Minerva A component-based framework for active documents

Markus Reitz [1] and Christian Stenzel [2]

Software Technology Group
University of Kaiserslautern
Kaiserslautern, Germany

Abstract

Active documents as next generation document technologies may offer a solution to the system immanent problems of traditional information distribution techniques. The purpose of the *Minerva* framework is to serve as a test bed and implementation base for these active document models and applications. Among other things, *Minerva* combines a component-oriented plugin mechanism, flexible and extensible user interactions, and platform independence in a homogenous way. *Minerva* documents are based on nonlinear ordering of information, have a notion of persistency, consistency, a role concept, stateful information and a high degree of flexibility using component-technologies which have been developed during the EASYCOMP project.

1 Introduction

For centuries, the printing technique developed by Gutenberg was the only way to distribute large amounts of information within small periods of time to a large number of people. Technology improvements increased the cost-effectiveness of the production process, but the core concept remained unchanged: Any kind of information is represented statically; a snapshot taken by the author at a certain point of time. This is the reason for the following problems:

(1) Interactions between users and the system which is responsible for information management and rendering are impossible.

[1] Email: markusreitz@gmx.de
[2] Email: c.stenzel@t-online.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

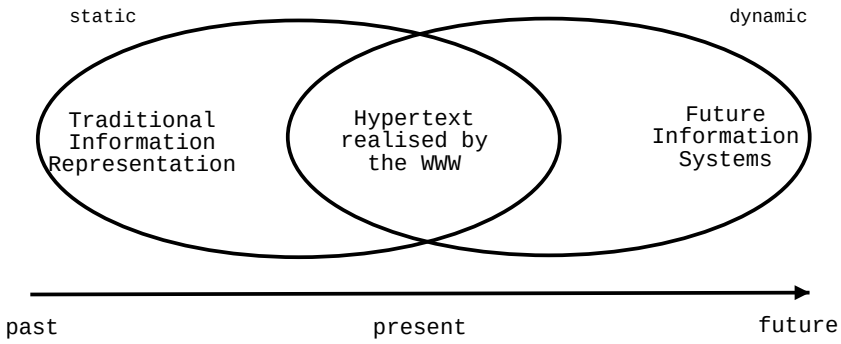

Fig. 1. The WWW as an intermediate step in the transition process from static to dynamic information systems.

(2) Documents are stateless. As a consequence, dynamic content, i.e. time dependent content whose appearance changes over the document or session life time, may not be presented adequately when a "snapshot" based system is used for this purpose.

As these problems are system immanent, being direct consequences of the core concept, only changes in the base technology are able to point out a solution for these problems. These solutions lead away from the dead-end of static information rendering.

Over the last twenty years, computer technology has shaped the emerging information society and is nowadays an integral part of our daily life. Increasing computing power allows for the realisation of visionary ideas like those of Vannevar Bush[5], Douglas Engelbart[6] or Ted Nelson[7]. The breakthrough of the World Wide Web which was invented by Tim Berners Lee in the early nineties of the last century radically changed the way of information retrieval and usage. It can be regarded as a first (rough) approximation of an dynamic information system. Future systems will extend these principles towards a complete solution of the aforementioned problems, overcoming the limitations of snapshot systems. At the moment, the domain of active documents is in permanent flux. Therefore, the development of modular and extensible frameworks supporting rapid prototyping of new concepts and techniques has highest priority. Interestingly, there are no or just a few frameworks which can serve for this purpose.

For the rest of this paper by the term *Active Documents* we mean documents, which provide high-level concepts including polymorphic appearance, statefulness, multi-user capability, and system-user interaction. These concepts are explained later on in this paper.

The purpose of the *Minerva* framework is to serve as a test bed and implementation base for future active document models and applications. With the help of the framework functionality, developers are able to focus their work on new aspects to be implemented without wasting time for standard feature implementation. The core document management functionality provided by the

framework and the flexible, modular, component-based approach allows for quick integration of additional functionality, resulting in an improved overall feature set. As the following sections will show, the requirement of dynamicity quite naturally leads to an component-based approach. Using this point of view, weaving of components and texts can be regarded as a composition process.

The *Minerva* framework is also able to handle documents using functionality which is not located at the same computer the document is rendered on. Therefore, it is possible to establish a kind of *abstract platform*, which is especially helpful when using active documents in mobile or web-based scenarios, enabling the integration of a document viewer into a browser, for example. The paradigm "The network is the document" is a potential consequence of the abstract platform, allowing to ultimatively define minimal requirements without limiting flexibility.

The rest of this paper is structured as follows. In section 2, we briefly give an overview of past and current active document technology and how the notion of active documents developed with the progress of computer technology. We will point out that there still exists a gap between the user-need of flexible and platform independent documents and document composition techniques, and the functionality offered by state-of-the-art document frameworks, even if they are component-oriented.

Section 3 identifies concepts which in our view aid to overcome these shortcomings. In section 4 we introduce the *Minerva* framework in more detail and show how the identified general concepts are practically realised by it. We thereby focus on non-linearity and stateful document presentation. Further we present techniques used in the framework for enabling documents to access distributed functionality. Finally, we draw some preliminary conclusions from our experiences with the framework and give an outlook of future applications and stages of extension.

2 Foundations for active document technologies

Active document technology has been in the main research focus for several years. Nevertheless, the herewith induced transition from an application-centric point of view to a task-centric has not yet really influenced mainstream software. Instead of extending a word-processor to gain the required features for the fulfillment of a certain task, the user has to accept and somehow circumvent the limitations of the word-processing system being used. Software systems are still monolithic, i.e. functionality may only be introduced by changing large amounts of code. As the points of change may be spread over the whole system, this task is tedious and error-prone.

In the nineties of the last century, the available processing power of standard PCs reached a level that could cope with the high demands of multi-media based active document systems. Apple Computer Inc. as a research pioneer in

the area of active document technology invented OpenDoc[4], one of the first commercial technologies that offered the concepts and techniques required for creation and usage of active documents. However, the support of OpenDoc was dropped with the introduction of MacOS X because of several difficulties. Although the abstract document model defined by OpenDoc was operating system independent, many inherent platform dependencies complicated the development of cross platform active documents. Each component used in an OpenDoc document is responsible for its correct rendering at the right position in the document. Despite of a platform independent document model, a platform independent drawing model that would allow uniform access to the graphics hardware on different platforms did not exist. As a consequence, the rendering part of a component had to be reimplemented for each platform, a task that unnecessarily increased the overall development time of OpenDoc components.

Microsoft comparatively early became aware of the need for powerful component models and support for active documents, too. Starting with OLE (Object Linking and Embedding) which supports document composition on the coarse-grain level, Microsoft refined and generalised the necessary feature sets. The results were the general purpose component models COM, DCOM and the latest incarnation COM+ which may be generically used for software development tasks, but also for the implementation of active document systems. In contrast to Apple's approach which focused on active documents, Microsoft seemed to regard this technology as a starting point for further development of component systems, not as the goal component technology has to prepare the ground for. Because of the dependence on Windows, COM+ is one of the most frequently used component models. Nevertheless, it is not platform independent, therefore not a universal component model and so not suited for active document types conforming to our requirements.

The unexpected success of the Java programming language for general software development tasks outside the web-browser applet domain very soon created the need for a component model. The JavaBeans specification[8] is targeted at general software development tasks. In combination with SWING, SUN's platform independent Windowing Toolkit for the Java programming language, rudimentary support for cross-platform active document systems is available.

3 Concepts for active document frameworks

As the last paragraphs have shown, the importance of component technology for general software development processes has long been recognised by major IT companies. Unfortunately, only a small part of the inventiveness was trimmed towards active documents with Apple as one out of a few companies which investigated this topic intensely. As a result of the negligence of active document concepts, state of the art component models are extremely

generalised (CORBA, COM+, JavaBeans) and detached from higher order concepts that would help to structure such systems. Nonetheless, component technology is essential for flexible and extensible active document frameworks.

From a conceptual point of view, the use of active documents should lead to optimal information utilisation. The information society is confronted with the problem of information explosion and overload, an issue for which active document technologies in our view can contribute adequate solutions. Optimisation means enhancement of existing technologies where appropriate and introduction of new techniques and concepts where necessary.

Therefore, this section enumerates shortcomings of traditional document technology and identifies key concepts essential for active document framework implementations which help to overcome these issues.

3.1 Shortcomings of classic document technology

An in-depth look at the drawbacks of nowadays information rendering systems, predominantly books, reveals the following issues of traditional techniques:

- linear ordering of information, non-linearity unsupported
- static content (→ snapshot property)
- stateless document storage
- dependency on well-known data types, no extensibility to cope with future requirements
- missing transconsistency (hot update capability)
- no or limited environment-awareness

For centuries, information having a time-independent representation was dominating. Nowadays, the amount of continuous and time-dependent data is permanently increasing. Gutenberg's technique was targeted at the properties of discrete information and is therefore not able to adequately cope with this switch-over in the requirements.

3.2 Required properties of active document frameworks

The above mentioned drawbacks can be smoothed out when using computer technology for document rendering purposes. The flexibility as a result of the programmability is superior to the use of "dumb" paper used in traditional printing techniques and may add the following features to documents:

Polymorphism

The added value generated by computer-based documents results in a new document type: the *hyper*document resp. the *polymorphic document*. A polymorphic document is a document having a stateful representation, i.e. the data itself and the way it is presented depends on

- the current document user,
- the previous document usage pattern (i.e. usage history) of the current user.

Polymorphism can be used to realise user preferences and personalised documents.

Statefulness

A stateful document presentation is closely related to the concept of polymorphism. Statefulness helps conserving polymorphism over consecutive user sessions or separating different user groups. Note that statefulness implies polymorphism, but not vice versa. On the other hand, statefulness implies persistency. More than one level of statefulness is possible: One can think of an 1:1 relation between users and documents, or a scenario, where for every user and every document more than one state is maintained.

Non-linearity

Although the concept of hypertext, i.e. among other things non-linearity, had its breakthrough in the early 90s by the introduction of the WWW by Tim Berners-Lee, the idea of information fragments being connected by a kind of web is considerably older. The article "As we may think", written by Vannevar Bush in 1945, described the realisation of these concepts by a fictitious machine called *Memex*. Inspired by Bush's work, Ted Nelson coined the word *hypertext* in the mid 60s.

Managing information in a non-linear fashion is superior to the linear approaches obeyed for centuries by classical information storage media like books, for example. Instead of following a predetermined path through the document being defined by the author without any alternatives, non-linearity permits to use the document according to personal preferences of varying readers.

With these considerations in mind, non-linearity is an essential part of the *Polymorphic Document*. In combination with the concept of state-preserving document usage and embedded components which are capable of user interactions, the concept of a *polymorphic document* can be realised with the help of software systems.

Web of components

When implementing polymorphic documents with the help of software, the different parts (e.g. pages, paragraphs, figures etc.) may be identified with components. The dependencies and possible interactions between these objects form a graph structure (cf. Fig.3.2). Therefore, a polymorphic document consists of a web of interacting objects. From this point of view, a supporting framework offers algorithms, which manipulate the graph representing the document.

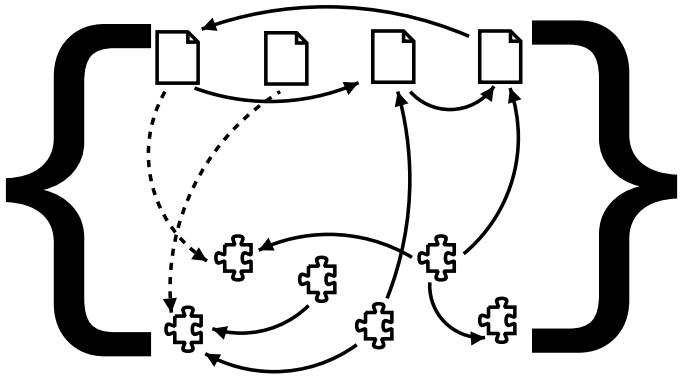

Fig. 2. Active documents are two-staged, complex data-structures, whereby elements of the two stages may interact with each other. Pages may have dependencies to other pages, resulting in non-linear navigation capabilities or may depend on components, resulting in polymorphic behaviour. Components are embedded into pages, i.e. their environment, and may interact with it or with other components, resulting in dynamic, user-specific document behaviour.

Components as building blocks

Although the feature set of early HTML versions was not suitable for all purposes, HTML is still *the* language of the web. The increasing number of additional requirements was incorporated into the existing design using two different techniques:

(1) By extending HTML itself

(2) By using special plugins for the rendering of new data types (SVG, Flash etc.)

The result is an inhomogenous and hard to handle conglomerate of technologies which complicates the creation of cross-platform documents. This emphasises the fact that the only constant is change and future document management systems have to provide mechanisms for the handling of arbitrary data types. A component-based approach where components are used as building blocks for the construction of active documents seems to be a promising solution for this problem.

Due to the well-defined interfaces of components being used in such a document system, persistence issues could be solved easily, too.

Environment awareness

Current component models have only limited *environment awareness*, a property which is essential for active document systems. By environment awareness we mean the ability of components to inspect their local context and trigger actions when necessary.

Example 3.1 (Environment aware component) *Suppose a document consisting of several pages. A per-page glossary, which lists important keywords and their corresponding explanations, may be realised by instantiating a spe-*

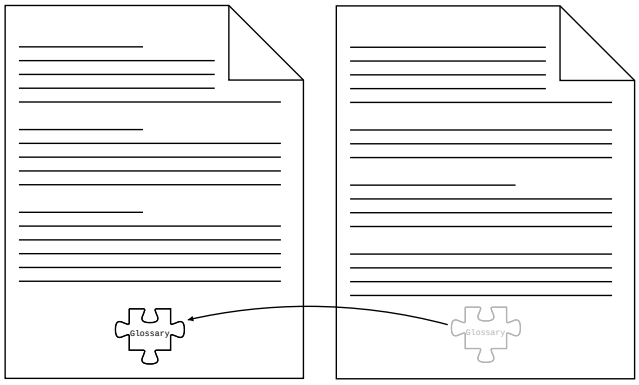

Fig. 3. A glossary as an example for an environment aware component.

cific component. Additionally, this component has to adapt itself when moved or copied to another page (cf. Fig. 3.2).

To fulfill these requirements, knowledge about the surrounding environment is needed. Querying for keywords and explanations of the embedding page results in the desired functionality.

Explicit links between single components or connections between components and certain parts of the text are in our view not sufficient to solve the above stated problem. This category of limited environment interaction also encompasses, for example, connections between chart components and data areas in contemporary spreadsheet applications. Therefore, a powerful extension of the rudimentary available concept of environment awareness has to be designed.

4 From theory to practice: *Minerva*

Theoretical concepts of active document technology have not been investigated with the same intensity when compared to COTS (Component Of The Shelf) approaches. Because of the flux in this domain, a complete reference implementation is not worthwhile at this point of time. A framework which aggregates the most common functionality in a platform independent way is an important premise for fast progress. Interestingly, there are no or just a few frameworks which can serve for this purpose. *Minerva* tries to close this gap by being a helpful test environment for future active document application development.

The *Minerva* framework tries to be a kind of partial reference implementation for the fulfillment of the above-mentioned requirements for future document systems. It is important to emphasise that *Minerva* is not a fully-fledged active document solution but a construction kit with typical modules already implemented. The programmer can make use of these features, but is not forced to. Modules can be substituted but do not have to.

The following paragraphs give an overview of the core functionality which

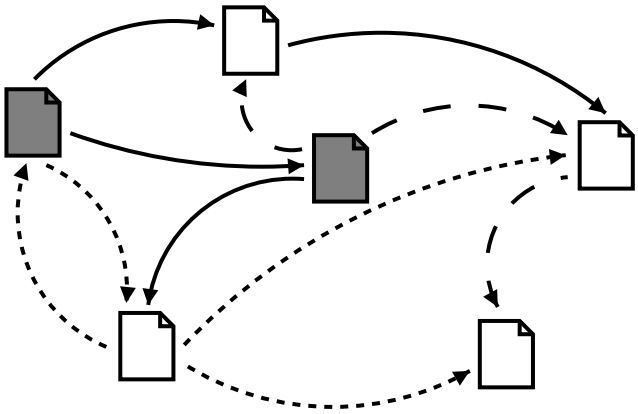

Fig. 4. Using the concept of non-linearity results in several distinct document paths, each of them representing a different document view. Initially reachable pages are marked in grey color.

is, besides generic management functionality for the web of components, essential for the active document developer.

4.1 Non-linear documents

Separately defining each possible path through the web of information is a complex and therefore error-prone task. The *Minerva* framework takes a slightly different approach. Explicit path definitions are replaced by implicit ones using so-called *entry* and *exit conditions*. A document is formed by an arbitrary number of pages, the building blocks for non-linear navigation. Each of these pages is associated with its individual entry and exit condition. If a user wants to enter a page, the corresponding entry condition has to be checked. Access is only granted if the condition is fulfilled, i.e. evaluated to *true*. The same principle is applied in case of exit conditions: A page can only be left if and only if the associated exit condition is evaluated to *true*. By using this concept, authors are forced to take a viewpoint that is focused on the *local context* of the page whose constraints have to be defined (cf. Fig 4). In opposition, using *global contexts*, i.e. explicit enumerations of possible document paths, is often more complex and not applicable when designing larger documents. From this point of view, the concept obeyed by the *Minerva* framework is designed to be more scaleable and therefore more feasible.

Notation of navigation constraints

To define real-world entry and exit conditions in documents, an adequate mechanism for the combination of simple expressions to complex ones is needed. Because of the possibility to hierarchically nest element tags which can easily be parsed, XML is used as textual representation of entry and exit conditions. To be able to express any boolean function, the operators AND, OR

and NOT are defined using well-known semantics. Variables which should be handled as parameters of these operators are enclosed by the start and end tag representing the operator in XML notation.

Available variables

At the moment, the *Minerva* framework supports three kinds of variables:

Component name Using the name of a component, its current state of consistency can be queried. Therefore, this variable has the value *true* if the associated component is in state of consistency, otherwise its value is *false*. The method being used to actually calculate the consistency state is determined in the component's implementation. For example, the multiple choice component is consistent, if the user has selected the correct answer. Not only components can be referenced by their (unique) ID: every page has a unique name and is therefore referenceable by this mechanism, too. The corresponding variable is evaluated to *true*, if all components of the page are in a consistent state.

Visited attribute Besides their unique name, every page owns a *visited* attribute that can be referenced using the well-known dot notation. The attribute is *true* if the corresponding page was already visited by the user and *false* in any other case.

Visitable attribute The second attribute that is existent in every page component is called *visitable*. There is a direct correspondence between the entry condition of the associated page and its *visitable* attribute: the attribute is *true* if the corresponding entry condition is *true* and *false* in case of a *false*-evaluated entry condition. As in case of *visited* attributes, the access is performed using dot notation.

The syntax and semantics of the operators which can make use of these variables is as follows:

`<AND>` $v_0\ v_1\ \ldots\ v_n$ `</AND>` The result of the `AND` operator is *true* if and only if all enclosed variables are evaluated to *true*, otherwise its value is *false*.

`<OR>` $v_0\ v_1\ \ldots\ v_n$ `</OR>` The result of the `OR` operator is *true* if at least one of the enclosed variables is evaluated to *true*, otherwise its value is *false*.

`<NOT>` $v_0\ v_1\ \ldots\ v_n$ `</NOT>` The result of the `NOT` operator is *true* if none of the enclosed variables is evaluated to *true*, otherwise its value is *false*.

For brevity, the well-established relational operators

- equal ($=$)
- greater ($>$)
- greater or equal ($\geq$)
- lesser ($<$)
- lesser or equal ($\leq$)

can also be expressed using XML notation. These operators allow for short-sized formula definitions because relational operators can substitute formulas operating over permutations of variables.

Example 4.1 (Short-cuts) *The expression*

```
<equal count="2" lookFor="true">
  v1 v2 v3
</equal>
```

is a short-cut for the lengthy expression

```
<or>
  <and> v1 v2 </and>
  <and> v1 v3 </and>
  <and> v2 v3 </and>
<or>
```

Using this construction kit, all boolean functions suitable for entry and exit conditions can be build easily. During document usage by readers, the framework is responsible for monitoring and evaluating the entry and exit conditions of all pages belonging to a document. The list of pages actually reachable, i.e. those whose entry condition is *true*, is determined on demand and presented to the user for navigation purposes. From this point of view, the system acts as a pathfinder, guiding the user on his "travel" through the document.

Example 4.2 (Entry condition) *Consider the following entry condition:*

```
<and>
  P1.visited
  P2.visited
  P1:MC1
</and>
```

P1 and P2 refer to pages and P1:MC1 denotes the consistency state of a multiple choice component embedded into page P1. The page belonging to this entry condition can be entered if the following criteria are met:

- *Page P1 was visited by the user*
- *Page P2 was visited by the user*
- *The multiple choice component is in state of consistency, i.e. the user has selected the correct answer(s)*

Display conditions

Besides entry and exit conditions, which may be used to control the accessability of information on the per page level, *Minerva* offers another type of condition which can be used to control the information rendering of page elements. The so-called *display conditions* control the visibility of text areas which have been marked up using the concept of styles already known in conventional word-processing systems. As in the case of entry and exit conditions, display conditions are expressed using the already introduced XML-notation.

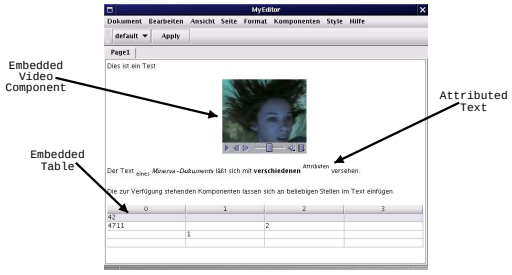

Fig. 5. The framework-based document editor in action. Components can be embedded at nearly any position of the text.

4.2 Component model

The ability to embed more or less active components into the text of a page is a main feature that distinguishes framework based documents from classical (paper-) representations. To be embeddable into a page, a *Minerva* based component only has to implement a special interface. Any component conforming to this interface can be used and manipulated and its state saved and restored by the framework facilities. The component developer does not need to know anything about the framework internals – beyond using the SWING GUI – when using this functionality. Handling components on the interface-level means implementing a black box component model [3].

Interface categories

A closer look at the interface that has to be implemented by framework components reveals the existence of several functional groups, i.e. categories:

(1) Configuration management
(2) ID and namespace management
(3) Management of the component's consistency condition
(4) Persistency
(5) Management of views (see the next section)

Component configuration

The common interface of components usable by the *Minerva* framework also implies a configuration procedure for common aspects of components. These "invariant" parts of a component may be configured with the help of a default configuration dialog.

[3] Although the majority of framework functionality can be assigned to black box component model concepts, there exist some grey box oriented parts. For example, component developers can dynamically extend the component's standard runtime configuration dialog using a pre-defined extension hook. Therefore, *Minerva* uses a component model which is a mixture of a black box and a grey box component model.

Every component inserted into a document has an ID which allows referencing this component by other components or other parts of the framework unambigiously. When using the mechanisms offered by the framework, the system takes care of the uniqueness, avoiding name-clashes which may cause problems during document usage. Another part of the common configuration dialog is the definition of the component's consistency condition which will be discussed in more detail in the next paragraph.

To be as flexible as possible, the common configuration dialog offered by the framework has to be extensible with respect to additional component properties which are specific ones. The concept of a hook, i.e. a well-defined point of interaction between custom code and the *Minerva* framework, is used to cope with this requirement. Defining additional property editors allows for a flexible but partially framework-controlled dialog which reduces the overall component development time.

This is comparable to the concept of property editors used by JavaBeans, which allow entering component specific values and data types. In contrast, the concept obeyed by *Minerva* is not a per-property one but a distinction between variant and invariant properties of a component.

Component consistency

The concept of component consistency is the foundation of polymorphic documents. By defining a two-valued consistency (i.e. consistency states *true* and *false*), a component can propagate information to the surrounding shell, in this case, the embedding page or the document the component is an element of.

The consistency state may be queried by any other component or by framework facilities whose reaction can depend on the result of this query. From the developer's point of view, the consistency definition of *Minerva* is two-staged:

Internal consistency is defined by the component developer and mainly focuses on internal aspects of a component [4].

External consistency is defined by the document author and is a superposition of the internal state of consistency and additional, user-defined constraints. From a conceptual point of view, the external state of consistency is calculated by $c_{external} = c_{user-defined} \cdot c_{internal}$.

From the viewpoint of a framework user, i.e. document author or document reader, the component's state of consistency is equivalent to its external consistency constraint. In constraints used for the definition of page conditions, this external consistency is referenced using the unique full-qualified component name [5].

[4] Focussing just on internal factors is not a must. There could be situations where the internal state of consistency could depend on other, i.e. external, constraints.

[5] The full qualified component name consists of the component's environment identificator, i.e. the ID of the embedding page, followed by the unique component ID, both separated

As the consistency state of a component may be part of a page condition, this mechanism allows the realisation of quite sophisticated polymorphic documents. Suppose there is a multiple choice component, whose internal state of consistency depends on the user-selection out of a set of possible answers. The component is consistent, i.e. a consistency query results in *true*, if and only if the user has selected the right answers out of the available pool of answers. By including the state of such a multiple choice component into the display condition(s) of pages, a mathematical course may present the full text or just the definitions, depending on the user-selection.

User-defined component consistency constraints

User-defined component consistency constraints may be used for grouping components logically. Suppose a set of multiple choice components a user has to answer. By making the consistency of one component dependend on the state of consistency of the other components, this component can serve as the master, i.e. the component whose state of consistency is regarded as the aggregation of the consistency of all slave components. Instead of listing all multiple choice components, a document author can reach the same effect by just referencing the master component.

4.3 Hot-update capability - Views

When inserted into text, a component is able to display its data at the position of insertion. Under some circumstances, displaying data just at one place is not sufficient. To circumvent this limitation, *Minerva* introduces the so called multi-view concept, which is similar to the one used in OpenDoc. A component can be associated with an arbitrary, user-defined number of additional views, i.e. additional locations in the document where components can display their data. Note that these extra views can be located everywhere in the document - there is no spatial dependency to the original insertion position.

An example for a multi-view capable component used in our case studies is the formula component, which, besides displaying its textual representation, supports the possibility of drawing a graphical visualisation anywhere in the document. Consistency between the component's state and its associated views is ensured by the framework.

4.4 State preserving document usage

Because of the polymorphic behaviour of *Minerva* documents which is caused by session-awareness and non-linearity, the framework has to offer appropriate mechanisms to distinguish different document users, i.e. different document sessions. If a user decides to end the current document session, snapshots

by a colon. For example, the full qualified name of a component C1 embedded into a page P1 is P1:C1.

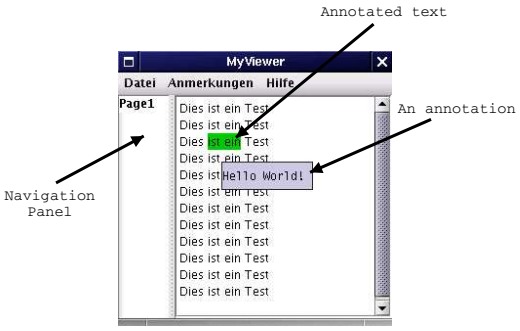

Fig. 6. A framework based viewer application in action. Annotations inserted into the text appear as tool tips when the mouse is moved over the corresponding area.

of the states of all components embedded into the text and the information concerning already visited pages (so called *history information*) have to be taken and stored using serialisation mechanisms.

When a reader starts using the document again, the system checks for the availability of session data. If on-hand, this data is used to reconstruct the document state at the time of the last usage. Besides being able to restore the last valid state of the document, this mechanism can be used to establish feedback loops between users and the author. There are several scenarios one can think of:

(1) By analysing the session information, the author is able to track the usage of the document. The knowledge gained by this process can be utilised to optimise the document according to the usage patterns of the users.

(2) Session information can be used to support the role allocation in a teacher-student scenario. Students process a questionnaire which is part of the document and send their session information back to the teacher. The teacher can read in the session information and analyse the data of the questionnaire.

(3) By using annotations (consult the next paragraph for further details) students can request for help in case of occuring problems when using the document created by the author.

4.5 Annotation management

Utilising annotations to associate text fragments with additional, personal information is a well-known metaphor when working with books. Transferring this feature to hypertext environments adds extra functionality to the framework and helps to increase the acceptance because of well-known behaviour.

Annotations can be associated with any text part of the document. To insert an annotation, the user selects a text fragment and enters the desired annotation. Everytime the mouse cursor is moved over such a marked text area, the associated annotation is displayed.

4.6 The network is the document

By providing the *Minerva* framework with an extensible plugin mechanism, a well-known problem when deploying component-oriented (meaning dynamically linked) software occurs: The active document may rely on special features of the platform when rendering its embedded components. Disregarding this problem means to entail every user of a document to install all necessary software to ensure the proper functionality.

Unlike other approaches for active documents, the *Minerva* framework provides means to handle this problem. It enables the author of a document to specify the locations of the services the document's components require to work properly. With that information, the document viewer can access this services transparently.

In the spirit of HTML, active documents may be spread over a computer network, so different parts of the document are stored on different servers. From that point of view, our concept of distributing document functionality over the network is complimentary to distributing the document content.

Abstract platform

In contrast to applications where the host operating system can be used to fulfill requests for specific features, an active document based approach should depend on software on the document hosting system as few as possible. As a first approach, functionality which is not realised by the framework itself can only be accessed using framework mechanisms as indirection. Because there is no direct access to operating system dependent features, but access via a well-defined and stable interface, the framework realises a kind of *abstract platform*.

Nevertheless, the concept of an abstract platform in its simplest incarnation requires the presence of all additional software on the local system. This assumption is easy to fulfill if the required additional software is part of the operating system by default. In all other cases, this requirement is at least problematic.

An example for this is the LaTeX-component of the *Minerva* framework. This component is used in the context of learning courses and enables the user to input a mathematical formula using LaTeX syntax. This formula can be transformed into an image, which is displayed using the aforementioned view mechanism. To render the formula entered by the user, the component has to make use of the services of a standard LaTeX installation. To assume that for every host this active document is rendered on, a LaTeX distribution is installed and available to the user is just illusory.

Remote features

Instead, requests should be transparently delegated to servers in the network which are able to handle the requests and send back the result to the

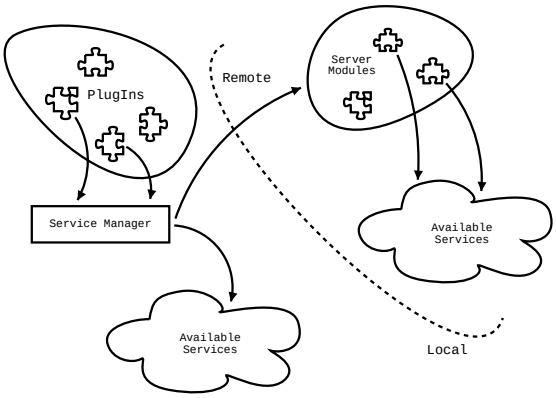

Fig. 7. Schematic illustration of the interaction between local plugins and remote modules handling incoming service requests.

requester. In case of the LaTeX component, a service server handles the incoming request and sends back the resulting graphical representation which then can be displayed by the initiating component or its associated views. Instead of processing the LaTeX commands on the client side, the necessary instructions are transferred to the server which performs the task and sends back the results.

There are several constellations where this kind of delegation could appear. To be able to serve all those requests and to be as flexible as possible, the client based component concept is complemented by modules on the server side as displayed in Figure 7. Each component making use of remote functionality is associated to a module deployed on some server in the network. The cooperation between component and module gives maximum flexibility and platform independence as long as a network connection is available.

While initially the operating system and all software on the host form the *abstract platform*, with our approach taken in the *Minerva* implementation, it is the network which can be seen as the *abstract platform.*

The concept of the *abstract platform* enabled us to implement a document viewer as conventional Java-applet. With that, an active *Minerva* document can be viewed in any Java-enabled browser without the need to install any of the required system services the document makes use of. Therefore the deployment of documents and document rendering software is extremely simplified when compared to a distribution approach using self-contained packages. The pattern of communication between the applet and the services is sketched in Figure 8.

5 Evaluation and conclusions

As we have seen in the preceeding sections, the *Minerva* framework is an appropriate foundation for the development of active documents. The framework establishes a component-oriented document model by an extensible plu-

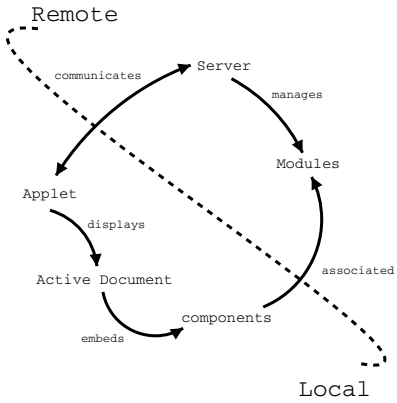

Fig. 8. Illustration of the concept used in case of network-based documents.

gin mechanism. Therefore, a *Minerva* document can actually be characterised as a web of components.

The concept of an abstract platform with components on the client side and corresponding modules on the server side distributed over the net allows for the reduction of platform dependencies to a minimum.

With the help of the framework, we implemented a standalone application for the development and usage of active documents. The aforementioned applet can serve as a mobile access to course materials of computer science lectures. The functionality of the already developed components is sufficient for frequently occuring tasks typical for undergraduate lectures. This includes a formula component which is capable of displaying the full set of LaTeX mathematical symbols, the integration of Jython [6] and the BeanShell [7], a function plotter and multiple choice tests. With the help of the latter component we are able to implement semi-automatic gradable online-exams.

Most of the documents we have created using the *Minerva* framework focus on the e-Learning domain. We plan to integrate the *Minerva* framework and active documents in general into the education of undergraduates more tightly. To this end, we converted conventional lecture scripts into active documents ready to use for the students.

With the help of the feedback which is gained from the users of such materials using the framework's annotation management mechanism, we hope to improve *Minerva* and the corresponding lectures in the future.

[6] Jython is a full compliant implementation of Python 2.1 which additionally allows to use a synthesis of the Python and the Java programming language. See [2] for more details about Jython and [3] for details about the Python programming language.

[7] We use the BeanShell as an interpreter for user-entered Java code. Nevertheless, the BeanShell is by far more than a simple interpreter. Weak typing, interactive mode etc. can help the developer in rapid prototyping of applications. For further information see [1].

Acknowledgements
We gratefully acknowledge the financial support provided by the European Union as part of the EASYCOMP project (IST-1999-14191). We would like to thank Arnd Poetzsch-Heffter for his helpful comments and suggestions.

References

[1] The Beanshell Project Home Page. http://www.beanshell.org.

[2] The Jython Project Home Page. http://www.jython.org.

[3] The Python Project Home Page. http://www.python.org.

[4] Apple Computers Inc. OpenDoc – Information for Developers. http://developer.apple.com/documentation/macos8/Legacy/OpenDoc/opendoc.html.

[5] Vannevar Bush. As we may think. http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm.

[6] Douglas Engelbart. The NLS Demo. http://sloan.stanford.edu/MouseSite/1968Demo.html.

[7] Theodor Nelson. Project Xanadu. http://xanadu.net.

[8] Sun Microsoft Systems. JavaBeans Specification. http://java.sun.com/products/javabeans/spec.html.