Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 116 (2005) 171–184
www.elsevier.com/locate/entcs

Towards Testing and Analysis of Systems that Use Serialization

Giovanni Denaro and Leonardo Mariani

Università degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Via Bicocca degli Arcimboldi, 8
I-20126 - Milano, Italy
{denaro, mariani} @disco.unimib.it

Abstract

Object serialization facilitates the flattening of structured objects into byte streams and is therefore important for all component-based applications that strongly rely on data-exchange among components. Unfortunately, implementing and controlling the serialization mechanisms may expose the software to subtle faults. This paper paves the way towards testing and analysis techniques specifically tailored to the assessment of software that uses serialization. In particular, we introduce a taxonomy of abstractions and terms to semantically characterize and classify the main data-exchange cases, which serialization can be involved with. The resulting conceptual framework provides a means to forecast how erroneous implementations of serialization would look like in different cases, thus enabling the focusing of testing and analysis techniques to address serialization-related faults.

Keywords: object serialization, testing, software analysis, data exchange

1 Introduction

Component-based software applications are increasingly popular in several application domains. In the general setting, these applications are made out of a set of both independently developed and independently deployed software components that accomplish a common goal by coordinating and exchanging data in either distributed or local environments. High level communication

[1] This work has been partially founded by the Italian Government in the context of the QUACK project (QUACK: A Platform for the Quality of New Generation Integrated Embedded Systems.)

doi:10.1016/j.entcs.2004.02.075

mechanisms tend to facilitate the interactions among components. In particular, it is often extremely useful that the exchanged data appear as *objects*, i.e., instances of abstract data types with their run-time state. For example, in distributed component technologies, component methods can be invoked remotely with object parameters that are passed across process and network boundaries [1]. Similarly, in mobile code technologies, components (a.k.a. executing units) can migrate through the network along with sets of local data objects [2]. As a further example, a common implementation of data persistence is achieved by hibernating and resuming objects into and from the file system. However, although objects may have complex structures, in the general case the data-exchange takes place through low-level *ducts*, which support just a byte-stream abstraction, such as, files and TCP sockets.

Object serialization allows to bridge the gap, facilitating the flattening of objects into byte streams. When an object is serialized, both the encapsulated values and types are flattened in the target byte stream with sufficient information to insure that the equivalent typed object can be later recreated; referenced objects (if any) are recursively serialized as well. Deserialization is the symmetric process of recreating the object (or the graph of objects) from the serialized representation. Programming languages such as Java and C#, which are widely used for implementing component-based software, provide built-in primitives for serializing (deserializing) objects and mechanisms for controlling the serialization process to some extent [12,13]. As an example of these latters, Java allows to explicitly specify object references that must be ignored during serialization, thus preventing some parts of an object graph from being serialized when the root object is serialized. An analogous facility is available in C#. Controlling serialization is crucially useful in many practical situations. Interestingly, Ghezzi, Martena and Picco describe a method for optimizing the performance of remote method invocations based on controlling serialization: their method reduces the network overhead by pruning off unused subgraphs of the objects passed as parameters [3].

Unfortunately, changing the default behavior of serialization exposes the software to subtle faults. Consider for example the case in which two components exchange an object and this involves serialization. Subtle faults and failures may show up if the receiving component assumes that the object structure is as defined in the sender component, but instead the structure was modified during serialization. The first part of this paper reports a number of sample cases of software faults that can be accounted to the use of serialization. However, to the best of our knowledge, in the current research and industrial practice there is lack of testing and analysis techniques that address correctness of the software in presence of serialization. This motivates our research on testing and analysis techniques specifically tailored to the assessment of software that uses serialization.

As a first milestone towards this goal, the main contribution of this paper is the definition of a conceptual framework for reasoning about serialization. We introduce a taxonomy of abstractions and terms to semantically characterize and classify the main data-exchange cases, which serialization can be involved with. We draw the link between the identified semantic cases and the linguistic support for serialization. Our conceptual framework provides a means to forecast how erroneous implementations of serialization would look like in different cases. This enables focusing of testing and analysis techniques to address serialization-related faults. As preliminary evaluation of our research, we sketch how the defined conceptual framework may facilitate the adaptation of the ideas of traditional data-flow testing [6,5] for verifying serialization. We illustrate the framework and the examples referring to Java, but we believe that the ideas can be easily ported to other programming language.

Part of the work presented in this paper relates to the concepts presented by Fuggetta, Picco and Vigna in [2]. Referring to mobile code systems, these authors distinguish mobility mechanisms for three elements of a program: code, execution state and data space. In particular, data mobility involves transfer of structured data across computing environments and is often practically accomplished by means of serialization [10]. Our taxonomy of data-exchange cases can be regarded to as an extension of the data mobility cases presented in [2].

The paper is organized as follows. Section 2 illustrates some sample software faults that can be accounted to the use of serialization. Section 3 proposes a conceptual framework for reasoning on testing and analysis techniques for serialization. Section 4 exemplifies how our framework may support the definition of dataflow-based testing of serialization related faults. Section 5 summarizes the contributions of this paper and sketches the future agenda of our research.

2 Serialization Faults in Java

Java provides primitives and mechanisms for using and controlling serialization [4]. For objects to be serializable, they are required to implement the interface `Java.io.Serializable`. This interface just works as a marker of serializable objects, while it does not contain any further definition. A serializable object can be flattened into a stream using the method `void writeObject(Object)` of the class `ObjectOutputStream` and serialized objects can be recreated from their flattened representations using the method

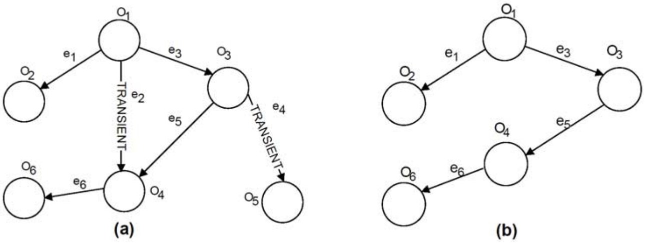

Fig. 1. (a) represents the structure of the object O_1, while (b) is the serialized version of O_1

`void readObject(Object)` of the class `ObjectInputStream`. By default, when an object is serialized, its whole object graph is serialized, i.e., all directly and indirectly referenced objects are recursively serialized as well. The default behavior of serialization can be altered by marking object references as `transient`, thus interrupting the recursion in the a branch of the object graph, or redefining the methods `readObject` and `writeObject` for specific classes of objects. Serialization can be the source of subtle faults in a software system. Referring to Java, this section provides examples of faults that can be accounted to serialization in specific cases.

2.1 Transitive Closure of the Serialization

The state of a complex object is often composed of other objects. For instance the state of a `Person` object might include an `Address` object containing information like street, city, zip code and so forth. By default in Java the serialization of a `Person` object would imply the serialization of the corresponding `Address` object. In general, the serialization of an object `o` is performed by recursively serializing all objects referenced in `o`.

In many cases however, serializing any referenced object is not the best option. For example, if we send across the network an object that contains a reference to a local service provider, in most of the cases we do not want to serialize the service provider state along with the state of our object. Object attributes that must not be considered during serialization can be specified with the keyword `transient`. Referring to the example, if we specify the reference to the local service provider as `transient`, its value is ignored during serialization and instead a *null* value is flattened into the stream. Figure 1 illustrates this behavior for a generic object graph: the serialization of the object `o1` in Figure 1 (a) results in the graph in Figure 1 (b), where the `transient` edges e_2 and $e4$ have been removed.

Accessing transient attributes after deserialization may cause failures. For example, let us consider the following piece of code:

```
// The SERIALIZED OBJECT
public class Person implements java.io.Serializable{
    private transient Person parent;
    ...
    public Person getParent() {return parent;}
    ...
}

// the FAULY DESERIALIZATION
//...opening the stream "targetStream"
Person p = (Person)targetStream.readObject(); // deserializing the object
Person p2= p.getParent();
System.out.println(p2.getName()); //Very likely this is a fault!!
//closing the stream "targetStream"...
```

The attribute `parent` of the class `Person` is specified as `transient` and thus it is skipped when a person object is serialized. The last instruction of the above code is doomed to fail: because of deserialization, `p2` will have a `null` value and the call of the method `getName` will result in a `NullPointerException`.

2.2 Static Attributes and Replication of Constructors

Static attributes address the definition of properties that are shared among all objects of a class. In Java the values of the static attributes are not serialized by default. Thus, when an object is resumed from a stream, its static attributes are inherited from the new runtime context. This however may be a potential source of faults if all possible bindings are not considered. For example, let us consider a class `WebPage` that contains a static attribute `style`, which references an object of class `PageStyle` storing style attributes for all pages in a given context:

```
public class WebPage implements java.io.Serializable{
  public static PageStyle style;
  ...
}
```

When a `WebPage` object is serialized, the contained static reference is ignored. When the object is deserialized, `style` is set to the `PageStyle` object in the new environment. However, if the attribute is undefined in the new environment, the attribute `style` of a deserialized `WebPage` object will inherit the default reference value, i.e., a `null` reference. This can produce undesired effects (e.g., a `NullPointerException`) if the value of `style` is not checked before use.

In some cases, dealing with static attributes may require to modify the default behavior of serialization. Very frequently this involves reexecuting part of the code from *constructor* method(s) during deserialization. In fact, construction and deserialization are similar operations somehow: although deserialization does not create a new object, an object is added to the system in both cases. For instance, let us consider the following code:

```
public class Son implements java.io.Serializable{
    public static int numberOfSons;
    Son() {numberOfSons++;}
    ...
}
```

The static attribute `numberOfSons` is meant to count the number of instances of the class `Son` that are present in the system at any moment. When an object of this class is created, the constructor increments the instance counter whose value is shared among all objects of the class. When an object of the class `Son` is resumed from a stream, deserialization should work such that it updates the instance counter, as well. A way of dealing with this requirement is to define a customized `readObject` method in the class `Son`, imitating the constructor behavior, as follows:

```
private void readObject(ObjectInputStream ois)
  throws IOException, ClassNotFoundException {
    ois.defaultReadObject();
    synchronized(Son.class){
        numberOfSons++;
    }
}
```

Replication of the code from constructors is often the case in practice for customizing the deserialization process in presence of static attributes. However, for complex programs, it can be difficult to correctly choose which constructor and which parts must be replicated. This generates a new source of potential faults.

2.3 Class Inheritance

Serialization of objects belonging to classes that inherit from other classes, requires to hibernate an object state that spans among all classes in a branch of the inheritance tree. This behavior does not cause problems as long as all involved classes implement the `java.io.Serializable` interface, but unexpected results can be observed if some of these classes is not serializable. Let us consider the following piece of code:

```
public class Person {
    private String name;
    private String surname;
    Person() {name=""; surname="";}
    Person(String n, String s){name=n; surname=s;}
}

public class Student extends Person implements java.io.Serializable{
    private String universityId;
    ...
}
```

The non-serializable class `Person` has two string attributes, `name` and `surname`, a constructor without parameters that initializes both attributes to empty strings, and a further constructor to provide specific values for the attributes. The serializable class `Student` inherits from `Person` and adds the further attribute `universityId`. If we serialize and deserialize the student instance {"*Leonardo*", "*Mariani*", "*UMB1556445*"}, we obtain as result a student with `universityId` "*UMB1556445*", but empty name and surname. This is due to non-serializability of the class `Person`. In general, when a class in the inheritance tree is not serializable, the default behavior of serialization ignores its attributes and reconstruct them at deserialization time by means of the default constructor. In the example, `name` and `surname` are set during deserialization using the constructor without parameters of the class `Person`.

Non serializable classes in the inheritance tree of a serializable class require special consideration and may induce subtle faults.

2.4 Incompatible Class Versions

In component based applications, it is likely that different components may refer to different versions of the same class, for instance because platforms have been updated at the different moments. It is possible that a component receives a serialized object that does not match the held version of the class. Deserialization would fail in this case.

The following piece of code shows two possible versions of the class `Point` that use Cartesian and polar coordinates, respectively:

```
//Cartesian coordinates version
public class Point implements java.io.Serializable{
  private double  x;
  private double y;
  ...
}

//Polar coordinates version
public class Point implements java.io.Serializable{
  private double r;
  private double phi;
  ...
}
```

In general, the problem of converting an object of class version C_1 in an object of class version C_2 includes the conversion of three data elements: the object state variables, the metadata and the serialVersionUID of the class. This latter is a static attribute assigned to classes at compile-time. It represents the class version as a long integer number. Conversion can be accomplished during the serialization process in two ways:

- redefining the behavior of serialization, such that it writes data in the stream according to the format of version C_2 of the class.
- redefining the behavior of deserialization, such that it reads data from the stream according to the format of the version C_1 and converts it on-the-fly in the format of version C_2.

We do not show a sample code in this paper for space limitations. However, converting between class versions requires a complex management. The need of keeping the coherence across different states, metadata and serialVersionUIDs can easily induce faults in the code.

2.5 Keeping Identity

The existence of a cycle in the transitive closure of the references of a given object could, in principle, cause the same object to be serialized multiple times in the same stream. To avoid this situation, the default serialization maintains memory of already serialized objects and inserts only a token into the stream when the same object occurs again. Tokens are such that they uniquely identify the objects. This mechanism prevents infinite recursion, but also hinders the serialization of a new copy of an already serialized version, which sometimes can be the desired behavior.

For example, consider a server and a client components that communicate through a TCP socket, as in the following piece of code:

```
public class Client {
    public static void main(String args[]) {
      ...
      //Initializing communicazion
      Socket soc = ....
      ObjectOutputStream s = new ObjectOutputStream(soc.getOutputStream());

      Counter c = new Counter();
      s.writeObject(c);
      c.Inc();
      s.writeObject(c);
      s.close();
      ...
  }
}

public class Server {
    public static void main(String args[]) {
      ...
      // opening communication
      Socket soc = ...
      ObjectInputStream s = new ObjectInputStream(soc.getInputStream());

      Counter c = (Counter) s.readObject();
      System.out.println(c);
      c = (Counter) s.readObject();
      System.out.println(c);
      s.close();
      ...
    }
}
```

After establishing the communication, the client sends to the server two objects of the class `Counter`. (Although not explicitly shown, the class `Counter` is supposed to provide the functionality of a simple counter that is initialized to zero on construction and can be incremented of a unit with the method `Inc`.) The client initializes a counter, serializes it and sends it to the server through the TCP socket. The operation is repeated twice and the counter is incremented in between. The server receives the objects and prints their values on the screen. In the showed example, the communication results in a failure: the server prints the number 0 twice, instead of printing 0 and 1 as expected. The reason for such faulty behavior is that the `Counter` object is not serialized the second time that the client writes it into the stream. Instead, a token that refers to the previous copy of the counter is inserted into the stream.

A possible solution to this problem, can exploit the method `reset()` for resetting the state of the stream between two serializations of the same object. In this way, the `Counter` object is stored twice. However, let us notice that the second object is serialized as a new object and does not replace the previous one. This can be in turn the source of faults in other cases: for example, because if in subsequent communications the server fails in handling two instances of an object that is present in a single copy in the client.

3 A Reasoning Framework for Serialization

So far we showed how the use of serialization may induce subtle faults in component-based systems. This calls for testing and analysis techniques specifically targeted to reveal this type of faults. However, defining testing and analysis techniques for this purpose, is not straightforward. In this section, we discuss the proposal of a conceptual framework that classifies the main data management strategies, which may occur with serialization. Goal of such classification is to facilitate the definition of techniques for assessing serialization.

Figure 2 (a) illustrates the proposed classification of data management strategies. Every entry in this classification represents a particular way of

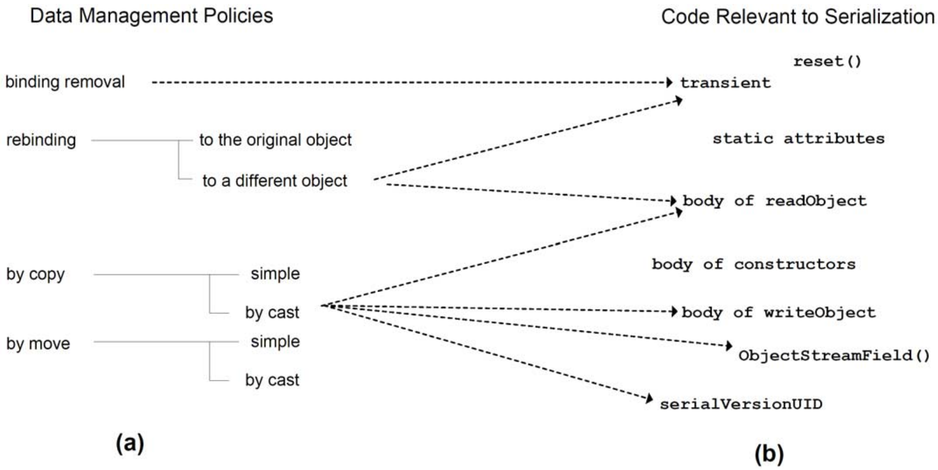

Fig. 2. A taxonomy of semantic categories of data-exchange

handling a graph of objects when it participates in a data-exchange between two components. The semantics of each entry can be informally given as follows:

Binding removal occurs when an object graph is excluded from the data-exchange with the semantics that it will not be available to the receiving component. Typically, *binding removal* is used to detach a subgraph of an object that is not needed in the target environment.

Rebinding occurs when an object graph is excluded from the data-exchange with the semantics that the corresponding reference will be relinked in the target environment to an object of the same type. *Rebinding* is possible in two different ways, depending on the identity of the relinked objects. *Rebinding to the original object* occurs when the original subgraph is relinked through a link that spans across the target and the original environments. Conversely, *rebinding to a different object* occurs when the actual relinked objects are different from the original object graph. For example, *rebinding to the original object* is used in Java-RMI ([8]) when a remotely accessible object is passed as parameter to a remote method (i.e., a method of a remote component), resulting in the object remaining local and only a remote reference being passed to the remote component: the object graph is not transferred, while instead as a result of the invocation, it is relinked from within the remote component. *Rebinding to a different object* happens for example in the case of a software agent that migrates across the network and links to resources in the reached hosts, after having detached the corresponding resources in the original host.

By copy specifies a strategy that occurs when a copy of an object graph is created in the target environment, i.e., the root object is copied and the referenced objects (if any) are recursively copied as well (this is called *by copy simple*). In Java-RMI, this is the default strategy for passing non-remotely accessible objects as parameters of remote calls. A variant of this strategy does not perform a mirror copy, but also converts the object graph format across the original and the target environment. We refer to this strategy as *copy by cast*. *Copy by cast* may be for example required to guarantee compatibility between different versions of the object classes.

By move specifies a strategy that is equal to *by copy*, but also involve elimination of the object graph from the original environment. Analogous considerations apply for the strategy *move by cast* with respect to *copy by cast*.

A set of the above strategies may coexist in the same data-exchange case. For example, a graph of objects may be transferred *by copy*, but some of its subgraphs may be either transferred *by move* or handled by *binding removal* or *rebinding*.

Furthermore, we are studying the connections between the strategies classified in our conceptual framework and the implementation support for serialization in Java. Figure 2 (b) preliminary illustrates this idea in some cases: the keyword `transient` can be relevant to *binding removals*; `transient` and the body of `read object` can be relevant to *rebinding to new objects*; the body of `read object`, the body of `write object`, the procedure `ObjectStreamField` and the keyword `SerialVersionUID` can be relevant to *copy by cast* strategies. The associations represented in the figure are not meant to be exhaustive. However we envision the possibility that refining our understanding of these associations, can be useful for the implementation of tools that support perspective testing and analysis techniques. For example, based on these associations, a tool might be able to automatically identify specific data management strategies in the code. Other tools could exploit the associations to instrument the relevant code, such to address the analysis of a specific data management strategy.

The use of this classification framework to support the definition of a testing technique for serialization is exemplified in the next section.

4 Establishing Dataflow Testing of Serialization

Let us consider the definition of a testing technique for serialization, based on extending the well-known dataflow testing criteria ([6,5]) in order to deal with serialization faults.

Dataflow testing is based on exercising *def-use pairs* (i.e., the program paths between a definition and an use of a given datum), aiming at discovering flawed interactions between the way in which data are assigned values and the way in which such values affect the computation. This idea seems to satisfy with minor modifications the requirements of testing for serialization. In fact, generalizing the examples from Section 2, (1) serialization-related problems show up on using objects (after deserialization) and (2) the faults are made for mis-comprehension of how the objects have been modified (through the serialization process) with respect to their initial definition (before serialization). For instance, an object relinked in the target environment may not conform to the assumptions of the original environment, thus becoming a source of mis-uses. Test-cases that cause object definitions before serialization and uses of the corresponding objects after deserialization, may increase the chances to reveal such faults. However, the indiscriminate application of this criterion would be hindered in practice by the large number of combinations of objects definitions and uses that generally exist(considering also definition and uses of the subparts of the objects). The number of test-cases tends to explode further in presence of polymorfism [9].

Our classification framework provides a means for distinguishing the test-cases that would be overkilled or redundant, thus increasing the chances of practical feasibility of the technique.

Let us consider first object subgraphs corresponding to *binding removals*. In this case, it is always an error to access such subgraphs after deserialization. The corresponding faults can be easily discovered by static checking the existence of the erroneous uses in target environments, thus testing def-use pairs is not needed for all *binding removals*. Then, let us consider the object subgraphs corresponding to *rebinding to the original object*. In this case, a single test case would suffice to assess that the pruned references have been correctly relinked, while one can rely on unit testing of the original components for what concerns other interactions. Thus, testing all related def-use pairs would be redundant in this case. Next, let us object subgraphs corresponding to *rebinding to a different object*. These are cases in which it makes sense indeed to exercise the related def-use pairs: the corresponding test-cases have the chance to discover a fault if either some object was not correctly relinked during deserialization or the relinked objects conflict with the assumptions of the original environment. Finally, let us consider objects and object subgraphs exchanged using *by copy* and *by move* strategies. These generate interesting test-cases only when they feature also the *by cast* strategy. As far as serialization is under concern the exchanged objects that are exact copies of the original one cannot fail the assumptions of the original environment. For *by move* objects however, when serialization occurs, it rises requirements for specific static checks in the original environment.

Summarizing, test-cases for serialization should be defined by considering all combinations of def-use pairs for object (sub)graphs corresponding to *rebinding to a different object*, *copy by cast* and *move by cast* strategies, where definitions take place in the original environment, uses in the target environments and serialization is used for exchanging the objects between the two environments. Moreover, singleton test-cases would be required for *rebindings to the original object* and static checks for *binding removals* and *by move* strategies, in the target and original environments, respectively. The sound and complete definition of a testing technique is not the subject of this paper. However, this example supports the possibility that the conceptual classification framework defined in this paper can provide a rationale to facilitate the definition of testing and analysis technique to address serialization.

5 Final Remarks and Future Work

Serialization is a powerful way to record and retrieve graph of objects into and from byte streams. Serialization mechanisms are frequently used in component-based application domains, e.g., distributed and mobile systems. The use of serialization mechanisms, however, may lead to subtle faults in the software. Serialization faults are difficult to reveal with traditional testing techniques.

The definition of testing techniques addressing serialization is not straightforward since both semantics of current operation and semantics of the serialization must be taken into account. To support testing of serialization, we developed a reasoning framework based on our knowledge on the possible data exchange strategies that may occur with serialization. We do not claim this framework to be complete or exhaustive, but we investigated its usefulness by showing that it allows to tune and extend dataflow testing for systems that use serialization.

We are currently working to refine and complete our conceptual framework for reasoning on serialization. In particular, we are currently investigating the connection between the data-exchange cases classified in the framework and the implementation support provided in Java for serialization. We are also conducting further research on testing and analysis techniques that can be defined based on our conceptual framework. We are refining the specialized dataflow testing approach that we preliminarily described in this paper. We are finally investigating assertion-based run time verification [7,11]. This latter could exploit our conceptual framework as a base for the definition of specialized assertions to address serialization.

References

[1] Emmerich, W., *Software engineering and middleware*, in: *Proceedings of the 22th International Conference on Software Engineering (ICSE-00)* (2000), pp. 117–132.

[2] Fuggetta, A., G. Picco and G. Vigna, *Understanding code mobility*, IEEE Transactions on Software Engineering **24** (1998), pp. 342–361.

[3] Ghezzi, C., V. Martena and G. Picco, *Enhancing remote method invocation through type-based static analysis*, in: *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE 2004), Barcelona, Spain*, 2004.

[4] Halloway, S. D., "Component Development for the Java Platform," Addison-Wesley, 2001.

[5] Harrold, M. J. and G. Rothermel, *Performing data flow testing on classes*, in: *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'94)* (1994), pp. 154–163.

[6] Laski, J. and B. Korel, *A data flow oriented program testing strategy*, IEEE Transactions on Software Engineering **9** (1983), pp. 347–354.

[7] Meyer, B., "Eiffel: The Language," Object-Oriented Series, Prentice Hall, New York, N.Y., 1992.

[8] Microsystems, S., *Java^{TM} remote method invocation specification*, Technical report, Sun Microsystems (2002).

[9] Orso, A., "Integration Testing of Object-Oriented Software," Ph.D. thesis, Politecnico di Milano, Italy (1999).

[10] Picco, G. P., *µCode: A Lightweight and Flexible Mobile Code Toolkit*, in: K. Rothermel and F. Hohl, editors, *Proceedings of the 2nd International Workshop on Mobile Agents*, Lecture Notes in Computer Science **1477** (1998), pp. 160–171.

[11] Rosenblum, D. S., *A practical approach to programming with assertions*, IEEE Transactions on Software Engineering **21** (1995), pp. 19–31.

[12] Sun Microsystems, Inc., *Java object serialization specification, rev. 1.4.4*, Technical report (2001).

[13] Wiltamuth, S. and A. Hejlsberg, *C# language specification*, Technical report, Microsoft Corporation (2003).