INFORMATICA, 1997, Vol. 8, No. 1, 57–82

SOFTWARE SYSTEM ENGINEERING: ANALYSIS OF THE DISCIPLINE

Albertas ČAPLINSKAS

Institute of Mathematics and Informatics
Akademijos 4, 2600 Vilnius, Lithuania
Email: alcapl@ktl.mii.lt

Abstract. Software system engineering has not yet developed an engineering science for its discipline. On the other hand, a lot of fundamental concepts, shared methods, techniques, patterns for structuring software systems, and languages for documenting design decisions has been accumulated over the years. To analyse and systematise the accumulated ideas is the main challenge for computer scientists today. The main objective of this paper is to analyse software system engineering both as a discipline and as an engineering science. A special attention is paid to conceptual modelling formalisms used in software system engineering.

Key words: software system engineering, software system engineering paradigms, software system engineering principles, conceptual modelling.

1. Introduction. Software engineering is now almost 30 years old. Meanwhile thousands of articles and a lot of books have been published on its theory and practice and a wide spectrum of methods and techniques that can be successfully applied in software projects has been developed. Despite those facts, software engineering has not yet developed an engineering science for its discipline (Tichy *et al.*, 1993) and according to Jackson it is not a discipline at all; it is an aspiration, as yet unachieved (Jackson, 1994).

The critical approval of the contemporary state-of-the-art in software engineering is shared by Andriole and Freeman (1993), Fenton (1993), Jackson (1994) and many other researchers. The state-of-the-art has been discussed at the Dagstuhl Workshop on Future Directions in Software Engineering and estimated as unsatisfactory (Tichy *et al.*, 1993). According to (Davis, 1996) software engineering is "perhaps more like custom home construction than either electrical engineering or oil on canvas".

On the other hand, Waserman argues that "despite the rapid changes in computing technology and software development, some fundamental concepts of software engineering have remained constant". These fundamental concepts form a kernel of the engineering science for software system engineering. Identification and systematisation of these concepts are the main challenge for the computer scientists today.

2. Synopsis. The main objective of this paper is to analyse software system engineering both as a discipline and as an engineering science. A special attention is paid to conceptual modelling formalisms used in software system engineering. The remainder of the paper is organised as follows. Section 3 provides the notation and terminology. Section 4 discusses the scope of the software system engineering discipline. Section 5 considers the structure of software system engineering, discusses its processes and objects. Section 6 discusses the paradigms and fundamental principles used in software system engineering. Section 7 deals with the models used in software system engineering and considers some differences between information modelling formalisms and object-oriented formalisms. Finally, Section 8 concludes the work.

3. Terminology and notation. The term *software engineering* has been introduced by the NATO Conference on Software Engineering in 1968 (Tichy *et al.*, 1993). Already from the very beginning, the term has two different meanings. It was used in a narrow sense, as "the disciplined application of principles, methods and tools to the requirements analysis, design, implementation, operation and maintenance of software comprising computer programs, operating procedures and associated documentation" (McDermid, 1985), and in a wide sense, as the discipline that seeks to devise techniques for software development (Ramamoorthy *et al.*, 1984). Later definition uses the term *software* instead *computer programs, operating procedures and associated documentation.* These two meanings are quite distinct. The first concentrates on computer programs, whereas the second one is concerned with software systems, and a software system is thought of as an entity that includes computer programs only as one of its many components. This dichotomy remained up till now though the term software engineering currently is most often used in a wide sense. Some representative examples are the following:

"The entire range of activities used to design and develop soft-

ware, with some connotation of "good practice"."

(Dictionary of Computing, 1990)

"1. The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software: that is the application of engineering to software.
2. The study of approaches as in (1)."

(IEEE Std 610.12–1990, 1994)

"That form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems."

(Humphrey, 1993)

"The application of tools, methods, and disciplines to produce and maintain an automated solution to real-world problem."

(Blum, 1992)

In 1990 Tayer and Roice proposed the term software system engineering and defined it as follow:

"1. A technical and management process. The technical process is the analytical effort necessary to transform an operational need into a software design of the proper size and configuration, and its documentation in requirements specifications. The management process involves assessing the risk and cost, integrating, integrating the engineering specialities and design groups, maintaining configuration control, and continuously auditing the effort to ensure that cost, schedule, and technical performance objectives are satisfied to meet the original operational need [adapted from (Sailor, 1990)]. Software system engineering has the same relationship to software engineering as system engineering has to hardware engineering (all types).
2. A special case of system engineering."

(Thayer and Thayer, 1990)

The term software system engineering has also been used by Andriole and Freeman and defined as a discipline that combines the essential elements of system engineering and software engineering and is addressed to the creation of complex software-intensive systems (Andriole and Freeman, 1993).

In this paper, we use the term software system engineering to address the discipline that is concerned with software system development and define it as follows:

DEFINITION 1.

1. Software system engineering is a special case of system engineering that deals with the industrial development of special systems, software systems.

2. Software system engineering is the study of what principles, methods, techniques, tools, languages, and procedures are expedient to apply in order to develop a software system in an industrial way and how those principles, methods, techniques, tools, languages, and procedures have to be applied to be effective.

By a *software system* we mean a collection of related computer programs, protocols, interfaces, files, data and knowledge bases, and, maybe, other components intended for solving a real world problem. By *industrial* we mean the development of a software system that must satisfy the timing, marketing, engineering, customer service, and other requirements. Specific features of industrial development which distinguish it from custom home construction are planning, teamwork, automation, quality management, standardisation and depersonalisation of the created artefacts, and mass production.

We use the term *software engineering* only in a narrow sense.

According to Wasserman "the cognitive leap from understanding a problem to implementing a system is too great to proceed without including analysis and design models as key deliverables in the development process" (Wasserman, 1996). In software system engineering we deal with many kinds of models. Most important models are application domain models, models of a system to be developed, and software process models. Application domain models capture analysis and design information and are used for communicating this information to others. Software process models capture technology information on the development process itself.

To model the application domain, the system to be developed, and the software process we use some modelling formalisms. There is an important distinction between the modelling formalism and its notation (its representation). We define the modelling formalism as follows:

DEFINITION 2. A modelling formalism is a four-tuple

$$\Psi = \langle \alpha, \Xi, \Phi, \Omega \rangle,$$

where

α is a set of modelling primitives,

Ξ is a set of term constructors,

Φ is a set of formula constructors,

Ω is a reasoning method.

In the modelling formalism Ψ, modelling primitives and all what can be produced from them by means of Ξ-constructors are regarded as terms. The Φ-constructors are used to produce formulas. All what can be produced from the modelling primitives by means of Φ-constructors are regarded as formulas. The terms are used to model a structure of the original and the formulas are used to model statements about the original.

The terms and formulas are to be expressed by some textual, graphical, or mixed notation. Modelling primitives and compound constructions of the formalism Ψ can be represented in many different ways. Software system engineering has no standards in this matter. A lot of different modelling formalisms are used and each modelling formalism usually has several notation. This situation is probably reasonable because application domains (or software systems) to be modelled are sometimes quite different and any "standard" notation cannot be suitable in all cases. However, it is helpful to use some "standard" notation to discuss and compare different modelling formalisms. To this end this paper will keep to the following conventions, adapted from (Aït-Kaci and Nasr, 1986; Kifer *et al.*, 1993).

An *attribute signature* $Attr \Rightarrow$ **rs** defines a unary function $Attr$ that maps instances of a given class or given relation into a *range set* **rd**. In the case of a class attribute, we use $\overline{\Rightarrow}$ instead of $\Rightarrow$. Which range sets are allowed depends on the modelling formalism. The range sets which model weak entities are marked by "$*$".

We distinguish built-in range sets, declared range sets, and derived range sets. Built-in range sets are modelling primitives. In the attribute signature a built-in range set we denote by name (e.g., $Age \Rightarrow$ **integer**). Declared range sets are constructed by means of Φ-constructors. In the attribute signature a declared range set we denote by name, too. (e.g., $Passenger \Rightarrow$ **Person**).

Derived range sets are constructed using Ξ-constructors. The construction may be described in the attribute signature or in the previous text. Some examples are as follows:

- $Date \Rightarrow \{\textbf{month} \times \textbf{day} \times \textbf{year}\}$

- .....
  m: month ← **m:** {**"January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"**};
  d: date ← **d:** {**month** × **day** × **year**};

- .....
  $Date \Rightarrow$ **date**

- **x: Supplier** ← **x:** {**Person** ∪ **Organisation**};
  $Supplier \Rightarrow$ **Supplier**

- **v: name** ← **(v: string)&(length(v)** ⩽ **30);**
  $First\_name \Rightarrow$ **name**

An *attribute signature* $Attr \Rightarrow\Rightarrow$ **rs** : (*mincard, maxcard*) defines a unary multivalued function *Attr* that maps instances of a given class or given relation into subsets of a range set **rd** with a minimal cardinality *mincard* and a maximal cardinality *maxcard* (e.g., $Name \Rightarrow\Rightarrow$ **name(1,3)**). In the case of a class attribute, we use $\overline{\Rightarrow}\,\overline{\Rightarrow}$ instead of $\Rightarrow\Rightarrow$.

An *attribute signature* $Attr \Rightarrow$ **rs | default: d** defines an attribute with default value **d**. In the case of a class attribute, we use $\overline{\Rightarrow}$ instead of $\Rightarrow$.

The notation $Attr \rightarrow$ **v** is used to denote that the function *Attr* maps the given instance of given class to **v** (**v** ∈ **rs**) and $Attr \rightarrow\!\!\!\rightarrow$ **s** is used to denote that the multivalued function *Attr* maps the given instance of a given class to the set **s** (**s** ⊂ **rs**). We use $\bullet\!\rightarrow$ or $\bullet\!\rightarrow\ \bullet\!\rightarrow$ to describe inheritable constants.

A *method signature* $M@x_1, \ldots, x_n \Rightarrow$ **rs** defines a function M arity $n+1$ that maps instances of some given class and given parameters into range set **rs**. In the case of a class method, we use $\overline{\Rightarrow}$ instead of $\Rightarrow$. In the case of multivalued methods, we use $\Rightarrow\Rightarrow$ or $\overline{\Rightarrow}\,\overline{\Rightarrow}$.

A *signature* **Cls**[...] defines a class or a relation. The structure of the signature depends on the modelling formalism. Some examples are as follows:

Person[$First\_name \Rightarrow$ **name** : (1, 3);

$Name \Rightarrow$ **string**;
$Identity\_number \Rightarrow$ {**n | (n: integer) & (length(n) = 6)**};
$Native\_language \Rightarrow$ (**language | default: "Lithuanian"**);
{$Identity\_number$}]

The notation {$Identity\_number$} is used to denote key attributes.

- **Ownership** [$Owner \Rightarrow$ **Person**; $Property \Rightarrow$ **Car**]

- **Employee** [$Average\_salary$ @ $\overline{\Rightarrow}$ **real**;
  public $Employ$ @ **f: string, n: string, a: string, s: real** $\Rightarrow$
  this{(**this**.$First\_name$ = **f**) & (**this**.$Name$ = **n**)
  this.$Appointment$= **a**) & (**this**.$Salary$ = **s**)};
  $First\_name \Rightarrow$ **name** : (1, 3);
  $Name \Rightarrow$ **string**;
  $Nationality \bullet\!\!\rightarrow$ "**Lithuanian**";
  $Salary \Rightarrow$ **integer**;
  public $Dismiss$ @ $\Rightarrow$
  |($Salary \rightarrow$ **x**) $\Rightarrow$ (**x** $\leqslant$ **1500**)]

The part-of relation is described by signature $\mathbf{Rel_{part\text{-}of}}$[...].

We use ":" to represent a class membership (e.g., **john: Person**) and "::" to denote a subclass relationship (e.g., **Employee:: Person**). The notation $\langle \mathbf{Cls_1}, \ldots, \mathbf{Cls_n} \rangle_{(\mathbf{x,y})}$:: **Cls** is used to describe partition of class **Cls**. The partition can be total (**x = t**) or not (**x = p**). It can be strong ($\mathbf{Cls_i} \cap \mathbf{Cls_j}$ for all i, j = $1, \ldots, n$) or not. A strong partition is denoted by **y = s**, and a weak partition is denoted by **y = w**. Some examples are as follows:

- $\langle$**Child, Teenager, Adult**$\rangle_{(t,s)}$:: **Person**

- $\langle$**Russian_speaking, English_speaking**$\rangle_{(p,w)}$:: **Person**

4. The scope of the discipline. The scope of the software system engineering discipline has been analysed by Andriole and Freeman (1993). They suggest that two dimensions (time and activity) provide a framework for describing the scope of the discipline and use this framework to analyse and compare systems engineering and software engineering (in a wide sense) activities over the time frame of a system's life.

For educational purpose we need to describe the scope of system software engineering in terms of subdisciplines that can be taught independently taking

Software System Engineering	Requirements Engineering	Software Engineering	Data Engineering	Knowledge Engineering	Interface Engineering	Maintenance and Reengineering	Tolls Theory
Theoretical Foundamentals	+	+	+	+	+	+	+
Engineering Fundamentals	+	+	+	+	+	+	+
Architecture		+	+	+	+	+	+
Project Management	+	+	+	+	+	+	
Quality Management	+	+	+	+	+	+	+
Configuration Management	+	+	+	+	+	+	+

Fig. 1. The scope of Software System Engineering.

into account the dependencies among them. In this case, the framework for describing the scope of the discipline is implied by the nature of knowledge used by software system engineers. The knowledge can be structured in many different ways. We distinguish specific knowledge that is used to do specific work in a certain phase of a project only and general knowledge. As general knowledge we regard knowledge that is used in all phases of the project. Fig. 1 presents a description of the scope of the discipline according to this framework. The description follows the turned out tradition. General and specific knowledge may be split into subdisciplines in many other ways, too. The subdiscipline "Theoretical fundamentals" forms a scientific basis for software system engi-

neering. It must teach about concepts, paradigms, principles, models, modelling formalisms, and modelling languages used in software system engineering.

The subdiscipline "Engineering fundamentals" has been suggested at Dagstuhl Workshop on "Future Directions in Software Engineering" (Tichy *et al.*, 1993). It studies software parts and tools for the purpose of creating software systems, and analyses when and where these parts and tools can be efficiently applied (Tichy *et al.*, 1993). The subdiscipline "Architecture" studies a high-level organisation of software system components and interaction between those elements. As noted by Garlan (1995), the architecture of a software system has long been recognised as an important issue (Dijkstra, 1968; Parnas *et al.*, 1985) and recently has begun to emerge as an explicit field of study (Tichy *et al.*, 1993). Although there is currently no common accepted definition of the term "software system architecture", a lot of shared methods, techniques, patterns for structuring software systems and a lot of languages for documenting design decisions have been developed over the years.

The subdiscipline "Tools theory" deals with programming environments, CASE-systems, generators, and other analysis, design, programming and testing tools. Although we have well-formed theory for some tools (compilers, debuggers, etc.), a single tool construction theory is still the main challenge. As in the case of the software system architecture, only a repertoire of shared methods, techniques and patterns exists in this field at present.

Other subdisciplines mentioned in Fig. 1 are traditional and we will not discuss them there.

5. Objects and processes. Andriole and Freeman (1993) argue that in order to understand a discipline means to look at its structure (processes and objects). They posited a set of basic software system engineering processes and suggested a schema for describing the objects created by those processes. The proposed set of basic processes includes analysis, specification, design, programming, testing, verification, validation, and modification. The schema for describing the objects provides development prologues, technical descriptions, system aggregations, installed systems, and derived information.

The ISO/IEC standard 12207–1 (1994) provides a framework for the software life cycle. The framework consists of major processes for a software development, maintenance and usage. This standard groups the processes into primary, supporting, and organisational ones. The standard applies to the acqui-

sition, supply, development, operation, and maintenance of software systems, software products and services.

We argue that the grouping of processes into primary, supporting and organisational provides a proper framework to define the structure of software system engineering as an engineering discipline, too.

Primary (or basic) processes are those without using of which it is impossible to develop or to use a software system. The set of basic processes proposed by Andriole and Freeman is insufficient for this aim. On the other hand, it includes some processes that are not basic in the sense of our definition. We suggest that the set of primary processes includes analysis, specification, design, implementation, integration, preparation to operation, operation, maintenance (reengineering), and retirement. We define the primary processes as follows:

- *analysis*: modelling something done with the aim to understand it;
- *specification*: modelling something to be created in terms of its external characteristics done with the aim to establish its design and implementation objectives and constraints;
- *design*: modelling something to be created in terms of its internal characteristics done with the aim to meet its specifications in the best way;
- *implementation*: building parts of something to be created done with the aim to obtain its specified behaviour and in the way provided by its design;
- *integration*: assembling the parts of something to be created into the whole;
- *preparation to operation*: accumulation of the resources necessary to operate something, its delivery, installation and adaptation to the operation requirements;
- *operation*: support the behaviour of something provided by its specification;
- *maintenance*: changing internal or external characteristics of something to be in operation;
- *retirement*: stepwise withdrawal from the operation something to be in operation.

It should be noted that many of our definitions are suggested by Andriole and Freeman (1993).

According to ISO/IEC 12207–1 (1994) a supporting process supports some basic process as an integral part with a distinct purpose and contributes to

its success and quality. Supporting processes are testing, verification, validation, documenting, configuration management, reviewing, auditing, and problem solving. This list of processes is suggested by ISO/IEC 12207–1 (1994). By problem solving the standard means "a process for analysing and removing the problems (including non-conformance), whatever their nature or source, that are discovered during the development, operation, maintenance, or other processes".

Organisational processes are employed to establish an underlying structure made up of associated processes and personnel, and continuously to improve it (ISO/IEC DIS 12207–1, 1994). We agree with the standard that organisational processes are management, infrastructure establishing, improving other processes, and training.

In principle, we agree with the schema for describing the objects proposed by Andriole and Freeman (1993), however, suggest that it should be filled up with "system distributive".

6. Principles and paradigms. Andriole and Freeman (1993) argue that in order to understand a discipline, its fundamental principles, paradigms and constraints should be identified, too. They identified six fundamental principles (modularity, information-hiding, abstraction, step-wise refinement, decomposition, systematic processes), four paradigms (structured development, formal development, evolving development, and kernel development) and six constraints (incompleteness and inaccuracy of software representations, absence of robust models of standard systems, paucity of observed data concerning the results of software engineering processes, inability to observe many things of interest, and incomplete and changing map between reality and models of reality) of software system engineering.

The list of principles presented by Andriole and Freeman is incomplete and not all the principles listed in it are fundamental and independent. For example, modularity is a concretisation of decomposition and step-wise refinement is a concretisation of abstraction. We identified nine common accepted independent fundamental principles:

decomposition principle: apply a process to a composite object, decompose the object into independent parts (modules) and apply this process to each part separately;

abstraction principle: apply a process to an object iterative by increasing

in each iteration the level of detailing of the object and ignoring those details that are not relevant to the current purpose;

structurisation principle: build all objects from well-defined primitives using the well-studied design patterns only;

open system principle: build all objects in such a way that internal behaviour any of their parts can be changed without influence on other parts;

uniformity principle: apply the same standards to all objects of the same class (modules, documents, etc.);

black-box principle: build all objects so that they could be used without knowledge of their internal structure;

conceptualisation principle: build software system so that it reflect the aplication (problem) domain structure;

metaphorisation principle: build user interface as an application domain metaphor taking into account users' mentality and experience;

user comfortability principle: a user interface has to be user friendly, adaptive, require as little users' efforts as possible, and not cause a psychological discomfort.

It should be noted that all fundamental principles can be concretised and applied in many ways. A taxonomy of the principles used in software system engineering can be built. For example, information hiding is a concretisation of the black-box principle, structured programming is a concretisation of the structurization principle, etc.

According to Andriole and Freeman (1993) the software system engineering paradigms are overall patterns of actions. We argue that the list of paradigms identified by Andriole and Freeman (1993) can be improved because the structured development paradigm as it is defined in this list (phases, defined work products, set relationships between activities) is a project structuring but not engineering paradigm. We propose the following list of paradigms:

top-down development: starting with a set of system requirements, decompose its and map using several intermediate levels of abstraction into subsystems, modules, and other parts of the desired system;

bottom-up development: starting with a set of low level primitives, compose a desired system using several intermediate levels of abstraction;

evolving development: continuous development of the system using a number of intermediate prototypes;

Kernel development: starting with a small, central core of functionality and mechanisms, grow the desired system by accretion of additional mechanisms (1993);

parameteric development: starting with a generic system procurement (usually called a shell), produce a desired system using a set of adaptation and concretization tools;

formal development: starting with a formal specification of a desired system, to generate it using automatic programming tools.

Of course, software system engineering paradigms are not necessarily mutually exclusive and several different paradigms may be used to develop a software system.

7. Models. In software system engineering we deal with many different models. The most important models are application domain models, models of system to be developed and software process models.

Coad and Yordan (1990) argue that four major approaches to application domain modelling used in software system engineering are functional decomposition, data flow models, information modelling, and object-oriented modelling. Entity life cycle models should be added to this list.

Coad and Yordan (1990) noted that the object-oriented approach (e.g., Rubin and Goldberg (1992)) and the information modelling approach (e.g., Chen (1976); Flavin (1981); Shlaer and Mellor (1988)) are quite distinct. Although later uses concept of object, it is rather relation-oriented. In this section we analyse differences between the two mentioned approaches in more precise terms. The analysis concentrates on term representation. It should be noted, that information modelling and object-oriented modelling essentially differs in reasoning mechanisms too, however, in this work we don't try to consider the reasoning issues at all.

DEFINITION 3. An information modelling formalism is an eight-tuple

$$\mathbf{F_I} = \langle \mathbf{Dom}, \mathbf{Cnst_G}, \mathbf{Cls}, \mathbf{Atr}, \mathbf{R}, \mathbf{Rl}, \Xi_\mathbf{I}, \Phi_\mathbf{I} \rangle,$$

where

Dom is a finite nonempty set of symbols called *domain names*,
$\mathbf{Cnst_G}$ is a nonempty set of symbols called *general constants*,
Cls is a finite nonempty set of symbols called class names,

Atr is a finite nonempty set of symbols called functional constants or attribute names,

R is a finite nonempty set of symbols called predicate constants or relations names,

Rl is a finite nonempty set of symbols called role names,

$\Xi_{\mathbf{I}}$ is a set of term construction rules,

$\Phi_{\mathbf{I}}$ is a set of formula construction rules.

In this definition a set of modelling primitives is defined as follow

$$\alpha = \langle \mathbf{Dom}, \mathbf{Cnst_G}, \mathbf{Cls}, \mathbf{Attr}, \mathbf{R}, \mathbf{Rl} \rangle.$$

The reasoning mechanism remained undefined because we did not consider it here. In addition, the following assumption has been made:

1. A many-to-one mapping **type** that maps general constants to domain names is defined. We say that the general constant **c** is a constant of type **d**, if and only if **type**(**c**) = **d**. The set of all constants of type **d** is called a intension of domain **d** or a *set of possible values* of domain **d**.

2. An expression $\sigma_{\mathbf{k}}$ is associated with each class name **k** in **Cls**. This expression is called a *signature* of class **k** and has a special form defined bellow. The signature $\sigma_{\mathbf{k}}$ is a term construction rule that defines the class **k** as an intensional set of terms. We denote this set as $\mathbf{Int_k}$.

3. An expression $\sigma_{\mathbf{attr}}$ is associated with each attribute name **attr** in **Attr**. This expression is called a *signature* of the attribute **attr** and has a special form defined in Section 3. A signature defines one of the following functions:

- a function **attr:** $\overline{\mathbf{Ext}}_{\mathbf{k}} \times \mathbf{T} \Rightarrow \mathbf{d}$ that at each time moment $\mathbf{t} \in \mathbf{T}$ maps the term names of class **k** into the values of domain **d**,
- a function **attr:** $\{\mathbf{k}\} \times \mathbf{T} \overline{\Rightarrow}\ \mathbf{d}$ that at each time moment $\mathbf{t} \in \mathbf{T}$ maps the name of class **k** into the values of domain **d** (in this case, we say that attribute **attr** is a *class attribute*),
- a function **attr:** $\overline{\mathbf{Ext}}_{\mathbf{r}} \times \mathbf{T} \Rightarrow \mathbf{d}$ that at each time moment $\mathbf{t} \in \mathbf{T}$ maps the relationship names of relation **r** into the values of domain **d**,
- a function **attr:** $\{\mathbf{r}\} \times \mathbf{T} \overline{\Rightarrow}\ \mathbf{d}$ that at each time moment $\mathbf{t} \in \mathbf{T}$ maps the name of relation **r** into the values of domain **d** (in this case, we say that attribute **attr** is a *relation attribute*).

We also say in all the mentioned cases that the attribute **attr** is of type **d**. We define the set $\overline{\mathbf{Ext}}$ later on (see Definition 4).

4. An expression $\sigma_{\mathbf{r}}$ is associated with each relation name **r** in **R**. This expression is called a *signature* of relation **r** and is of a special form defined bellow. The signature $\sigma_{\mathbf{r}}$ is a formula construction rule and defines the relation **r** as an intensional set of relationships. We denote this set as $\mathbf{Int_r}$.

5. The expression

$$\mathbf{v} \Rightarrow \mathbf{k}(\nu, \mu), \quad \mathbf{k} \in \mathbf{Cls}$$

is associated with each role name **v** in **Vd**. This expression is called a *signature* of the role **v**. The signature of role **v** defines the class **k** which plays this role and cardinality of the role.

DEFINITION 4. In formalism $\mathbf{F_I}$, an information model of application domain Δ at the time moment **t** is a four-tuple

$$\mathbf{M}_{\Delta,\mathbf{t},\mathbf{F_I}} = \langle \mathbf{Cnst}_{\Delta,\mathbf{A},\mathbf{t}}, \mathbf{Trm}_{\mathbf{t}}, \mathbf{Cnst}_{\Delta,\mathbf{R},\mathbf{t}}, \mathbf{Rls}_{\mathbf{t}} \rangle,$$

where

$\mathbf{Cnst}_{\Delta,\mathbf{A},\mathbf{t}}$ is a finite set of symbols called *application constants* or *objects names*,

$\mathbf{Trm}_{\mathbf{t}}$ is a finite set of objects called *terms*,

$\mathbf{Cnst}_{\Delta,\mathbf{R},\mathbf{t}}$ is a finite set of symbols called *relationship names*,

$\mathbf{Rls}_{\mathbf{t}}$ is a finite set of objects called *relationships*.

In this definition we postulate the existence of one-to-one mapping

$$\eta_{\mathbf{F},\Delta} : \mathbf{Concepts}_{\Delta} \Longleftrightarrow (\mathbf{Dom} \cup \mathbf{Cls}),$$

that maps application domain concepts to domains and classes of the formalism $\mathbf{F_I}$. In addition, the following assumptions have been made.

1. A many-to-one mapping **instance_of** that maps application constants to class names is defined. We say that the application constant **t** is an instance of class **k**, if and only if **instance_of(t)=k**. The set of all application constants of class **k** at the moment **t** is denoted by $\overline{\mathbf{Ext}}_{\mathbf{k},\mathbf{t}}$ and a set of all the possible application constants of class **k** is denoted by $\overline{\mathbf{Ext}}_{\mathbf{k}}$.

2. A many-to-one mapping **class:** $\mathbf{Trm}_{\mathbf{t}} \Rightarrow \mathbf{Cls}$ that maps terms to class names is defined. We say that the term **t** is a term of class **k** if and only if **class(t)=k**. The set of all the terms of class **k** at the moment **t** we denote $\{\mathbf{Trm}\}_{\mathbf{k},\mathbf{t}}$ and a set of all terms of class **k** is denoted as $\{\mathbf{Trm}\}_{\mathbf{k}}$. We expect

that the terms of class **k** meet the signature $\sigma_{\mathbf{k}}$. Additionally, a one-to-one mapping

$$\textbf{object\_name: } \{\textbf{Trm}\}_{\mathbf{k},\mathbf{t}} \Longleftrightarrow \overline{\textbf{Ext}}_{\mathbf{k},\mathbf{t}},$$

that maps the terms of the class **k** to the application constants of class **k** and *vice versa*, is defined.

3. A many-to-one mapping **relationship_of** that maps relationship names to relation names is defined. We say that the relationship name ρ is an instance of relation **r** if and only if **relationship_of**$(\rho) = \mathbf{r}$. The set of all relationship names of relation **r** at the moment **t** is denoted by $\overline{\textbf{Ext}}_{\mathbf{r},\mathbf{t}}$ and the set of all the possible relationship names of relation **r** is denoted by $\overline{\textbf{Ext}}_{\mathbf{r}}$.

4. A many-to-one mapping **relation:** $\textbf{Rls}_{\mathbf{t}} \Rightarrow \mathbf{R}$ that maps relationships to relation names is defined. We say that the relationship ρ is a relationship of relation **r**, if and only if **relation**$(\rho) = \mathbf{r}$. We denote the set of all the relationships of relation **r** at the moment **t** by $\{\textbf{Rls}\}_{\mathbf{r},\mathbf{t}}$ and a set of all the relationships of relation **r** by $\{\textbf{Rls}\}_{\mathbf{r}}$. We expect that relationships of relation **r** meet the signature $\sigma_{\mathbf{r}}$. In addition, a one-to-one mapping

$$\textbf{relationship\_name}: \{\textbf{Rls}\}_{\mathbf{r},\mathbf{t}} \Longleftrightarrow \overline{\textbf{Ext}}_{\mathbf{r},\mathbf{t}},$$

that maps the relationships of relation **r** to the relationship names and *vice versa*, is defined.

DEFINITION 5. In formalism $\mathbf{F_I}$, a class is a three-tuple

$$\textbf{class}_{\mathbf{k}} = \langle \mathbf{k}, \textbf{Int}_{\mathbf{k}}, \textbf{Ext}_{\mathbf{k}} \rangle, \quad \mathbf{k} \in \textbf{Cls},$$

where

k is the name of the class,
$\textbf{Int}_{\mathbf{k}}$ is the intension of the class,
$\textbf{Ext}_{\mathbf{k}}$ is the extension of the class,
$\textbf{Ext}_{\mathbf{k}} = \overline{\textbf{Ext}}_{\mathbf{k}} \cup \{\textbf{Trm}\}_{\mathbf{k}}$.

DEFINITION 6. A signature $\sigma_{\mathbf{k}}$ of the class **k** is the expression

$$\mathbf{k}\Big[\psi_{\mathbf{k}}^{\mathbf{1}} \overline{\Rightarrow}\ (\delta_{\mathbf{1}} | \textbf{default: } \mathbf{g_1}); \ldots; \psi_{\mathbf{k}}^{\mathbf{m}} \overline{\Rightarrow}\ (\delta_{\mathbf{m}} | \textbf{default: } \mathbf{g_m});$$

$$\psi_{\mathbf{k}}^{\mathbf{i}_1} \rightarrow \mathbf{h_1}; \ldots; \psi_{\mathbf{k}}^{\mathbf{i}_r} \rightarrow \mathbf{h_r};$$

$$\mathbf{f}_{\mathbf{k}}^{\mathbf{1}} \Rightarrow (\mathbf{d}_{\mathbf{1}} | \mathbf{default:}\ \mathbf{e}_{\mathbf{1}}); \ldots; \mathbf{f}_{\mathbf{k}}^{\mathbf{n}} \Rightarrow (\mathbf{d}_{\mathbf{n}} | \mathbf{default:}\ \mathbf{e}_{\mathbf{n}});$$

$$\phi_{\mathbf{k}}^{\mathbf{1}} \bullet \rightarrow \mathbf{a}_{\mathbf{1}}; \ldots; \phi_{\mathbf{k}}^{\mathbf{1}} \bullet \rightarrow \mathbf{a}_{\mathbf{1}};$$

$$\left\{\mathbf{f}_{\mathbf{k}}^{\mathbf{j}_1}, \ldots, \mathbf{f}_{\mathbf{k}}^{\mathbf{j}_p}\right\}; \ldots; \left\{\mathbf{f}_{\mathbf{k}}^{\mathbf{b}_1}, \ldots, \mathbf{f}_{\mathbf{k}}^{\mathbf{a}_w}\right\} \left| \gamma_{\mathbf{k}}^{\mathbf{1}}; \ldots; \gamma_{\mathbf{k}}^{\mathbf{s}} \right],$$

where

$\mathbf{f}, \phi$ are term attribute names,

$\mathbf{d}, \delta$ are domain names,

$\mathbf{e}$, $\mathbf{g}$, $\mathbf{h}$ are values,

$\{\}$ denotes a key attribute list,

γ denotes the integrity constraint.

If an attribute is a multivalued attribute, we use $\overline{\Rightarrow}\ \overline{\Rightarrow}$, $\rightarrow\rightarrow$, $\Rightarrow\Rightarrow$ or $\bullet\rightarrow\ \bullet\rightarrow$ instead of $\overline{\Rightarrow}$, $\rightarrow$, $\Rightarrow$, $\bullet\rightarrow$ respectively.

DEFINITION 7. A signature $\sigma_{\mathbf{r}}$ of the relation $\mathbf{r}$ is the expression

$$\mathbf{r}\left[\mathbf{f}_{\mathbf{r}}^{\mathbf{1}} \Rightarrow \mathbf{k}_{\mathbf{1}} : (\nu_{\mathbf{1}}, \mu_{\mathbf{1}}); \ldots; \mathbf{f}_{\mathbf{r}}^{\mathbf{n}} \Rightarrow \mathbf{k}_{\mathbf{n}} : (\nu_{\mathbf{n}}, \mu_{\mathbf{n}});\right.$$

$$\langle \phi_{\mathbf{r}}^{\mathbf{1}} \Rightarrow \mathbf{d}_{\mathbf{1}}; \ldots; \phi_{\mathbf{r}}^{\mathbf{m}} \Rightarrow \mathbf{d}_{\mathbf{m}} \rangle$$

$$\left. | \gamma_{\mathbf{k}}^{\mathbf{1}}; \ldots; \gamma_{\mathbf{k}}^{\mathbf{s}} \right],$$

where

$\mathbf{k}$ is a class name,

$\mathbf{f}$ is a role name,

ϕ is a relationship attribute name,

$\mathbf{d}$ is a domain name,

γ denotes the integrity constraint,

(ν, μ) denotes the cardinality of a role.

If an attribute is a multivalued attribute we use the notation $\Rightarrow\Rightarrow$ instead of $\Rightarrow$. If the relation $\mathbf{r}$ is an aggregation relation we use the notation $\mathbf{r}_{\mathbf{part\_of}}$ instead of $\mathbf{r}$.

DEFINITION 8.

i) In information modelling formalism $\mathbf{F}_{\mathbf{I}}$, a class signature is an object molecule.

ii) There exist no other object molecules except that defined in i).

DEFINITION 9.

i) In information modelling formalism $\mathbf{F_I}$, a relation signature is a predicate molecule.

ii) There exist no other predicate molecules except that defined in i).

DEFINITION 10.

i) Let $\mathbf{k}, \mathbf{k_1}, \ldots, \mathbf{k_n}$ be class names from **Cls**. Then the statements

$\mathbf{k_i} :: \mathbf{k}, \quad \mathbf{i} = \mathbf{1}, \ldots, \mathbf{n},$

$\langle \mathbf{k_1}, \ldots, \mathbf{k_n} \rangle_{(\mathbf{t,s})} :: \mathbf{k},$

$\langle \mathbf{k_1}, \ldots, \mathbf{k_n} \rangle_{(\mathbf{t,w})} :: \mathbf{k},$

$\langle \mathbf{k_1}, \ldots, \mathbf{k_n} \rangle_{(\mathbf{p,s})} :: \mathbf{k},$

$\langle \mathbf{k_1}, \ldots, \mathbf{k_n} \rangle_{(\mathbf{p,w})} :: \mathbf{k}$

are intensional molecular formulas.

ii) In information modelling formalism $\mathbf{F_I}$, object molecules are intensional molecular formulas.

iii) In information modelling formalism $\mathbf{F_I}$, predicate molecules are intensional molecular formulas.

iv) There exist no other intensional molecular formulas except those defined in i), ii) and iii).

DEFINITION 11.

i) In information modelling formalism $\mathbf{F_I}$, intensional molecular formulas are intensional formulas.

ii) If **F** and **G** are intensional formulas, then **F& G** is an intensional formula.

iii) There exist no other intensional formulas except those defined in i) and ii).

DEFINITION 12.

i) Let $\sigma_{\mathbf{k}}$ be a signature of the class **k**. Then the object

$$\mathbf{e_k}\left[\mathbf{f_k^1} \to \mathbf{b_1}; \ldots; \mathbf{f_k^n} \to \mathbf{b_n}; \phi_{\mathbf{k}}^{\mathbf{1}} \to \mathbf{a_1}; \ldots; \phi_{\mathbf{k}}^{\mathbf{r}} \to \mathbf{a_r}\right],$$

where $\mathbf{e_k} \in \overline{\mathbf{Ext}}_{\mathbf{k}}$ and $\mathbf{b_1}, \ldots, \mathbf{b_n}, \mathbf{a_1}, \ldots, \mathbf{a_r}$ are term attribute values satisfying the integrity constraints $\gamma_{\mathbf{k}}^{\mathbf{1}}; \ldots; \gamma_{\mathbf{k}}^{\mathbf{s}}$ is a term of the class **k**.

ii) Let $\sigma_{\mathbf{k}}$ be a signature of the class **k**. Then the object

$$\mathbf{k}\left[\psi_{\mathbf{k}}^{\mathbf{1}} \to \mathbf{c_1}; \ldots; \psi_{\mathbf{k}}^{\mathbf{m}} \to \mathbf{c_m}; \right.$$
$$\left. \psi_{\mathbf{k}}^{\mathbf{i_1}} \to \mathbf{h_1}; \ldots; \psi_{\mathbf{k}}^{\mathbf{i_r}} \to \mathbf{h_r}\right],$$

where $\mathbf{k}$ is a class name and $\mathbf{c_1}, \ldots, \mathbf{c_m}$, $\mathbf{h_1}, \ldots, \mathbf{h_r}$ are class attribute values satisfying the integrity constraints $\gamma_{\mathbf{k}}^{1}; \ldots; \gamma_{\mathbf{k}}^{s}$ is a term that presents the class $\mathbf{k}$.

iii) In information modelling formalism $\mathbf{F_I}$ exist no other terms except those defined in i) and ii).

If an attribute is a multivalued attribute, we use the notation $\rightarrow\!\!\rightarrow$ instead of $\rightarrow$.

Relationships are defined in a similar way.

Let us consider now the object-oriented formalisms.

DEFINITION 13. An object-oriented modelling formalism is a six-tuple

$$\mathbf{F_{Obj}} = \langle \mathbf{Cls}, \mathbf{Cnst_G}, \mathbf{Atr}, \mathbf{Op}, \Xi_{\mathbf{Obj}}, \Phi_{\mathbf{Obj}} \rangle,$$

where

Cls is a finite nonempty set of symbols called *class names*,

$\mathbf{Cnst_G}$ is a nonempty set of symbols called *general constants*,

Atr is a finite nonempty set of symbols called *functional constants* or *attribute names*,

Op is a finite nonempty set of symbols called *method names* or *operation names*,

$\Xi_{\mathbf{I}}$ is a set of term construction rules,

$\Phi_{\mathbf{I}}$ is a set of formula construction rules.

In this definition, a set of modelling primitives is defined as follows

$$\alpha = \langle \mathbf{Cls}, \mathbf{Cnst_G}, \mathbf{Attr}, \mathbf{Op} \rangle.$$

The reasoning mechanism remained undefined because we did not consider it here. In addition, the following assumptions have been made:

1. The set **Cls** is defined as $\mathbf{Cls} = \mathbf{Cls_1} \cup \mathbf{Cls_2}$ where $\mathbf{Cls_1}$ is a nonempty set of symbols called names of *lexical classes* and $\mathbf{Cls_1}$ is a set of symbols called names of *nonlexical classes*.

2. An expression $\sigma_{\mathbf{k}}$ is associated with each class name $\mathbf{k}$ in **Cls**. This expression is called a *signature* of class $\mathbf{k}$ and has a special form defined bellow. The signature $\sigma_{\mathbf{k}}$ is a term construction rule that defines the class $\mathbf{k}$ as an intensional set of terms. We denote this set as $\mathbf{Int_k}$.

3. A many-to-one mapping **type** that maps general constants to names of lexical classes is defined. We say that the general constant $\mathbf{c}$ is a constant of

type **k**, if and only if **type(c)=k**. The set of all constants of type **d** is called **intension** of nonlexical class **k** and denoted as $\mathbf{Int_k}$.

4. An expression $\sigma_{\mathbf{attr}}$ is associated with each attribute name **attr** in **Attr**. This expression is called a *signature* of the attribute **attr** and has a special form defined in Section 3. A signature defines one of the following functions:

- a function **attr:** $\overline{\mathbf{Ext}}_{\mathbf{k}} \times \mathbf{T} \Rightarrow \mathbf{k_1}, \mathbf{k} \in \mathbf{Cls_2}, \mathbf{k_1} \in \mathbf{Cls}$ that at each time moment $\mathbf{t} \in \mathbf{T}$ maps the term names of nonlexical class **k** into the terms of class $\mathbf{k_1}$,
- a function **attr:** $\{\mathbf{k}\} \times \mathbf{T} \overline{\Rightarrow} \ \mathbf{k_1}, \mathbf{k_1} \in \mathbf{Cls}$ that at each time moment $\mathbf{t} \in \mathbf{T}$ maps the name of class **k** into the terms of class $\mathbf{k_1}$ (in this case, we say that the attribute **attr** is a *class attribute*).

In both cases, we also say that the attribute **attr** is of type $\mathbf{k_1}$.

5. An expression $\sigma_{\mathbf{op}}$ is associated with each operation name **op** in **Op**. This expression is called a *signature* of the operation **op** and has a special form defined in Section 3. The signature defines one of the following functions:

- a function

$$\mathbf{op} : \overline{\mathbf{Ext}}_{\mathbf{k}} \times \overline{\mathbf{Ext}}_{\mathbf{k_1}} \times \ldots \times \overline{\mathbf{Ext}}_{\mathbf{k_n}} \Rightarrow \kappa,$$
$$\mathbf{k} \in \mathbf{Cls_2}, \ \kappa, \mathbf{k_1}, \ldots, \mathbf{k_n} \in \mathbf{Cls}, \ \mathbf{n} > 0,$$

  that maps the term names of nonlexical class **k** and parameters into the terms of class κ,
- a function

$$\mathbf{op} : \{\mathbf{k}\} \times \overline{\mathbf{Ext}}_{\mathbf{k_1}} \times \ldots \times \overline{\mathbf{Ext}}_{\mathbf{k_n}} \Rightarrow \kappa,$$
$$\mathbf{k} \in \mathbf{Cls_2}, \ \kappa, \mathbf{k_1}, \ldots, \mathbf{k_n} \in \mathbf{Cls}, \ \mathbf{n} > 0,$$

  that maps the name of class **k** and parameters into the terms of class $\mathbf{k_1}$ (in this case, we say that operation **op** is a *class operation*).

DEFINITION 14. In the formalism $\mathbf{F_{Obj}}$, an object-oriented model of application domain Δ at the moment **t** is a four-touple

$$\mathbf{M}_{\Delta,\mathbf{t},\mathbf{F_{Obj}}} = \langle \mathbf{Cnst}_{\Delta,\mathbf{t}}, \mathbf{Obj}_{\mathbf{t}}, \mathbf{Frm}_{\mathbf{t}} \rangle,$$

where

$\mathbf{Cnst}_{\Delta,\mathbf{t}}$ is a finite set of symbols called *application constants* or *objects names*,

$\mathbf{Obj}_{\mathbf{t}}$ is a finite set of objects (*terms*),

$\mathbf{Frm}_{\mathbf{t}}$ is a finite set of formulas that are true at the moment **t**.

In this definition we postulate the existence of one-to-one mapping

$$\eta_{\mathbf{F},\Delta}: \mathbf{Concepts}_{\Delta} \Longleftrightarrow \mathbf{Cls}$$

that maps application domain concepts to classes of the formalism $\mathbf{F_{Obj}}$. In addition, the following assumptions have been made:

1. A many-to-one mapping **instance_of** that maps object names to class names is defined. We say that the application constant **t** is an instance of class **k**, if and only if **insatance_of(t)=k**. The set of all the application constants of class **k** at the moment **t** is denoted by $\overline{\mathbf{Ext}}_{\mathbf{k},\mathbf{t}}$ and a set of all possible application constants of class **k** is denoted by $\overline{\mathbf{Ext}}_{\mathbf{k}}$.

2. A many-to-one mapping **class:** $\mathbf{Obj}_{\mathbf{t}} \Rightarrow \mathbf{Cls}_2$ that maps objects to names of nonlexical classes is defined. We say that the term **t** is a term of class **k**, if and only if **class(t)=k**. The set of all the objects of class **k** is denoted as $\{\mathbf{Obj}\}_{\mathbf{k}}$. We expect that the terms of class **k** meet the signature $\sigma_{\mathbf{k}}$. Besides, a one-to-one mapping

$$\mathbf{object\_name}:\{\mathbf{Obj}\}_{\mathbf{k},\mathbf{t}} \Leftrightarrow \overline{\mathbf{Ext}}_{\mathbf{k},\mathbf{t}}$$

that maps the objects of t class **k** to the object names and *vice versa,* is defined.

In the formalism $\mathbf{F_{Obj}}$, a class is defined analogously as in the formalism $\mathbf{F_I}$ (see Definition 5).

DEFINITION 15. In the object-oriented formalism $\mathbf{F_{Obj}}$, a signature $\sigma_{\mathbf{k}}$ of the class **k** is the expression

$$\begin{aligned}
&\mathbf{k}[\psi^1_{\mathbf{k}} \overline{\Rightarrow}\ (\delta_1 \mid \text{default: } \mathbf{g}_1); \ldots; \psi^{\mathbf{m}}_{\mathbf{k}} \overline{\Rightarrow}\ (\delta_{\mathbf{m}} \mid \text{default: } \mathbf{g}_{\mathbf{m}});\\
&\psi^{\mathbf{i}_1}_{\mathbf{k}} \rightarrow \mathbf{h}_1; \ldots; \psi^{\mathbf{i}_r}_{\mathbf{k}} \rightarrow \mathbf{h}_{\mathbf{r}};\\
&[T^1_\theta] : \{P^1_\theta\}\xi^1_\theta\theta^1_{\mathbf{k}}@\kappa^1_1, \ldots, \kappa^{\mathbf{I}_1}_1\{Z^1_\theta\} \Rightarrow \kappa_1; \ldots;\\
&[T^{\mathbf{t}}_\theta] :\ \{P^t_\theta\}\xi^t_\theta\theta^{\mathbf{t}}_{\mathbf{k}}@\kappa^1_{\mathbf{t}}, \ldots, \kappa^{I_t}_{\mathbf{t}}\{Z^t_\theta\} \Rightarrow \kappa_{\mathbf{t}};\\
&\mathbf{f}^1_{\mathbf{k}} \Rightarrow (\mathbf{d}_1 \mid \mathbf{default}\ : e_1); \ldots; \mathbf{f}^{\mathbf{n}}_{\mathbf{k}} \Rightarrow (\mathbf{d}_{\mathbf{n}} \mid \mathbf{default:}\ \mathbf{e}_{\mathbf{n}});\\
&\phi^1_{\mathbf{k}} \bullet \mathbf{a}_1; \ldots; \phi^1_{\mathbf{k}} \bullet \rightarrow \mathbf{a}_1;\\
&[T^1_\omega] : \{P^1_\omega\}\xi^1_\omega\omega^1_{\mathbf{k}}@\kappa^1_1, \ldots, \kappa^{\mathbf{z}_1}_1\{Z^1_\omega\} \Rightarrow \kappa_1; \ldots;\\
&[T^{\mathbf{W}}_\omega] :\ \{P^{\mathbf{W}}_\omega\}\xi^{\mathbf{W}}_\omega\omega^{\mathbf{W}}_{\mathbf{k}}@\kappa^1_{\mathbf{W}}, \ldots, \kappa^{\mathbf{Z}_W}_{\mathbf{W}}\{Z^{\mathbf{W}}_\omega\} \Rightarrow \kappa_{\mathbf{W}}; \mid \gamma^1_{\mathbf{k}}; \ldots; \gamma^{\mathbf{s}}_{\mathbf{k}}],
\end{aligned}$$

where

$\mathbf{f}, \phi$ are object attribute names,

ω is an object operation name,

e, **g**, **h** are values,

θ is a class operation name,

κ, σ are class names,

P denotes operation preconditions,

Z denotes operation postconditions,

T denotes triggers,

ξ denotes specifiers *public* and *private*,

γ denotes the integrity constraint.

If an attribute is a multivalued attribute we use $\overline{\Rightarrow}$ $\overline{\Rightarrow}$, $\rightarrow\!\!\rightarrow$, $\Rightarrow\!\!\Rightarrow$ or $\bullet \rightarrow \bullet \rightarrow$ instead of $\overline{\Rightarrow}$, $\rightarrow$, $\Rightarrow$, $\bullet \rightarrow$ respectively.

DEFINITION 16.

i) In the object-oriented formalism $\mathbf{F_{Obj}}$, a class signature is an object molecule.

ii) There exist no other object molecules except that defined in i).

DEFINITION 17.

i) Let $\mathbf{k}, \mathbf{k_1}, \ldots, \mathbf{k_n}$ be class names from **Cls**. Then the statements

$$\begin{aligned}
&\mathbf{k_i} :: \mathbf{k}, \mathbf{i} = 1, \ldots, \mathbf{n},\\
&< \mathbf{k}_1 \ldots, \mathbf{k}_n >_{(\mathbf{t},\mathbf{s})} :: \mathbf{k},\\
&< \mathbf{k}_1 \ldots, \mathbf{k}_n >_{(\mathbf{t},\mathbf{w})} :: \mathbf{k},\\
&< \mathbf{k}_1 \ldots, \mathbf{k}_n >_{(\mathbf{p},\mathbf{s})} :: \mathbf{k},\\
&< \mathbf{k}_1 \ldots, \mathbf{k}_n >_{(\mathbf{p},\mathbf{w})} :: \mathbf{k}
\end{aligned}$$

are intensional molecular formulas.

ii) In the object-oriented formalism $\mathbf{F_{Obj}}$, object molecules are intensional molecular formulas.

iii) There exist no other intensional molecular formulas except those defined in i) and ii).

DEFINITION 18.

i) In the object-oriented formalism $\mathbf{F_{Obj}}$, intensional molecular formulas are intensional.

ii) If **F** and **G** are intensional formulas, then **F**&**G** is an intensional formula.

iii) There exist no other intensional formulas except those defined in i) and ii).

DEFINITION 19.

i) Let $\sigma_{\mathbf{k}}$ be a signature of the class $\mathbf{k}$. Then the object

$$\begin{aligned}\mathbf{e_k}[&\mathbf{f}_{\mathbf{k}}^{\mathbf{1}} \to \mathbf{b}_1; \ldots; \mathbf{f}_{\mathbf{k}}^{\mathbf{n}} \to \mathbf{b_n};\\ &\phi_{\mathbf{k}}^{\mathbf{1}} \to \mathbf{a}_1; \ldots; \phi_{\mathbf{k}}^{\mathbf{r}} \to \mathbf{a_r};\\ &\omega_{\mathbf{k}}^{\mathbf{1}} \to \mathbf{p}_1; \ldots; \omega_{\mathbf{k}}^{\mathbf{w}} \to \mathbf{p_w}],\end{aligned}$$

where $\mathbf{e_k} \in \overline{\mathbf{Ext_k}}$, $\mathbf{b_j} \in \delta_{\mathbf{j}}$, $\mathbf{a_i} \in \delta_{\mathbf{i}}$, $\delta_{\mathbf{i}}, \delta_{\mathbf{j}} \in \mathbf{Cls}$, $i = 1, 2, \ldots, \mathbf{r}$, $\mathbf{j} = 1, 2, \ldots, \mathbf{n}$, $\mathbf{p}_1, \ldots, \mathbf{p_w}$ are values of the object operations, and all the values satisfy integrity constraints $\gamma_{\mathbf{k}}^{\mathbf{1}}; \ldots; \gamma_{\mathbf{k}}^{\mathbf{s}}$ is an object of the class $\mathbf{k}$.

ii) Let $\sigma_{\mathbf{k}}$ be a signature of the class $\mathbf{k}$. Then the object

$$\begin{aligned}\mathbf{k}[&\psi_{\mathbf{k}}^{\mathbf{1}} \to \mathbf{c}_1; \ldots; \psi_{\mathbf{k}}^{\mathbf{m}} \to \mathbf{c_m};\\ &\theta_{\mathbf{k}}^{\mathbf{1}} \to \mathbf{q}_1; \ldots; \theta_{\mathbf{k}}^{\mathbf{t}} \to \mathbf{q_t}],\end{aligned}$$

where $\mathbf{k}$ is class name $\mathbf{c_i} \in \kappa_{\mathbf{i}}$, $\kappa_{\mathbf{i}} \in \mathbf{Cls}, \mathbf{i} = 1, 2, \ldots, \mathbf{m}$, $\mathbf{q}_1, \ldots, \mathbf{q_t}$ are values of the class operations, and all the values satisfy integrity constraints $\gamma_{\mathbf{k}}^{\mathbf{1}}; \ldots; \gamma_{\mathbf{k}}^{\mathbf{s}}$ is an object that represents the class $\mathbf{k}$.

iii) In the object-oriented formalism $\mathbf{F_{Obj}}$, exist no other objects except those defined in i) and ii).

If an attribute is a multivalued attribute, we use the notation $\to\!\!\to$ instead of $\to$.

Now we can compare information modelling and object-oriented modelling formalisms.

1. Application domain entities have descriptive, organisational and operational characteristics. Object-oriented modelling formalisms can model characteristics of all kinds. Information modelling formalisms can model descriptive and organisational characteristics only.

2. Object-oriented modelling formalisms model descriptive and organisational characteristics in a uniform way (by attributes). Information modelling formalisms model descriptive characteristics by attributes and organisational characteristics by relations.

Of course, a strict boundary between information modelling formalisms and object-oriented modelling formalisms does not exist because various hybrid formalisms can be defined.

8. Concluding remarks. This paper analyses the software system engineering discipline and its science. It aims to identify and systematise software engineering paradigms, principles, processes, and objects. An attempt to adapt the notation developed in object-oriented logics (Aït-Kaci and Nasr, 1986; Kifer *et al.*, 1993) to the description of conceptual modelling formalisms has been made.

Using this notation information modelling formalisms and object-oriented formalism have been described and compared.

REFERENCES

Andriole, S.J., and P.A. Freeman (1993). Software systems engineering: the case for a new discipline. *Software Engineering Journal,* **8**(3), 165–179.

Azuma, M., and D. Mole (1994). Software management practice and metrics in the European Community and Japan: some results of a survey. *J. Systems and Software,* **26**, 5–18.

Aï t-Kaci, H., and R. Nasr (1986). LOGIN: a logic programming language with built-in inheritance. *J. Logic Programming,* **3**, 185–215.

Blum, B.I. (1992). *Software Engineering, A Holistic View.* Oxford University Press, New York.

Chen, P. (1976). The entity-relationship model. Toward a unified view of data. *ACM Transactions on Database Systems,* **1**, 9–36.

Coad, P., and E. Yordan (1990). Object-oriented analysis. In R.H. Thayer and M. Dorfman (Eds.), *Tutorial: System and Software Requirements Engineering.* IEEE Computer Society Press, California. pp. 272–289.

Davis, A.M. (1996). Art or engineering, one more time. *IEEE Software,* **13**(6), 4–5.

Denning, P.J., D.E. Comer, D. Gries, M.C. Mulder, A. Tucker, A.J. Turner, and P.R. Young (1989). Computing as a discipline. *Communications of the ACM,* **34**(1), 9–23.

Dictionary of Computing. (1990) Third edition. Oxford, New York: Oxford University Press.

Dijkstra, E.W. (1968). The structure of "THE"-multiprogramming system. *Communication of the ACM,* **11**(5), 341–346.

Fenton, N. (1993). How effective are software engineering methods? *J. Systems and Software,* **22**(2), 141–146.

Fenton, N.E., B. Littlewood and S. Page (1992). Evaluating software engineering standards and methods. In R.H. Thayer and A.D. McGettrick (Eds.), *Software Engineering: An European Perspective.* IEEE Computer Society Press. pp. 463–470.

Flavin, M. (1981). *Fundamental concepts of information modelling.* Englewood Cliffs, N.J: Yourdan Press/Prentice-Hall.

Garlan, D. (1995). Research directions in software architecture. *ACM Computing Surveys,* **27**(2), 257–261.

Glass, R.L. (1995). A structure-based critique of contemporary computing research. *J. Systems and Software,* **28**(1), 3–7.

Hetzel, B. (1993). *Making Software Measurement Work.* QED.

Humphrey, W.S. (1993). Software engineering. In A. Ralston, and E.D.Reilly (Eds.), *Encyclopaedia of Computer Science.* Van Nostrand Reinhold.

Humphrey, W.S. (1994). The personal process in software engineering. *Software Process Newsletter,* **13**(1), 1–3.

IEEE Std 610 (1994). Standard glossary of software engineering terminology. In *IEEE Software Engineering Standards Collection,* 1994 edition, New York.

ISO/IEC DIS 12207–1 (1994). *Information Technology. Software. Part 1: Software Life-Cycle Process.* International Organization for Standardization, International Electrotechnical Commission.

Jackson, M. (1994). Problems, methods and specialisation. *Software Engineering Journal,* **9**(6), 249–255.

Kifer, M., G. Lausen and J. Wu (1993). *Logical Foundations of Object-Oriented and Frame-Based Languages.* Technical Report 93/06, Department of Computer Science SUNY at Stony Brook, Stony Brook, NY 11794.

McDermid (1985). The IEEE and software engineers. *Software&Microsystems* , **4**(2), 45–48.

Morrison, J., and J.F. George (1995). Exploring the software engineering component in MIS research. *Communications of the ACM,* **38**(7), 80–91.

Parnas, D.I., P.C. Clements and D.M. Weiss (1985). The modular structure of complex systems. *IEEE Transactions on Software Engineering,* **SE-11**, 259–266.

Ramamoorthy, C.V., A. Prakash, W.T. Tsai, and Y. Usuda (1984). Software engineering: problems and perspectives. *Computer,* October 191–209.

Reddy, S (1994). Interview with Stephen R. Schach. *Crossroads,* **1**(2), http://info.acm.org/crossroads/xrds1-2/schach.html.

Rubin, K.S., and A. Goldberg (1992). Object behaviour analysis. *Communication of the ACM,* **35**(9), 48–62.

Sailor, J.D. (1990). System engineering overview. In R.H. Thayer and M. Dorfman (Eds.), *Tutorial: System and Software Requirements Engineering.* IEEE Computer Society Press, Washington.

Schlaer, S., and S. Mellor (1988). *Object-oriented System Analysis: Modelling the World in Data.* Prentice-Hall, Englewood Cliffs, N.J.

Tichy, W.F., N. Habermann, and L. Prechelt (1993). Summary of the Dagstuhl workshop on future directions in software engineering. *ACM SIGSOFT Software Engineering Notes,* **18**(1), 35–48.

Thayer, R.H., W.W. Royce (1990). Software system engineering. In R.H. Thayer and M. Dorfman (Eds.), *Tutorial: System and Software Requirements Engineering.* IEEE Computer Society Press, Los Alamitos, California. pp. 77-116.

Thayer, R.H., and M.C. Thayer (1990). Glossary. In R.H. Thayer and M. Dorfman (Eds.), *Tutorial: System and Software Requirements Engineering* IEEE Computer Society Press, Loss Alamitos, California. pp. 605–676.

Wasserman, A.I (1996). Toward a discipline of software engineering. *IEEE Software,* **13**(6), 23–31.

Received November 1996

A. Čaplinskas is head of the Software Engineering Department at the Institute of Mathematics and Informatics in Vilnius, teaches at the Vilnius University and Vilnius Gediminas Technical University. He graduated from the Moscow Lomonosov State University (Faculty of Mathematics) in 1966. Prior to joining the Institute he spent four years on the research staff at the Vilnius University. In 1970 he joined the Institute and has been concerned with information system development ever since. From 1976 to 1990, he was the leader of the project VILNIUS which was aimed at the development of the CASE- system for knowledge-based software systems. Čaplinskas was the author or a co-author four books and over sixty papers in information system engineering, software engineering, knowledge-based systems and related areas. His current interests include legislative engineering, national information infrastructures, software system engineering and knowledge representation.

PROGRAMŲ SISTEMŲ INŽINERIJOS ANALIZĖ

Albertas ČAPLINSKAS

Straipsnyje analizuojama programų sistemų inžinerija kaip savarankiška inžinerijos mokslo šaka. Bandoma išryškinti ir sistematizuoti tos mokslo šakos paradigmas ir principus bei jos nagrinėjamus procesus ir objektus. Siūloma programų sistemų inžinerijoje naudojamiems koncepcinio modeliavimo formalizmams aprašyti pritaikyti moderniojoje logikoje sukurtą objektinių logikų aprašymo notaciją. Naudojant tokią notaciją, formalizuojami ir lyginami informacinio ir objektinio modeliavimo formalizmai.