Multimedia Metacomputing[†]

Ulrich Marder Jernej Kovse

University of Kaiserslautern
Dept. of Computer Science
P. O. Box 3049
D-67653 Kaiserslautern
Germany

{marder,kovse}@informatik.uni-kl.de

Abstract

The concept of multimedia metacomputing involves the formation of a large scale loosely coupled multiprocessing environment capable of performing complex transformations on media objects. The transformations are provided in the form of operations integrated in special media processing components. The components are described by signatures that denote the runtime environments required for component deployment, the types of media objects the component operations accept and emit and a formal description of transformations they perform. The multiprocessing environment also connects a set of heterogeneous processing resources in which the components are dynamically deployed in order to carry out the transformations. The existing Internet infrastructure is used to connect storages of media processing components, available processing resources and the system controlling the transformation process. By such an environment, we try to realize the concept of delivering global media data without the need to generate specially adapted materialization of the media data in advance. An open "plugable" environment provides the possibilities for both vendors of media processing components as well as providers of processing resources to exploit the potential of the business model involved in offering and providing multimedia services using the existing Internet infrastructure.

1 Introduction

Over the last couple of years, the Internet has significantly improved in the sense of the variety of different media types it involves. The introduction of complex media types, such as graphics, sound and video clips has made the usage of various Internet services, ranging from electronic mail to the World Wide Web (Web), more appealing. However, the existing Internet infrastructure along with its protocols today still is mainly used to merely support the exchange of media objects. It would be useful if we could find a way to combine this exchange with the possibilities of media processing. This way, the Internet infrastructure would be used to form a large, loosely coupled multiprocessing environment where various kinds of media objects could efficiently be found and processed according to the requirements that may be posed by human as well as certain types of software agents.

[†] This work is supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Sonderforschungsbereich (SFB) 501 "Development of Large Systems with Generic Methods".

During the last decade, metacomputing concepts have been invented to support the dynamic distribution of processing components in high-performance multiprocessing environments. While early approaches were targeted at homogeneous massive parallel systems [9], newer approaches, e. g. [3], often exploit the advantages of distributed component architectures. Our proposal of a multimedia metacomputing environment enhances the component-based approach with dynamic configuration, optimization, and multimedia-specific semantics. In particular, this kind of environment involves the following parts:

- mechanisms supporting storage and retrieval of various types of processing components that enable media objects to be transformed according to specific user requirements,
- processing and communication infrastructure supporting the transfer of media objects between processing resources used to carry out the transformations specified by chosen processing components,
- a special control system supporting scheduling and dynamic migration of processing components between the resources as well as initialization and gathering of the results of the media transformation process taking into consideration the availability and the existing processing load for a certain resource,
- a semantic model supporting the description of multimedia processing tasks independently from concrete processing components, optimization strategies, and materialization of media objects.

In the following, we describe the requirements that each of the parts has to fulfill in order to be able to form the heterogeneous open multimedia metacomputing environment.

2 Component-based Multimedia Metacomputing

2.1 Providing and storing media processing components

Using the definition provided by the Unified Modeling Language (UML) Specification [7], a component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files. As such, a component may itself conform to and provide the realization of a set of interfaces, which represent services implemented by the elements resident in the component [7]. Over the last couple of years, the so-called Component-based Software Development (CBSD) [1] has become highly popular primarily because of its promise of reducing costs and time needed to produce software products using components as their building blocks. Today, component technologies such as JavaBeans [8] may be used to support the CBSD.

Using the term media processing component, we refer to a software component providing a set of services used to transform the state of a media object. In our case, it is not necessary that such a component conforms to one of the predefined technologies, such as JavaBeans, for example. A component should merely be deployable in the sense of being able to find an appropriate run-time environment tied to a processing resource where the operations provided by a component may actually be applied to a media object. However, an important aspect of using predefined technologies is that the issues of specifying a set of interfaces, component deployment, and component cooperation are already defined, which makes the usage and combination of such components easier.

After a deployed component receives a media object, it applies a sequence of transformations and delivers a transformed object as its output. An audio transcription component, for in-

stance, first performs a speech recognition and then generates a text object containing the transcript. The transformation process is configurable by a set of parameters, which makes it possible for the user agents to influence the process of applying the transformations.

2.1.1 Describing component services

In our scenario, components are stored using a special storage mechanism. In order to be able to successfully locate and retrieve components according to the transformations of the media object that have to be carried out, a component has to provide not only processing functionality, but also a formal description of the transformation process it supports. Also, the types of media objects the transformations may be applied to, have to be precisely specified. This way, the result of the process of applying transformation operations is exactly defined and the control system knows what kind of result a media object transformation delivers. We call such additional information related to transformation functionality a *component signature*. In case a component provides a formal description of its set of interfaces, as it is the case with the majority of common component technologies, the component signature is to be comprehended as an upgrade of such a description that defines not only which operations may be invoked, but also the exact effects of applying the transformation operations to a media object. Component signatures may not be provided directly by media processing components and may therefore be stored separately. Hence, relationships need to be established between storage representations of components and component signatures. Figure 1 illustrates possible usages of provided component signatures.

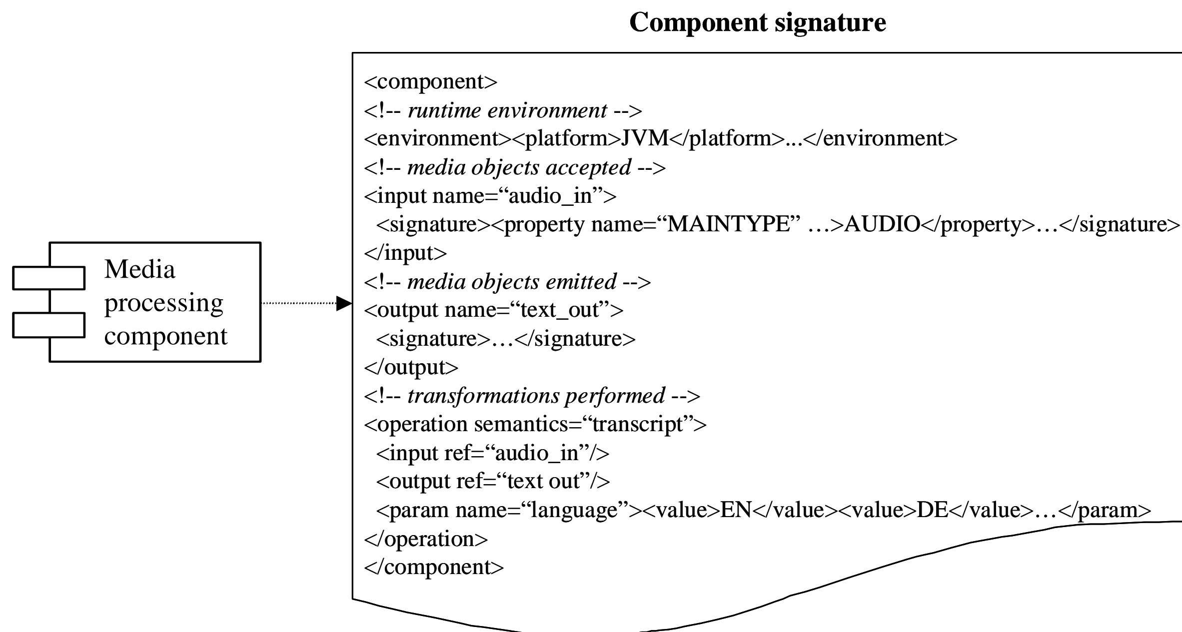


Figure 1: Media Processing Component with Sample Component Signature (shortened)

2.1.2 Managing dependencies between component versions

In a lot of cases, not only components, but also relationships declaring dependencies and possibilities of cooperation between them have to be stored and managed. For example, a component declares by its set of interfaces that it is capable of carrying out an operation that performs a media object transformation as required by the user. However, in the course of this transformation, services of another component are required. For this reason, relationships between components have to be established to make the control system aware of this depend-

ency. This makes it possible to successfully deploy both components in appropriate run-time environments, so that the media object transformation can be carried out. For example, the transcription component mentioned earlier could as well be realized as a composition of two other components: speech recognition and text generation. However, the storage mechanism should also be capable of storing and managing different versions of the same media processing components. A new component version may provide improved functionality related to media processing, but may prove to be incompatible with other components the initial component depends on. Therefore, various versions of the same component should be stored and managed by the storage mechanism. Moreover, the relationships between the components that define the dependencies should be refined in such a fashion that it is possible to choose and deploy the appropriate configuration of component versions that fits the desired context of a media object transformation. Hence, following a similar approach as with components, *configuration signatures* are required. Figure 2 illustrates an example of configurations of different component versions. Mahnke et al. [4] describe more general advantages of using customized version control in repositories.

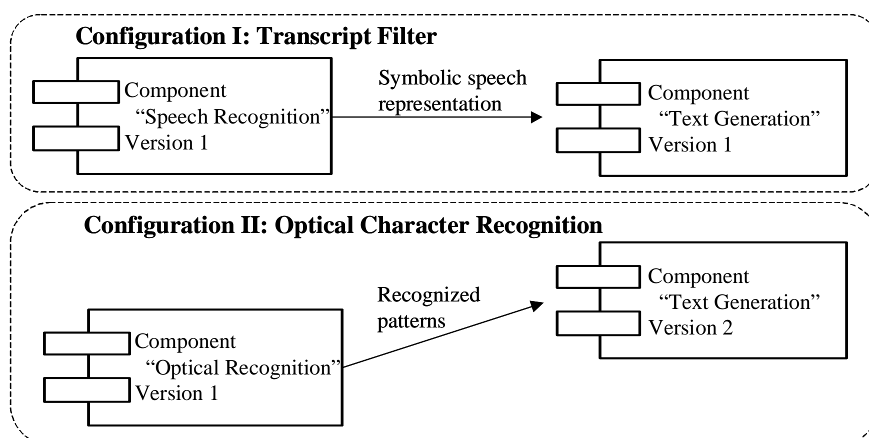


Figure 2: Configuration Example

2.1.3 Storing media processing components

Because of the complex requirements related to storing meta information to support searching of the appropriate components and managing valid configurations of component versions, we think that the services of a file system offer only limited functionality to provide storage facilities for media processing components. Therefore, it is essential to provide storage facilities in a form of special component repositories. Such a repository usually provides standard amenities of a database management system (DBMS), such as data model, query facility, view mechanism, and integrity control. However, in our case the functionality is upgraded using special value-added repository services, such as efficient searching for stored components according to desired functionality as well as version and configuration control.

Note that a repository may be distributed in general such that various media processing components may actually be stored at different locations. Various component vendors may produce components providing media processing functionality, provide component signatures for these components, store them at their own location and register them with the repository. This way, a repository seamlessly integrates various storage locations in a single virtual storage environment, which makes it possible for the control system to find and retrieve the components in a simple fashion (see Figure 3).

2.2 Processing and communication infrastructure

The *processing and communication infrastructure* enables the control system to locate the resources available for media processing, transfer data related to a media object to these

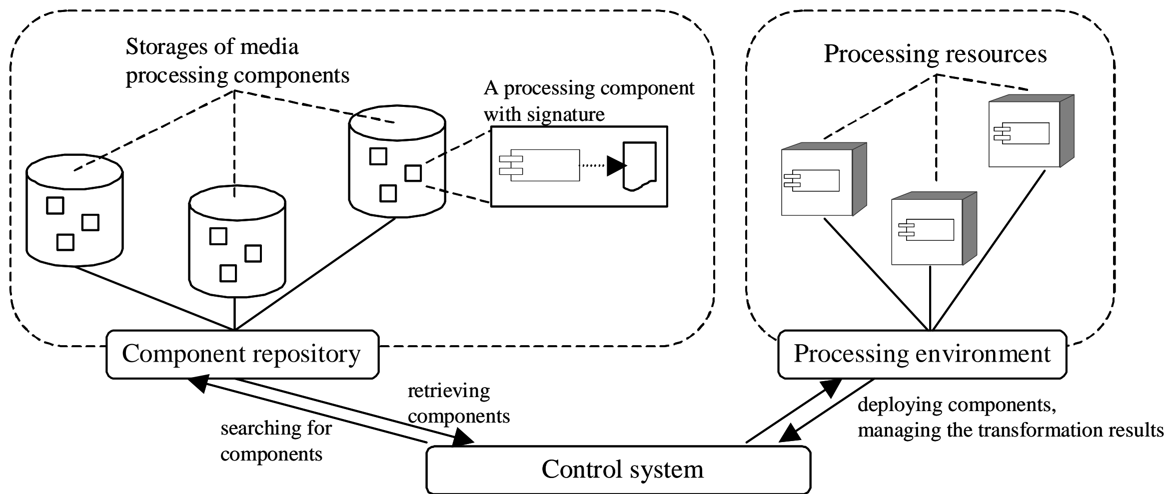


Figure 3: General Architecture of the Multimedia Metacomputing Environment

resources and initiate the transformation process that may take place at various resources in a parallel or sequential manner.

Processing resources form special run-time environments in which media processing components are deployed in order to perform transformations on the media objects. Note, since our multiprocessing environment involves multiple types of different processing components, the communication infrastructure links heterogeneous run-time environments. Run-time environments register with the control system, where the exact environment type and the possibilities of component deployment are described.

2.3 Control system

A special *control system* is used to direct the transfer of components to appropriate run-time environments as well as the transfer of media objects between the components on various stages of the performed transformations.

As an input, the control system receives data related to the media object and a formal description of a transformation that has to be performed on the object. The description may be given as a list of commands in a special-purpose high-level language like VMML [6] (cf. Figure 4). The system tries to analyze the description to obtain a list of basic transformation tasks needed. Using special query capabilities of the distributed repository of media processing components, it tries to locate the components capable of performing the requested transformations. In the process of searching for the components, component signatures stored in the repository are used. In case the component is capable of carrying out the transformation, additional information about dependencies of the component to other components may be delivered so that it is possible to choose a valid configuration of cooperating components. Next, run-time environments needed to deploy the components have to be chosen among the environments that have registered with the control system. After component deployment, a media object that needs to be transformed is passed as an argument to the components along with additional arguments used to configure the transformation process performed by the components. The process of deploying appropriate components to run-time environments, configuring them and passing the media object is repeated for each of the basic transformation tasks. As a result of this process, an object transformed according to the user specification is delivered to the user.

Note that due to different types of media processing components, the way of accessing a component by a control system and retrieving the results of the transformation process may

vary. In this aspect, the usage of predefined component models proves to be easier, since such models already define the way clients access component services, pass a media object and other configuration parameters, and retrieve the results of the transformation of the media object. However, the usage of other components, such as binary executables deployed in an operation system environment requires additional functionality needed to access the services to enable the communication with the control system. This functionality may be provided by a component vendor as a separate part of the software that is deployed in the same run-time environment and is used as a mediator between the control system and the actual component performing media transformation.

```
<?xml version="1.0" encoding="UTF-8"?>
<?doctype vmd system "vmd.dtd"?>
<vmdesc>
  <source>
    <moid alias="bc_video" ext_ref="CNN_db/CNN_Videos/4711"/>
  </source>
  <virtual name="TranscriptedSpeech">
    <signature>
        <property name="Maintype" class="Typespec">Text</property>
        <property name="Subtype" class="Typespec">Plain</property>
        <property name="Encoding" class="Typespec">UTF-8</property>
    </signature>
    <transformation name="transcription">
      <operation semantics="transcript">
        <input alias="i1" ref="bc_video"/>
        <param name="language" value="EN"/>
      </operation>
    </transformation>
  </virtual>
  …
</vmdesc>
```

Figure 4: Sample Client Request using VirtualMedia Markup Language (VMML)

3 Semantic Model for Multimedia Metacomputing

The component-based metacomputing foundations described in the previous section form a necessary prerequisite for multimedia metacomputing. We need, however, also a semantic model clearly specifying

- an abstract collaboration model,
- the external representation of media objects, operations, and client requests, and
- how client requests are pre-processed (transformed) to become executable by the metacomputing environment.

This model supplies the user or application programmer with everything needed to create both ad-hoc requests and metaprograms (e. g., request templates). The model also precisely describes how these requests get transformed into plans executable within a given metacomputing environment. The advantages of letting users make requests instead of directly creating plans are:

Ease of use: Requests are much simpler to create, because one does not have to deal with finding components implementing a certain operation, manipulating the media objects in order to fit them to the selected operation's signature, and so on.

Stableness: Requests are stable while plans are not. The reason is the inherent unstableness of a Web-based metacomputing environment, in which resources may become unavailable, replaced, or updated from time to time. A plan fails, if one of the required resources is not

available or compatible anymore, whereas a request would result in an alternative plan (if one exists).

Optimization: A request may be transformed into different plans depending on volatile conditions. We may, for example, consider time constraints, cost limits, utilization of resources, and exploitation of redundancy.

Due to space limitations, we only sketch the major aspects of the model in the following sections.

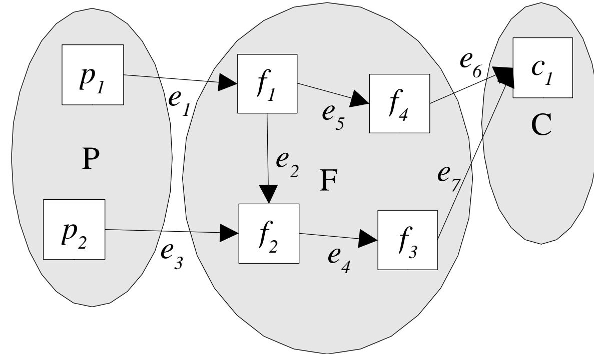


Figure 5: Illustration of the Collaboration Model

3.1 Abstract Collaboration Model

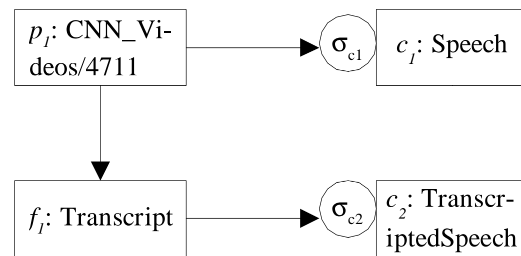
Our collaboration model is based on filter graphs (cf. Figure 5) similar to those introduced in [2]. The start nodes of the graph are media producers p_i (media objects managed by some server, maybe even live media sources) and the end nodes are media consumers c_i (e. g., client applications). The intermediate nodes are media filters f_i, the basic operations of a media computation, while the edges of the graph represent media streams flowing from one filter (or media producer) to another filter (or media consumer).

The graph in Figure 5 can be interpreted as follows. There is a producer p_1 creating a media object sent to filter f_1. Filter f_1 generates from its input two media objects which are sent to filters f_2 and f_4, respectively, and so on. Thus, the graph nodes are not bound to any real instances of media servers or filters, therefore we call this an abstract collaboration model.

3.2 Abstract Media Semantics

Turning our collaboration model into a multimedia metacomputing model requires the addition of some more specific media semantics to the graphs. This is done by means of signatures. A signature is a set of properties belonging to any of the following property categories: type, quality, content, functional, and non-functional specifications.

Figure 6 shows an example of a filter graph with signatures, in which the path $p_1 \rightarrow f_1 \rightarrow c_2$ corresponds to the sample request presented previously in Figure 4. The rectangular nodes contain node signatures. An edge has two signatures, one for each end. Edge signatures may be empty. In Figure 6, non-empty edge signatures are drawn as a circle between edge and node.



σ_{c1}:
[Typespec]
Maintype=Audio
Subtype=Waveform
Encoding=WAV
[Quality]
Sampling_Frequency=44100
Sample_Depth=16

σ_{c2}:
[Typespec]
Maintype=Text
Subtype=Plain
Encoding=UTF-8

Figure 6: Sample Request Graph with Signatures

A graph like the one in Figure 6 describes the computation of some media objects on a very abstract level, yet reflecting the full semantics from the client perspective.

3.3 Request Processing

The first thing to note is that the addition of edge signatures may turn the edges into a sort of "magic channels". Consider, for example, the edge between p_1 and c_1 in Figure 6. The signature σ_{c1} contains some type properties of the media object emitted by this channel, but what goes into this channel fully depends on the internal representation of the media object referenced by p_1. Thus, we can get edges with incompatible signatures at both ends. Apparently, there should be some kind of hidden computation within the channel, hence, the name "magic channel".

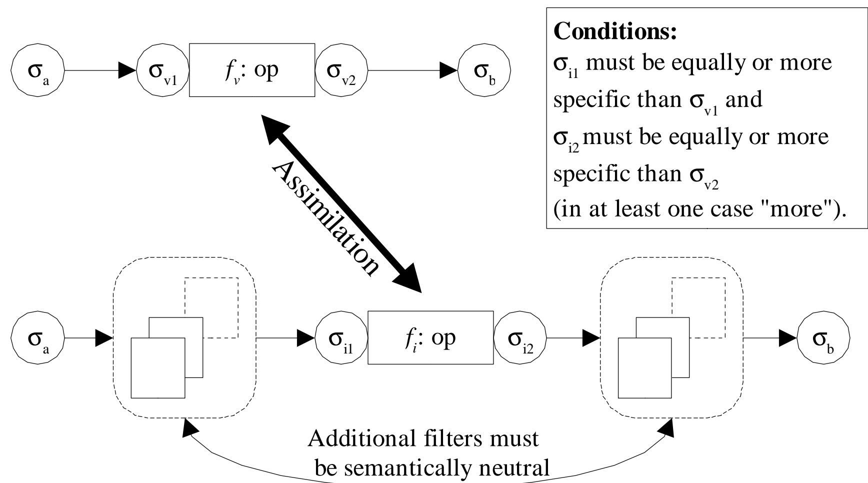


Figure 7: Equivalence of Abstract Operations (Filters) and (possible) Implementations defined as Semantic Assimilation

Consequently, one of the major purposes of request processing is revealing the semantics of "magic channels" in such a way that the request becomes executable by configuring and integrating appropriate components. In other words, the request graph has to be transformed into a semantically equivalent graph that contains only deployable components as nodes and edges with compatible media signatures at both ends. This automatic process has to be supported by formally specified semantic equivalence relations.

The three basic equivalence relations are *neutrality*, *reversibility*, and *permutability*, thus defining which media operations are considered semantically neutral, which operation is the inverse of another, and which groups of operations are semantically independent from each other, respectively. The semantics of media *composition* and *decomposition* is defined by generalized forms of these relations. The most important equivalence relation is called *semantic assimilation* (see Figure 7), which defines the relation between an abstract media operation (usually specified in a user request) and a component or configuration of components implementing that operation. More detailed explanations and a discussion of other important issues like, for instance, materialization management can be found in [5,6].

4 Conclusions

In this paper, we described the so-called multimedia metacomputing approach that aims at the formation of a large scale loosely coupled multiprocessing environment providing a distributed architecture to perform transformations on media objects. Basically, the following conclusions emerge from our previous discussion:

- operations that perform the transformations on media objects can be provided in the form of special media processing components,

- each media processing component should provide a signature to formally describe the run-time environment it requires during its deployment, types of media objects it accepts and the transformations it performs,
- it proves to be essential to exploit services of a repository as a distributed storage mechanism for processing components; in comparison to other solutions, a repository may provide additional functionality related to component versioning as well as combining component versions into valid configurations capable of cooperation in the process of media object transformation,
- an abstract semantic model has to be provided to ensure (semantically) correct request processing and robustness of client programs against changes of the metacomputing environment like, for example, exchange of components, processors, or media object materialization.

A global multimedia metacomputing environment would, in principle, allow to deliver global media data to any client and any kind of multimedia device without need to generate especially adapted materialization of the media data in advance. Moreover, computationally complex transformations and manipulations of the data are dynamically delegated to the most appropriate processing resources at run-time, thus optimizing response time and utilization of expensive special-purpose hardware. Ultimately, a “plugable” model for vendors of components providing media transformations and processing resource providers could form the (technical) foundation of a flexible business model for offering and vending multimedia services over the Internet. Albeit, our future work will be related to:

- further exploring the possibilities of using existing component technologies, such as JavaBeans in our approach,
- developing a mechanism used to dynamically evaluate the performance of processing components deployed in run-time environments while carrying out transformations on the media objects; using this mechanism, the results obtained are stored in a special history database and later used by the control system in order to achieve improved response times in the instances of subsequent media transformations.

References

1. Brown, A. W.: Large Scale Component Based Development, Prentice Hall, 2000.
2. Candan, K. S., Subrahmanian, V. S., Venkat Rangan, P.: Towards a Theory of Collaborative Multimedia. In: Proc. IEEE International Conference on Multimedia Computing and Systems (Hiroshima, Japan, June 96), 1996, pp. 279–282.
3. Hawick, K. A., James, H. A., Silis, A. J., et al.: DISCWorld: An Environment for Service-Based Metacomputing. In: Future Generation Computer Systems, 15 (5–6), 1999, pp. 623–635.
4. Mahnke, W., Ritter, N., Steiert, H.-P.: Towards Generating Object-Relational Software Engineering Repositories. In: Proc. 8th GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", BTW ’99 (Freiburg, Germany, March 1–3), Buchmann, A. (ed.), Informatik aktuell, Springer-Verlag, March 1999, pp. 251-270.
5. Marder, U.: On Realizing Transformation Independence in Open, Distributed Multimedia Information Systems. In: Proc. 9th GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", BTW '2001 (Oldenburg, Germany, March 7–9), Heuer, A., Leymann, F., Priebe, D. (eds.), Springer-Verlag, Heidelberg, Berlin, March 2001, pp. 424–433.
6. Marder, U.: Transformation Independence in Multimedia Database Systems. SFB-Report 11/2000, SFB 501, University of Kaiserslautern, Nov. 2000, 24 pages.
7. OMG, Unified Modeling Language Specification, version 1.3, OMG Document ad/00-03-01, March 2000.
8. Roman, E.: Mastering Enterprise JavaBeans, John Wiley and Sons, 1999.
9. Smarr, L., Catlett, C. E.: Metacomputing. In: Comm. ACM, Vol. 35 No. 6, June 1992, pp. 44–52.