The Strong Object Invariant

Dušan Malbaški [1], Aleksandar Kupusinac [1]

[1] Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia

***Abstract* – The concept of an invariant is fundamental to object-oriented programming, because it provides information on the overall behaviour of the class and/or its objects. An invariant is a predicate, that is true in every state that is proclaimed as valid. A strong invariant is a predicate, that is true in every valid state and false in every invalid state. Basically, we can divide them into two categories: object invariants and class invariants. Object invariants describe the consistency of object, i.e. non-static fields. Analysis of invariants takes the most important place in object-oriented program verification and can be directed in two ways – *as prescribed* and *as described*. This paper considers both analyses which are based on the strongest dynamic postconditions of methods with the guard as the precondition, thus, determining all possible transitions and only them. In addition, since dynamic postconditions are logical functions of the initial-final states, our solution is based solely on the first-order predicate logic.**

***Keywords* –** Invariants, Object-oriented programming, Program correctness, Program verification.

1. Introduction

Object-oriented programming has been created through evolution, as a response to the software market requests. Therefore, already during the development of theory and practice the factual situation was established in which there are different views on the fundamental concepts of object-oriented methodology – class, object and invariant. The terminology used in object-oriented programming is not satisfactory, because in addition to synonymy, it often meets reckless use of the entrenched terms from other fields of science (especially philosophy). The main problem with the definitions in object-oriented programming is that they all have a free form not allowing the object and the class to have independent definitions. In this paper we explain the conceptual definitions of class, object and invariant [1] that are based on concepts. By definition, a concept is the thought on the essential characteristics of the unit of observation [2]. With this approach, we can clearly explain what is the class invariant, and what is object invariant.

Analysis of invariants is a central point when it comes to analyzing the semantics of class or semantics of object-oriented programs [3]. The importance of invariants in the analysis of the class semantics is clearly described by Bertrand Meyer [4]: "To me the notion of the invariant is one of the most illuminating concepts that can be learned from the object-oriented method. Only when I have derived the invariant (for a class that I write) or read and understood it (for someone else's class) do I feel that I know what the class is about.". Invariant is every predicate that is true in any valid state, while may or may not be true in an invalid state. In particular, the predicate that uniquely separates the set of valid and set of invalid states, we call the **strong invariant**. In other words, a strong invariant is any predicate that is true in every valid state and false in every invalid state. However, since all the strong invariants uniquely separate sets of valid and invalid states, we conclude that all strong invariants are uniform to the level of equivalence [5].

Based on conceptual definitions, invariants in the class are basically divided into object and class invariants. Invariant property means that all the methods are activated by one and only one object-representative (i.e. ***this***) and describes all valid states in which that object can be found. Analysis of invariants can be directed in two ways:

- *as prescribed* – invariant is given in advance and we prove the class or object correctness with respect to the given invariant [6],
- *as described* – we assume that the object or class behaves correctly and based on that we derive the invariant [7].

In this paper we will consider both analyses of object invariants based on the strongest dynamic postconditions of methods with the guard as the precondition, provided that it determines all reachable states and only them. In the Section 2 we consider conceptual definitions of class, object and invariant. Basic elements of the analysis of invariants are given in the Section 3. In the Sections 4 and 5 we consider *as prescribed* and *as described* analyses, respectively.

2. Conceptual definitions

During the design of an information system, an object-oriented developer would spend most of his time for modelling. However, it should be noted here that a developer deals with thoughts, not with actual

participants of the information system. Therefore, our presentation will begin with a definition of the concept:

- The **concept** is the thought of essential characteristics of the unit of observation [2],

where the term *unit of observation* should be understood in the broadest sense, as a unit of observation and/or thinking. Characteristics of the unit can be divided into essential and inessential. The thought of any characteristic is called **property**, and the thought of the essential characteristic is called the **essential property** [2]. Inessential properties can be derived from the essential. For example, properties of the concept TRIANGLE are *being a convex polygon* and *having three sides*, while *the altitudes equality* in an equilateral triangle is not an essential property, because it follows from the equality of its sides.

Concepts can be classified in various ways, but for our consideration, the most important is a division between **individual** and **class concepts**. Individual concepts refer to individual units of observation. Individual units that have common properties constitute a class, and the thought of a given class is the class concept. Thus, individual concepts HAYDN, MOZART and BEETHOVEN are the thoughts of famous composers who have in common that they belong to the period of classicism, and that their work is more or less related to Vienna. Based on common properties, they constitute a class, and thought of this class is the class concept VIENNA CLASSICISTS. Obviously, individual concepts may or may not be realistic, while the class concepts are not. For example, the class concept VIENNA CLASSICIST is not realistic, while the individual concept MOZART is realistic (composer Mozart existed in the period 1756-1791 yr.). However, both individual and class concepts TRIANGLE are not realistic. The more abstract class concepts can be obtained from class concepts by the method of abstraction, thus creating a hierarchy of classes, which has a tree structure. For instance, from the class concept VIENNA CLASSICIST, we can obtain more abstract class concept COMPOSER, and from this more abstract class concept ARTIST etc.

It is customary to begin learning object-oriented programming with a consideration of the term **entity**. By ISO definition, an entity is any concrete or abstract thing that exists, that existed or could exist, including connections between these things. We prefer to treat entity as a synonym for the unit of observation and find that there is no need to put it to the forefront, because as already stated, information system developer deals with concepts, i.e. thoughts.

Consideration of all essential properties of a concept in general is often a difficult philosophical question, since it is not an easy task do decide whether some feature is essential or not. In order to avoid such difficulties we stick to the problem domain. In contrast to the logician, the software developer does not have to take into account all essential properties, but just those that are relevant, i.e. those that are of interest for a given problem domain. For example, designer of information systems for tax administration for the concept CITIZEN will choose properties *name*, *identification number*, *address*, *data on income* etc., but certainly will not choose the properties *height* and *weight*, although every citizen has them. On the other hand, the designer of information systems for health care institutions, next to the *name*, *identification number*, *address*, etc., will safely select the *height* and *weight*, because these properties are essential in determining the patient treatment, while dropping *data on income*. Let us introduce the following definitions:

- Concept properties that are of interest in a given problem domain are called relevant properties [5], [8].
- Software modelling is a procedure of selecting concept features that are *relevant* to a given problem domain. The appropriate result is called software model [1].

Now, we can define a class:

- The class is a software model of the class concept and it has identity, state and behaviour.

The class concept properties are divided into two groups: **features** and **fragments**. The class concept may also contain other class concepts, and we call them fragments. For instance, class concept CAR has the feature *colour* and the fragment MOTOR. The features can be descriptive, such as *colour*, *weight*, etc., and operational as well as the *possibility of movement*, *ability to fly* and so on. Now we present the conceptual definition of the object:

- The **object** is a software model of the individual concept and it has identity, state and behaviour.

These definitions have two major advantages over the various definitions that can be found in the literature on object-oriented programming. Firstly, conceptual definitions are based on concepts, i.e. the well-elaborated and clear system of terms and their meanings. Secondly, the conceptual definitions of class and object are equal in the semantic sense, i.e. class is not defined over the object or vice versa.

Obviously, a conceptual aspect of class and object puts them in equal position. The difference is that the class corresponds to a class concept, and the object to an individual concept. Classes and objects are interrelated by an assumption, which actually has the power of a postulate:

- For every object there exists a class that has all of its relevant properties; we say that the object belongs to that class.

For example, each object *isosceles triangle* (with specific side length values) belongs to the class *Isosceles triangle*. Fragments and features of the class appear in the class case, while the fragments and features of the individual object appear in the individual case. For instance, if the class feature is *colour*, then the individual feature appears as *white* or *green*.

The concepts can be complex which means that their fragments themselves may have fragments, and so on. A book consists of chapters, chapters contain paragraphs, paragraphs lines and lines contain characters. The **concept structure** is the set consisting of the concept itself, its fragments, and further their fragments, etc., all moderated by the relation *to be a fragment*. If the concept does not include fragments, but only features, we say that it has a simple structure. If it contains at least one fragment, then the concept has a complex structure. For example, the concept CAR contains fragment MOTOR, so it has complex structure. The concept POINT contains only features, such as coordinate values, so we say that it has a simple structure. Since the object models an individual concept, by modelling the concept structure we get the object structure. Similarly, we can talk about simple and complex object structures.

The essential characteristics of an object are that it has the identity, state and behaviour [9]. The conceptual definition is not in conflict with these essential characteristics. Indeed, since each individual concept has identity that uniquely defines it, this property is mapped on the object, as its model, and each object has identity that uniquely defines it. The state of an object with a simple structure determines its descriptive features. The state of an object with complex structure determines its descriptive features, as well as states of its fragments. If we now consider this conclusion using a different criterion, i.e. if we firstly look the object state set, then it follows that the descriptive features are derived from the state, i.e. they are functions of the state. In the implementation phase the data-members represent descriptive features, while the object-members represent fragments. Operational characteristics determine the object behaviour, which is described by its methods. By the definition of the object, it logically follows that the objects of the same class have the same structure and same behaviour. Class and object are related exactly as the data type and variable in standard programming languages. For example, *int* variables are occurrences (instances) of *int* type and objects are occurrences (instances) of their class.

Among all the concept features, both descriptive and operational, some never change: they constitute an **invariant**. For the concept TRIANGLE, no matter whether it is an individual or class concept, feature $a+b>c$ is an invariant, where a, b and c are the descriptive features that represent side lengths, and feature $\alpha+\beta+\gamma=180^{\circ}$, where α, β and γ are the descriptive features that represent the values of internal angles. Obviously such an invariant holds, regardless of the changes (transformations) on the concept. Apparently, there are infinitely many invariants, all of which together build the **concept essence**.

Modelling assumes the choice of the finite set of relevant features, both descriptive and operational. At the same time, it implies the choice of invariant features that are relevant to the particular model. Such features are called **invariant relevant features**. All invariant relevant features describe the concept essence in a given problem domain. Above mentioned features $a+b>c$ and $\alpha+\beta+\gamma=180^{\circ}$ are invariant. However, if the triangle is modelled only by its side lengths a, b and c, then $a+b>c$ is the relevant invariant feature, while $\alpha+\beta+\gamma=180^{\circ}$ is not. In the implementation phase we obtain an object or class with a finite set of abstract states. Even than, invariant relevant features remain valid in the form of restrictions adopted on the final set of abstract states. Let us now define the invariant in an object and a class:

- **Invariant in the object** is the restriction of invariant relevant feature of the appropriate individual concept defined on the set of abstract states [1].
- **Invariant in the class** is the restriction of invariant relevant feature of the appropriate class concept defined on the set of abstract states [1].

Based on the fact that the invariant feature can be descriptive as well as operational, after modelling, we get several types of invariants [7]:

- **Field invariants** - come from the relevant invariant descriptive features, and after modelling represent the relation over the class fields. For example, in the previous example,

$a+b>c$ is a field invariant, where a, b and c are the class fields.

- **Functional invariants** - come from the operational invariant relevant features, and after modelling connect methods in sense of implementation. For example successive application of the methods *push* and *pop* leaves the stack in the same state.
- **Relationship invariants** - a typical example is the relationship cardinality between concepts, e.g. relationship cardinality between concepts TRIANGLE and VERTEX is 1:3.
- **Mixed forms**.

In addition, there are invariants that are defined in the implementation phase, although they may not be in the problem domain. For example, a sequential stack with capacity C is restricted to C or less elements, while for the linked stack such limitation does not exist.

Consider now an object with a simple structure containing only the data-members of some type. Say, if data-member a (the triangle side length) is of the type *real number*, the question is - does this type matches the corresponding relevant concept feature? Although everything seems logical, unfortunately, the answer is negative, because the length of the triangle side may be of the type *length*, only. Such type does not exist in any programming language, but we have to improvise and say that the data-member is of the type *positive real number*. However, it does not end the problem, because the question is - can any three positive real numbers represent the side lengths of a triangle? Again the answer is negative, because these three numbers must satisfy the triangle inequality theorem (the sum of the lengths of any two sides of triangle is greater that the length of the third side). In other words, the invariant relevant feature of the concept TRIANGLE must be valid. It is now abundantly clear problem - on the one hand we have requests coming from the nature of the concept feature, and on the other hand we have the reality imposed by the descriptive power of programming languages. Therefore, it is fully justified to divide the final set of abstract states into two disjoint subsets, which are sets of **valid** and **invalid** states. For example, when it comes to a triangle, state (3, 4, 5) is valid, while states (-3, -4, -5) and (5, 1, 1) are invalid.

3. Basic elements of the analysis of object invariants

Let the class K contain only non-static fields $\phi_1, \phi_2, ..., \phi_n$. To the class K we assign a finite nonempty set of abstract states U_K. The set of all fields of the class K is denoted by Φ_K. Assuming that there is at least one field in the class it follows that $\Phi_K \neq \emptyset$, so $U_K \neq \emptyset$.

Definition 1. (Abstract state space) *The abstract space of the class K is a nonempty finite set of abstract states U_K that is assigned to the class K.*

The abstract state space U_K we divide into two disjoint parts, namely the sets of *valid* states V_K and *invalid* states N_K. The set of valid states V_K is assumed to be nonempty. Null-state o is a valid (quasi) state that an object occupies before its construction or after its destruction, i.e. when $this = null$, where $this$ stands for the representative object. The set of all methods in the class K is denoted by M_K provided that $M_K \neq \emptyset$.

Definition 2. (Invariant) *Let the class K contain only non-static fields. An invariant in the class K is every predicate I_K defined over its abstract state space U_K that is true in every valid state.*

Definition 3. (Strong invariant) *Let the class K contain only non-static fields. Strong invariant in the class K is any predicate IS_K defined over U_K with the properties:*

1.) IS_K is an invariant in the class K,
2.) If I is an invariant in the class K then $\forall s \in U_K, IS_K(s) \Rightarrow I(s)$.

Note that the invariant must be true in all valid states, whilst in invalid states it may or may not. In the class K a strong invariant always exists and it is unique up to the level of equivalency [7]. Strong invariant completely defines the set of valid states, i.e. $V_K = \{s \mid s \in U_K, IS_K(s)\}$.

Hoare triples [10], [11] are two special formulas of the first-order predicate calculus [12] defined over the abstract state space U_K. We introduce dynamic form of the total correctness formula (*DTCF*) in the object oriented environment:

$$\forall s[P(s) \Rightarrow \exists s' m(s,s') \wedge \forall s'(m(s,s') \Rightarrow \hat{Q}(s,s'))]$$

shortly denoted by $\{P\}m\{\hat{Q}\}$, where the postcondition $\hat{Q}(s,s')$ is now dynamic [13], meaning that it is a function of initial and final states, thus representing the

transition function. In order to avoid confusion we will denote dynamic postconditions by the sign " $\wedge$ ". The structures of static and dynamic total correctness formulas are the same. The predicate $\hat{Q}(s,s')$ can be formally replaced by the predicate $R(s')$ such that $\forall s \forall s' \hat{Q}(s,s') \Leftrightarrow R(s')$, so the mathematical apparatus developed in the scope of static postconditions may be used for dynamic without any restrictions, e.g. all general laws of Hoare logic hold in dynamic environment as well, such as the laws of consequence, conjunction, disjunction etc. The interpretation of formula $\{P\}m\{\hat{Q}\}$ is the following – if the predicate P is true in some state then the method m terminates from that state and the transition will occur that satisfies the dynamic predicate $\hat{Q}$. The set of all states from which a method terminates is described by a special predicate called *guard* [14], [15].

Definition 4. (Guard) *The guard of the method* m *denoted by* $\Gamma(m)$ *is a predicate with the following properties:*

1.) $\{\Gamma(m)\}m\{\top\}$, *i.e. if the precondition* $\Gamma(m)$ *is satisfied than the method* m *must terminate,*

2.) $\{P\}m\{\top\} \Rightarrow \forall s(P(s) \Rightarrow \Gamma(m)(s))$.

Definition 5. (Strongest dynamic postcondition) *The strongest dynamic postcondition of the method* m *with respect to the precondition* P *is predicate* $s\hat{d}p(m,P)$ *if:*

1.) $\{P\}m\{s\hat{d}p(m,P)\}$,

2.) $\{P\}m\{\hat{Q}\} \Rightarrow \forall s \forall s'(s\hat{d}p(m,P)(s,s') \Rightarrow \hat{Q}(s,s'))$.

The most important case is the strongest dynamic postcondition with the guard Γ as the precondition. Let m be a method of the class K. The dynamic predicate $s\hat{d}p(m,\Gamma(m))$ determines all valid transitions that can be accomplished by m and only them [13].

4. As Prescribed Analysis

In the *as prescribed* analysis invariant is given in advance and we prove the class or object correctness with respect to the given invariant [6].

Definition 6. (Class correctness) *Class* K, *with strong invariant* IS_K, *is correct iff:*

$$\{\Gamma(m) \wedge IS_K\}m\{IS_K\},$$

for all methods $m \in M_K$.

Theorem 1. (*As prescribed* analysis) *Let* IS_K *be strong invariant of the class* K. *If*

$$s\hat{d}p(m,\Gamma(m) \wedge IS_K) \Rightarrow IS_K,$$

where $m \in M_K$, *then class* K *is correct.*

Proof.

Let $m \in M_K$ *and*

$$s\hat{d}p(m,\Gamma(m) \wedge IS_K) \Rightarrow IS_K. \quad (1)$$

Based on property (1) from the Definition 5, we infer

$$\{\Gamma(m) \wedge IS_K\}m\{s\hat{d}p(m,\Gamma(m) \wedge IS_K)\}. \quad (2)$$

Based on the second Hoare law of consequence, by (1) and (2) we obtain

$$\{\Gamma(m) \wedge IS_K\}m\{IS_K\}$$

Q.E.D.

Now, let us consider the following class written in Java:

```
public class K {
  private int n;
  public K() { n=0; }
  public void m() {
    n++;
    if(n>10)
       throw new RuntimeException();
  }
}
```

with the strong invariant:

$$IS_K \equiv this = null \vee (this \neq null \wedge 0 \leq n \leq 10).$$

Based on the class K, we obtain:

$$\Gamma(K) \equiv this = null,$$
$$\Gamma(m) \equiv this \neq null \wedge n \leq 9,$$
$$s\hat{d}p(K,\Gamma(K)) \equiv this = null \wedge this' \neq null \wedge n' = 0,$$
$$s\hat{d}p(m,\Gamma(m)) \equiv this \neq null \wedge this' \neq null \wedge n \leq 9 \wedge n' = n+1$$

Since

$$\Gamma(K) \wedge IS_K \equiv this = null,$$
$$\Gamma(m) \wedge IS_K \equiv this \neq null \wedge 0 \leq n \leq 9,$$

we obtain

$$s\hat{d}p(K,\Gamma(K) \wedge IS_K) \equiv this = null \wedge this' \neq null \wedge n' = 0$$

,

$$s\hat{d}p(m,\Gamma(m) \wedge IS_K) \equiv this \neq null \wedge this' \neq null \wedge 0 \leq n \leq 9 \wedge n' = n+1$$

i.e.

$s\hat{d}p(K, \Gamma(K) \wedge IS_K) \Rightarrow IS_K$,
$s\hat{d}p(m, \Gamma(m) \wedge IS_K) \Rightarrow IS_K$

and we conclude that class K is correct.

5. As Described Analysis

In the *as described* analysis we assume that the object or class behaves correctly and based on that we derive the invariant [7]. Let the predicate Z_0 describe some valid states of the class K and only them. Let us form the following recursive sequence of predicates:

Algorithm 1.

$$Z_1 \equiv Z_0 \vee s\hat{d}p(m_1, Z_0) \vee s\hat{d}p(m_2, Z_0) \vee \ldots \vee s\hat{d}p(m_n, Z_0),$$
$$Z_2 \equiv Z_1 \vee s\hat{d}p(m_1, Z_1) \vee s\hat{d}p(m_2, Z_1) \vee \ldots \vee s\hat{d}p(m_n, Z_1),$$
$$Z_3 \equiv Z_2 \vee s\hat{d}p(m_1, Z_2) \vee s\hat{d}p(m_2, Z_2) \vee \ldots \vee s\hat{d}p(m_n, Z_2),$$
$\ldots$

where $m_1, m_2, \ldots, m_n \in M_K$. Consequently,

$$Z_0 \subseteq Z_1 \subseteq Z_2 \subseteq \ldots.$$

Let $Z = \{Z_0, Z_1, Z_2, \ldots\}$. The set Z is a complete chain since each if its subsets has the greatest lower bound and the least upper bound. Now, consider the mapping $\sigma : Z \to Z$ of the form

$$\sigma(z) = z \vee s\hat{d}p(m_1, z) \vee s\hat{d}p(m_2, z) \vee \ldots \vee s\hat{d}p(m_n, z),$$

where $z \in Z$. Obviously, it follows that

$$Z_{i+1} \equiv \sigma(Z_i),\ i = 0, 1, 2, \ldots.$$

Lemma 1. The function σ is monotonic on the chain Z.

Proof.
Suppose that $x \Rightarrow y$, where $x, y \in Z$. Let $x \equiv Z_j$, then $y \equiv Z_k$, where $k \geq j$. According to the property $Z_{i+1} \equiv \sigma(Z_i)$, $i = 0, 1, 2, \ldots$, we obtain $\sigma(x) \equiv Z_{j+1}$ and $\sigma(y) \equiv Z_{k+1}$ and conclude that $Z_{j+1} \Rightarrow Z_{k+1}$, i.e. $\sigma(x) \Rightarrow \sigma(y)$. Based on that, we conclude that the function σ is monotonic on the chain Z.

Q.E.D.

Lemma 2. The function σ has the least fixpoint on the chain Z.

Proof.
By Tarski theorem [16] and Lemma 1, the function σ has a least fixpoint Z_{fix} on the complete chain Z, for which $\sigma(Z_{fix}) \equiv Z_{fix}$.

Q.E.D.

Further, if Z_{fix} is the least fixpoint (Lemma 2) then obviously

$$(\forall j \geq fix)\ Z_j \equiv Z_{fix},$$

where $fix, j \in \{0, 1, 2, \ldots\}$. For the sake of simplicity we introduce the function $fixp$:

$$Z_{fix} \equiv fixp(Z_0, \sigma).$$

Theorem 2. (*As described* analysis) The predicate $Z_{fix} \equiv fixp(O, \sigma)$ is a strong object invariant in the class K, where $O \equiv this = null$ stands for null-predicate.

Proof.
The predicate $Z_0 \equiv O$ describes null-state and only it. By definition null-state is valid, so all states reachable from it after fix transitions are also valid. Thus the left to right implication holds, i.e. $\forall s\, Z_{fix}(s) \Rightarrow IS_K(s)$. Now, suppose that t is a valid state such that $Z_{fix}(t) \equiv \perp$. That means that the state t is not reachable after fix transitions when starting in null-state described by Z_0. But then t can not be valid, so right to left implication also holds, i.e. $\forall s\, IS_K(s) \Rightarrow Z_{fix}(s)$. Since both implications hold, the equivalence $\forall s\, Z_{fix}(s) \Leftrightarrow IS_K(s)$ also holds and that proves the theorem.

Q.E.D.

Now, let us consider the previous Java class:

```
public class K {
  private int n;
  public K() { n=0; }
  public void m() {
    n++;
    if(n>10)
```

```
      throw new RuntimeException();
  }
}
```

Based on the class K, we obtain:

$$\Gamma(K) \equiv this = null,$$
$$\Gamma(m) \equiv this \neq null \wedge n \leq 9,$$
$$\hat{sdp}(K, \Gamma(K)) \equiv this = null \wedge this' \neq null \wedge n' = 0,$$
$$\hat{sdp}(m, \Gamma(m)) \equiv this \neq null \wedge this' \neq null \wedge n \leq 9 \wedge n' = n + 1$$

.

We adopt that $Z_0 \equiv this = null$ and form the following recursive sequence of predicates:

$$Z_1 \equiv Z_0 \vee (this \neq null \wedge n = 0),$$
$$Z_2 \equiv Z_1 \vee (this \neq null \wedge n \in \{0, 1\}),$$
$$\ldots$$
$$Z_{11} \equiv Z_{10} \vee (this \neq null \wedge n \in \{0, 1, \ldots, 10\}),$$
$$Z_{12} \equiv Z_{11} \vee (this \neq null \wedge n \in \{0, 1, \ldots, 10\}).$$

Since $Z_{11} \equiv Z_{12}$ we conclude that the predicate Z_{11} is a fixpoint and

$$IS_K \equiv this = null \vee (this \neq null \wedge 0 \leq n \leq 10)$$

is the strong object invariant.

6. Conclusions

Conceptual definitions, given in this paper, allow independent consideration of class and object characteristics. From this conclusion, it is possible to clearly classify invariants in the class, which is the basis for the analysis of invariants. Analysis of class invariants has a central place in the class semantics, and therefore in the semantics of object-oriented programs. Analysis of invariants can be *as prescribed*, where the invariant is given in advance and *as described*, where the invariant is derived from the class. In this paper we have presented both analyses of object invariants that are based on the strongest dynamic postconditions of methods with the guard as the precondition.

Acknowledgements

This work was partially supported by the Ministry of Science and Education of the Republic of Serbia within the projects: ON 174026 and III 044006.

References

[1] Malbaški, D. and Kupusinac, A : Classification of Invariants in Class Based on Conceptual Definitions, *15th International Scientific Conference on Industrial Systems (IS'11)*, Novi Sad, Serbia, 14. – 16. September, 2011.

[2] Petrović, G. : *Logic*, (*in Serbian*), Školska knjiga, Zagreb, Yugoslavia, 1981.

[3] Logozzo, F. : Class invariants as abstract interpretation of trace semantics, *Computer Languages, Systems & Structures*, Elsevier, vol. 35, num. 2, pp. 100-142, July 2009.

[4] Meyer, B. : *Object-Oriented Software Construction*, (*2nd edition*), Prentice Hall, 1997.

[5] Kupusinac, A. and Malbaški, D. : Class invariant in the object-oriented programming and its application, *(in Serbian)*, In *Proc. 15th TELFOR*, Belgrade, Serbia, 20-22. November, 2007, pp. 589-592, ISBN: 978-86-7466-301-1.

[6] Kupusinac, A. and Malbaški, D. : General Aspects of the *As Prescribed* Analysis of Invariants in the Class, In *Proc. 19th TELFOR*, Belgrade, Serbia, 22-24. November, 2011, pp. 1379-1381, ISBN: 978-1-4577-1498-6.

[7] Malbaški, D. : The invariants in the class, *(in Serbian)*, Faculty of Technical Sciences, Novi Sad, Serbia, Tech. Rep. 021-21/24, 2010.

[8] Kupusinac, A. : Class invariant in the object-oriented programming, MSc thesis, *(in Serbian)*, Faculty of Technical Sciences, Novi Sad, Serbia, 2008. (Mentor: Prof. dr Dušan Malbaški)

[9] Booch, G. : *Object-Oriented Analysis and Design, (2nd edition)*, Addison-Wesley, 1994.

[10] Hoare, C. A. R. : An Axiomatic Basis for Computer Programming, *Comunications of the ACM*, vol. 12, num. 10, pp. 576-585, October 1969.

[11] Gordon, M. J. C. : *Programming Language Theory and its Implementation*, Prentice-Hall, 1988.

[12] Hoare, C. A. R. and Jifeng, H. : *Unifying Theories of Programming*, Prentice-Hall, 1998.

[13] Kupusinac, A. : Analysis of Characteristics of Dynamic Postconditions in Hoare Triples, Ph.D. thesis, *(in Serbian)*, Faculty of Technical Sciences, Novi Sad, Serbia, 2010. (Mentor: Prof. dr Dušan Malbaški)

[14] Dijkstra, E. W. : Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Comunications of the ACM*, vol. 18, num. 8, pp. 453–457, August 1975.

[15] Dijkstra, E. W. : *A Discipline of Programming*. Prentice-Hall, 1976.

[16] Tarski, A. : Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Pacific Journal of Mathematics*, vol. 5, pp. 285-309, 1955.

Corresponding author: Aleksandar Kupusinac
Institution: Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
E-mail: sasak@uns.ac.rs