimation process needs to be transparent. This is achieved with formalization. For example, when comparing the FPA and MK II FPA methods, it becomes clear that the MK II FPA does not use data elements to calculate size and will consequently produce different results on models having only attributes and no methods.

To evaluate both the GASS model and the transformation algorithms for OO development, the application portfolio defined by Fetcke (1999b) was used. The application portfolio that contains five applications was converted into class diagrams. The class diagram for the application *W*, which represents the complete application portfolio, is presented in Fig. 12.

The class diagram was then converted to XMI format and imported to our tool that supports conversion to the GASS model and three size estimation algorithms, namely FPA, MK II FPA and COSMIC-FFP. The tool was developed in Java and supports size estimation, repository management and project comparison. The results for the reference application portfolio are in Table 6.

Table 6 summarizes size in function points for the applications W, M, C, LC and LS that are part of the application portfolio defined by Fetcke (1999b). For each application the size was calculated using three methods. The results are grouped in columns by each method and labeled with the method name. In each method group four sub-columns can be found. In the first sub-column, labeled Fetcke, the original human expert size estimates calculated by Thomas Fetcke in (Fetcke, 1999b) can be found. These values are used as reference values for our research since Fetcke previously conducted a controlled experiment and published a detailed analysis of the application portfolio size estimation procedure. In sub-columns S1, S2 and S3 the sizes for three variations of the UML abstraction model are given. The S1 is the so-called ‘‘normal’’ abstraction model. An instance of such a model for application *W* can be found in Fig. 12. The S1 abstraction model has only methods and attributes needed to solve the given business problem. In practice it is called an analysis class diagram. Since tha MK II FPA and COSMIC-FFP do not evaluate data functions directly, the abstraction model was changed twice. In the first change (Scenario 2—S2), for each attribute the two methods were added—get⟨attributeName⟩ and set⟨attributeName⟩. In the second change (Scenario 3—S3) only one method per attribute was added. The goal of S2 and S3 was to find the most appropriate UML abstraction model for MK II FPA and COSMIC-FFP since both methods underestimated the application size in comparison to Fetcke's reference estimates for all five applications. The last two rows in Table 6 show the mean value and standard deviation of scenarios S1–S3 from the reference value (Fetcke) for all three methods. For the FPA method it is obvious that in S2 and S3 it measures data functions' contribution more than

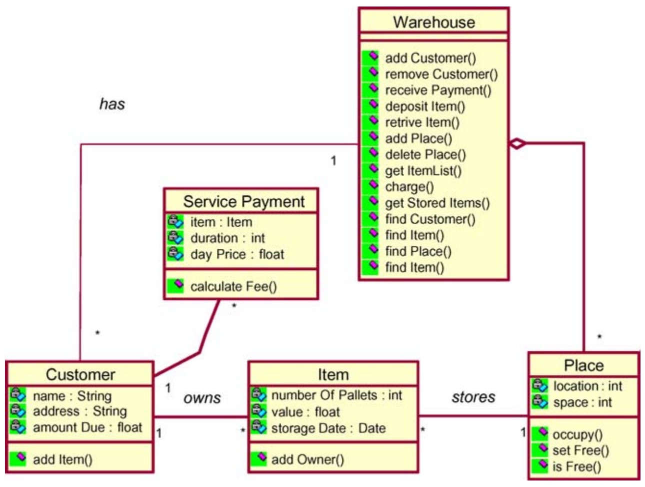


Fig. 12. Class diagram for application portolio (*W*).

Table 6
Measurement results for different methods and test cases

	FPA				MKII FPA				COSMIC-FFP			
	Fetcke	S1	S2	S3	Fetcke	S1	S2	S3	Fetcke	S1	S2	S3
W	77	98	164	131	72.96	46.15	89.64	66.06	81	27.80	54.00	39.80
M	40	56	164	131	32.40	25.45	91.30	69.05	38	15.33	55.00	41.60
C	49	75	129	102	46.72	37.35	75.11	56.44	51	22.50	45.25	34.00
LC	56	83	143	113	48.96	38.84	78.02	55.78	52	23.40	47.00	33.60
LS	31	44	80	62	24.00	21.58	44.82	31.54	29	13.00	27.00	19.00
Mean		20.6	85.40	57.20		11.13	30.77	13.52		29.79	11.35	18.04
Standard deviation		6.11	26.69	21.54		9.26	16.56	12.98		14.07	10.44	14.24

once and therefore overestimates size in both S2 and S3. For Mk II FPA the scenarios S1 and S3 are quite close, therefore it is difficult to decide which approach to take. The COSMIC-FFP gives the best result with S2, followed by S3.

7. Conclusion and future work

In this research, an important topic for size estimation was addressed. Each FSM method should have a formal definition that is easy to use and implement without the need for expert intervention during the measurement process. Only methods that have such a formal definition can be automated and integrated with analysis models already in use. In this case, it is not important which models the development team uses, the only question that arises is: Are the transformation rules already defined? In this research, a GASS model was introduced that adds a formal definition to existing FSM methods. The transformation rules for several FSM methods were also defined. The dominance of UML in object-oriented analysis demanded the definition of transformation rules for UML diagrams. The rules were defined for diagrams that are crucial in the size estimation process. The role of other diagrams was also briefly described. To promote the GASS model's practical application, algorithms in symbolic code were added and some ideas for the GASS model's use were also outlined. In the practical part of this paper, the suitability of the sample UML abstraction models for different FSM methods were tested using the GASS data model. The analysis showed that the FPA method is the most suitable FSM method for the tested abstraction models. The MK II FPA also performed well without changes. However the estimated size was always underestimated. Further analysis may prove that adding a method for each attribute in the analysis class diagram is a better solution. In our test sample the COSMIC-FFP performed with significant underestimates. The statistical analysis showed that the abstraction model should address the presence of attributes in the model with additional methods. The added methods balance the difference and make an adequate substitution for the data functions in the model.

In the future, the characteristics of the FSM method will be written in XML to enable the dynamic implementation of new FSM methods in FSM tools.

References

Albrecht, A., 1979. Measuring application development productivity. In: IBM Applications Development Symposium, pp. 83–92.

Antoniol, G., Lokan, C., Caldiera, G., Fiutem, R., 1999. A function point-like measure for object-oriented software. Empirical Software Engineering 4, 263–287.

Antoniol, G., Fiutem, R., Lokan, C., 2003. Object-oriented function points: an empirical validation. Empirical Software Engineering 8, 225–254.

COSMIC, 2003. COSMIC-FFP Measurement Manual—The COSMIC Implementation Guide for ISO/IEC 19761:2003, version 2.2, Common Software Measurement International Consortium (COSMIC).

Diab, H., Frappier, M., St Denis, R., 2002. A formal definition of function points for automated measurement of B specifications. In: Formal Methods and Software Engineering, Proceedings, pp. 483–494.

Fetcke, T., 1999a. A generalized structure for function point analysis. In: Proceedings of International Workshop on Software Measurement (IWSM'99), Mont-Tremblant, Canada, pp. 1–11.

Fetcke, T., 1999b. The warehouse software portfolio: a case study in functional size measurement. Technical Report Number 99-20, ISSN 1436-9915, TU Berlin.

IFPUG, 2004. Function Point Counting Practices Manual, Release 4.2, International Function Point Users Group, Princeton Junction, USA, January 2004.

ISBSG, 2001. Practical Project Estimation, A toolkit for estimating software development effort and duration. International Software Benchmarking Standards Group.

ISO, 1998. ISO/IEC TR 14143-1. Information technology—Software measurement-Functional size measurement, Part 1: Definition of concepts, first edition, ISO/IEC.

ISO, 2002a. ISO/IEC TR 14143-2. Information technology—Software measurement—Functional size measurement, Part 2: Conformity evaluation of software size measurement methods to ISO/IEC, first edition, ISO/IEC, 14143-1: 1998.

ISO, 2002b. ISO/IEC TR 14143-4. Information technology—Software measurement—Functional size measurement, Part 4: Reference model, first edition. ISO/IEC.

ISO, 2003. ISO/IEC TR 14143-3. Information technology—Software measurement—Functional size measurement, Part 3: Verification of functional size measurement methods, first edition. ISO/IEC.

Jeffery, D.R., Low, G.C., Barnes, M., 1993. A comparison of function point counting techniques. IEEE Transactions on Software Engineering 19, 529–532.

Lokan, C., 1999. An empirical study of the correlations between function point elements. In: Proceedings of METRICS'99: Sixth International Symposium on Software Metrics, pp. 200–206.

Lokan, C.J., 2000. An empirical analysis of function point adjustment factors. Information and Software Technology 9, 649–659.

Lorenz, M., Kidd, J., 1994. Object Oriented Software Metrics. Prentice Hall.

OMG, 2001. Unified Modeling Language Specification, version 1.4., Object Management Group.

Uemura, T., Kusumoto, S., Inoue, K., 1999. Function point measurement tool for UML design specification. In: Proceedings of the Sixth International Symposium on Software Metrics, pp. 62–69.

Uemura, T., Kusumoto, S., Inoue, K., 2001. Function-point analysis using design specifications based on the unified modelling language. Journal of Software Maintenance and Evolution-Research and Practice 13, 223–243.

UKSMA, 1998. UKSMA. Mk II Function Point Analysis, Counting Practices Manual, version 1.31, United Kingdom Software Metrics Association (UKSMA).

Živkovič, A., Hericko, M., Kralj, T., 2003. Empirical assessment of methods for software size estimation. Informatica (Ljubljana) 4, 425–432.

Živkovič, A., Hericko, M., Brumen, B., Beloglavec, S., Rozman, I., 2005a. The impact of details in the class diagram on software size estimation. Informatica (Lithuania) 2, 195–311.

Živkovič, A., Rozman, I., Heričko, M., 2005b. Automated software size estimation based on function points using UML models. Information & Software Technology 47, 881–890.

Marjan Heričko is an Associate Professor at the University of Maribor, Faculty of EE&CS, Institute of Informatics. He received his M.Sc. (1993) and Ph.D. (1998) in computer science from the University of Maribor. His research interests include all aspects of IS development with emphasis on metrics, software patterns, process models and modeling.

Ivan Rozman received the Ph.D. degree from University of Maribor in 1983. He is a full professor of software engineering at the Faculty of EE and CS. Prof. Rozman is author and co-author of numerous articles published in different scientific journals and a member of several program committees at domestic and international conferences. He is currently Rector of the University of Maribor.

Aleš Živkovič is a teaching assistant at the University of Maribor. His research work covers different aspect of object technology with the emphasis on UML, software processes, Java platform and metrics. He gained his practical experiences in cooperation with industry on several projects. Aleš received his master degree in 2000 and Ph.D. degree in 2005 both from University of Maribor.