Electronic Notes in Theoretical Computer Science 72 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume72.html 11 pages

UPGRADE: A Framework for Building Graph-Based Interactive Tools

B. Böhlen, D. Jäger, A. Schleicher, B. Westfechtel

Department of Computer Science III
Aachen University of Technology
D-52056 Aachen, Germany
westfechtel@i3.informatik.rwth-aachen.de

Abstract

Construction of interactive tools for visual languages is a challenging task. The UPGRADE framework leverages tool builders by integrating application logic and GUI components. It is based on attributed graphs as its internal data model. At the user interface (external representation), graphs can be rendered in multiple ways, including graphics, trees, text and tables. The framework is open, e.g., third-party viewer components may be plugged into the framework.

1 Introduction

Visual languages have become popular in a wide range of application domains. For example, in software engineering a big variety of visual languages is used, including e.g. PERT or GANTT charts for project management, class diagrams, entity-relationship diagrams or data flow diagrams for requirements engineering and design, Petri nets, state charts, etc.

Building tools for visual languages raises a couple of interesting and difficult problems, including the definition of the syntax and semantics of a graphical language, the specification of commands for editing, analyzing, and interpreting, the rendering of diagrams on the screen (in particular, the generation of a nice layout), etc. *UPGRADE* (*U*niversal *P*latform for *GRA*ph-Based *DE*velopment) addresses these challenges. UPGRADE is a framework for developing graph-based applications — not only in software engineering, but also in other domains. The development of UPGRADE was driven by the following requirements:

(i) *Reusability and customizability.* Obviously, reuse is the key issue when developing a framework. On the one hand, the framework's components have to be generally applicable to cover a broad range of possible applications within the desired domain. On the other hand, the resulting tool should provide a specific user interface as well as a specific visualization of the tool's internal data structures. To jointly achieve both goals, the framework's components have to be highly customizable.

(ii) *Decoupling of application logic and user interface.* The application logic is the specific part of every visual application. Consequently, the architecture of the UPGRADE framework has to be clearly separated from it.

(iii) *Independence of a specific storage device.* The framework should be able to operate on top of different kinds of storage devices, e.g. RDBMS or OODBMS or even the data storages of other applications. Thus, the application logic must provide an interface which abstracts from the underlying data storage.

(iv) *Common and generic data model.* At the interface between the application logic and the user interface layer, a common and generic data model is needed. For the UPGRADE framework, we have chosen graphs as a general data model because our tools are supposed to handle documents in visual languages, which in many cases can be easily represented as graphs.

(v) *Distribution.* Generally, tools should allow different users from different sites to access a document in parallel. Therefore, the framework provides support for distribution and synchronization of multiple tool instances.

(vi) *Platform independence.* The UPGRADE framework is written in Java and can be used on every platform on which Java is available.

(vii) *Openness and extensibility.* Even if a framework is highly customizable, this mechanism naturally has its limitations. If the built-in framework components should not suffice to implement a tool as intended, it must be easy to integrate third party user interface components or adapt the existing components by inheriting from them.

In the following, we will show how these requirements are met by the UPGRADE framework, allowing for a convenient construction of visual applications which use graphs as their internal data model.

2 A Sample Application

On top of UPGRADE, several applications have been developed up to now, e.g. tools for high-level authoring support [10] or for software reengineering [16]. As a case study in this paper we will use the *AHEAD* system, an *A*daptable and *H*uman-Centered *E*nvironment for the M*A*nagement of *D*evelopment Processes [12]. In the sequel, we briefly sketch user interface and functionality of the AHEAD system as far as it is required in the context of this paper. In the next sections, we will describe how UPGRADE has been used for the realization of the AHEAD system.

AHEAD is a management system which has been applied to development processes in different engineering disciplines such as chemical, mechanical, and software engineering. It combines project management, engineering/product data management, and workflow management in a single, integrated system. The screen shot displayed in Figure 1 illustrates the user interface of AHEAD. It shows three win-

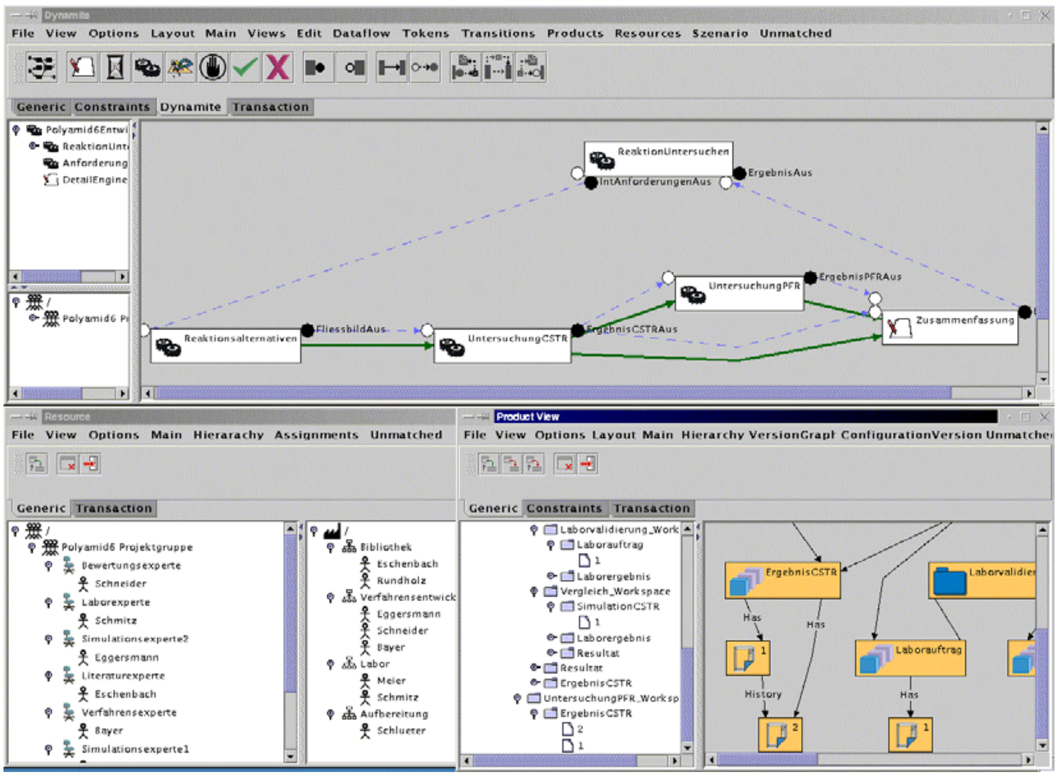


Fig. 1. The AHEAD management system

dows, each of which offers multiple views. In addition to tree and graph views displayed in this screen shot, AHEAD provides table views, e.g., an agenda of tasks assigned to an engineer.

The activity window on the top presents a graphical view on a task net (right-hand side) as well as tree views showing the task hierarchy (top) and the team hierarchy (bottom). In the graphical view, each box represents a task whose execution state is symbolized by an icon (e.g., gear wheels for the state Active). White and black circles stand for inputs and outputs which are connected by data flows (dashed lines). Furthermore, the execution order of tasks is represented by control flows. Project managers are offered commands for editing task nets, assign tasks to engineers, analyzing their execution state, and performing state transitions.

The resource window (bottom left) is used to organize the project team. The right view shows the engineers employed by the respective organization, while the right view displays the positions in a project team. The project manager may define the positions of the project team and subsequently assign engineers to positions.

Finally, the product window (bottom right) is used to manage the products of a development process, i.e., documents such as e.g. requirements definitions, software architectures in software engineering or flow sheets and simulation models in chemical engineering. These documents are subject to version control and are organized in a workspace hierarchy. The left view displays the hierarchy, the right view also shows non-hierarchical relationships.

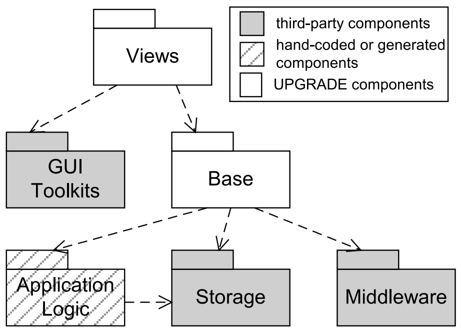


Fig. 2. Framework context

3 Architecture

The UPGRADE framework has been used to build the AHEAD system. How this has been done will be explained in the next section. Before that, we present the framework itself. The architecture is described with the help of UML diagrams [2].

Figure 2 displays the framework's two most important subsystems in relation to third-party components used by the framework and components to be developed when building an application on top of it. UPGRADE internally uses a graph data model including inheritance on attributed nodes and edges. This data model is very general and thus serves a large variety of application domains. However, third-party storage-devices usually do not directly support this data model. Therefore, the `Base` package's main responsibility is to provide connectivity to a storage device holding the application's operational data and abstraction from the storage device's internal data model. The application logic in the form of complex data manipulation operations is made available to the framework through the `Base` package. In addition, the latter uses a middleware component to provide ready-to-use distribution functionality.

The `Views` package's responsibility is to render the data on the screen with the help of suitable GUI components. For this purpose, it uses third-party GUI-toolkits like Sun's second-generation toolkit Swing and ILOG JViews which ships with efficient data structures to hold representation data to enhance rendering speed. The `Views` package is dependent on the `Base` package as it needs access to the offered operations (e.g. to offer them in menus and to query the user for their respective input parameters) and to the data in case of view updates.

Figure 3 shows a simplified class diagram of the framework's architecture. In the sequel, we will explain the architecture bottom-up. The classes `Applica-tionLogic` and `Storage` are not part of the UPGRADE framework. Rather, they belong to the application for which interactive tools are to be developed.

Central to the `Base` package is the class `Base Filter`. It is this class' responsibility to realize the connection to and abstraction from a chosen storage device and to the application logic's offered operations. Above the base filter, UPGRADE assumes a graph data model with attributed nodes and edges which is consistent with the GXL [11] standard graph interchange format. The base filter

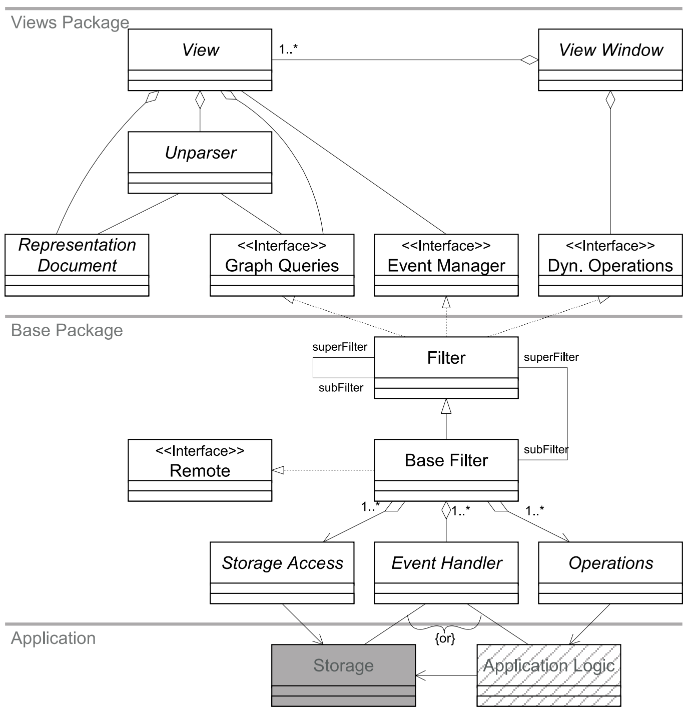


Fig. 3. Class Diagram

aggregates three classes: `Storage Acess` provides read access to the application data and handles the mapping of the application data model to the UPGRADE data model. `Operations` encapsulates the write operations offered by the application. This class can either be generated from an interface description (in case the application logic has been implemented e.g. in C or C++) or might be a generic implementation to read out Java-libraries via reflection. Finally, `Event Handler` serves as a collector of changes performed to the data within the storage device during the execution of an operation. After execution has finished the change events are propagated through the filter stack (see below) and result in automatic updates of all connected views.

The base filter is a special kind of `Filter`. A filter's general duty is to offer access to the logical data, the generated events and the offered operations, which is ensured by the three interfaces each filter has to implement. Generally, this duty is already sufficiently fulfilled by the base filter. However, a filter stack may be used to implement further abstraction steps. The framework contains generic and configurable filters to block out certain edge and node types or to map complex constructs from the logical data to nodes and edges within the internal data model.

In addition to providing a significant abstraction step by hiding away the internals of the storage device and application logic subsystems, the base filter also provides ready-to-use distribution support to enable the building of multi-user, multi-client tools. The latter is based on the freely available middleware Voyager, which allows for the distribution of Java classes, simply by letting the class to be distributed implement the interface `Remote`.

All classes above the base filter are part of the client. A client consists of one or many `View Windows`, each of which contains an arbitrary number of views (like diagrams, trees or text). A view window contains menus and tool bars allowing the invocation of the operations and displays a dialog to request the operation's parameters from the user. The operations offered and their respective parameters are read out via the `Dynamic Operations` interface implemented by the filter. The user may then fill the dialog by selecting objects within the contained views or by typing in data manually.

A `View` is responsible for rendering the information on the screen. The elements to render are held within a representation document. A view's unparser reads out the logical data via the `Graph Queries` interface from its associated filter whenever an update of its owning view is necessary, and fills the view's representation document. The necessity of an update is signaled by the filter with which the view has registered itself through the `Event Manager` interface to receive the set of change events. A view and its respective filter are designed along the observer pattern [9]. Third-party GUI components plugged into the framework usually ship with their own representation document (e.g., in Swing it is called model). Thus, the relations between a view, its representation document and its unparser are very tight. We use the adapter design pattern to implant new components into the framework (cf. Section 4).

4 Application Development

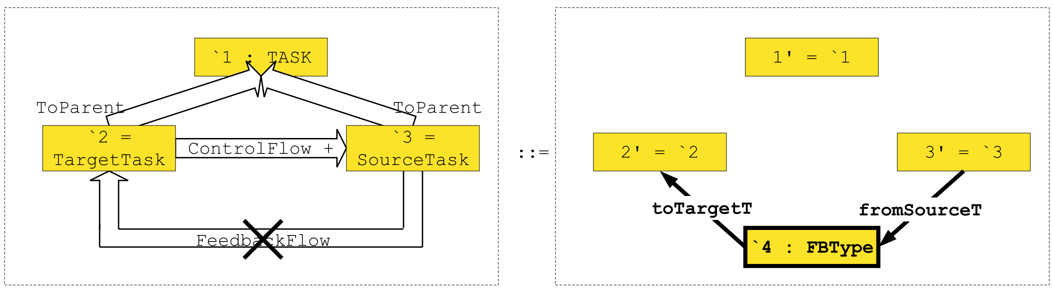
Building a full-fledged application using the UPGRADE framework comprises various steps. In this section, we will first describe how the application logic can be attached to the framework. Then we will discuss how the framework can be extended to fit the needs of a particular application. At the end, we will describe how the different components of the framework can be configured to suit a tool builder's requirements without extending the framework.

The first step in building an application based on UPGRADE is to provide the application logic. This can be done by implementing it manually using for example an OODB as a storage device. However, this may be a tedious and error-prone task. Rather, we use the *PROGRES environment* [19] for generating the application logic from a formal specification. PROGRES is a visual language based on programmed graph transformations.

In the AHEAD system introduced in Section 2, visual languages for activities, products, and resources have been defined in PROGRES. It goes beyond the scope of this paper to discuss the specification underlying the AHEAD system in detail; the interested reader is referred e.g. to [15].

An example of a PROGRES graph rewrite rule operating on the internal graph representation is given in Figure 4. The rule defines the graph transformation to be performed when the user intends to insert a feedback flow into the task net. The tasks to be connected are supplied as input parameters; the created node representing the feedback flow is returned as an output parameter. The left-hand side defines

```
production CreateFeedbackFlow
                (SourceTask, TargetTask : TASK; out NewFeedback : FBType) =
```



```
   return NewFeedback := 4';
end;
```

Fig. 4. Specification of a graph transformation

the graph pattern to be replaced. In addition to feedback flow's source and target task nodes, the graph pattern contains another task node representing the common parent of both tasks. Further positive and negative application conditions ensure that certain paths do exist or must not exist, respectively. For example, the source of the feedback flow to be created must be a transitive control flow successor of the target. Finally, the right-hand side says that a feedback node has to be created and connected with the respective task nodes.

Access to the application logic is provided by a generic base layer. In this base layer, which implements the abstract classes provided by the framework, the following tasks must be accomplished: map the storage device's model to the UPGRADE graph model, provide some means to invoke the operations of the application logic, and generate events to reflect changes in the storage device caused by these operations. The application logic generated from the formal PROGRES specification can be accessed by using a generic wrapper. Thus, the base layer is implemented in a generic way such that it can be used for any PROGRES specification without further adaptation.

These steps are necessary because UPGRADE's graph model is richer than the one used by PROGRES. For example, in Figure 4 the feedback flow is represented by a node of type `FBType` and two edges instead of one edge, because the PROGRES graph model does not support attributed edges. But with such an edge-node-edge pattern it is possible to simulate them. When the PROGRES graph model is mapped to the UPGRADE graph model, simple transformation rules are defined to map an edge-node-edge pattern into an attributed edge. If the application logic is backed by an OODBMS, other transformation rules have to be defined to map the object oriented data model of the application to the UPGRADE graph model. For instance, it is longer necessary to handle the edge-node-edge pattern, but attributes of certain types may be converted to edges.

The framework provides everything that is needed for rapid prototyping. With the application logic generated by the PROGRES environment, a rapid prototype can be created without extending or configuring the framework. Such a basic prototype displays a default graph view and may be used for validating the application logic. However, it does not yet present an appealing user interface as shown in

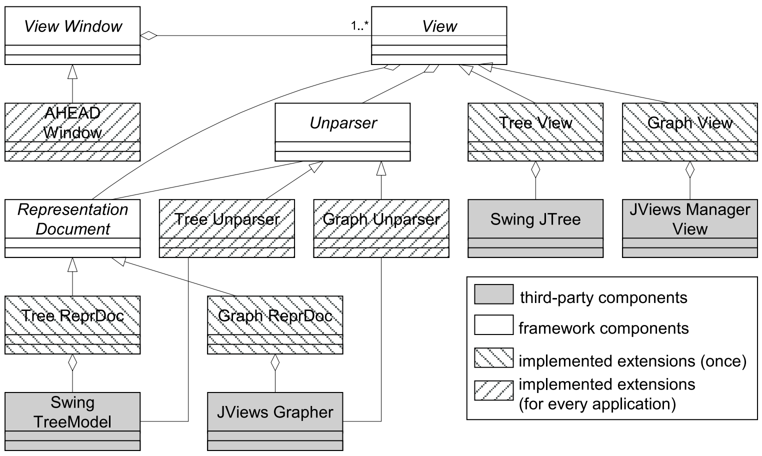


Fig. 5. Extended framework

Figure 1.

The UPGRADE framework is open and extensible. Figure 5 shows an *extension* of the core framework displayed in Figure 3. Note that the core framework does not provide specific views, windows, unparsers, etc. The core may be extended by reusing third-party components (grey boxes). Additional components may have to be implemented (hatched boxes) either once or for each application. For example, we have used Java Swing and ILOG JViews for implementing tree and graph views, respectively.

In the sequel, we will sketch how a third-party component is integrated into the framework. As an example, we will show which steps are necessary to integrate ILOG JViews into the framework as a view for displaying graphs. First, two classes have to be implemented to integrate the new graph view into the framework, a `Graph Unparser` by implementing the `Unparser` interface and a `Graph View` by extending the `View` class. These classes can be found in Figure 5. Our experience showed that introducing a third class serving as a model for the view, called `Graph ReprDoc` in this case, contributes to the extensibility of the entire view. These three classes make up a variant of the Model-View-Controller pattern presented in [9].

The class `Graph View` utilizes `JViews Manager View` to display the contents of our graph model which provides a convenient interface for our unparsers above the JViews graph model provided by `JViews Grapher`. Special graphic representation elements for nodes and edges have been implemented using the standard JViews classes. A detailed look at all these classes reveals that the only difference compared to a standard ILOG JViews application is the unparser which is responsible for translating database updates into representation updates.

A few classes have to be implemented for each application. For example, in the case of the AHEAD system we have implemented an `AHEAD Window` offering a tree view and a graph view on a dynamic task net. Furthermore, the respective unparsers for filling the representation documents depend on the application logic and therefore have to be implemented as well.

The implemented framework extension can be further enhanced by making use of UPGRADE's manifold *configuration capabilities*. These can be used to define the menu structure, provide a tool-bar, describe the specific local layout of graph nodes and edges, tree nodes and table cells. Additionally, the filters can be configured to block out node and edge types or to transform one graph pattern into another. For example, the last option has been used to realize the transformation of a edge-node-edge graph pattern to an attributed edge.

UPGRADE offers a style-sheet mechanism for view elements. This mechanism enables the tool builder not just to determine a view element's local layout according to its type. Moreover the element can be rendered according to its internal state retrieved from the application logic. For example, task nodes in dynamic task nets can be configured to be rendered as rectangles with an icon at the left-hand side symbolizing the task's execution state and the task's name printed at the right. Furthermore, parameter nodes can be rendered as white and black circles with a description written near to the parameter.

Every visual language has special requirements according to the layout of its representation elements. Besides standard layout algorithms like Sugiyama, for hierarchical layouts, or Giotto, for orthogonal layouts, the framework includes a *constraint based layout algorithm*. Constraints are specified in a declarative way and are generated by the unparser for the graph representation. For example, for dynamic task nets we specified constraints that attach the parameter nodes to the left and right of a task node.

5 Related Work

Many tools for visual languages have been built up to date, but usually these are designed for a specific language (e.g., LabVIEW [13] and Prograph [3]). In contrast, UPGRADE offers a reusable framework which is independent of the visual language to be supported. Therefore, it is closely related to meta-CASE tools.

Often, the term meta-CASE tool is used for tools dealing with graphical languages. At the logical level, a language is typically defined in terms of a graph-like or ER data model. At the representation level, graphical representations are considered most important. However, other representations such as trees, tables, or text are offered as well. For programming extensions, a meta-CASE tool typically provides either a scripting language or an application programming interface (API) for Java, C++, etc. As examples, we may list MetaEdit+ [14]and KOGGE [4].

Most meta-CASE tools suffer from several restrictions. First, the language definition includes only the structure. As a consequence, generated graphical editors provide only primitive commands such as insertion or deletion of single nodes and edges. Complex commands have to be programmed with the help of scripts or the API. UPGRADE solves this problem through its integration with PROGRES, which supports the declarative specification of complex operations with graph rewrite rules.

Second, meta-CASE tools are tied to a specific data model and store their data in a home-grown repository. In contrast, UPGRADE defines a standard graph interface in line with the GXL standard graph exchange format [11]. Thus, UPGRADE can be integrated with any storage device by implementing that interface.

Third, meta-CASE tools can be extended only to a limited extent, using a scripting language or an API. This is supported in UPGRADE, as well. In addition, UPGRADE provides standard interfaces for plugging in GUI components as required. Therefore, it is more open than current meta-CASE tools.

There is a small set of competing research prototypes which also rely on graph transformations and therefore address the first restriction in the list above. This includes e.g. DiaGen [17], GenGEd [1], Fujaba [8] and AGG [7]. However, these systems do not strive for providing an extensible, reusable and open framework, i.e., they fail to address restrictions 2 and 3.

6 Conclusion

We have presented UPGRADE, a framework for building tools for visual languages. UPGRADE relies on a graph model for defining the application logic, abstracts from the actual storage device through a graph-based interface, decouples the application logic from the user interface, includes multi-user and distribution support, and provides hooks for plugging in external GUI components, e.g., from ILOG JViews or Swing. It is implemented in Java and hence provides platform independence. So far, the implementation comprises about 70 packages, 250 classes, and 70,000 lines of code. Currently, we are using UPGRADE in our research group for several projects in the areas of book authoring, reverse engineering and process management. However, we are planning to deliver the framework to external users as well.

References

[1] Bardohl, R., *GenGed: A generic graphical editor for visual languages based on algebraic graph grammars*, in: *Proceedings of the 1998 Symposium on Visual Languages (VL '98)*, Halifax, Canada, 1998, pp. 48–55.

[2] Booch, G., J. Rumbaugh and I. Jacobson, "The Unified Modeling Language User Guide," Addison Wesley, Reading, Massachusetts, 1998.

[3] Cox, P., F. Giles and T. Pietrzykowski, *Prograph*, in: M. M. Burnett, A. Goldberg and T. G. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, Manning Publications Co., Greenwich, 1995 pp. 45–66.

[4] Ebert, J., R. Süttenbach and I. Uhe, *Meta-CASE in practice: A case for KOGGE*, in: *Proc. 9th International Conference on Advanced Information Systems Engineering (CAiSE'97)*, LNCS 1250, Barcelona, Spain, 1997, pp. 203–216.

[5] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, "Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools," World Scientific, Singapore, 1999.

[6] Engels, G. and G. Rozenberg, editors, "Proc. TAGT '98 — 6th International Workshop on Theory and Application of Graph Transformation," Springer-Verlag, Paderborn, Germany, 1998.

[7] Ermel, C., M. Rudolf and G. Taentzer, *The AGG approach: Language and environment*, in: Ehrig et al. [5] pp. 551–604.

[8] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java*, in: Engels and Rozenberg [6], pp. 296–309.

[9] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Software," Addison-Wesley, Reading, Massachusetts, 1995.

[10] Gatzemeier, F. and O. Meyer, *Improving the publication chain through high-level authoring support*, in: Nagl and Schürr [18], pp. 255–262.

[11] Holt, R., A. Winter and A. Schürr, *GXL: Toward a standard exchange format*, in: *Proceedings 7th Working Conference on Reverse Eng. (WCRE 2000)*, Brisbane, Australia, 2000.

[12] Jäger, D., A. Schleicher and B. Westfechtel, *AHEAD: A graph-based system for modeling and managing development processes*, in: Nagl and Schürr [18], pp. 325–340.

[13] Johnson, G., "LabVIEW Graphical Programming," McGraw-Hill, Maidenhead, England, 1994.

[14] Kelly, S., K. Lyytinen and M. Rossi, *MetaEdit+: A fully configurable and multi-tool CASE and CAME environment*, in: *Proc. 8th International Conference on Advanced Information Systems Engineering (CAiSE'96)*, LNCS 1080, Heraklion, Greece, 1996, pp. 1–21.

[15] Krapp, C.-A., S. Krüppel, A. Schleicher and B. Westfechtel, *Graph-based models for managing development processes, resources, and products*, in: Engels and Rozenberg [6], pp. 455–474.

[16] Marburger, A. and D. Herzberg, *E-CARES research project: Understanding complex legacy telecommunication systems*, in: P. Sousa and J. Ebert, editors, *Proc. 5th European Conference on Software Maintenance and Reengineering (CSMR '2001)*, Lisboa, Portugal, 2000, pp. 139–147.

[17] Minas, M. and G. Viehstaedt, *DiaGen: A generator for diagram editors providing direct manipulation nad execution of diagrams*, in: *Proc. 11th IEEE Symposium on Visual Languages (VL'95)*, Darmstadt, Germany, 1995, pp. 203–210.

[18] Nagl, M. and A. Schürr, editors, "Proc. AGTIVE — Applications of Graph Transformations with Industrial Relevance," Castle Rolduc, The Netherlands, 1999.

[19] Schürr, A., A. Winter and A. Zündorf, *The PROGRES approach: Language and environment*, in: Ehrig et al. [5] pp. 487–550.