J Braz Comput Soc
DOI 10.1007/s13173-012-0083-5

ORIGINAL PAPER

On-the-fly extraction of hierarchical object graphs

Hugo de Brito · Humberto Torres Marques-Neto · Ricardo Terra · Henrique Rocha · Marco Tulio Valente

Received: 17 November 2011 / Accepted: 16 July 2012
© The Brazilian Computer Society 2012

Abstract Reverse engineering techniques are usually applied to extract concrete architecture models. However, these techniques usually extract models that just reveal static architectures, such as class diagrams. On the other hand, the extraction of dynamic architecture models is particularly useful for an initial understanding on how a system works or to evaluate the impact of possible maintenance tasks. This paper describes an approach to extract hierarchical object graphs (OGs) from running systems. The proposed graphs have the following distinguishing features: (a) they support the summarization of objects in domains, (b) they support the complete spectrum of relations and entities that are common in object-oriented systems, (c) they support multithreading systems, and (d) they include a language to alert about expected (or unexpected) relations between the extracted objects. We also describe the design and implementation of a tool for visualizing the proposed OGs. Finally, we provide two case studies. The first study shows how our approach can contribute to understand the running architecture of two systems (myAppointments and JHotDraw). The second study illustrates how OGs can help to locate defective software components in the JHotDraw system.

Keywords Software architecture · Software models · Object graphs · Reverse engineering

H. de Brito · H. T. Marques-Neto
Department of Computer Science, PUC Minas,
Belo Horizonte, Brazil
e-mail: hugobrito@pucminas.br

H. T. Marques-Neto
e-mail: humberto@pucminas.br

R. Terra · H. Rocha · M. T. Valente (✉)
Department of Computer Science, UFMG,
Belo Horizonte, Brazil
e-mail: mtov@dcc.ufmg.br

R. Terra
e-mail: terra@dcc.ufmg.br

H. Rocha
e-mail: hscr@dcc.ufmg.br

1 Introduction

A common definition (or view) describes software architecture as the main components of a system, including the acceptable and unacceptable relations among them [7,13,20]. However, despite their unquestionable importance, architectural models and abstractions are usually not documented, or when they are, the available documentation normally does not reflect the actual architecture followed by the implementation of the target systems [9,16,19].

In such scenarios, reverse engineering techniques can be applied to reify information about a target system architecture [10,27]. Usually, those techniques extract models that reveal the static architecture, including class and package diagrams [12] or dependency structure matrices [22]. As one of their distinguishing advantages, static models can be retrieved directly from the source code (i.e. without requiring the execution of the target system). However, static models only show a partial snapshot of the relations, connections, and dependencies that are actually established during the execution of the modeled system. For example, static diagrams cannot reveal relations due to polymorphism, dynamic method calls, or reflection. Furthermore, they do not include information on the order in which the represented relations are established. In other words, static diagrams do not provide a clear roadmap to developers that need to understand a given system. Finally, static diagrams do not take into account relations and dependencies established by distinct threads,

Published online: 05 September 2012

which makes the task of understanding concurrent systems complex.

On the other hand, reverse engineering techniques can also be applied to extract models that reveal dynamic architectures, such as object and sequence diagrams [12]. Dynamic diagrams explicitly represent the control flow of the target system and therefore they provide an order that can be followed when initially reasoning about the system. Moreover, dynamic diagrams can express relations due to polymorphism or reflection [23,29]. In contrast, dynamic diagrams present major problems regarding their scalability. Because they typically do not make any distinction between lower-level objects (such as instances of `java.util.Date`) and architectural relevant objects (such as collections of `Customer` objects), dynamic diagrams may have thousands of nodes even for small-sized systems [1,2].

The available solutions to increase the scalability of dynamic diagrams are centered on the same principle: to group objects into coarse-grained and hierarchical structures. In the highest level of such structures, only architectural relevant groups of objects are displayed (usually called domains [1], components [18], clusters [5], etc.). It is also possible to expand such higher-level groups to provide more details about their elements. This process can be repeated several times, until reaching a flat object graph (OG), where each node corresponds to a runtime object. Basically, there are two approaches to group objects into coarser-grained structures: automatic approaches (for example, using clustering algorithms [5,6]) and manual approaches (for example, using annotations [1,2]). Typically, automatic solutions do not derive groups of objects similar to those expected by the system's architects and maintainers. On the other hand, solutions based on annotations are invasive, requiring the annotation of each architectural relevant class (for example, the classes of the `Model` layer must be annotated with a `@Model` annotation).

This paper is a revised and extended version of a previous conference paper presenting an on-the-fly and non-invasive approach to extract hierarchical OGs from running systems [4]. It also describes a non-invasive tool to extract and display the proposed graphs. This tool can be plugged to existing systems and thus it supports the on-the-fly visualization of the proposed graphs (i.e. the graphs are displayed and updated as the host system executes). This property distinguishes the proposed tool from other reverse engineering systems, where it is usually required to first execute the target system to generate a trace that is then displayed off-line. Finally, we report two case studies on using OGs. The first study illustrates how the proposed OGs and supporting tool can help to recover and to reason about the dynamic architecture of two systems: myAppointments (a personal information manager system) and JHotDraw (a well-known framework for creating drawing applications). The second study describes how OGs can help to locate the defective software components responsible for an incorrect behavior in the JHotDraw system, as reported in real corrective maintenance requests retrieved from JHotDraw's bug tracking platform.

The remainder of this paper is organized as follows: Section 2 describes the proposed OGs, including a description on their main elements and examples. Section 3 describes the language used to define visual alerts when expected (or unexpected) dynamic relations are established in the extracted OGs. Section 4 presents the OG tool that extracts and displays OGs. In Sect. 5, we present the first case study (on extracting dynamic architectures). Section 6 presents the second study (on the use of OGs in real JHotDraw's corrective maintenance tasks). Finally, Sect. 7 discusses related work and Sect. 8 concludes the paper.

2 Object graphs

The graphs proposed in the paper have been designed to support the following requirements: (a) they should be able to express the different types of relations available in object-oriented systems, including relations due to dynamic calls and reflection; (b) they should support the creation of coarse-grained groups of objects to increase readability and scalability; (c) they should provide means to distinguish objects created by different threads; (d) in order to provide support to dynamic architecture conformance, it should be possible to highlight relations that are expected—or that are not expected—when running a system. Finally, it should be possible to extract OGs from running systems in a non-invasive way.

Formal definition An OG is a directed graph that represents the dynamic behavior of the objects in an existing system. In an OG, the nodes denote objects (and classes with static members) and the edges represent possible relations between the represented nodes. In formal terms, an OG is defined as a graph (*Nodes*, *Edges*), where *Nodes* and *Edges* are the following sets:

$$
\begin{aligned}
Nodes &= Type \times Name \\
Type &= \{object, class\} \\
Name &= UnsignedInt \times String \times String \\
Edges &= Nodes \times Nodes
\end{aligned}
$$

where *Type* is a set with the two possible types of a node (which can represent objects or classes) and *Name* is a tuple with three fields: the insertion order of the node (a non-negative integer), the name of the class of the node (a string), and the node's color (a string). Finally, *Edges* are ordered pairs of *Nodes*.

In the following paragraphs, we provide more details on this definition, including information on how OGs must be displayed.

Nodes As defined by the set *Type*, there are two types of nodes in an OG. Nodes in the form of a circle denote objects. Nodes in the form of a square represent classes. Circle-shaped nodes have the same life span of the objects they model (in other words, a circle node is inserted in an OG when an object is created in the host program; likewise, it is removed when the represented object is destroyed by the garbage collector). Square-shaped nodes are created to model accesses to static members of a class. Therefore, only classes with static members accessed by objects are represented in an OG.

As defined by the set *Name*, the name of a node is a tuple with three fields. The first field is a sequential non-negative integer that indicates the order in which the nodes have been inserted in the graph. By convention, the first node receives the number zero (typically, this node denotes the class containing the application's `main` method). The goal of this number is to guide the developers when "reading" the graph. The second field indicates the class name of the represented object (in the case of circular nodes) or the name of the class whose static member has been accessed (in the case of square nodes). Finally, the third field represents the node's color. In OGs, colors are used to distinguish circular nodes created by different threads. Nodes created by the `main` thread have a white color and a fresh color is automatically assigned to nodes representing objects created by other threads.

Edges As defined by the set *Edges*, edges denote relations between objects and classes. Suppose that o_1 and o_2 are circle-shaped nodes (representing objects) and that c_1 and c_2 are square-shaped nodes (representing classes). The directed edge (o_1, o_2) indicates that o_1—at some point during its life span—has obtained a reference to o_2. This reference could have been acquired by an object's field, by a local variable, or by a method's formal parameter. Similarly, the directed edge (o_1, c_1) indicates that o_1—at some point during its life span—has called a static method implemented by c_1. On the other hand, an edge (c_1, o_1) indicates that a static member of c_1—at some point of the program's execution—has obtained a reference to o_1. Finally, the edge (c_1, c_2) indicates that c_1 has accessed a static member of c_2.

Edges are inserted in an OG as soon as the represented relation is established during the execution of the host program. When a node is removed from the graph, its incoming and outcoming edges are also removed. Furthermore, for the sake of readability, edges denoting loops (i.e. edges starting and ending in the same node) are not represented.

Example (*Nodes and Edges*) Consider the code shown in Listing 1.

```
1  public class Main {
2      private static Invoice invoice;
3      public static void main(String[] args) {
4          invoice= new Invoice();
5          invoice.load();
6      }
7  }
8  public class Invoice {
9      private Collection<Product> listProducts;
10     public void load(){
11         listProducts= new ArrayList<Product>();
12         Product p= new Product();
13         listProducts.add(p);
14     }
15 }
```

Listing 1: Nodes and Edges Example

In this code, the `Main` class creates an object of type `Invoice` and calls the `load` method (lines 4–5). This method creates and adds a `Product` to an `ArrayList` (lines 11–13). Figure 1 presents the OG generated by the execution of the code fragment shown in this listing. This OG has one square-shaped node (representing the class with the `main` method) and three circle-shaped nodes, representing the `Invoice`, `ArrayList`, and `Product` objects.

The extracted OG illustrates in a compact way the runtime behavior of the presented program fragment. Following the sequential integers associated with each node, it is possible to conclude that initially the `Main` class (node 0) has accessed an `Invoice` object (node 1). Next, this `Invoice` object has accessed an `ArrayList` object (node 2). Finally, a `Product` has been created (node 3). This `Product` instance has been accessed by the `Invoice` object (responsible for its creation) and by the `ArrayList` object (responsible for its storage).

Example (*Threads*) Consider the code presented in Listing 2. In this code, the `Main` class creates and activates two `Box` threads (lines 3–4). Each thread creates a `Product` object (line 10). Figure 2 presents the OG generated by the execution of this program. In this OG, the `Main` class (node 0) has references to two `Box` objects (nodes 1 and 3). Moreover, we can verify that each `Box` references its own `Product`

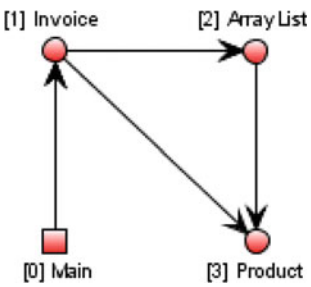


Fig. 1 OG for the nodes and edges example

```
1  public class Main {
2      public static void main(String[] args) {
3          new Box().start();
4          new Box().start();
5      }
6  }
7
8  public class Box extends Thread {
9      public void run() {
10         new Product();
11     }
12 }
```

Listing 2: Threads Example

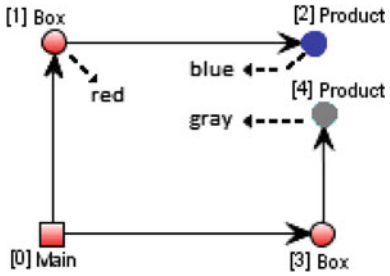


Fig. 2 OG for the threads example (the *names* of the *colors* are only illustrative)

object (nodes 2 and 4). More importantly, nodes denoting `Product` objects have different colors, because they have been created by different threads.

Packages and domains As it is common when extracting runtime diagrams, the number of nodes and edges in an OG can grow rapidly, even for small applications. Therefore, to promote the scalability of OGs, there are two forms of summarization: by packages or by domains. When package summarization is enabled, all the objects and classes from a given package are represented as a single node. In such compacted graphs, suppose two nodes representing packages p_1 and p_2. In this case, an edge (p_1, p_2) indicates that at least one element summarized by p_1 is connected to an element summarized by p_2.

The second form of summarization is by domain. Basically, in the particular context of this paper, a domain is a group of nodes explicitly defined by developers using the following syntax:

```
domain <name>:<classes>
```

where `<name>` is the domain name and `<classes>` is a list of classes separated by commas. For summarization purposes, objects from the specified classes will be represented in the graph by a single node, in the form of a hexagon. Moreover, to facilitate the specification of domains, classes can be defined using regular expressions (e.g. `model.*DAO` denotes the classes in the `model` package whose names end with `DAO`).

Domain-based summarization is more flexible than summarization by packages, because developers can explicitly define the domain names—to resemble, for example, architectural relevant components and abstractions. Moreover, developers have the freedom to define the members of a domain, by mapping classes to their respective domains. By contrast, summarization by packages is more rigid, since it assumes that architectural relevant components can be extracted automatically from the package hierarchy. From our experience with OGs, the usual procedure is to start by using OGs with package summarization, especially when no other form of documentation is available. After an initial understanding of the architecture, maintainers usually get enough knowledge to define their own domains (e.g. domains that summarize packages related to persistence, when the maintenance task does not require changes in persistence concerns).

Example (*Domains*) Consider a hypothetical system following the model–view–controller (MVC) architecture [17]. To provide a high-level picture for this architecture, the domains presented in Listing 3 have been defined. In this listing, the `View` domain denotes instances of the `myapp.view.IView` class and of its subclasses (as prescribed by the operator `+`) (line 1). The `Controller` domain includes objects from any class implemented in the `myapp.controller` package (line 2). The `Model` domain includes objects whose class names begin with `myapp.model` and end with the string `DAO` (line 3). In the specification of domains, the operator `**` denotes classes from packages with a given prefix. For example, the `Swing` and `Hibernate` domains include, respectively, objects from classes in the `javax.swing` and `org.hibernate` packages, as well as objects from classes implemented in inner packages (lines 4 and 5).

```
1 domain View:  myapp.view.IView+
2 domain Controller: myapp.controller.*
3 domain Model: "myapp.model.[a-zA-Z0-9/.]*DAO"
4 domain Swing: javax.swing.**
5 domain Hibernate: org.hibernate.**
```

Listing 3: Domains for a system based on the MVC architecture

Figure 3 presents the OG extracted for the MVC-based system considered in this example. First, we can observe that the nodes associated with domains are displayed as hexagons. However, there is a single node in the form of a circle (node 3, `Util`), representing an object whose class has not been included in any of the defined domains. In other words, objects or classes that are members of a defined domain are summarized by a hexagonal node; objects or classes that are not captured by any defined domain continue to be represented by circles (in the case of objects) or squares (in the case of classes).

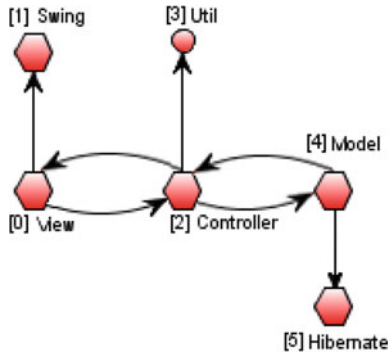


Fig. 3 OG for a system based on the MVC architecture

As can be observed in the OG presented in Fig. 3, the target system's architecture follows the MVC pattern. For example, there is a bidirectional communication link between the `View` and `Controller` domains, and between the `Controller` and `Model` domains. Furthermore, the OG reveals that the `Controller` acts as a mediator between the `View` and the `Model`, as expected in MVC architectures. We can also observe that only the `View` relies on services provided by the Swing framework (for GUI concerns) and that only the `Model` is coupled to the Hibernate framework (for persistence concerns).

Detailed information on edges It is also possible to display detailed information on the object-oriented relations modeled by an OG's edges. Suppose that o_1 and o_2 are nodes in an OG and (o_1,o_2) is an edge connecting such nodes. An edge's name is a structure in the following format:

```
Edge_Order [O1_Order] Location [Suffix] > [O2_Order] O2_Service [Suffix]
```

The members of this structure are

- `Edge_Order` is a sequential non-negative integer that indicates the order in which the edges have been created in the graph. This integer makes possible a sequential reading of the graph's edges.
- `O1_Order` is a sequential non-negative integer enclosed by brackets that indicates the order in which the node o_1 was inserted in the graph.
- `Location` represents the program location where the relation was established.
- `O2_Order` is a sequential non-negative integer enclosed by brackets that indicates the order in which the node o_2 was inserted in the graph.
- `O2_Service` represents the service provided by o_2 that has been accessed to establish the edge.
- `Suffix` provides information about both the `Location` and `Target` elements. It can assume one of the following values:
  - *()* indicates access to methods.
  - *(MS)* indicates access to static methods.
  - *(C)* indicates access to constructors.
  - *(A)* indicates access to attributes.
  - *(AS)* indicates access to static attributes.
  - *<new>* indicates that an object has been created.

Example (*information on edges*) Listing 4 shows information on the edges of the *OG* presented in Fig. 1. In this listing, line 1 indicates that the static method `Main.main` (suffix MS) has a static field (suffix `AS`) that references an instance of the class `Invoice`. Line 3 indicates that at the location `Invoice.load()` the source object has created an `ArrayList`. Next, at the same location, this `ArrayList` object has been assigned to the field `listProducts` (suffix `A`, line 4). Finally, the `ArrayList.add()` method has been called (line 6).

```
1 01-[0]Main.main(MS) > [1]Invoice.invoice(AS)
2 ...
3 03-[1]Invoice.load() > [2]ArrayList.<new>
4 04-[1]Invoice.load() > [1]Invoice.listProducts(A)
5 ...
6 06-[1]Invoice.load() > [2]ArrayList.add()
```

Listing 4: Detailed information on OG edges

3 Alert language

To provide support for dynamic architecture conformance using OGs, we have defined a small language to trigger visual alerts when expected (or unexpected) relations are established in an extracted OG. Since our approach is based on dynamic analysis, the proposed language can check relations due to dynamic calls or reflection. For example, consider a system that relies on the *data access objects* (DAO) pattern for handling data [11]. In this case, to analyze the runtime behavior of the system when it is persisting data, we can define an alert to be triggered whenever an expected DAO service is called (which in some frameworks is implemented using reflection).

Syntax Alerts are defined according to following grammar:

```
<alert_clause> ::= alert <domain> {, <domain>}
                   <relation> <domain> {, <domain>}

<domain> ::= [!] <string> | *
<relation> ::= access | create | depend
```

In this grammar, non-terminal symbols are written between `and` (e.g. `domain`). Brackets denote optional symbols (e.g. `[!]`, indicating that `!` is optional). Braces indicate that the delimited element may have zero or more repetitions

(e.g. `domain`). Terminal symbols are written without special delimiters (e.g. `alert`, `access`, etc.). The non-terminal `string` denotes a sequence of characters. In the specification of domains, the operator `!` means complement. For example, `!A` denotes a domain containing any object that is not included in `A`. Finally, the symbol `*` matches any object, regardless of its domain.

According to this grammar, an alert clause defines a relation between two domains. The alert will be activated when the defined relation is detected at runtime. When specifying alerts, the following relations between domains can be specified:

- `depend`: represents any kind of relation between elements of an object-oriented program.
- `access`: represents two particular types of relations: accesses to fields or method calls. Thus, `access` is a particular case of a `depend` relation. For example, an object may hold a reference to an object in another domain (`depend`), but it may not use its services (`access`). A typical example is an object received as argument in a Facade method and that is just passed to another method behind the Facade (i.e. the Facade does not call any method or access any field from this object).
- `create`: denotes that an object from the source domain has created an object from the target domain.

To illustrate the proposed syntax, suppose the following alert clauses—where `A`, `B`, and `C` are domains and `R` is a relation type (i.e. `depend`, `access`, `create`):

- `alert A R B`: This alert will be activated when an element at the domain `A` has established a relation of type `R` with an element at the domain `B`.
- `alert A R !B`: This alert will be activated when an element at the domain `A` has established a relation of type `R` with an element not included in the domain `B`.
- `alert !A R B`: This alert will be activated when an element not included in the domain `A` has established a relation of type `R` with an element from the domain `B`.

Visual interface Alerts are displayed in two ways: (a) changing the edge's color on the relations responsible for the alert; (b) generating a message on a dedicated alert window with detailed information on the alert (e.g. source and target node, type of relation, etc.).

3.1 Example 1

Listing 5 illustrates three examples of alert specification (using the domains defined in Fig. 3). In this code, we first define that an alert must be raised when any object accesses the `Hibernate` domain (line 1). We also define an alert to capture accesses from the `myapp.Util` class to other classes (line 2). This alert checks whether utility classes are self-contained (i.e. whether they only provide services to client domains). Finally, we define an alert to check whether `DAOImpl` objects are created only by their respective Factory class (line 3).

```
1 alert * access Hibernate
2 alert myapp.Util access !myapp.Util
3 alert !DAOFactory create DAOImpl
```

Listing 5: Alert Language example

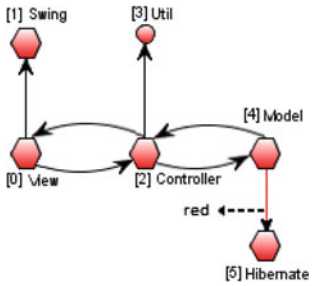


Fig. 4 OG with an alert enabled due to a dependency from Model to Hibernate

Figure 4 shows an OG with an alert enabled. In this OG, the edge between the `Model` and the `Hibernate` domains has the color red, indicating that—at some point during the program's execution—an object located in the `Model` domain has accessed a service provided by an object in the `Hibernate` domain, which represents a violation to the first alert in Listing 5. Furthermore, this alert is explained in a separate alert window, with detailed information on the source and target objects responsible for its activation.

3.2 Example 2

This second example is based on a common scenario when accessing databases in Java. Usually, this task is performed by creating an object from a specific DBMS class (that represents the database driver). Usually, the qualified name for this class is stored in a text file or is directly hard-coded in the program, as illustrated in Listing 6 (line 1). More specifically, this example relies on the Java reflection API to open a connection to the HSQLDB database manager system.

```
1 Class.forName("org.hsqldb.JdbcDriver");
2 ...
3 Connection conn= DB.getConnection(...);
```

Listing 6: Code that registers a HSQLDB driver

As usual, changing the DBMS without a previous detailed analysis can raise several problems. For instance, SQL statements that have specific HSQLDB instructions will stop working. To avoid this problem, we can define an alert to

monitor DBMS changes, as illustrated in Listing 7. This definition alerts when the DB class—responsible for the DBMS connection—creates an object that is not a HSQLDB driver or that is not part of the Java SQL API.

```
1 domain JavaSql: java.sql.**
2 domain DB:      myapp.model.DB
3 alert DB create !org.hsqldb.JdbcDriver, !JavaSql
```

Listing 7: Alert to monitor DBMS changes

4 OG tool

This section presents the OG tool that extracts and displays OGs. It is a non-invasive tool that can be plugged into an existing Java system to visualize the graphs proposed in this paper. Figure 5 shows the tool's main screen. In order to describe this interface, labels (from *A* to *I*) are used to show the interface's main components. The labels in this figure are described next.

- Label A: represents the number of nodes in the extracted OG. This information can be used, for example, to start an investigation on an alternative summarization strategy (in the case of graphs with a massive number of nodes).
- Label B: represents two visualization features provided by the tool. The first feature allows users to choose the graph's layout and consequently to organize its visualization. The second feature is used for *transforming* or *picking* the graph. When users choose *Transforming*, they can translate, move, or zoom in/out the graph. On the other hand, if they want to organize the nodes by themselves, they can rely on the *Picking* functionality.
- Label C: embraces two command buttons—*Capture* and *Clear*. The first command is used to enable the retrieving of OGs—from the current state of the target system execution—and the second command clears the captured OG.
- Labels D, E, and F: allow users to show the nodes' names, to reduce the size of the text fonts to improve visualization, and to enable the summarization of nodes according to the package structure, respectively.
- Label G: graphical panel where the extracted OG is displayed.
- Label H: embraces two command buttons—*All Edges* and *Clear*. The first command displays information on the edges in an OG. The second command clears the information on the extracted edges.
- Label I: text panel to display information on the OG's edges.

4.1 Running the OG tool

In our current implementation, the OG tool instruments the target program using a generic aspect implemented in AspectJ [15,26]. Therefore, to execute the tool, the users must first execute the AspectJ weaver to instrument the target code with the aspects provided by the tool's implementation. After this preliminary instrumentation phase, the target system can be executed as usual. During its execution, the target system will behave exactly as prescribed by the original code, with the exception the OG tool's interface (Fig. 5), which is shown in a separate window.

5 Dynamic architecture extraction examples

This section provides concrete examples of OGs for two systems: myAppointments and JHotDraw. myAppointments is a small personal information manager system that follows the MVC architectural pattern. Basically, myAppointments allows users to create, search, update, and remove

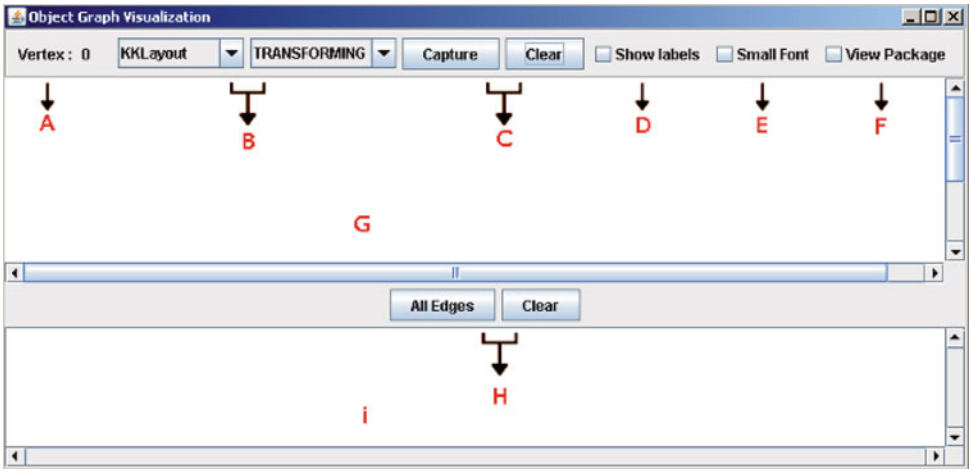


Fig. 5 OG tool

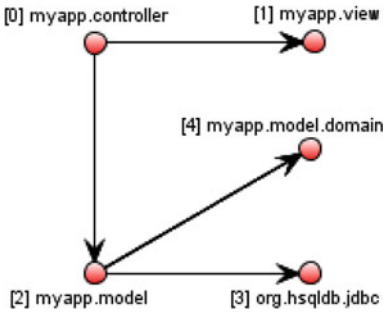


Fig. 6 myAppointments' OG for the feature appointment's removal, summarized at the package level

appointments. The system has been originally designed to illustrate the application of static architecture conformance techniques [19]. The second system, JHotDraw, is a well-known framework for the creation of drawing applications.

5.1 myAppointments

Suppose that one of the myAppointments' developers needs to apply a change in the modules of the system responsible for removing appointments. Suppose also that the developer does not have a deep knowledge on such modules (for example he has started recently to maintain this part of the system). Therefore, he can use the OG tool to extract an OG that represents only appointment removals. Initially, this OG can be extracted using package summarization (since the developer does not have enough knowledge to define domains for the system). Later, he can zoom into the extracted graph, in order to get more information at the level of plain objects.

Figure 6 shows the first OG extracted for the feature appointment's removal. This OG has five nodes representing the following packages: `myapp.controller` (Controller concerns), `myapp.view` (View concerns), `myapp.model` (Model concerns), `org.hsqldb.jdbc` (Persistence concerns), and `myapp.model.domain` (domain concerns). As we can observe, the OG shows that the Controller communicates with the View and the Model and that the Model communicates with the Persistence and Domain packages.

The previous OG can also be viewed at the level of plain objects, as presented in Fig. 7. Although we have argued previously in this paper that plain OGs are not scalable, for a single and delimited feature like the one in this example, they can show valuable information for developers. As we can observe, the new graph has seven nodes (instead of five nodes, as in the case of summarization at the package level): `AgendaController` (representing the application entry point), `AgendaView`, `AgendaDAO`, `DB`, `JDBCConnection`, `DAOCommand`, and `App`.

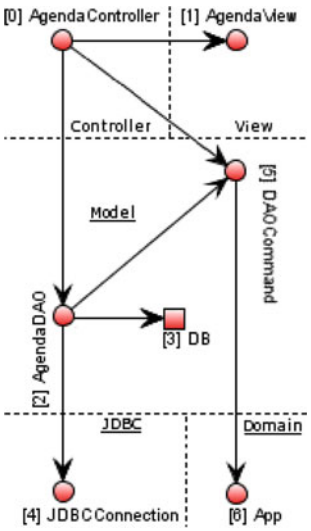


Fig. 7 Plain myAppointments' OG for the feature appointment's removal

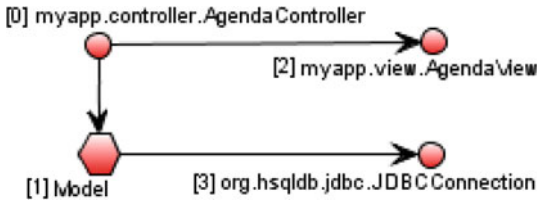


Fig. 8 myappointments' OG, with a `Model` domain enabled

This new graph presents more information on the system's behavior. For example, it reveals that DAO objects are used for database access, and that the communication with the database relies on JDBC drivers.

Finally, the developer can define domains to better represent the system's objects. For example, suppose the developer defines the following domain for the objects in `myapp.model.**` packages:

```
domain Model: myapp.model.**
```

Figure 8 shows the OG with this domain enabled. In this third OG, the nodes associated to Model objects or classes—objects `AgendaDAO`, `DAOCommand`, `App`, and the class `DB`—have been summarized in a single node, called `Model`. In this way, the new OG has only four nodes, which makes it easier to understand.

To conclude, depending on the understanding task under development, the approach can provide graphs with more information than a standard summarization by package. On the other hand, whenever needed, it can also provide higher-level graphs than those retrieved by package summarization.

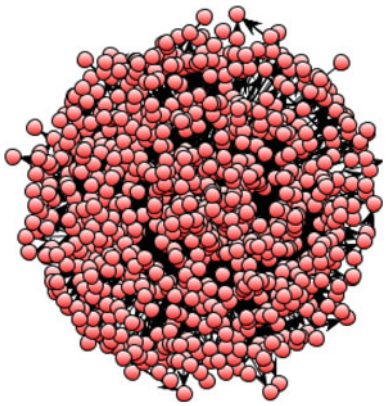


Fig. 9 JHotDraw's OG without summarization

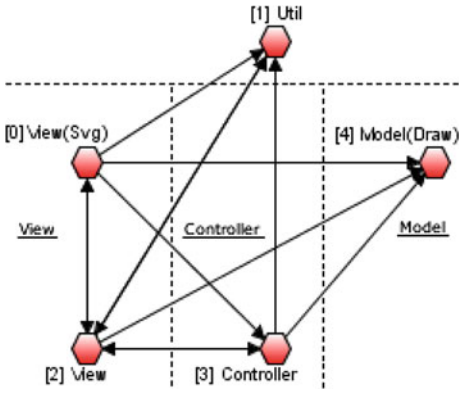


Fig. 10 JHotDraw's OG using domain-based summarization (edited to include the layer's *names* and *dashed lines* separating the layers)

5.2 JHotDraw

First, we have extracted a plain graph for JHotDraw, without any form of summarization. As can be observed in Fig. 9, the extracted OG has thousands of nodes and edges, which precludes its application in reengineering tasks.[1]

Next, to get an initial view of JHotDraw dynamic architecture, we extracted a second OG using domains for a better summarization. The domain definition was based on the class division proposed by Abi-Antoun and Aldrich [2] for JHotDraw. According to this definition, presentation objects (such as `DrawingEditor` and `DrawingView` instances) are located in two domains with the `View` prefix. Objects responsible for the presentation logic (such as `Tool`, `Command`, and `Undoable` instances) are located in the `Controller` domain. Finally, model objects (such as `Drawing`, `Figure`, and `Handle` instances) are located in a domain called `Model`.

Therefore, as presented in Fig. 10, we defined five domains: one for utility classes, two related to the `View`, one to the `Controller`, and one to the `Model` layer. To improve readability, we used an OG tool's resource that provides bidirectional edges to connect nodes that communicate in both ways. Unlike Fig. 9, Fig. 10 can be used by architects and developers to reason about JHotDraw's implementation. For example, this OG reveals the three layers that define the MVC pattern followed by JHotDraw architecture.

This example illustrates the importance of first relying on a coarse-grained view of the target system (probably based on domains), which contributes to get a first understanding of the system's main components and relations. After retrieving this first view, architects and maintainers can, for example, zoom into particular components, to study their internal elements and relations.

[1] More specifically, this graph has 9,950 nodes and 37,976 edges. Despite this fact, it has been retrieved promptly after JHotDraw has been started. Indeed, we have not observed any important performance overhead when using JHotDraw with OGs enabled.

6 Case study: corrective maintenance tasks

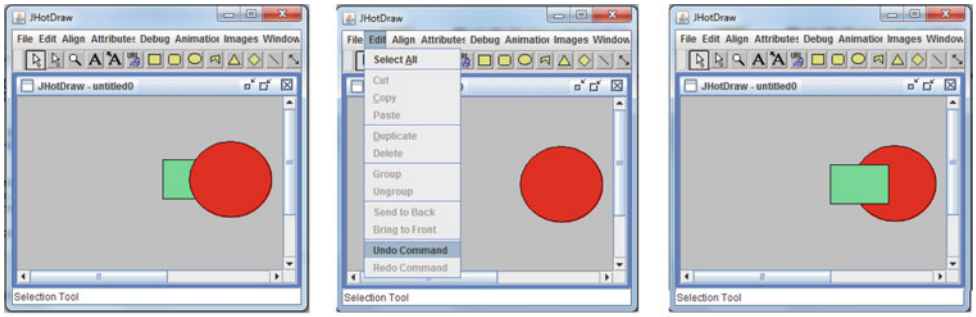
Using JHotDraw as our target system, we designed a study to illustrate how OGs support corrective maintenance tasks. Since our tool provides visualization of OGs in an on-the-fly way, it can be used to retrieve OGs that describe the runtime configuration of the objects in the target system just before or after a given failure has been observed. We claim that such OGs provide valuable information to locate and to discover the static components (i.e. classes and methods) that generated the reported failure.

To support our claim this section illustrates the use of OGs to correct the following bugs reported by JHotDraw's users:

- Bug 1850703 (Opened 2007-12-14): "Redoing Figure delete change order"
- Bug 1989778 (Opened 2008-06-10): "Pick & Apply Attributes"

We selected these bugs based on the following criteria: (a) they have been reported in the last five years (i.e. we filtered out requests with more than five years); (b) they denote corrective maintenance tasks (i.e. we filtered out evolutive maintenance tasks); (c) they imply an incorrect behavior of the system (i.e. we filtered out maintenance that just requires changing the name or color of a UI label, for example); and (d) they do not abort JHotDraw's execution with an unhandled exception (in fact, in such cases the stack trace provides valuable information to locate the failure).

In the following subsections, we describe how we have used the proposed OGs to locate the components responsible for these two bugs.



(a) Original drawing **(b)** Rectangle deleted **(c)** Rectangle restored on top

Fig. 11 Example of Bug 1850703

6.1 Bug 1850703: "Redoing Figure delete change order"

To locate the source of this bug, we performed the following tasks:

Task #1 Based on the bug's description we reproduced its occurrence in a concrete drawing, as illustrated in Fig. 11. Figure 11a shows the original drawing. In Fig. 11b, we have deleted the rectangle and prepared to execute an undo command. Figure 11c shows that after the undo the rectangle has appeared on top of the circle (and not below the circle as in the original drawing).

Task #2 Figure 12 shows the OG extracted by the OG tool just before selecting the undo command (i.e. in the state captured in Fig. 11b).

Task #3 We sequentially inspected the OG's edges to locate possible methods related to the "depth" of a figure in a drawing. As illustrated in Fig. 12, a call to the method `AnimationDecorator.getZValue()` (node 6) coming from a `BouncingDraw` object (node 5) called our attention (since the suffix `Zvalue` reminds the depth of a figure in the current drawing). In fact, by retrieving JHotDraw's code where this bug has been fixed, it was possible to assert that the changes have been confined to the method `BouncingDraw.add()`, which was calling `getZValue()` in an incorrect way.

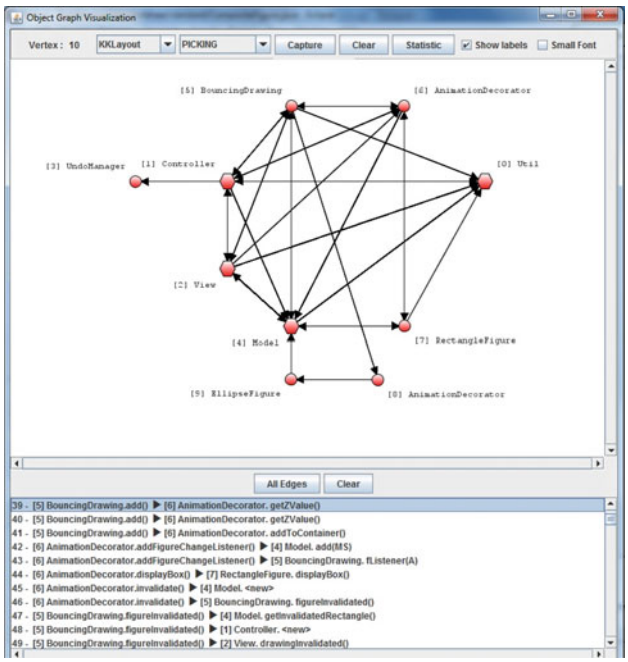


Fig. 12 OG for Bug 1850703

6.2 Bug 1989778: "Pick & Apply Attributes"

To locate the source of this bug, we performed the following tasks:

Task #1 Following the description at SourceForge, we were able to reproduce the bug in the following way: (a) we created a diagram with one circle and one rectangle, with different filling colors; (b) we marked the circle and selected the "PickAttribute" button; (c) we marked the rectangle and selected the "ApplyAttribute" button. Differently from the normal behavior, the rectangle's color has not changed (in fact, the change was only applied after we managed to unmark the box).

Task #2 Figure 13 shows the OG extracted by our supporting tool just after selecting the "ApplyAttribute" command.

Task #3 In the extracted OG, the existence of an object of the class `ApplyAttributeAction` has initially attracted our attention (node 0, Fig. 13). After discovering this object, we carefully inspected its outgoing edges and we were attracted by an edge to an object of the class `RectangleFigure` (node 6), since in our example we were applying the selected attributes to an rectangle. Finally, by inspecting the calls responsible to this edge—listed in a lower panel in the OG tool window—we discovered a call to a method named `setAttribute()`. In fact, by retrieving JHotDraw's code where this bug has been fixed, it was possible to assert that the method `ApplyAttributeAction.apply Attributes()` was the source of the reported bug. More

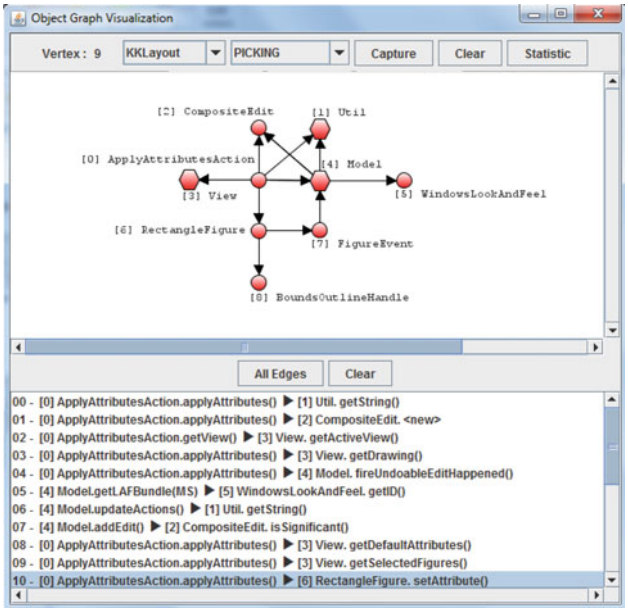


Fig. 13 OG for Bug 1989778

specifically, in this method, a call to a `Figure.changed()` method was missing after calling `setAttribute()`.

6.3 Discussion

Our intention with this case study was to provide initial evidence that the proposed OGs can play an important role in corrective software maintenance tasks. Particularly, the study showed that OGs can be a more effective tool to locate defective program components than for example traditional debuggers. Basically, debuggers usually require maintainers to have a previous knowledge of the source code to define breakpoints near the defective program elements. When this knowledge is not available, debuggers may require maintainers to navigate through several program elements until they discover the components related to the bug reported in the maintenance request. On the other hand, when the bug generates an incorrect behavior in a particular and reproducible state of the program's execution—as in our two examples—the OG tool promptly provides a snapshot describing the dynamic state of the instrumented system. As we have reported, by manually inspecting this snapshot it is possible to discover the exact methods that must be changed to correct the failure. However, it is also important to highlight that the proposed OGs are tightly coupled to the particular execution in which the bug has been reproduced. Because different OGs can be extracted on each execution, it is possible that some graphs do not provide enough information—including both nodes and edges—to correctly understand and locate the defective program components. Therefore, to minimize the chances of having incomplete OGs, it is important that the bugs under analysis have a precise and non-intermittent beha .

Threats to validity Our study presents at least two threats to validity. First, we have evaluated a single system (JHotDraw). Therefore, as usual in empirical software engineering research, we are not claiming that our findings can be generalized to other systems. On the other hand, we have considered real bugs from a system commonly used in software reengineering papers. The second threat is due to the fact that the failure locations have been discovered by ourselves (i.e. the maintainers were the authors of the OG tool). On one hand, this can raise questions on the reproducibility of our findings when the OG tool is used by maintainers that do not have the same expertise on our approach. On the other hand, although we are experts in OGs, we had no knowledge about JHotDraws's architecture, source code, and even its main functionalities, before the study.

7 Related work

Related work can be arranged in three groups: tools and approaches based on static analysis, tools and approaches based on dynamic analysis, and languages for architecture analysis.

Static analysis Scholia is an approach to statically extract hierarchical runtime architectures from object-oriented systems [1,2]. However, there are two main differences between the graphs retrieved by Scholia and the OGs proposed in this paper. First, Scholia relies exclusively on static analysis techniques to retrieve dynamic object-oriented relations. Therefore, at the best, the relations retrieved by Scholia represent an approximation for the concrete relations established in a particular execution of the target system. For example, Scholia cannot capture information about the cardinality of a relation (e.g. the approach can indicate that a collection is composed by elements of a type `A`, but it cannot infer how many objects in fact exist in the collection). As a second difference, Scholia relies on explicit annotations in the code to define the hierarchy that should be followed to display the runtime architecture. This requirement may hamper the application of Scholia in real software development scenarios, due to the effort required to annotate a large and complex system. Moreover, developers are usually reluctant to insert annotations in an existing codebase to avoid the well-known maintenance problems that characterize this technique (a phenomenon usually referred as the annotation-hell [21]). Finally, Scholia also provides support to architecture conformance, i.e. it is possible to check and compare the retrieved diagrams with an intended architecture model.

Womble is a lightweight approach to recover object diagrams by means of static analysis techniques [14]. Therefore, it shares the same advantages and disadvantages of Scholia

Table 1 Comparison with related tools

Feature	OG	Scholia	Womble	Discotect	Briand et al. [8]
Static/dynamic analysis	Dynamic	Static	Static	Dynamic	Dynamic
Extracted model	Objects	Objects	Objects	C&C	Sequence
Code instrumentation	AOP	Annotations	No	Mapping	AOP
On-the-fly/off-line	On-the-fly	Off-line	Off-line	Off-line	Off-line
Summarization	Yes	Yes	No	Yes	No
Conformance	Yes	Yes	No	No	No
Distributed systems	No	No	No	No	Yes

regarding the precision of the retrieved relations. However, unlike Scholia, Womble does not provide means for summarizing runtime objects into coarse-grained components. Therefore, the graphs extracted by Womble have thousands of objects, even for small systems.

Dynamic analysis Discotect is a tool designed to recover dynamic architectures [23,29]. However, instead of hierarchical object diagrams, Discotect extracts flat models based on connectors and components (C&C). For this purpose, Discotect requires developers to provide a map between the runtime trace and architectural events. Although it is less invasive than source code annotations, this map is more complex and requires more information on the target program than the definition of domains in OGs.

Briand et al. [8] have proposed an approach for reverse engineering UML sequence diagrams using dynamic analysis. Similar to the tool described in this paper, their approach relies on aspect-oriented programming for instrumenting the target code. However, their approach is off-line, i.e. in a first step, the instrumented system is executed to generate a trace file; in a second step, this file is off-line processed to generate sequence diagrams. Furthermore, their approach retrieves flat sequence diagrams, and therefore it suffers from the scalability problems that are common to non-hierarchical reverse engineering approaches based on dynamic analysis. On the other hand, they can retrieve sequence diagrams both for centralized and for distributed systems based on Java RMI [28].

Table 1 summarizes the major differences between our approach and the aforementioned systems.

Languages ArchJava is an architecture definition language (ADL) that extends Java with architecture abstractions, like components and connectors [3]. Therefore, ArchJava requires developers to migrate their systems to a new language. OG's alert language has been inspired by the language dependency constraint language (DCL) [24,25]. Basically, DCL allows developers to define acceptable and unacceptable dependencies according to a system's designed architecture. Once defined, such constraints are verified by a conformance tool integrated to the Eclipse platform. Therefore, DCL is an architecture conformance language based on static analysis.

8 Conclusions

In this paper we have presented an on-the-fly and non-invasive approach to extract hierarchical OGs from running systems. As proposed, OGs have the following distinguishing features: (a) they support the classification of objects in coarse-grained entities, called domains; (b) they support the whole spectrum of dynamic relations that can be established in object-oriented systems; (c) they can distinguish objects created by different threads; and (d) by means of an alert language, they can highlight relations that are expected—or that are not expected—between running objects. We have also presented a non-invasive tool to extract and display the proposed graphs. This tool can be weaved to an existing system and therefore it supports on-the-fly visualization of the proposed graphs (i.e. the graphs are displayed and updated as the host system executes). We used this tool to extract real OGs for two systems (myAppointments and JHotDraw). We also reported a study where OGs have been successfully used to locate the defective program elements responsible for bugs reported by real users of the JHotDraw system.

As future work, we intend to (a) apply our extraction tool to other systems, preferably using as subjects professional software maintainers; (b) implement the OG tool as an Eclipse plugin; and (c) evaluate the performance overhead introduced by the instrumentation of the code using aspects; and (d) investigate the benefits of combining our approach with static analysis based techniques, for example, to avoid the extraction of OGs with incomplete sets of nodes or edges.

References

1. Abi-Antoun M, Aldrich J (2009) Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In: 24th Conference on object-oriented programming, systems, languages, and applications (OOPSLA), pp 321–340

2. Abi-Antoun M, Aldrich J (2009) Static extraction of sound hierarchical runtime object graphs. In: 4th International workshop on types in language design and implementation (TLDI), pp 51–64
3. Aldrich J, Chambers C, Notkin D (2002) ArchJava: connecting software architecture to implementation. In: 22nd International conference on software engineering (ICSE), pp 187–197
4. Alves H, Rocha H, Terra R, Valente MT (2010) Uma abordagem para recuperação da arquitetura dinâmica de sistemas de software. In: IV Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS), pp 145–154
5. Anquetil N, Lethbridge TC (1999) Experiments with clustering as a software remodularization method. In: 5th Working conference on reverse engineering (WCRE), pp 235–255
6. Anquetil N, Lethbridge TC (2009) Ten years later, experiments with clustering as a software remodularization method. In: 16th Working conference on reverse engineering (WCRE), p 7
7. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Addison-Wesley, Reading
8. Briand LC, Labiche Y, Leduc J (2006) Toward the reverse engineering of UML sequence diagrams for distributed Java software. IEEE Trans Softw Eng 32(9):642–663
9. Clements P, Shaw M (2009) The golden age of software architecture revisited. IEEE Softw 26(4):70–72
10. Ducasse S, Pollet D (2009) Software architecture reconstruction: a process-oriented taxonomy. IEEE Trans Softw Eng 35(4):573–591
11. Fowler M (2002) Patterns of enterprise application architecture. Addison-Wesley, Reading
12. Fowler M (2003) UML distilled: a brief guide to the standard object modeling language. Addison-Wesley, Reading
13. Garlan D, Shaw M (1996) Software architecture: perspectives on an emerging discipline. Prentice-Hall, Englewood Cliffs
14. Jackson D, Waingold A (2001) Lightweight extraction of object models from bytecode. IEEE Trans Softw Eng 27(2):156–169
15. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG (2001) An overview of AspectJ. In: 15th European conference on object-oriented programming (ECOOP). LNCS, vol 2072. Springer, Berlin, pp 327–355
16. Knodel J, Muthig D, Naab M, Lindvall M (2006) Static evaluation of software architectures. In: 10th European conference on software maintenance and reengineering (CSMR), pp 279–294
17. Krasner GE, Pope ST (1988) A cookbook for using the model–view–controller user interface paradigm in Smalltalk-80. J Object Oriented Program 1(3):26–49
18. Medvidovic N, Taylor RN (2000) A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng 26(1):70–93
19. Passos L, Terra R, Diniz R, Valente MT, Mendonta N (2010) Static architecture-conformance checking: an illustrative overview. IEEE Softw 27(5):82–89
20. Perry DE, Wolf AL (1992) Foundations for the study of software architecture. Softw Eng Notes 17(4):40–52
21. Rocha H, Valente MT (2011) How annotations are used in Java: an empirical study. In: 23rd International conference on software engineering and knowledge engineering (SEKE), pp 426–431
22. Sangal N, Jordan E, Sinha V, Jackson D (2005) Using dependency models to manage complex software architecture. In: 20th Conference on object-oriented programming, systems, languages, and applications (OOPSLA), pp 167–176
23. Schmerl BR, Aldrich J, Garlan D, Kazman R, Yan H (2006) Discovering architectures from running systems. IEEE Trans Softw Eng 32(7):454–466
24. Terra R, Valente MT (2008) Towards a dependency constraint language to manage software architectures. In: Second European conference on software architecture (ECSA). Lecture notes in computer science, vol 5292. Springer, Berlin, pp 256–263
25. Terra R, Valente MT (2009) A dependency constraint language to manage object-oriented software architectures. Softw Pract Exp 32(12):1073–1094
26. Tirelo F, Bigonha R, Bigonha M, Valente MT (2004) Desenvolvimento de Software Orientado por Aspectos. In: XXIII Jornada de Atualização em Informática (JAI), XXIV Congresso da Sociedade Brasileira de Computação
27. Tonella P (2005) Reverse engineering of object oriented code (tutorial). In: 27th International conference on software engineering (ICSE), pp 724–725
28. Wollrath A, Riggs R, Waldo J (1996) A distributed object model for the Java system. In: 2nd Conference on object-oriented technologies and systems, pp 219–232
29. Yan H, Garlan D, Schmerl BR, Aldrich J, Kazman R (2004) DiscoTect: a system for discovering architectures from running systems. In: 26th International conference on software engineering (ICSE), pp 470–479