UPGRADE: A Framework for Building Graph-Based Software Engineering Tools

Dirk Jäger, Ansgar Schleicher, Bernhard Westfechtel
Department of Computer Science, Aachen University of Technology
Ahornstr. 55, D-52074 Aachen
E-mail: `(jaeger | schleich | bernhard)@i3.informatik.rwth-aachen.de`

Aachener Informatik-Berichte 00-3

Abstract

Construction of software engineering tools is a challenging task. In the different phases of the software life cycle, a wide variety of software documents is created, including e.g. class diagrams, project plans, entity-relationship diagrams, etc. As a consequence, an equally large set of supporting tools has to be developed with affordable effort.

The UPGRADE framework addresses this challenge. All kinds of software documents are internally represented as graphs; the application code operating on these graphs is generated from a high-level specification. At the user interface (external representation), graphs can be rendered in multiple ways, including graphics, trees, and tables. The framework is open, e.g., third-party viewer components may be plugged into the framework. In this way, the effort for building tools is reduced even further.

There is also another important connotation of openness: UPGRADE supports rapid prototyping of tools generated from graph-based specifications. Many other generators are closed in the sense that the generated code cannot be extended or adapted manually. In UPGRADE, developers can start from a rapid prototype and can then tailor it to suit the specific needs of the respective application.

1 Introduction

Throughout the software life cycle, many different kinds of software documents are created. To this end, a wide variety of notations and 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, software specification languages, as well as textual and visual programming languages.

Many of these notations are graphical ones. Building tools for graphical 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.

The UPGRADE framework addresses these challenges. UPGRADE is a framework for developing graph-based applications — not only in software engineering, but also in other domains (e.g., it is currently being used for building tools supporting chemical engineers). UPGRADE is characterized by the following features:

1. UPGRADE is based on graphs. All kinds of software documents are uniformly represented as attributed graphs.

2. Both the structure of these graphs and the operations performed on them are specified at a high level of abstraction. The specification is written in a language (PROGRES) which is based on programmed graph rewriting.

3. From the specification, code is generated which realizes the database functionality of the respective graph-based application.

4. An initial end-user prototype may also be generated. This prototype provides a graphical user interface to the graph-based application.

5. Starting from an initial prototype, a full-fleged application may be developed. This application may offer multiple views. Either new views are programmed "from scratch", or they are created by configuring already available standard components.

6. To reduce the effort of implementing the framework itself, we are using public-domain and commercial components as far as possible. Moreover, UPGRADE is an open framework inasmuch as application programmers may plug in further components as necessary.

UPGRADE is a successor to ECU, a framework for generating tools from graph-based specifications that was described in [13]. ECU and UPGRADE share several important features. In particular, both are based on code generated from PROGRES specifications and support rapid prototyping (features 1–4 of the list above). However, there are also some crucial differences:

- In ECU, it was essentially not possible to go beyond the generated prototype (feature 5). Some interactive configuration facilities were offered, but ECU was not meant as a framework for application development. In particular, ECU is partially written in Tcl/Tk. Unfortunately, Tcl/Tk made software maintenance very difficult, and it effectively excluded the use of ECU as an application development framework. In contrast, UPGRADE is written in Java, which makes application development much easier.

- ECU was a closed environment (feature 6). In contrast, UPGRADE is open and allows to integrate further external components. Moreover, the UPGRADE framework already contains several public-domain and commercial components.

The rest of this paper is structured as follows: Section 2 introduces the running example to be used throughout the paper. Section 3 provides an overview of the UPGRADE architecture and briefly explains how an initial prototype may be generated from a graph-based specification. In Section 4, the initial prototype is elaborated into a full-fledged application. Section 5 discusses related work, and Section 6 concludes the paper.

2 A Sample Application

On top of UPGRADE, we are currently developing AHEAD, an Adaptable and Human-Centered Environment for the MAnagement of Development Processes [14]. In particular, AHEAD contains tools for managing activities that are based on dynamic task nets [11, 12]. In this paper, we focus on the project management tool which provides a graphical view on a task net and offers commands for planning, analyzing, and monitoring the project status.

A task describes a unit of work to be done. Tasks are organized hierarchically, i.e., a task may be decomposed into a set of subtasks. Control flows define the order in which the subtasks are to be executed. Data flows connect input and output parameters of tasks. In this way, they describe which data are passed among the subtasks. Finally, feedback flows are oriented oppositely to control flows. They are used in case of problems (e.g., errors detected during testing) requiring the re-execution of previous tasks. Typically, a task net cannot be defined in advance and then be executed according to this definition. Rather, the definition — the tasks representing the actual work units as well as their relationships — constantly changes during execution.

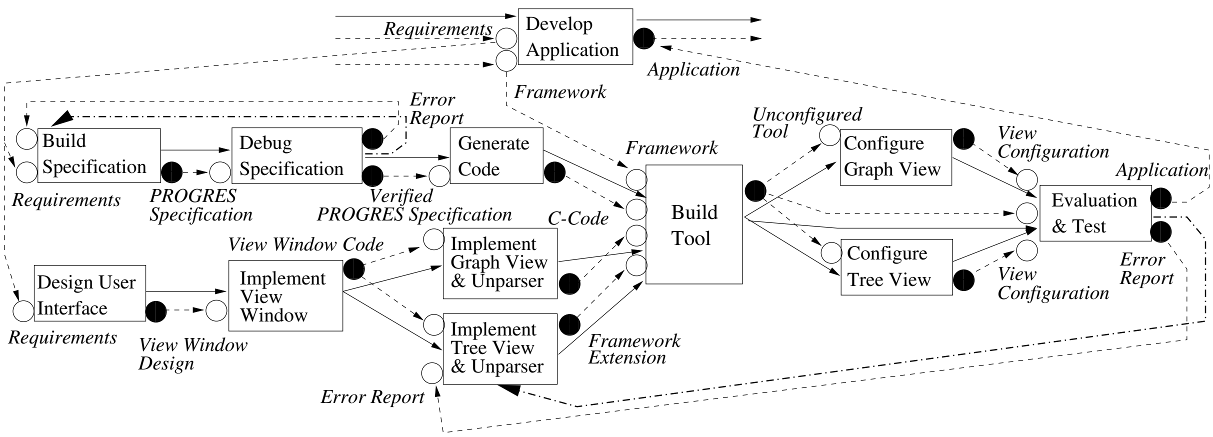


Figure 1: Dynamic task net (external representation)

The task net of Figure 1 does not only serve to illustrate the concepts introduced above; in addition, it explains the process of developing an application with the help of the UPGRADE framework. The task shown on the top represents the process as a whole. The incoming requirements are passed down to the subtasks on the left-hand side; conversely, the developed application is passed upwards by the subtask on the right-hand side.

Development starts with writing a PROGRES specification [20] defining the underlying graph and its operations. The specification is debugged within the PROGRES environment which offers an interpreter for executing graph transformations. In the case of an error, feedback occurs (represented by a feedback flow), and the writer of the specification receives an error report. As soon as the specification is considered correct, code is generated from it.

The diagram shown in Figure 1 is close to the external representation that is eventually rendered on the screen. In contrast, the PROGRES specification defines an internal representation that is required for the detailed and unambiguous description of the tool's

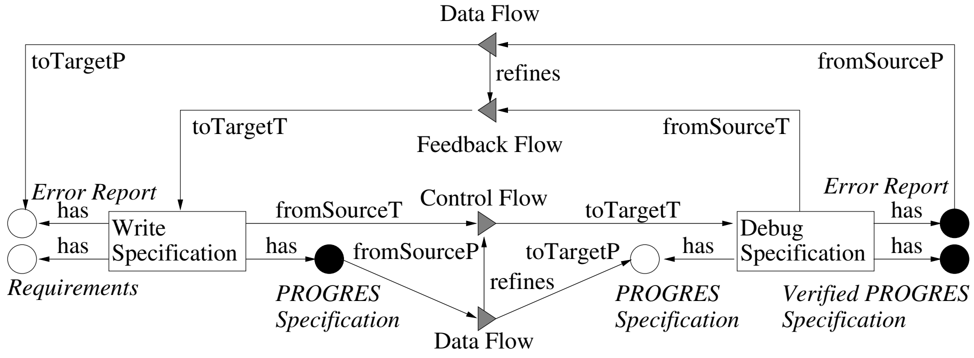
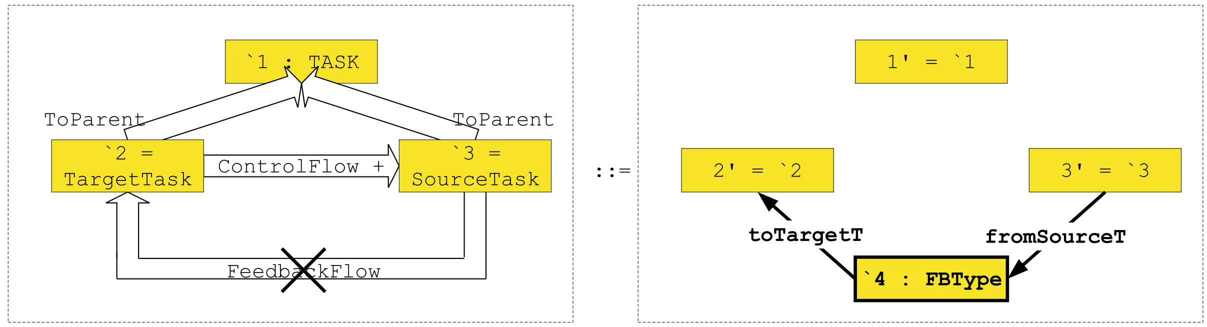


Figure 2: Dynamic task net (internal representation)

database functionality. Figure 2 displays a cut-out of the internal representation of the task net in Figure 1. The internal representation is a typed graph whose nodes are attributed; however, edges may not carry attributes. Furthermore, edges between edges are not allowed. For these reasons, relationships often have to be represented by nodes and adjacent edges.

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




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

Figure 3: Specification of a graph transformation

An example of a graph rewrite rule operating on the internal graph representation is given in Figure 3. 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 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.

Let us now return to the development process of Figure 1. In the simplest case, the UPGRADE framework is used for rapid prototyping. Then, a graphical tool may be immediately generated from the graph specification. In our example, however, this approach is not sufficient. First, we cannot simply display the internal graph 1:1 to the end user. Second, further views are required, e.g., a tree view showing the work breakdown structure.

Therefore, we have to design an appropriate user interface (lower left corner). This involves the design of a view window, which contains a tree view and a graph view. The realization of both views is divided into two steps (to be explained more thoroughly later). In a first step, an unparser is implemented which generates the respective view from the underlying graph. After that, we may build an initial version of our desired tool, consisting of the framework, the generated code, and the implemented views. In a second step, interactive configuration facilities that are available for tool builders are used to realize further adjustments. Finally, we may evaluate and test the configured tool.

3 Architecture

In contrast to our first approach in developing a framework around the code generated from PROGRES-specifications, the UPGRADE framework's aim is to not only support rapid prototyping but also enable a specificator to build full-fledged user-friendly applications with little development effort.

With this aim in mind, it was necessary to build a highly configurable and extendable framework around the generated code. This flexibility has to be reflected within the framework's design which has to offer well separated, easily identifiable and loosely coupled extensible components.

3.1 Generic Architecture

This section will describe the framework's overall architecture which, when unextended, provides ready-to-use rapid-prototyping functionality. Extensibility issues will be discussed in Section 4. Figure 4 displays a rough overview of the framework's generated, third-party and self-developed components. It centers around the generated code which itself is based on the PROGRES Graph Code (PGC) library. The PGC library is an extension layer to the graph-based database management system GRAS for the execution of PROGRES-specifications.

The framework's base layer is responsible for offering the generated operations to the connected views and for handling change events generated by GRAS, which are propagated through the PGC layer. The base layer provides an abstraction from the underlying DBMS GRAS and its proprietary data model. Above the base layer all components work on the UPGRADE-specific data model. Through this abstraction layer GRAS (and the PGC-layer) could be replaced by a different DBMS or even a storing mechanism in volatile memory as long as a filter implements the mapping between UPGRADE and the storage data model.

The view layer's responsibilities are to read out the logical graph data structure via the base layer and fill representation data structures which can be rendered by GUI-components. The view layer is heavily based on third-party toolkits. JViews provides efficient data structures for large representation documents and thus enhances rendering speed. Swing is Sun's second generation toolkit for building graphical user interfaces

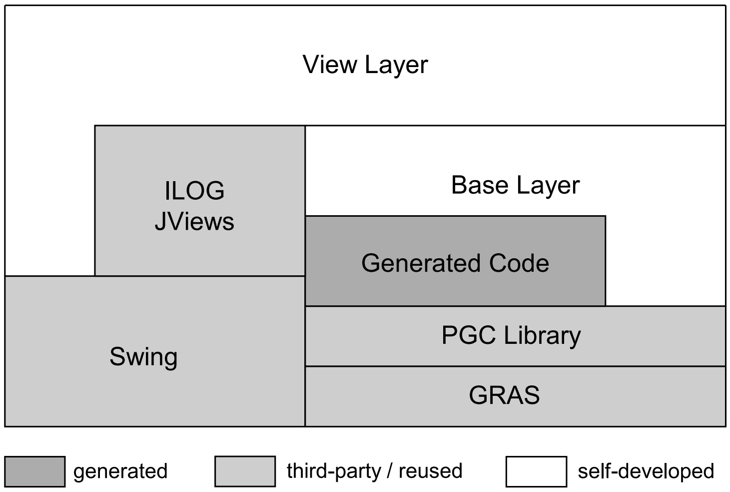


Figure 4: Layered framework architecture

and provides many pluggable components to implement specific views onto the graph like tables or trees.

Presenting the complete design of the framework is not the intention of this paper. However, the main ideas of the object-oriented design can be explained by a (simplified) excerpt which is displayed in Figure 5.

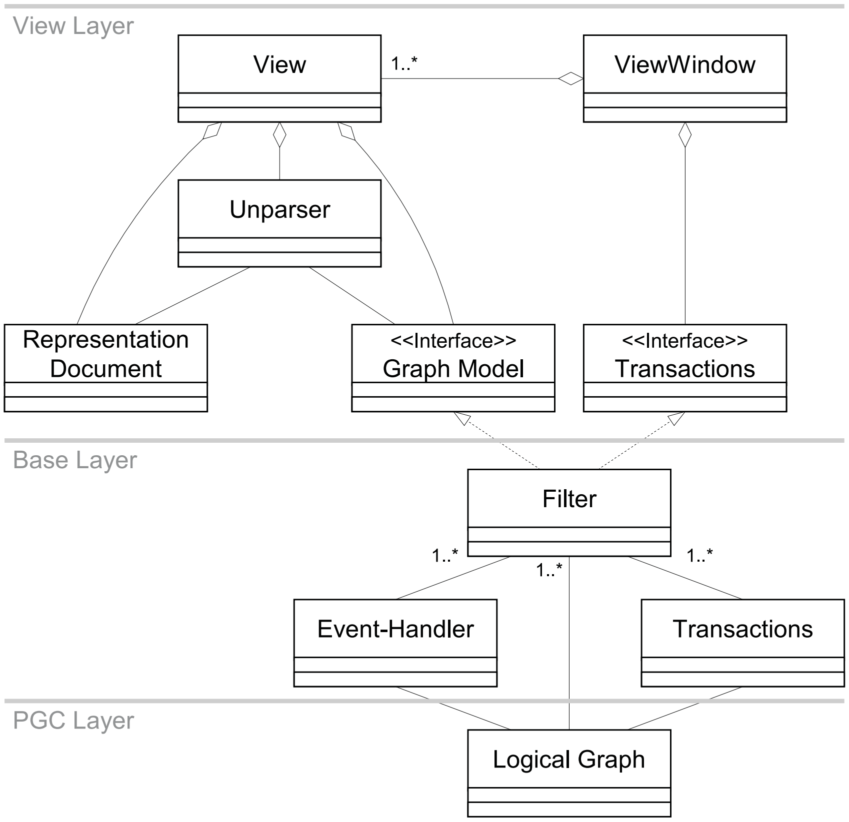


Figure 5: Class diagram

It contains classes from the base-, the view- and the PGC-layer. At the very top stands a `ViewWindow` which is responsible for holding a set of views and for offering the operations available to manipulate the logical graph to the user.

Every View contains its own `Unparser`, its `Representation Document` and an object implementing the `Graph Model` interface which is always the case with objects of class `Filter`. The tasks of these components are as follows:

- The `Graph Model` interface hides the graph data model used by GRAS and offers methods to read out the graph via operations consistent with the UPGRADE graph data model.
- The `Unparser` uses this interface to read out the graph and fill the view's representation document (called `Model` in Swing-terminology).
- It is the `View`'s responsibility to render the contents of the representation document onto the screen.

As it needs an object implementing the graph model interface, every view also contains its own `filter` object. Filters are responsible for the following tasks:

- They implement the mapping from the GRAS data model to the UPGRADE data model.
- They can be configured to block out any set of nodes and edges belonging to certain types and thus significantly reduce the size of the graph for classes accessing the graph via a filter.
- They register with the event handler of the graph database and forward the mapped change events to the owning views.
- They make the operations of the generated C- code library available to the Java-implemented framework. The view window can then build its menus via the `Transactions`-interface using reflection.

It is necessary for a view to contain its own instance of each of these classes for the following reasons: A representation document is usually specific with respect to the GUI-component used. Additionally each instance of a GUI-component might display different cutouts of the graph with different layout alternatives. Only these cutouts are kept within a representation document. For this reason representation documents cannot be shared between views. The same reasons apply to the owning of an unparser as it has to fill representation documents through varying interfaces and with varying data. Through the filters the graph may be reduced to a meaningful structure with respect to the owning view (e.g. if a component for displaying tree-structures is used, the filter might exclude all edges other than those defining the hierarchy within the graph).

The use of an own instance of each of these classes within every view implies another important benefit: No synchronization has to take place between different views. As soon as an update becomes necessary all views can update in parallel. The only synchronization point is the access to the logical graph.

Underneath the filters we find the generated operations operating on the logical graph, the logical graph itself and a class whose objects are responsible for collecting change events from the logical graph which are generated when the graph is manipulated.

The resulting structure allows for the distribution of applications built on the basis of this framework. A client contains everything above and including the filter. A filter stub is placed on the server side and encapsulates the event handler, the logical graph and the generated code. Through this distribution multi-client, multi-user operation of a tool is enabled without any further implementation steps.

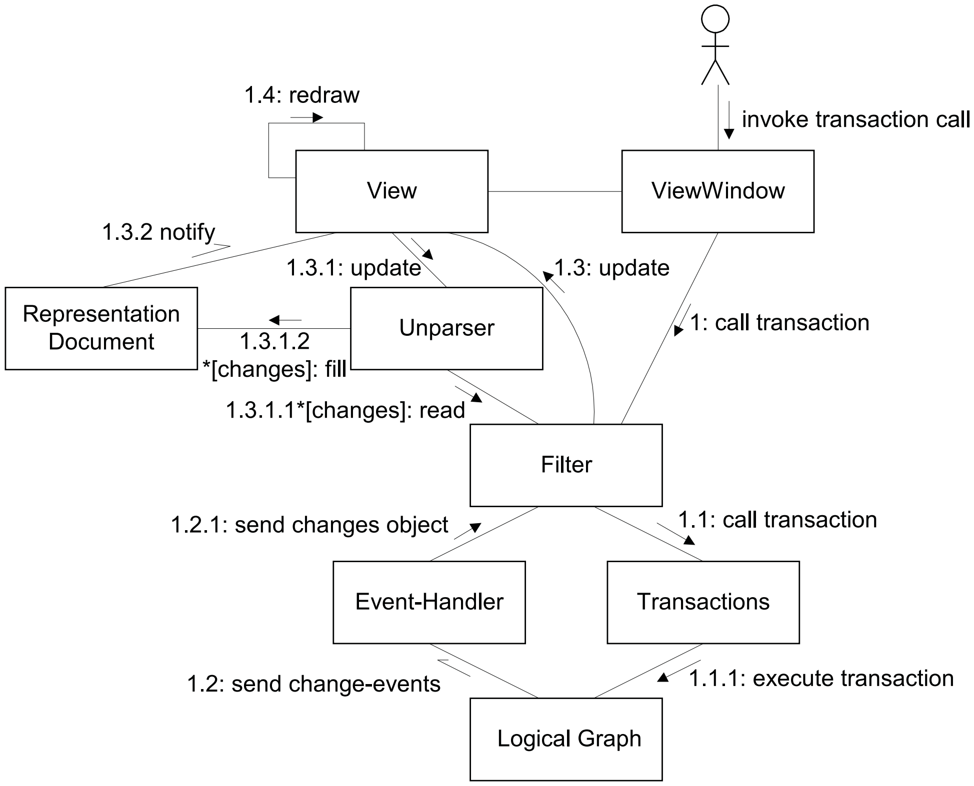


Figure 6: Collaboration diagram

To further illustrate the interaction between these classes the collaboration diagram in figure 6 shows an operation call and its propagation through the framework.

Initially, the user calls an operation to perform changes on the graph. He does so by selecting a menu item or a button in the toolbar. He is asked to fill the parameters of the operation which are an UPGRADE-specific mapping of the operation's parameters within the formal PROGRES- specification. The window handles this event and invokes an operation call within the filter of the current view (1). The filter maps the given actual parameters of the operation to the formal parameters in the GRAS graph data model and invokes the call to the actual generated code (1.1). The latter then manipulates the logical graph within the database (1.1.1). During manipulation events are generated by the DBMS informing any listener of added and deleted nodes and edges and changed attributes. The event-handler collects these events and deletes superfluous events (e.g. nodes that have been created and deleted within the same operation) (1.2). When the operation has completed its execution the event- handler sends the compacted changes-object to every registered listener (1.2.1). In this case listeners are always the filters owned by one specific view. A filter then calls the update-method of its owning view (1.3) who forwards the update request to its unparser (1.3.1). The unparser subsequently iterates over the changes contained in the changes object and reads out the logical graph via the filter to receive information of the change's nature (1.3.1.1). It then propagates the graph changes into the representation document of its owning view if necessary (1.3.1.2). The representation document informs the view of the changes made to its internal data structures (1.3.1.3) (this mechanism is dependent on the GUI-component used) and the view redraws itself to reflect these changes (1.4). Finally, the user who invoked the operation call sees the result of the graph manipulation on the screen.

3.2 Building a Rapid Prototype

From a PROGRES specification a shared library can be generated which offers the specified operations. This library is based on the PROGRES graph code machine which implements heuristic graph pattern matching algorithms to find the left hand side of a graph production within the logical graph [1].

Additionally, PROGRES allows specifications to contain nondeterminism in two ways:

1. The left hand side of a graph production does not have to match exactly one part of the logical graph. Rather the pattern can be found multiple times. If an operation performed after a pattern match fails, the PROGRES graph code machine automatically invokes backtracking and tries to find a different match for the pattern within the logical graph. Thus, the failure of an operation only occurs after all pattern matching options have been tried.

2. Nondeterminism can be explicitly specified as a control flow structure. For example graph transformations can be selected nondeterministically from a specified set. The graph code machine then randomly tries to execute one of the graph transformations. If one transformation fails, backtracking is invoked to select another option.

These mechanisms allow for a very declarative, problem-oriented specification of an application's logic and are therefore ideally suited to rapid prototyping.

Having generated the code library from the specification, it is now possible to link it together with the standard, unextended and unconfigured framework (which provides default implementations of graph, table and tree views) to build an application prototype. A sample prototype built in this manner is shown in Figure 7. Please note that many enhancements to this fully generated prototype can be made without having to implement a single line of code. All configuration options described in Section 4 can be applied.

An unconfigured framework shows the graph in its internal representation (compare with Figure 2). However, our aim is to provide a tool which shows a task net in a way resembling the one shown in Figure 1. The configuration abilities of the framework are not sufficient to reach this level of presentation without implementing extensions to the framework. UPGRADE's full extension and configuration capabilities and their application to our example are presented in the following.

4 Application Development

One of the main reasons a successor to the ECU framework has been developed, is the latter's lack of extensibility. It was highly usable as a rapid prototyping environment and

[1] A detailed description of the PROGRES graph code library and its application within a rapid prototyping framework can be found in [13].

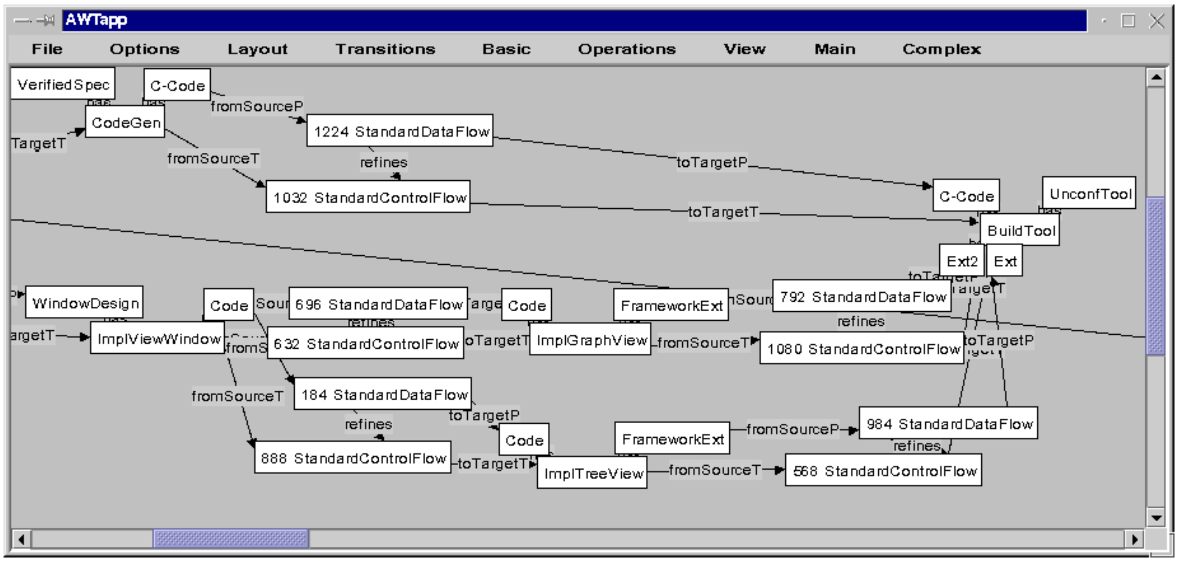


Figure 7: Rapid prototype

allowed configuration to some extent. However, building proper applications with a user-defined graphical interface was impossible. The UPGRADE-framework overcomes this deficit by providing an extendible and configurable view-layer and a generic, configurable, and reusable base-layer.

Building a full-fledged application from a PROGRES-specification using the UPGRADE-framework comprises various steps (similar to those depicted as an example in Figure 1). While specification development is equivalent to the rapid prototyping case, application development includes some implementation and configuration steps which will be described in the following.

4.1 Implementing Framework Extensions

Typically, all implementation steps necessary to build an application take place within the view layer. Generally spoken, an application consists of a set of view windows, containing a number of views of various types. The views have to be connected to a filter implementation.

Figure 8 shows an extended framework design containing all the classes that have to be manually implemented to build an application for process management on the basis of the DYNAMITE specification. First of all, a new window has to be implemented that contains more views than the default graph view. In this specific case it should contain a tree view to display the task hierarchy and a graph view to show the details of a task net's cutout.

If these specific views are not implemented yet (the framework ships with default implementations of a graph, a tree and a table view type) they have to be implemented manually. In this case we will not use the default implementations of the framework but provide our own graph and tree views. Implanting a new view type into the framework requires the realization of three classes per view type: a specific view class inheriting from the framework's class `View`, a specific unparser class inheriting from the framework's class

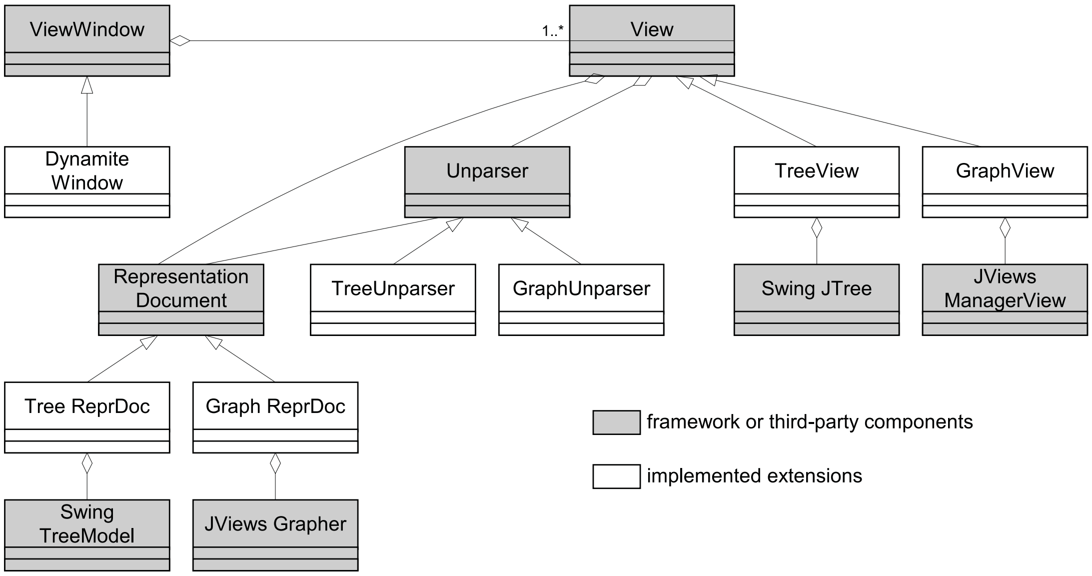


Figure 8: Extended framework

`Unparser` and a class implementing the suitable representation document inheriting from `Representation Document`.

Let us look at the example of a tree view implementation. The class `TreeView` is inherited from `View` but it does not provide the full functionality of a GUI-component for displaying trees. It rather serves as an adapter [2] which encapsulates the Swing-component `JTree` and re-implements the methods from class `View` which are needed by the framework to communicate with the third-party component `JTree`. In the same fashion a class `TreeReprDoc` serves as an adapter to the `TreeModel` from the Swing library. Finally, a specific unparsing strategy has to be implemented which is done within class `TreeUnparser` [3]. The tree unparser implements methods to read out the logical graph via a filter and feed the hierarchical structure implied within the logical graph into the TreeModel. It has to implement a graph traversing strategy and collect the information (e.g. attributes) of the nodes and edges relevant for the hierarchy.

Please note that the unparser communicates directly with the third-party component `TreeModel` and that the rendering of a tree to the screen is done by a third-party component as well which is again directly connected to the `TreeModel` class. Plugging a new GUI-component thus means implementing two adapter classes and a specific unparser. The overall implementation effort to incorporate a new GUI-component is reduced to 300 to 800 lines of code which can be implemented rather mechanically.

In case of the specific graph view the `ManagerView` and `Grapher` classes from the JViews-toolkit are encapsulated by the `GraphView` and `GraphReprDoc` classes. The unparser additionally creates constraints within the representation document that are used by the framework's generic constraint-based graph layout algorithm to layout the graph in a semantically meaningful way for the application domain. In the case of dynamic task nets, constraints are introduced to attach input parameter nodes to the left and output

[2]For details on the adapter design pattern see [8].

[3]The class `View` together with the abstract class `Unparser` and the specific unparser classes forms a strategy design pattern [8].

parameter nodes to the right hand side. Additionally, tasks are layed out from left to right according to their control flow relationships and parent tasks are placed above their children.

4.2 Configuring the Framework

The implemented framework extension, which provides a proper GUI to the user, can be further enhanced by making use of UPGRADE's manifold configuration capabilities. These can be used to influence the menu structure, provide a toolbar, 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. In the following the most important configuration facilities are described.

The commands available within the application stem from the generated code and are placed within a menu structure according to their grouping within the specification. However, this menu structure is quite arbitrary as it contains internal commands that should not be offered to the end user, and might contain an unintuitive grouping of commands. Therefore a tool builder can adapt the menu structure according to his needs. Initially, the menu is generated into an XML-file. Every entry within this file contains the command's name and its parameter profile. By restructuring this XML-file (deleting commands, regrouping commands) a tool-specific menu structure can be created. Additionally, toolbars can be configured in a similar manner. Commands are simply grouped into to toolbar-sections within the same XML-file and an icon to be displayed on the corresponding button is attached to each command description. A resulting menu structure and toolbar is displayed in Figure 9.

Every logical graph contains many node and edge types that have been introduced solely for implementation reasons. Since these are not needed within the framework's data structures it is quite useful to hide them away through the filters. The set of node and edge types to hide can be configured by the tool developer specifically for each view type and can be adapted at runtime by the user for every view instance. Additionally filters allow to map attributed nodes with two adjacent edges to an attributed edge. This can be used in many cases, for example UML editors (associations carry attributes like role names and cardinalities but should still be displayed as edges) or a tool to manipulate and execute dynamic task nets. In the latter case task relations like control flow or feedback flow relations are specified as node classes within the formal specification, because they carry attributes. When a task net is rendered to the screen, however, these relations should be displayed as edges.

Elements within a view should resemble the symbols of the visual language the tool supports. Therefore it has to be possible to configure an element's local layout according to the requirements of the specificator. For this purpose UPGRADE offers a stylesheet mechanism for view elements which enables the specificator to not only fix a view element's local layout according to its type but also according to its internal state. Within the logical graph one attribute of each node is responsible for identifying the node's state. This attribute is then matched to the corresponding stylesheet which determines the rendering of the element on the screen. The external shape (rectangle, circle, point etc.) [4] of an element, its rim size and color, its background color, the font color, family and

[4]Shapes not provided by the framework can easily be plugged in.

size, images to be displayed within the element, attributes to be displayed as text within or next to the node can be configured through user-friendly dialogs [5].

For example task nodes from dynamic task nets can be configured to be rendered as rectangles with an icon to the left hand side symbolizing the task's execution state and the task's name printed to the right of this icon. Parameter nodes can be rendered as white and black circles with a description written to the right hand side of the element.

The rendering of a task net on the screen after having implemented the view window and the two view types, after having generated appropriate layout constraints within the unparser (see above) and after having configured the menus, view elements and filters is presented in Figure 9. The task net displayed by the tool now very much resembles the manually drawn task net of Figure 1.

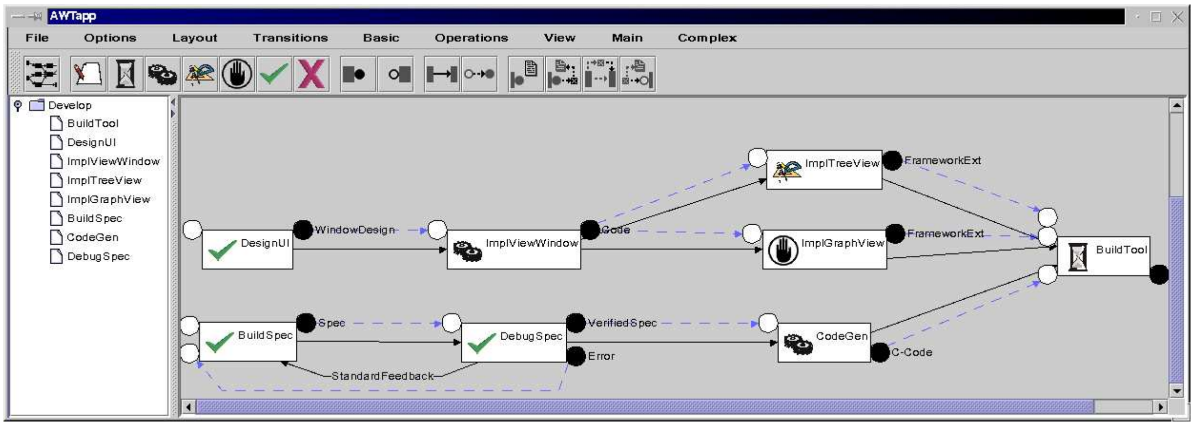


Figure 9: Customized application

5 Related Work

UPGRADE and PROGRES — the latter of which has been described extensively elsewhere [20] — jointly constitute a meta-CASE tool, i.e., a CASE tool for developing CASE tools. Usually, the term "meta-CASE" refers to CASE tools supporting graphical methods for requirements engineering and design. However, it can also be applied to environment generators for textual programming languages, which may be viewed as the first instances of meta-CASE tools. The Cornell Program Synthesizer Generator (CPSG [18]) serves as a well-known representative.

The PROGRES environment itself has been built with the help of a meta-CASE tool called IPSEN [16]. In IPSEN, software documents are internally represented as abstract syntax graphs, which generalize the abstract syntax trees employed in CPSG. Originally, only textual languages were supported; later on, IPSEN was generated to graphical languages[6]. PROGRES is a hybrid language mixing textual and graphical elements. However, in the case of a purely graphical language it is inconvenient to use abstract syntax graphs as an internal representation.

[5] The same mechanisms apply to the configuration of edges.

[6] Basically, graphical languages were implemented on top of textual ones.

While IPSEN is committed to abstract syntax graphs, arbitrary kinds of graphs may be specified in PROGRES (e.g., the task graphs used as a running example). Accordingly, we do not enforce any restrictions on the modeling of graphs when a tool is to be built with the help of the UPGRADE framework.

In contrast to IPSEN, UPGRADE focuses on graphical languages from the very beginning. At the logical level, there is a graphical language specified in PROGRES; at the representation level, graphs, trees, and tables are supported so far[7]. Therefore, UPGRADE is closely related to other meta-CASE tools focusing on graphical language support. However, at least two features distinguish UPGRADE from related CASE tools:

1. PROGRES is an extremely powerful language for graph-based specifications, featuring multiple inheritance, derived attributes and relations, graph rewrite rules with sophisticated pattern matching capabilities, control structures for programming with graph rewrite rules, and backtracking.

2. The architecture of the UPGRADE framework has been designed carefully such that multiple levels of abstraction are separated clearly. In particular, the logical graph is distinguished from its representations. Moreover, it is even possible to replace the graph database with the help of filters.

The main stream of meta-CASE tools for graphical languages is based on some sort of graph, ER, or object-oriented data model. Typically, the elements of a graphical language are specified by some schema defining types (e.g., of entities and relationships) and constraints (logical formulas). From the specification, a graphical tool may be generated that offers some standard functions for editing and analyzing diagrams according to the defined schema. ToolBuilder [1], MetaEdit+ [19, 17], KOGGE [4], and MetaBuilder [9] belong to this category of meta-CASE tools. A crucial advantage of our approach stems from the fact that complex operations are declaratively specified with the help of graph rewrite rules at a high level of abstraction.

There is a small set of competing research prototypes relying on graph transformations. This includes e.g. DiaGen and GenGEd, both of which are described in [2], Fujaba [7], and AGG [6]. All of these systems offer a less powerful specification language than PROGRES, e.g., none of them supports derived data or backtracking. In addition, logical and representation level are mingled together, resulting in a less flexible architecture.

Before building UPGRADE, we looked into other meta-CASE tools in order to save development effort. In particular, we investigated DV-Centro [3], which does offer an architecture with clearly separated layers of abstraction. Finally, however, we excluded DV-Centro because it would have forced us to mirror the logical PROGRES graph as a materialized DV-Centro data structure. As a result of this experience, we introduced the filter layer into the UPGRADE architecture. This layer makes it possible to replace the generated PROGRES code, and it also abstracts from the PROGRES data model. This is achieved through a view mechanism which does not require physical materialization.

6 Conclusion

We have presented UPGRADE, a novel framework for building graph-based software engineering tools. UPGRADE distinguishes itself from other meta-CASE tools with respect

[7]Maybe, text representations will be added later on.

to its powerful underlying specification language (PROGRES) that allows to declaratively specify complex graph transformations at a high level of abstraction. Furthermore, UPGRADE offers a flexible architecture that clearly delineates several levels of abstraction and allows for plugging in commercial components as required.

UPGRADE is a research prototype which is currently under development. It is implemented in Java; so far, the implementation comprises about 70 packages, 250 classes, and 70,000 lines of code. As said before, we are using the commercial user interface tool kit ILOG JViews for graph representations (the logical graphs are stored in our home-grown database system GRAS [15] that has been developed over more than a decade in our group).

Apart from tuning and debugging, there are still many issues to be addressed in future work. So far, we have mainly focused on graphical views. The incremental layout algorithm we are using performs satisfactorily, but customization still requires a lot of hand-coding. Similarly, we would like to improve the functionality as well as the customization facilities for other kinds of views (some interactive menu-based facilities are already integrated, but should be optimized and extended). We also have to explore multi-user issues more thoroughly, and we have not yet attempted to build complete environments operating on multiple graphs.

References

[1] A. Alderson, J. Cartnell, and A. Elloit. Toolbuilder: From CASE tool components to method engineering. In Gray et al. [10], pages 9–18.

[2] R. Bardohl, M. Minas, A. Schürr, and G. Taentzer. Application of graph transformation to visual languages. In Ehrig et al. [5], pages 105–180.

[3] P. C. Brown. Satisfying the graphical requirements of visual languages in the DV-Centro framework. In *Proceedings of the IEEE Symposium on Visual Languages (VL'97)*, pages 84–91, Capri, Italy, Sept. 1997. IEEE Computer Society Press.

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

[5] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific, Singapore, 1999.

[6] C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In Ehrig et al. [5], pages 551–604.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph grammar language based on the unified modeling language and Java. In G. Engels and G. Rozenberg, editors, *TAGT '98 — 6th International Workshop on Theory and Application of Graph Transformation*, number tr-ri-98-201 in Series in Computer Science, pages 112–121, Paderborn, Germany, Nov. 1998. Department of Computer Science, University of Paderborn.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[9] M. Gong, L. Scott, Y. Xiao, and R. Offen. A rapid development model for Meta-CASE tool design. In D. W. Embley and R. C. Goldstein, editors, *Conceptual Modeling — ER '97*, LNCS 1331, pages 464–477, Los Angeles, California, Nov. 1997. Springer-Verlag.

[10] J. Gray, J. Harvey, A. Liu, and L. Scott, editors. *Proceedings of the First International Symposium on Constructing Software Engineering Tools (CoSET'99)*, Los Angeles, California, May 1999. University of South Australia.

[11] P. Heimann, G. Joeris, C.-A. Krapp, and B. Westfechtel. DYNAMITE: Dynamic task nets for software process management. In *Proceedings of the 18th International Conference on Software Engineering*, pages 331–341, Berlin, Mar. 1996. IEEE Computer Society Press.

[12] P. Heimann, C.-A. Krapp, B. Westfechtel, and G. Joeris. Graph-based software process management. *International Journal of Software Engineering and Knowledge Engineering*, 7(4):431–455, Dec. 1997.

[13] D. Jäger. Generating tools from graph-based specifications. In Gray et al. [10], pages 97–107.

[14] D. Jäger, A. Schleicher, and B. Westfechtel. Ahead: A graph-based system for modeling and managing development processes. In M. Nagl and A. Schürr, editors, *AGTIVE — Applications of Graph Transformations with Industrial Relevance*, LNCS, Castle Rolduc, The Netherlands, Sept. 1999. Springer-Verlag. To appear.

[15] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, a graph-oriented software engineering database system. *Information Systems*, 20(1):21–51, Jan. 1995.

[16] P. Klein and A. Schürr. Constructing SDEs with the ipsen meta environment. In *Proceedings Software Engineering Environments*, pages 2–10, Cottbus, Germany, Apr. 1997. IEEE Computer Society Press.

[17] M. Koskinen and P. Martiin. Process support in MetaCASE: Implementing the conceptual basis for enactable process models in MetaEdit+. In *Proceedings Software Engineering Environments*, pages 110–122, Cottbus, Germany, Apr. 1997. IEEE Computer Society Press.

[18] T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, New York, 1988.

[19] M. Rossi. Construction of a CASE tool: The case of MetaEdit+. In Gray et al. [10], pages 33–40.

[20] A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [5], pages 487–550.