# Graph-oriented Storage for Fujaba Applications

Erhard Schultchen, Ulrike Ranger, and Boris Böhlen
[schultchen|ranger|boehlen]@cs.rwth-aachen.de
Department of Computer Science 3
RWTH Aachen University
52074 Aachen, GERMANY

## ABSTRACT
Fujaba supports the visual modeling of software applications and the generation of according Java code. During its execution, the runtime state of the generated applications can be saved and restored using the CoObRA framework. In this paper, we present the graph-oriented database DRAGOS for the persistent storage as alternative to CoObRA. Due to the extensive functionality of DRAGOS, the database offers the possibility for introducing new language features in the Fujaba modeling language. Furthermore, the usage of DRAGOS supports rapid prototyping of Fujaba specifications based on the UPGRADE framework.

## Keywords
Graph Storage, Modeling Language, Rapid Prototyping

## 1. INTRODUCTION
Fujaba facilitates the modeling of software applications in a visual way. For this purpose, the developer draws UML-like diagrams, e.g. class diagrams for the static structure and activity diagrams for the dynamic behavior of the software application. Based on this specification, Fujaba generates according Java code, which can be executed in a *Fujaba application*.

Internally, the specified application is translated into a *graph transformation system*, thus the generated code covers appropriate data structures and graph transformations. At runtime, the application's host graph consists of Java objects representing the graph nodes. The execution of the generated transformations leads to modifications of the host graph. All Java objects together form the *state* of the application, and are managed by the Java runtime environment.

Basically, the generated Java code does not provide any support for the persistent storage of the Java objects. To fill this gap, CoObRA [6] can be used, which is a lightweight framework for the storage of Java objects. By adapting the Fujaba code generation to CoObRA, the state of a Fujaba application is automatically stored persistently. A severe drawback of the CoObRA approach is that only *changes* made to objects are registered. CoObRA does not control the *access* to objects, which is required e.g. for providing consistency in case of concurrent changes. To ensure consistency, isolation of concurrent (both reading and writing) operations has to be ensured. For this, CoObRA only provides the isolation level *read uncommitted* known from databases. An application's object structure may therefore only be accessed by a single thread, otherwise the result is not predictable. When the generated application is used by several users concurrently, e.g. if it is part of a distributed system, this restriction is not acceptable.

In this paper, we present the graph-oriented database system DRAGOS[1] [1] as alternative to CoObRA. DRAGOS has been developed at our department, and is used by applications generated with the graph rewriting system PROGRES [7]. As Fujaba also uses graph transformations internally, we have realized the integration of DRAGOS in Fujaba. DRAGOS can utilize existing database management systems (DBMS) for storing graphs. Besides, the DBMS's functionality can be used by DRAGOS, e.g. to allow concurrent access with the DBMS's isolation level . In addition to the persistent storage of runtime data, DRAGOS enables new features in Fujaba, like the evaluation of *derived attributes*.

The paper is structured as follows: Section 2 introduces the DRAGOS architecture, and describes the mapping of the application's runtime data on the database. Based on the functionality of DRAGOS, section 3 shows possible extensions of the Fujaba modeling language. Additionally, the graphical prototyping framework UPGRADE is presented, which can be used for the visualization of Fujaba applications. A summary and an outlook on future work are given in section 4.

## 2. GRAPH-ORIENTED DATABASE
The graph-oriented database DRAGOS is designed to store typed, and attributed graphs in a repository. In this section, we present DRAGOS and its integration in Fujaba. First, the architecture of DRAGOS is described and an introduction to the basic functionality is given. Second, we present the DRAGOS graph model describing the stored graphs. Third, the integration in Fujaba is presented.

### 2.1 DRAGOS Architecture
An overview of the DRAGOS architecture is given in figure 1. The DRAGOS Kernel offers the basic functionality, which comprises defining, creating, modifying and querying graphs. In detail, the DRAGOS Kernel consists of the following components:

[1]DRAGOS is an acronym for *Database Repository for Applications using Graph-Oriented Storage*, and was formerly called GRAS/GXL.

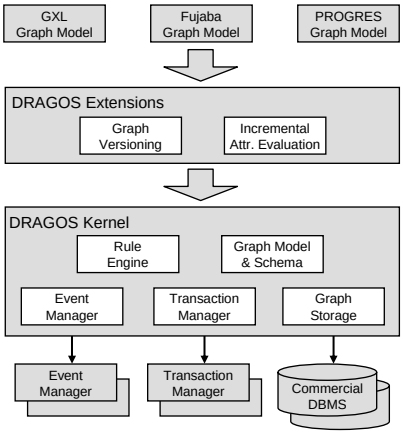


**Figure 1: DRAGOS architecture overview**

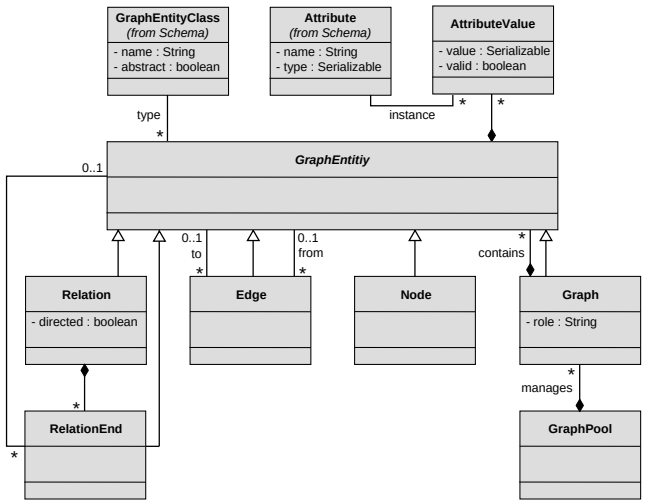


**Figure 2: DRAGOS graph model**

- The Event Manager propagates events raised on graph modifications to registered listeners. Users can choose between different Event Manager implementations by changing a configuration file. For example, building a distributed system consisting of several DRAGOS instances requires a special event manager. This manager has to propagate events between the different DRAGOS systems, which is not offered by the provided standard implementation.
- Using the Rule engine, applications may register event-condition-action rules to perform the defined actions when selected events are raised.
- The Transaction Manager allows to start, commit and rollback transactions. Analog to the Event Manager, the provided implementation can be exchanged easily.
- The Graph Storage provides access to an underlying data storage. For the persistent storage of graphs DRAGOS uses relational DBMS like PostgreSQL, but can also store volatile graphs in main memory[2]. Using existing DBMS allows to use their functionality, such as transactional operations with respect to the ACID properties. The DRAGOS Transaction Manager controls the DBMS transactions in this case.
- The Graph Model and the Graph Schema describe the structure of the stored graphs. The graph model is presented in detail in section 2.2.

Above the DRAGOS Kernel, extensions to the basic functionality can be implemented by extension modules. Until now, versioning of graphs (including the management of configurations) and an engine for the incremental evaluation of derived attributes are available. This modularization of the DRAGOS functionality allows applications to avoid unnecessary overhead, e.g. an application, which does not require versioning, does not activate this extension. Furthermore, DRAGOS can be easily extended by developing new extensions, which can be combined with existing ones.

On top of the DRAGOS architecture, applications are integrated by specialized graph models. This is presented in section 2.3 considering the Fujaba graph model.

[2]The in-memory storage can be serialized to disk when closing the graph pool and thus enables persistent storage, too.

## 2.2 Graph Model

In order to represent a graph in volatile or persistent memory, a formal notation about the entities a graph consists of is required. For this purpose, DRAGOS provides a general *graph model*, which is shown in figure 2. The interface of the Graph Storage (cf. fig. 1) is designed according to this graph model. For example, methods to retrieve all GraphEntitys or to get the source and target of an Edge are provided.

In the DRAGOS graph model, GraphPool contains a set of Graphs. A Graph holds a collection of GraphEntity, which is - according to the inheritance relation - either a Node, a binary relation (Edge), an n-ary Relation or another Graph. A Graph may be contained within another Graph, allowing hierarchical structures. Edges and Relations can connect arbitrary GraphEntitys, even if they are contained in different graphs within the same GraphPool.

Valid graph structures are defined by the graph schema, which is not shown here for the lack of space. In contrast to the graph model which defines the entities a graph consists of, the schema provides the typing system. For this, the graph schema defines classes for every element of the graph model. In figure 2, the GraphEntity refers to a type definition GraphEntityClass which is part of the schema. The schema contains a dedicated Class element for every subclass of GraphEntity.

The DRAGOS graph model allows all GraphEntitys to be attributed. As AttributeValue, serializable Java objects (including primitive data types) and references to other graph entities are supported. Attribute definitions are also part of the graph schema, as indicated by the *(from Schema)* marking in figure 2.

The DRAGOS graph model is based on the graph exchange format GXL [8]. However, GXL does not treat graphs as graph elements. As a consequence, e.g. edges are not al-

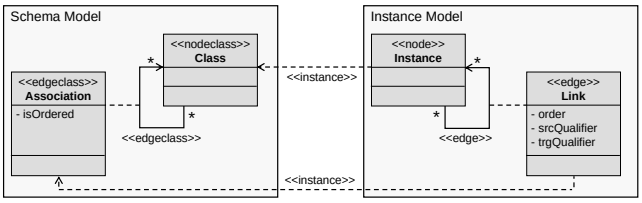


**Figure 3: Fujaba graph model and schema**

lowed to connect two graphs, but only elements contained in these graphs. In the DRAGOS graph model, a graph is a graph entity, too, and may therefore be incident to edges or relations. Furthermore, the DRAGOS model simplifies the GXL model to enable a more efficient implementation. For example, an attribute in the DRAGOS model cannot be attributed by another attribute, which is allowed in GXL to an arbitrary nesting depth. Following the GXL model, an attribute value would have to be treated as a graph element in order to identify it for attributing. In contrast to GXL, the DRAGOS model does not provide ordered edges or relations to reduce overhead for applications not using them. As we will present in the following, support for ordered edges can easily be added.

## 2.3 Graph Model Mapping

DRAGOS supports graph-oriented applications from various domains, and therefore does not tie applications to a common graph model. A common model would either restrict the applications' expressiveness or clutter applications with elements not required. Hence, DRAGOS allows applications to use a *specialized* graph model, which can be designed according to the applications' needs. The specialized graph model for Fujaba is implemented by mapping the Fujaba elements to the DRAGOS graph model. Using the tool SUMAGRAM [4], this mapping can be partially modeled graphically based on UML class diagrams.

For the support of Fujaba applications, we developed a specialized graph model reflecting object-oriented data in the graph database (cf. fig. 3), comprising a *schema model* and an *instance model*. The schema model is composed of Class and Association elements and describes the structure of a graph schema. The instance model consists of elements instantiating the schema model.

The mapping of the specialized graph model on the DRAGOS model is depicted by stereotypes enclosed in <<>>. Elements of the instance model are mapped to the DRAGOS graph model whereas elements of the schema model are mapped to the DRAGOS graph schema.

As mentioned above, the DRAGOS graph model does not provide *ordered edges*. As they are needed by Fujaba applications, these can be integrated into the specialized graph model. For ordered associations, the creation of Link instances automatically assigns an ordering index[3]. This behaviour is implemented as a method in the specialized graph model, which is depicted below in a simplified form:

```
public void createLink (Instance src, Instance trg,
                        Association type, int index) {
  Edge edge =
    this.graph.createEdge (src.getDragosNode(),
                           trg.getDragosNode(),
                           type.getDragosEdgeClass());
  if (type.isOrdered()) {
    reorderEdgesForInsert(src, trg, type, index);
    edge.setMetaAttribute(META_ATTR_ORDER, index);
  }
}
```

The method `createLink` gets two `Instance` objects and an `Association` object from the specialized graph model as parameters. In addition, the parameter `index` is passed as the insertion index for ordered associations; it is not considered for unordered ones. To create the link, according elements of the DRAGOS model are determined using the methods `getDragosNode` and `getDragosEdgeClass`. Using the returned elements of the DRAGOS model as parameters, the method `createEdge` is invoked on the `graph` object representing the application's runtime graph. A new edge of the given type is created between the two nodes by this method. If the association is ordered, existing instances connected to the same `src` or `trg` are reordered according to the insertion position `index`. After reordering, the insertion index is stored as a meta attribute of the new edge. Please note, although a Link class is depicted in figure 3, we do not create a `link` object corresponding to the `edge` object from the DRAGOS model. To fully reflect the depicted specialized graph model, the method `createLink` would have to create and return a new instance of the `Link` class. The reason this was omitted is efficency, as the Fujaba generated code does not use these objects. In fact, another method `retrieveLink` can be used to create these objects when required. This method is called by the UPGRADE framework (c.f. sec. 3.2). We omit to explain the support of qualified associations in this paper, as the implementation is similar to ordered associations.

## 2.4 Specialized code generation

With the specialized graph model, Fujaba applications can store their runtime state in the DRAGOS database. As next step, we have to modify the code generation to make use of DRAGOS. Based on CodeGen2 [3], we developed a plug-in for this purpose.

As basic idea, we aim to change the code generation as little as possible to preserve compatibility with existing applications. For example, we retain the accessor methods for attributes and associations, but change their implementation according to the specialized graph model. As an example, we present the set-method for a many–to–one association. With DRAGOS in use for storing the runtime state, the code is generated as follows[4]:

[3]We could have implemented the same behavior based on edges, which connect links to indicate the ordering. However, this would have increased the overhead for insertion and removal of these links.

[4]Code required for exception handling and transaction management has been excluded here.

```
public boolean setAR (AR value) { [...]
  if (value != null) {
    RepositoryManagerFactory.
      getDefaultRepository().
      createLink(this, "cR",
                 value, "aR",
                 "RM.CR:cR-aR:RM.AR");
  } else {
[...] }
```

This code block retrieves the `Repository` representing the runtime graph of the application. The `Repository` provides the method `createLink` for connecting two objects (`this` and `value`). As Fujaba uses role names to distinguish between the source and the target of a link, these are passed too (`cR` and `aR`). The association's name is passed as last parameter, which is constructed from the attached classes and the corresponding role names. The `createLink` method creates the desired edge regarding uniqueness constraints defined by the schema. If such a constraint is violated, an existing link of the same type is automatically removed. As the method `setAR(AR)` is also used to remove connections between objects by passing `null` as `value`, deletion of links has to be handled in the `else` part of the `if`-statement.

The current code generation creates accessor methods for attributes and associations. These methods could be completely abolished by using DRAGOS. Thus activity diagrams would solely utilize the methods provided by the specialized graph model for pattern matching and graph modifications. The generated code would become clearer because a large number of methods would not be created. However, other parts of the Fujaba environment such as the support for path expressions or the assignment of attributes rely on accessor methods. So, discarding accessors would require far reaching changes to the code generation. Hence we chose the more compatible approach by exchanging the accessors' implementations only.

# 3. APPLICATION AREAS

The usage of DRAGOS as underlying database for Fujaba applications enables several extensions. On the one hand, new features for the Fujaba modeling language can be supported, which are described in section 3.1. On the other hand, the modeled application can be visualized using the UPGRADE framework leading to a configurable graphical user interface shown in section 3.2.

## 3.1 Language Extensions

At runtime, Fujaba can only handle one host graph consisting of connected objects. I.e. separate graphs within one application and even isolated objects within one graph are not supported. This restriction is founded on the fact that Fujaba uses the Java runtime environment for managing its objects. Consequently, Fujaba only supports the search of object structures[5] *consisting of connected objects*. Additionally, these object structures need at least one defined *entry object* for their processing. For example, an entry object is given by the `this`-object or by a method parameter.

[5]To execute a graph transformation specified within a Fujaba activity, a certain object structure (*pattern*) has to be found within the host graph. This structure consists of all objects, which are either obligatory or have to be destroyed within the activity.

Using DRAGOS as underlying database, these limitation can be easily removed: DRAGOS supports the handling of different graphs simultaneously and of isolated objects within one host graph. For example, isolated objects can be found by querying the database for all objects of the requested object's type. This mechanism also enables the search of objects structures without any specific entry object. Another solution to enable isolated objects is to introduce a hidden (singleton) class referencing all objects. However, consistency of these references would have to be assured by the code generation.

Another limitation of Fujaba is the support of only intrinsic attributes within classes, whose concrete values are directly assigned within activities. This does not exploit the capabilities of using graphs as underlying data structure, as these also offer the possibility to define more complex attributes like *derived attributes* in PROGRES.

The values of derived attributes are not assigned directly, but are computed according to other graph objects and their attributes. These computations can be arbitrarily complex and refer to the entire host graph. The attribute is evaluated only on graph modifications, which impact the current attribute value. With the DRAGOS extension Incremental Attribute Evaluation, derived attributes can be easily integrated in Fujaba.

Fujaba allows the definition of *paths*, which aggregate an arbitrary number of associations (edges) of possible different types. Unfortunately, the usage of paths affects largely the runtime efficiency of the Fujaba application, as they have to be evaluated on every employment. The Incremental Attribute Evaluation extension also enables the definition of *static paths*, so that frequently used paths can be materialized in the database. Thus, the path is only evaluated on graph modifications, which change the path, improving the runtime efficiency.

With the support of isolated nodes, the abolishment of necessary entry objects, and the definition of derived attributes, DRAGOS offers new and useful modeling elements in the Fujaba language. Furthermore, the introduction of static paths improves the runtime efficiency of Fujaba applications.

## 3.2 Rapid Prototyping

Fujaba provides DOBS for executing the generated code and for visually representing the runtime state. However, DOBS allows only little customization to the representation and the user interface.

At our department, we developed the UPGRADE framework [2]. UPGRADE allows rapid prototyping by executing code generated from a PROGRES specification. The runtime graph of the generated application is presented visually. Furthermore, users can execute graph transformations and view their affects on the runtime graph. UPGRADE is highly customizable using configuration files, and can be extended through a programming interface. In addition, UPGRADE provides a filter stack to separate the user's view from the underlying data. This filter stack can be customized by the user. New filters can be implemented using the provided API. For layouting the graph structure,

UPGRADE includes a set of layout algorithms: E.g. the sugiyama layout for hierarchical graph structures and the nicolay layout for constraint-based layout.

As UPGRADE offers a lot of functionality, adapting it for Fujaba generated code is desirable. Until now, this was not possible because Fujaba applications do not offer an explicit graph model and schema. Concerning the graph schema, node types are provided through generated Java classes, but associations are not explicitly available. DOBS solves this problem by inferring associations from the respective access methods generated in the incident classes. This behavior would have to be re-implemented in UPGRADE. Concerning the graph model, UPGRADE requires methods to retrieve a list of all nodes and edges from the graph. Fujaba generated applications do not provide direct access to the graph's nodes because the Java runtime environment does not allow to list its objects. Appropriate support like an extra class referencing all objects would have to be added to the generated code. Accessing the graph's edges would require even more extensive changes to the generated code.

In addition to the graph model, UPGRADE requires events to be raised upon changes to the runtime graph for incrementally updating the display. With the standard Fujaba generated code, these events can only be obtained by registering a change listener at each node of the runtime graph. This approach is not efficient for large graphs, especially if only an excerpt of them is displayed. This might also cause memory leaks when change listeners are not properly removed from deleted nodes.

With the introduction of DRAGOS, this problem can be easily solved, as direct access to the graph elements and a central event manager are provided. A simple adapter class has to be implemented, so that UPGRADE can access the Fujaba graph model. This adapter is currently under development.

# 4. CONCLUSION

In this paper, we have presented the graph-oriented database DRAGOS for persistently storing the runtime data of Fujaba applications. For this purpose, we have shortly described the architecture of DRAGOS and its integration into Fujaba. With DRAGOS as underlying database, we have introduced possible extensions of Fujaba. This includes on the one hand extending the Fujaba modeling language, and on the other hand the generation of configurable graphical user interfaces for Fujaba applications.

DRAGOS can be used as an alternative to the existing CoObRA framework. The database offers more functionality than CoObRA, but still has problems in its runtime efficiency. Therefore, we are optimizing and improving the database. Additionally, we are investigating, how the presented language extensions can be integrated best into the Fujaba modeling language.

In addition to UPGRADE, we have also experimented with the re-engineering tool *RePLEX*, which is currently developed in the ECARES [5] project. The tool is implemented as a plug-in for the Eclipse IDE, using a hand-written GEF-based user interface and a Fujaba specification for the application logic. We generated code from the Fujaba specification using the modified code generation for DRAGOS. Furthermore, we modified the GEF edit parts so that change observers are no longer registered at the model objects directly, but at the DRAGOS event manager. This simplifies the management of these observers as already explained for the event management of UPGRADE. So, we provided RePLEX with a DRAGOS based persistency support with low implementation effort.

# 5. REFERENCES


[1] B. Böhlen. Specific graph models and their mappings to a common model. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, $2^{nd}$ Intern. Workshop, AGTIVE 2003*, volume 3062 of *Lect. Notes in Comp. Sci.*, pages 45–60. Springer-Verlag, 2004.

[2] B. Böhlen, D. Jäger, A. Schleicher, and B. Westfechtel. UPGRADE: A framework for building graph-based interactive tools. In T. Mens, A. Schürr, and G. Taentzer, editors, *Graph-Based Tools (GraBaTs 2002)*, volume 72 of *Electr. Notes in Theor. Comp. Sci.* Elsevier Science, 2002.

[3] L. Geiger, C. Schneider, and C. Reckord. Template- and modelbased code generation for MDA-tools. In H. Giese and A. Zündorf, editors, *Proceedings of the Fujaba Days 2005*, volume tr-ri-05-259 of *Technical Report*. University of Paderborn, Germany, 2005.

[4] E. C. Maes. Development of a supporter for the mapping of graph models for GRAS/GXL. Master's thesis, University of Utrecht, May 2005. Nr.: INF/SCR-04-66.

[5] C. Mosler. E-CARES Project: Reengineering of Telecommunication Systems. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, number 4143 in LNCS, pages 437–448, Braga, Portugal, 2006. Springer-Verlag.

[6] C. Schneider, A. Zündorf, and J. Niere. CoObRA - a small step for development tools to collaborative environments. In J. Grundy, R. Welland, and H. Stoeckle, editors, *Proceedings of the Workshop on Directions in Software Engineering Environments, $26^{th}$ International Conference on Software Engineering*, 2004.

[7] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Dissertation, RWTH Aachen, 1991.

[8] A. Winter. Exchanging graphs with GXL. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing – $9^{th}$ International Symposium, GD 2001*, volume 2265 of *Lect. Notes in Comp. Sci.*, pages 485–500. Springer-Verlag, 2001.