Multimedia Systems (2007) 12:403–421
DOI 10.1007/s00530-006-0050-0

REGULAR PAPER

User-author centered multimedia building blocks

André Santanchè · Claudia Bauzer Medeiros · Gilberto Zonta Pastorello Jr

Published online: 7 September 2006
© Springer-Verlag 2006

Abstract The advances of multimedia models and tools popularized the access and production of multimedia contents: in this new scenario, there is no longer a clear distinction between authors and end-users of a production. These user-authors often work in a collaborative way. As end-users, they collectively participate in interactive environments, consuming multimedia artifacts. In their authors' role, instead of starting from scratch, they often reuse others' productions, which can be decomposed, fusioned and transformed to meet their goals. Since the need for sharing and adapting productions is felt by many communities, there has been a proliferation of standards and mechanisms to exchange complex digital objects, for distinct application domains. However, these initiatives have created another level of complexity, since people have to define which share/reuse solution they want to adopt, and may even have to resort to programming tasks. They also lack effective strategies to combine these reused artifacts. This paper presents a solution to this demand, based on a user-author centered multimedia building block model—the digital content component (DCC). DCCs upgrade the notion of *digital objects* to *digital components*, as they homogenously wrap any kind of digital content (e.g., multimedia artifacts, software) inside a single component abstraction. The model is fully supported by a software infrastructure, which exploits the model's semantic power to automate low level technical activities, thereby freeing user-authors to concentrate on creative tasks. Model and infrastructure improve recent research initiatives to standardize the means of sharing and reuse domain specific digital contents. The paper's contributions are illustrated using examples implemented in a DCC-based authoring tool, in real life situations.

Keywords Content composition · Ontology-based annotation · Digital content component · Multimedia authoring · Reusable content · Reusable components

A. Santanchè (✉) · C. B. Medeiros · G. Z. Pastorello Jr
Institute of Computing, UNICAMP, CP 6176,
13084-971 Campinas, SP, Brazil
e-mail: santanch@ic.unicamp.br

C. B. Medeiros
e-mail: cmbm@ic.unicamp.br

G. Z. Pastorello Jr
e-mail: gilberto@ic.unicamp.br

1 Introduction

In the early days, the task of building multimedia applications was closely related to that of a software development process, being assigned to computing professionals in charge of writing code. This perspective propelled what we call "process-centric development". There was a clear distinction between the author (developer) and the end-user (consumer) of multimedia productions. This scenario progressively changed in many ways: (1) multimedia tools became *easier to use*, being accessible to nonprofessional developers; (2) the evolution of open standards combined with the Internet fostered the *sharing, reuse and adaptation of productions*; (3) in the multimedia context, as in other domains, software involves not only executable code, but also the digital content that this code handles, giving origin to what we call "content-centric development". The combination of these factors progressively shaped a new kind of user of multimedia applications, the user-author, illustrated in Fig. 1.

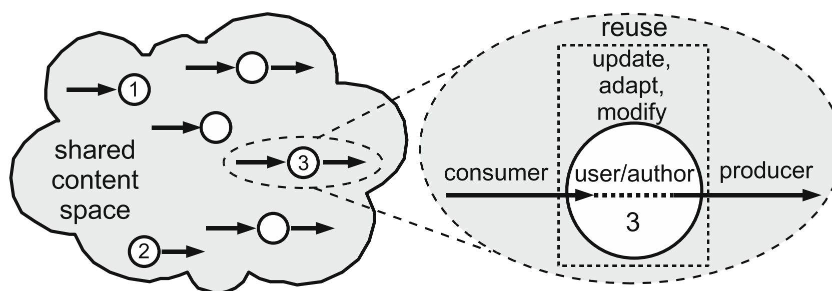

Fig. 1 Diagram illustrating our perspective of today's user

Under this perspective, these users can be seen as nodes of a shared content space, consuming multimedia artifacts (incoming arrows)and reshaping them through reuse (outgoing arrows). Node labelled (1) represents a user that is only a consumer; node (2) represents an author who works from scratch. User-authors alternate and combine their roles as creators and consumers, node (3). This collective usage scenario reflects today's reality, in which almost any user is also an author of some artifact (from simple text to complex multimedia presentations and software applications). As these artifacts travel among the nodes/users, they can be updated, adapted, modified, improved and shared again. This process of getting a content to update, adapt, modify and/or improve it, is the essence of the *reuse* concept. Being mainly noncomputer professionals, these users are propelled to become "reusers" in their authoring task, since it does not make sense to build an artifact from scratch when they have good material at hand and work under time and resources constraints. From now on, we will name this user "author", for short, adopting the term "user-author", whenever we want to emphasize these two interlaced roles.

We point out that, for us, multimedia authoring goes beyond creating productions using specialized authoring tools (e.g., Flash, Director or Toolbook). From an author's perspective, user-authoring means producing any digital content involving multimedia artifacts and taking advantage of tools available in a standard computational environment (e.g., text editors, presentation editors, spreadsheets, but also multimedia authoring tools). In this sense, the web can be seen as a virtual collaborative space for multimedia content production, where communities exchange digital artifacts. Moreover, we stress the need, in this context, to provide not only a model, but an infrastructure to implement the model and support its management and user-authoring. Present models and infrastructure are limited in aspects such as: (1) tradeoff between ease of use and reusability; (2) nature of content; (3) domain of application.

1.1 Tradeoff between ease of use and reusability

There are two perspectives to analyze authors' reuse practices. In the first perspective, authors reuse multimedia artifacts "as is", in the sense that they just take the artifact and insert it in a production, without modifications. In this case, authors can be portrayed as composers of multimedia artifacts, assembled from many sources. This kind of sharing and reuse is well supported by multimedia technologies when the shared/reused artifacts are basic multimedia files, e.g., an image, a video. In the second perspective, authors can decompose and adapt the reused production and fuse reused parts into a new production. This perspective involves the need for *complex digital objects* [4] to be shared. These objects can comprise many multimedia items and the relationships among them.

Both desirable factors in fostering user-author practices, "ease of use" and "flexibility in sharing/reuse", do not coexist harmoniously. Solutions that support ease of use (e.g., a family of interrelated tools) are limited to the formats supported by the tools themselves. On the other hand, solutions that are geared toward reuse are difficult to use. For instance, MPEG-21 [9], an initiative in the multimedia domain that is not constrained to specific tools or application types, and thus conducive to reuse, is difficult to use in authoring activities.

Our work overcomes this tradeoff between "*ease of use*" and "*freedom to share, adapt and reuse*". It presents a solution that combines author-friendliness with a model and infrastructure for sharing and reusing, which is not constrained to specific tools or types of products.

1.2 Content nature and application domain barriers

Content nature (executable software vs. content in general) and solutions driven by the application domain present limitations to user-authoring: (1) process-centric models are mainly focused in professional software

developers and program code, and do not contemplate other kinds of process descriptions accessible to non-professionals, e.g., workflows, spreadsheets; (2) content-centric models are designed to be used in specific application domains; (3) the process-centric × content-centric division is a barrier when the author wants to mix executable software and content, or when the artifact to be shared/reused cannot be classified inside one of these categories (e.g., a spreadsheet sometimes contains executable routines, sometimes not).

As will be seen, our approach introduces an upgrade from a "digital object" to a "digital component". These components, called DCCs, are generic "building blocks" that can be used by authors in their compositions, regardless of the nature of their content, and are not constrained to a specific family of tools or kinds of applications. DCCs are self-descriptive units, semantically annotated using taxonomic ontologies. An important strategy of our infrastructure is based on the notion of content-type driven execution, in which a given artifact "searches for" appropriate software to execute it, thus helping the consumer and production roles.

The main contributions of this paper are thus: (1) presentation of DCCs as a user-author centered building block in the multimedia context; (2) analysis of the notion of content-type driven execution under different guises and (3) presentation of a content-type driven execution strategy tailored to the context of user-author multimedia development.

These contributions arise from our analysis of requirements needed for full-fledged user-authoring. This analysis is itself a contribution, establishing guidelines against which other proposals can be evaluated. The DCC model is confronted favorably with complex digital object approaches, and mainly with the MPEG-21 standard that addresses the multimedia domain. The paper presents practical examples implemented in a DCC-based authoring tool, which illustrate the benefits of our solution. These examples come from the experience of the first author in developing an authoring environment that is being used in elementary and high schools in the city of Salvador, Brazil.

Our presentation first lays the foundations of the DCC model and infrastructure, being followed by the materialization of this model in the tasks of reusing and authoring. Section 2 presents an overview of related work. Section 3 presents the requirements that led to our user-author multimedia building block. Section 4 briefly presents our DCC model. Section 5 introduces the notions of content-type driven execution and of the companion DCC as central foundations to relate content resources with content handling software, and shows how these notions improve multimedia authoring. Sections 6 and 7 discuss retrieval mechanisms and some implementation issues. Sectin 8 summarizes how the DCC model and infrastructure meet the requirements of Sect. 3. Finally, Sect. 9 presents conclusions and ongoing efforts.

2 Related work

This section presents the research issues related with the main contributions of this paper. First, since DCCs define a model and infrastructure suitable for process-centric and content-centric development, the first three subsections analyse the philosophy behind these two currents, and models adopted by them for sharing/reusing artifacts. Section 2.4 compiles and classifies a set of strategies used for content-type driven execution.

2.1 Process-centric × content-centric development

In order to support authors who want to design and develop applications, a key question must be answered: what is the raw material employed by these authors in their work? The approach used to build the application will be defined by the raw material: content (content-centric development) or executable software (process-centric development).

In a typical content-centric development project, authors start from content resources, and transform, customize and combine them to form a resulting material, which can vary from a simple presentation to a sophisticated multimedia production. The backbone of content-centric development is thus formed by interrelating content artifacts. These artifacts in turn drive demands for software, e.g., a video file can require a video player routine to enable it being shown. Additional software routines can be inserted inside the content structure, like a Javascript routine inside a web document, with the content playing the central role. We can imagine such an application organized in layers, as illustrated in Fig. 2a, where the user-author roles of authoring and consuming are distinguished. The content layer comprises the main content artifacts used in the application; the software layer comprises all software routines requested to manipulate this content and the result layer corresponds to the in-memory composition of the content and software layers necessary to execute the application. As shown in the figure, in the content-centric approach the author deals with the content layer and the user interacts with the result.

In the process-centric approach, on the other hand, the process description plays the central role, being used to design or implement an application. Programming

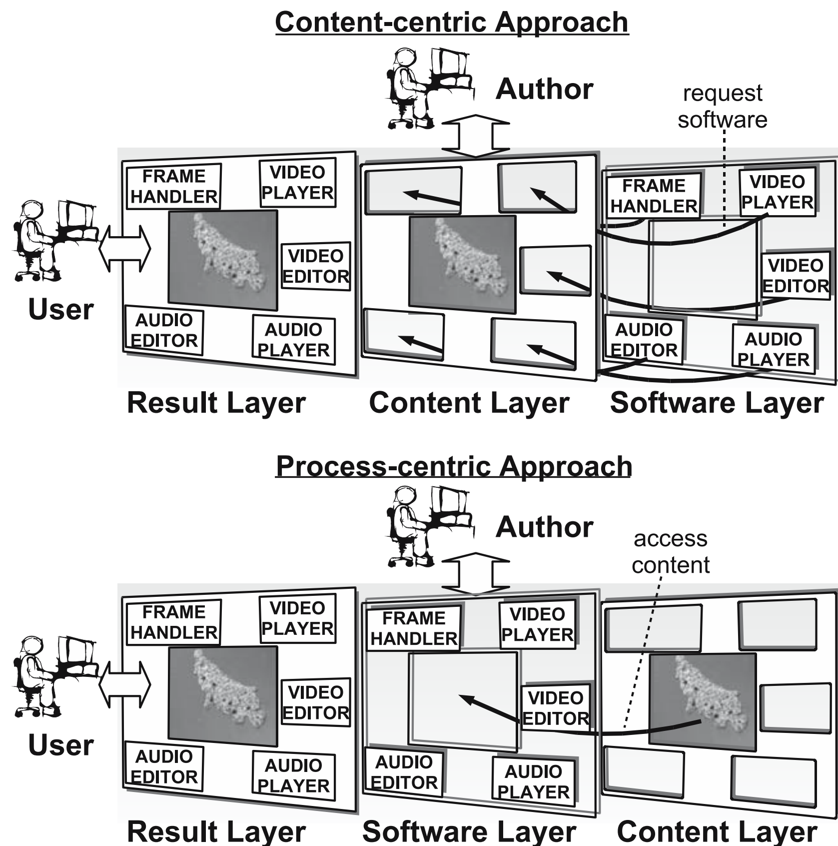

Fig. 2 Diagrams illustrating (**a**) content-centric and (**b**) process-centric approaches

languages are the usual ways to implement process-centric applications. There are, however, higher level approaches more suitable to the nonexpert authors, e.g., composition of software components and workflow specification. As illustrated in Fig. 2b, in the process-centric approach the authoring role deals with the software layer.

Since in the content-centric approach the raw material is the content, the strategies to produce, share and reuse content are based on content files or packages. The mechanisms to combine content pieces are either constrained to the formats supported by a tool or family of tools, or limited to a specific kind of product, as explained before. In a process-centric implementation, on the other hand, the raw material is the executable software, and the strategies to produce, share and reuse content emphasize executable units (software components, libraries, frameworks, software templates, etc.), which are prepared to be adapted and to work together with other software units.

The research and models aimed at sharing, reusing and composing productions are highly influenced by these two currents. On the one side, there are process-centric initiatives in the software engineering domain, where one of the main focus is on software components to encapsulate program code [13]. On the other side, there are many content-centric initiatives to systematize the packing, deployment, reuse and composition of domain specific content in areas such as education [1,15,33], digital libraries [10,35], multimedia [16] and software development related artifacts [25].

2.2 Software components (process-centric approach)

Software components have been the main unit adopted for program code reuse. There are many definitions for software component [14]. Even if they do not achieve total agreement, some characteristics are present in all definitions, or can be inferred from them: (1) a component is an entity meant to be composed; (2) each component publishes its functionality through a well-defined and open interface; (3) components can be nested into other components.

From a practical point-of-view, software components have additional characteristics, observed in the widespread component initiatives: (1) components contain some kind of binary code that implements the functionality declared in their interface and (2) the component interface and implementation are assembled into a standard package for deployment purposes.

The separation between interface and implementation resulted in a generic mechanism to explicitly express

how a component can be connected to other components, independent of its implementation details. Software components hide their heterogeneity inside a homogeneous capsule. For these reasons, the software component model has been adopted as a basis for the DCC model.

2.3 Complex digital objects (content-centric approach)

In multimedia authoring, the ability of sharing and reusing multimedia artifacts is essential. In the last years, many domain specific initiatives have been concerned with sharing and reusing digital content. Since each research domain uses its own terminology to refer to the sharable and reusable entities, we will use a term borrowed from digital libraries: *complex digital object* (or simply digital object) [4], whose concept and model can be considered a common foundation.

In the multimedia context, there are many standards for different kinds of media and their applications. Many of these standards overlap each other and produce competitive solutions to the same needs. The purpose of the MPEG-21 initiative [9] is to bring these standards together. MPEG-21 [16] is a framework that offers support for multimedia delivery and consumption, simplifying transactions and ensuring content interoperability. It defines many content/consumption related issues, like unique identification, rights and permissions through a specific language (Rights Expression Language) [37], etc. A fundamental piece of this framework is the *Digital Item*, which is a basic unit of reusable content representation, and is declared in the digital item declaration (DID) [8].

The engine responsible for digital item processing (DIP engine [9]) can be considered as a software framework that is extensible with software plug-ins. MPEG-21 methods (DIMs) can be attached to digital item descriptions and then can be related to digital content items, and can be shared inside complex digital objects.

The open archival information system (OAIS) is a reference model, whose purpose is to address preservation of complex digital objects over the long term, admitting impacts of changing technologies and user community [10]. As time goes by, appropriate tools to interpret, process and present a specific kind of content may not be available in the future. To deal with this problem, OAIS defines that each piece of content must be associated with a *representation information*, whose purpose is to map the data into more meaningful concepts. One possible kind of representation information is the *access software*, which can access and interpret the content. Metadata encoding & transmission standard (METS) is a standard related to OAIS that specifies an XML document format to represent metadata that is necessary for complex digital object management and exchange [35]. METS specifies a behavior support associated with complex digital objects. These declarations relate items of content with a web services API provided by Fedora [34], a general purpose repository service, which supports complex digital objects. It defines a special *disseminator* object that can process other objects, through web services requests. This model is well defined for objects inside the repository.

There are many initiatives working around the concept of Learning Objects [15], which can be conceived as an educational complex digital object. These educational initiatives have agreed over an architecture to enable the relationship between the educational content and the runtime environment (RTE), which is the software system where this content will be used. This relationship is useful when the RTE wants to track the interaction of a student with an educational object, e.g., what parts of an HTML tutorial a student visited, or the number of test questions the student answered correctly. There is an agreement over a proposal from the Aviation Industry CBT Committee (AICC) [19], which is based on the assumption that any educational content will be web-based. AICC defined an API that is responsible for specifying what services can be requested from the RTE and what information can be delivered to it.

2.4 Content-type driven execution

Many multimedia systems need to dynamically invoke software routines according to the type of the content to be handled, e.g., a web browser displaying a document with text, images and animations; a software to present slides that runs a video inside a slide, containing text and graphics. This section analyses a set of strategies adopted by this kind of system to dynamically associate content with blocks of software specialized in dealing with that content. Established strategies include software frameworks, active document components and software plug-ins.

2.4.1 Software frameworks

The more similar two content types are more closely related to their potential functionalities. Systems can exploit this aspect defining a set of software routines to be shared by content types based on their similarity. For example, many image file formats can have specific routines to decode their content, and share a library of routines that implement all other image related functionality. This has two benefits: the same code is applied to many similar content types, and the system deals

with the decoded images in a homogeneous way. These characteristics can be dealt with using software frameworks. For instance, object-oriented software frameworks usually define a generic class containing shared routines, and subclasses to implement the singularities of each content type. A framework example is Mozilla NGLayout [26], responsible for rendering web documents inside Mozilla web products, like browsers and e-mail clients. If an author wants to reuse a software framework to build a new system, this process will involve adapting program code. For instance, consider an author who wants to build a web page editing tool, and chose the Mozilla NGLayout framework to render the pages, then the author must adapt his/her code to properly embed the framework.

2.4.2 Software plug-ins

Software plug-in architectures enable to pack and deploy a set of software routines, related to content types, and to use them to dynamically extend systems to deal with new kinds of content. Some systems, like web browsers, have mechanisms to automatically identify a required plug-in for a new content type, and to find, load and execute it to deal with the content. Software plug-ins are more flexible reuse-wise than software frameworks. Instead of being deployed along with the host system, they can be fetched on demand. Therefore, new plug-ins can be developed to deal with new content types, without the need of modification of the host system code. However, plug-ins are usually designed geared to a specific system, e.g., Mozilla plug-ins, Eclipse plug-ins [5], Protégé plug-ins [24]. Like in software frameworks, it is necessary to adapt programming code to port (reuse) these plug-ins to a new system.

2.4.3 Active document components

A set of routines that deal with content types can be encapsulated inside a software component. Active document architectures, like Microsoft OLE [6] and Apple OpenDoc [3], allow systems to embed pieces whose types they do not support directly. Whenever the system needs to deal with these content pieces, it forwards the operation to the appropriate software component. Active document components and plug-ins operate in a very similar way. However, in the former the same component can be usually shared by many distinct applications, whereas in the latter a plug-in is designed for a specific application. On the other hand, active document components are highly dependent on a specific operating system.

The mechanisms to identify the content type in these three strategies are usually: (1) file extension, a poor and ambiguous mechanism (many formats have the same extension); (2) file header, not a standardized mechanism, since each content type has a distinct format to define its header; (3) MIME media type (RFC2046). As will be seen, we solve these issues through the notion of content-type driven execution, in which a given kind of media "finds" the appropriate software to run it. Media and software are encapsulated into our components; the discovery and combination mechanism is based on specific component interface matching characteristics.

3 Requirements for user-author multimedia building blocks

This section defines a set of requirements we consider necessary to create building blocks for user-author multimedia. The author adopts these blocks to produce, adapt, share and reuse any kind of digital content, regardless of its nature (executable software or not). At the same time, the conception enables exploring the specific functionality provided by each kind of digital content.

3.1 Breaking barriers between content- and process-centric development

As presented in Sect. 2.1, strategies to develop computer-based applications are highly influenced by the raw material employed in the work: content (content-centric development) or executable software (process-centric development). In particular, in the multimedia domain, both approaches can be adopted. Authors can follow a content-centric approach and produce multimedia presentations combining multimedia artifacts, which are presented following a time-line, or are organized over pages. On the other hand, they can follow a process-centric approach to build a multimedia application, using software routines. However, authoring models and mechanisms in process and content-centric currents follow parallel and distinct approaches to solve closely related problems. There is a lack of a unifying model to combine both.

The distinction between content and process-centric approaches is influenced by implementation concerns. In both cases the production is constrained to limits imposed by the layer where the authors work. Moreover, it is difficult to produce compositions that combine pieces of content and software, mixing both approaches. Furthermore, in the content-centric approach it is expected that software development experts will previously implement the software layer. Authoring

is expected to be limited to contents, since it is not envisaged that authors can contribute in writing and sharing software artifacts.

Here, our proposal is to reduce the distance between user and author. Thus, our first requirement is that a model must overcome the barriers between content- and process-centric approaches. The author should be able to combine pieces of content without needing to be concerned with their nature (software or content).

3.2 Providing a unified abstraction: potential × provided functionality

The content-centric approach works from "static" artifacts, in the sense that they can be seen as complex data (as opposed to software). Such artifacts may be constructed out of a variety of content pieces. The type associated with each such piece denotes which operations can be applied over it (e.g., a video content can be *played*; however, in order to be played it requires specific software). Similar to what is found in Internet media standards—e.g., MIME (RFC2046)—we use the term *content type* to denote the content, its internal representation and associated operations. We call the set of operations associated with a content type to be its "*potential functionality*", in the sense that their implementation is intrinsically not part of the content (e.g., the video player software is not part of the video).

In a process-centric approach, instead, we deal with executable instructions, and thus have a "*provided functionality*" inherent to any process description module (e.g., a software component, a workflow specification). In particular, a software component explicitly declares its provided functionality by means of its public facet, its interface. The software component model supports the distinction between public and private portions. Through this distinction, it is possible to control which aspects of a component are published (accessible to users). Interface specification can be seen as an abstraction of a component's functionality, and can be used for component discovery, selection and composition. Such an abstraction has a tight relationship with reusability [17].

In the content-centric approach, instead of an interface, there appears the notion of metadata as an abstraction of the content. Interfaces describe what a software "can do", whereas metadata describe what a content "is". There is no standard mechanism, however, to specify applicable operations, which, depending on the case, must be deduced from metadata.

Thus, a second requirement for user-authoring is to define a unified abstraction comprising process and content encapsulation, which is used in their reuse, discovery, selection and composition. Our solution to this unified abstraction is a combination of metadata and interface specification.

3.3 Exposing a homogeneous interface

The notion of composition appears in both content-centric and process-centric approaches, e.g., a software can be created by interlinking components, or a complex multimedia data artifact can emerge from the composition of distinct data blocks. Composition complexity dramatically increases with the amount of different blocks created, and the number of possible combinations grows dramatically. The composition procedure can be simplified if each piece that participates in it is encapsulated behind a homogeneous interface. In point II, the interface is used for abstraction, whereas here homogeneity fosters ease in composition.

The process-centric approach of software components takes advantage of this interface paradigm, which allows distinct software building tools to deal with the same set of components. A widespread example is the JavaBeans technology, where beans are homogeneously treated by building tools.

A third requirement is, therefore, using homogeneous interfaces to access content and software. In the content-centric approach, however, there is no such consensus. Some tools define their own proprietary format. Initiatives related with content reuse standardization propose domain specific pre-defined interfaces, which restricts their applicability in other areas.

3.4 Supporting content-type driven execution

The main content-centric reuse initiatives stress the importance of selecting appropriate program code, related to the reused content. In the multimedia context, MPEG-21 points out the need for specifying not only a standard for media exchange, but also a complete framework, including the software dimension [16]. Educational initiatives stress the necessity of defining standards in which reusable educational content pieces will dynamically interact with educational tools through an API. In the digital libraries context, the Open Archival Information System (OAIS) defines how to maintain software tools capable of interpreting specific content formats, which will be preserved in the long term [10].

The standardization efforts discussed partially deal with this issue, as seen in Sect. 2.3. The MPEG-21 methods (DIMs) are expressed as scripts. However, this approach to build software routines imposes some constraints on the user-author. First, since the methods have to use specific MPEG-21 libraries (DIBO), their func-

tionality is restricted. Second, MPEG-21 DIMs work as auxiliary routines, and have no structure appropriate for sharing and reusing among authors. The Fedora [34] repository service offers a means of attaching executable functionality to content. However, it lacks a strategy for sharing and reusing complex digital objects, and their related disseminators. The API defined by AICC within the learning objects [15] initiative also addresses this execution aspect. Unfortunately, the AICC standard is highly specialized in specific tasks envisaged in web activities for education.

Our fourth requirement is that there must be a mechanism that supports the selection of an appropriate software routine that handles a content according to its type.

4 A very brief overview of DCC

A *Digital content component* [29] is a unit of process and/or content reuse. From a high level point of view, it can be seen as content (data or software) encapsulated into a semantic description structure. It is comprised of four distinct sections: (1) the content itself (data or code), in its original format or a DCC composition; (2) the declaration, in XML, of an organization structure that defines how components within a DCC relate to each other; (3) a specification of DCC interfaces, using adapted versions of WSDL [11] and OWL-S [18]; (4) metadata to describe functionality, applicability, use restrictions, etc., using OWL [32].

Digital content component are assumed to be stored in repositories available on the web. Interface and metadata sections, respectively (3) and (4), are used to help retrieve the appropriate DCCs from the repositories and reuse them [30]. Furthermore, there is a DCC infrastructure that comprises an architecture to assemble DCCs into a desired product. A *DCC composition* is considered to be any digital artifact built by combining DCCs, and can vary from a multimedia document to a software application.

The DCC model was inspired by software engineering's software component paradigm. However, unlike software components, DCCs do not need to encapsulate binary program code to be useful as part of applications. It is possible to encapsulate inside a DCC only a multimedia artifact, other kinds of software (such as workflows), or both, and use it directly to compose an application. DCCs are thus more accessible to authors who are not experts in software development; they are based on an approach where the end-user is the author, and the components are the "raw material" [22,27].

We differentiate between two kinds of DCC, process and passive DCCs. The former encapsulates executable instructions, the later encapsulates any other kind of content. In order to handle operations accepted by a passive DCC, suitable software is needed. In our model, this role is performed by the so-called *companion DCC*, see Sect. 5.1. Going back to the video example, a video `V` can be encapsulated into a passive DCC-`V`, and video playing software `VP` into a process DCC-`VP`. If `VP` can play `V`, then DCC-`VP` is a companion to DCC-`V`. Other characteristics of DCC will be introduced via examples in subsequent sections. For internal details, not relevant to the paper, on DCCs, see [29].

5 Content-type driven authoring

A passive DCC is a component that encapsulates content. Its interface declares the operations that can be performed on this content. However, being passive, it does not have executable software to implement the operations explicitly declared on its interface. This raises questions that we will answer in this section. First, since a passive DCC declares operations and does not implement them, how are these declared operations associated with their respective code? The answer to this question is based on the notion of content-type driven execution and on the companion DCC strategy, treated in Sect. 5.1. Second, how can content-type driven execution be explored to create a meaningful author-friendly authoring environment? This is treated in Sect. 5.2. In our implementation, the subsystem responsible for supporting authoring and execution of any composition involving DCCs is called *execution engine*, described in Sect. 7.

5.1 DCC content-type driven execution

As discussed in Sect. 2.4, systems capable of handling more than one content type, e.g., web browsers and text processors, have mechanisms to delegate each content type to its respective content handler. We recall that content types implicitly or explicitly determine a content's potential functionality. This section analyses the DCC mechanism that makes some of the potential functionality operations effective.

In the content-centric approach, the content type is used to define the software blocks appropriate to deal with it. We thus now propose the notion of *content-type driven execution*, in contrast to the *process driven execution* of the process-centric approach. A well known example of this kind of execution is a web page, which can be taken as a combination of content pieces. In this case, for each kind of content piece (HTML document, image, Flash animation, MPEG video), the web browser

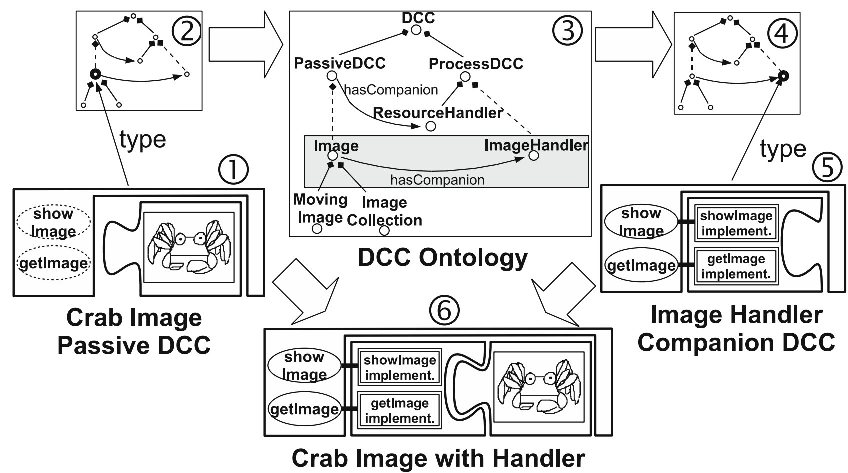

Fig. 3 DCC content-type driven execution diagram

invokes an internal specialized routine or a software plug-in to deal with it. The content type "drives" the execution.

A passive DCC is content-centric, and its interface defines how this content can be accessed. Since the program code for the operations declared in the interface is not embedded in a passive DCC, interface operations are implemented in a special kind of process DCC named *companion DCC*. The companion DCC lends its operations to a passive DCC in a way that is transparent to composition authors. The choice of the appropriate companion for a passive DCC is context sensitive, and is determined by the execution engine, when this passive DCC is used. This allows a homogeneous treatment of passive and process DCCs from the author's perspective. Moreover, the focus in the content is the best option for content-centric composition.

Figure 3 shows an example of content-type driven execution. In the figure, a passive DCC (a crab image file) is associated with a companion DCC (software that can display the image) based on its content type. *Taxonomic ontologies* play a central role in this matching process. We use the term taxonomic ontology, as defined by [12], to express a particular kind of ontology, whose purpose is to provide a referential vocabulary. Its structure organizes terms into generalization/specialization hierarchies, and semantic links to express synonymy, composition, and so on.

The taxonomic ontology, illustrated in the center of the figure, organizes and relates types of DCC, represented in the diagram by white filled circles. Lines with a diamond in one extremity represent subsumption relationships, e.g. `Passive DCC` subsumes `image`. Dashed lines indicate that some intermediate nodes were ommitted for simplicity. Each arrow represents that a property `has companion` that relates two nodes, which means that a passive DCC type is processed by the indicated companion DCC type.

As shown in the figure, any DCC has a DCC type, defined in the ontology. Type specification is carried out through an explicit reference in a DCC's metadata section, coded in OWL. Each DCC type represents a kind of process (`process DCC`) or content (`passive DCC`), and defines a minimal set of provided operations (process DCC) or potential operations (passive DCC) in its interface. These operations define the type's *minimal interface*, i.e., for any DCC to be considered as of that type, it must offer at least the operations of the type's minimal interface. If a DCC `A` subsumes a DCC `B`, then the minimal interface of `B` extends, or is equal to, the minimal interface of `A`. If a DCC `A` defines the property `has Companion` pointing to `C`, i.e., (`A` has companion `C`), then the minimal interface of `C` extends, or is equal to, the minimal interface of `A`. This guarantees that a companion DCC implements at least the operations declared in the minimal interface of any related passive DCC.

Figure 3 shows the cycle that associates a companion DCC to a passive DCC, following the numbers ① to ⑥. Consider the passive DCC that contains an image ①, and defines its interface operations of its potential functionality. A subset of these operations (`showImage` and `getImage`) is displayed in the figure. This passive DCC is related to the `Image` DCC type in the ontology ②, whose companion is the `ImageHandler` DCC ③. The execution engine asks the DCC repository manager for a DCC of this type, see Sect. 6. The selected DCC is loaded ⑤ and connected to the passive DCC, which it will process ⑥. Notice that the companion DCC declares a provided interface, which defines the

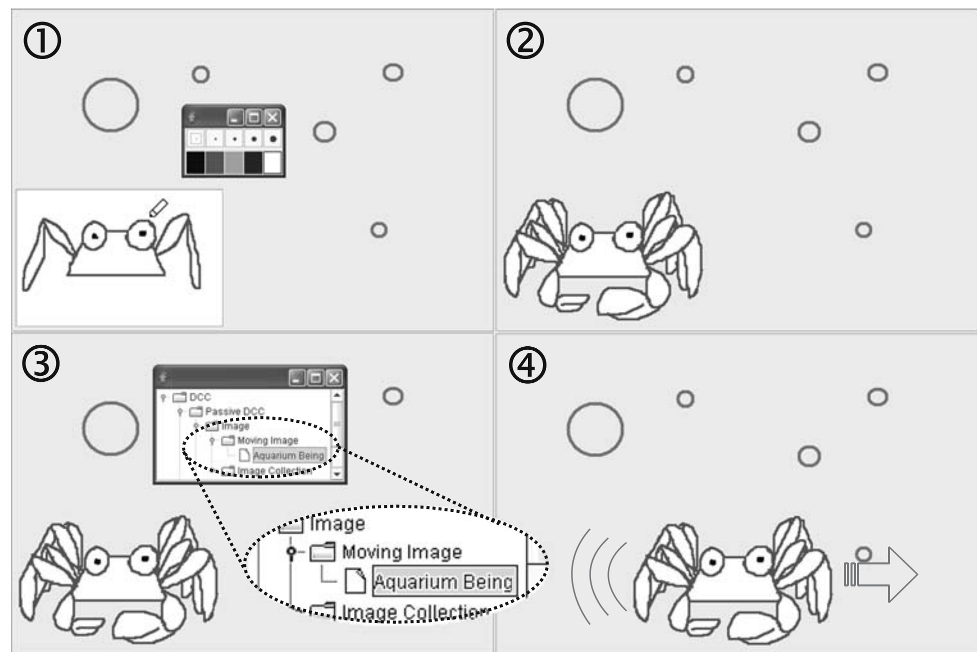

Fig. 4 Steps followed in Magic House authoring system to produce a crab DCC

same operations of the passive DCC, and implements them.

More than one companion DCC can be related to the same DCC type and be used for distinct contexts. In the example, we can have, for instance, three `ImageHandler` DCCs: (1) implemented in Java to run in a stand-alone application, (2) implemented in Java to run in a web browser (applet), and (3) implemented in C to run in a stand-alone application. Each can have context properties, in the metadata section, whose values are defined in specific taxonomic ontologies.

5.2 Authoring DCC multimedia artifacts

We now show how an author produces a multimedia artifact using DCCs. The presentation will use an example implemented in the Magic House environment, which is an educational authoring tool. It has been used in schools in the city of Salvador, Brazil in authoring multimedia learning material (e.g., animations employed in exploring laws of physics). The present version of our system is the result of user feedback, after over 5 years of use of Magic House. The previous Magic House environment was a process-centric development system, where teachers and/or students combined software components to build applications. We remarked that authoring was strongly influenced by reuse and exchange of such components. End-users needed help to find appropriate pieces to compose and to determine which software was needed to handle a given kind of media. This experience led to the notions of DCCs and companions.

Magic House is based on a combination of the graph-based authoring paradigm of [7] and the software components' visual editor modeling approach of tools such as Bean Builder. The main goal of this example is to discuss the content-type driven mechanism working behind the scenes, and to show how this mechanism explores the semantics associated to DCCs, to provide an author-friendly authoring environment. For simplicity, this example is based on a single DCC, but the usual Magic House's production contains many interconnected DCCs.

Figure 4 shows four Magic House screenshots that capture successive steps in a DCC production process; the result is an animated crab, which moves inside an aquarium. In the first step, the author requested the system to edit an image DCC retrieved from its DCC repository. Since a DCC of type image is passive and does not implement software to handle its content, the environment retrieves the respective companion DCC, based on type matching from the DCC taxonomic ontology, and context values. The latter are configurable parameters that specify work conditions, e.g., language. Following the cycle described in Sect. 5.1, the system finds a companion DCC appropriate to handling an image DCC. The result is shown in step `1` of Fig. 4. The companion DCC opens an image editing window inside the Magic House environment, where the author can edit the image which is inside the DCC.

Once editing is finished, the author indicates that this passive DCC is ready for the moment, and switches the Magic House environment from the editing mode to the execution mode. Here, the (edit-enabling) companion

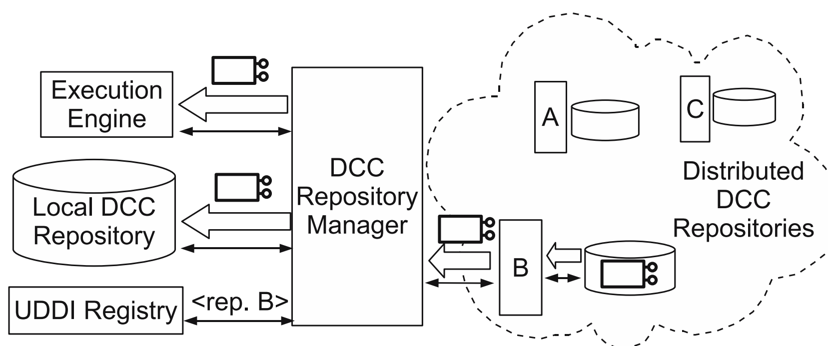

Fig. 5 Finding and retrieving a DCC based on type matching

DCC associated with the crab image DCC is replaced by another companion DCC, which also handles image DCCs, and whose context values define it as executable instead of editable. This new DCC is designed to be used in the execution mode, and only displays the crab image, as shown in the second step of the figure.

Suppose now the author wants to move the crab through the aquarium, rather than have a static image, he/she knows that there is a passive DCC type named `Aquarium Being` that is capable of moving through the aquarium. The `Aquarium Being` DCC encapsulates only the being's image; its companion DCC implements the program code to move the image. So, in step labeled `3` in Fig. 4, the author requests to the Magic House environment to redefine the type of crab image DCC. The system displays a window with the DCC ontology, from which the author can choose a new DCC type. The author selects the `Aquarium Being` DCC type. Once the change is accepted, when the author runs the application, the crab image moves through the aquarium, as illustrated in the fourth step of the figure.

This example shows how the content-type driven mechanisms provide an author-friendly environment. In steps `1` and `2`, different roles (author and user) are supported by switching context and thus the companions. Step `3` allows type changing and, as a consequence, new kinds of compositions. Authors are concerned with the content and the semantics they attribute to the content and, behind the scenes, the DCC infrastructure transforms the semantic indicators in executable behaviors.

6 DCC retrieval mechanism

We recall from the Introduction that user-author centered multimedia authoring, in our context, means: (1) ease of use in sharing, interact with and running a multimedia artifact and (2) the ability to find, reuse and combine pieces of process and passive DCCs in a given authoring step. This section shows how our DCC retrieval mechanism works, based on the notions of interfaces, metadata and domain ontologies. A DCC search process is roughly composed of two steps. First, the user specifies the requirements of a desired DCC, and the infrastructure returns available DCC types that meet these requirements. Next, the user chooses the desired type, and the infrastructure will return a DCC that matches the type. For details on the first step, see [30].

6.1 Finding/retrieving a DCC

Figure 5 illustrates the sequence of actions followed to find and retrieve a DCC given its DCC type. It shows the basic local infrastructure (a local DCC repository, execution engine and repository manager), which is replicated at each site where DCC authors exist (A, B, C). Thin arrows represent data exchange related to the DCC finding process, and thick arrows represent DCC retrieval, once found.

The process is started whenever a DCC is needed, in execution or authoring activities, either directly requested or as a companion request. All find/retrieve processes begin by a local search and proceed to a web-wide search if no local DCC satisfies the initial request. First, the `Execution Engine` requests a DCC from the `DCC Repository Manager`, informing its type. The manager searches in the `Local Repository` for DCCs of the given type.

If no local DCC satisfies the request, the repository manager queries a `UDDI registry` for the DCC type. Universal description, discovery and integration (UDDI) [36] is a standard for a web-based registry service, whose primary goal is to describe and discover web services [2]. UDDI supports the description of entities other than web services. Additionally, more recent versions of UDDI can accommodate and use identification taxonomies provided by third parties [36]. The DCC repository manager uses UDDI registry services to specify which repositories have a given DCC type, via the uniform resource identifier(URI) of each type. It is important to note that the DCC type ontology works as a

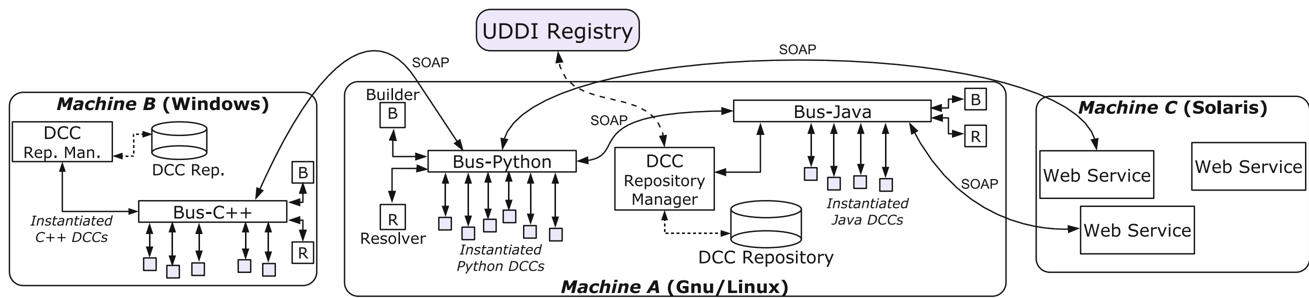

Fig. 6 Example of an arrangement with three machines adopting the Anima architecture

UDDI identification taxonomy, and can be used in DCC discovery.

The local repository manager uses information from the UDDI registry to get the Internet address of external repositories, which contain DCCs of the required type, and requests information about these DCCs from these repositories. This information is given to the execution engine, which will dynamically decide which is the most appropriate DCC for a given composition (e.g., see Sect. 5.1). Once the engine selects the appropriate component configuration, it asks the repository manager to provide it (either locally or remotely). When a DCC is retrieved from external repositories, the local repository manager stores a local copy of it to optimize subsequent retrieval requests, e.g., in the figure, a DCC was retrieved from local B.

6.2 Exploiting the DCC type ontology

A specific companion DCC may not be available to an author (e.g., if there is no `ImageHandler` DCC for an `Image` passive DCC). However, it is possible that the author does not want to take advantage of the full functionality of a companion, but just a subset thereof. In this case, the author may use a companion of a DCC whose type subsumes the type of the original DCC. Returning to the ontology in Fig. 3, the `Image` DCC type is subsumed by the `Passive DCC` type, which has a companion of `ResourceHandler` type. The `ResourceHandler` companion implements an operation that accesses the binary content of a passive DCC. It treats any passive DCC as a flat binary resource, without considering any format particularity. Thus, in the absence of an `Image` handler, the author may be satisfied by using a ResourceHandler companion.

In other words, in the DCC model, an author can define, during composition, that a passive DCC of type `A` can be adopted in lieu of DCC type `B`, when `B` subsumes `A`. We point out two advantages of this mechanism. First, as in the `Image` example, it simplifies composition and execution if the author does not need the most specialized companion DCC to deal with a given content. Second, this feature increases the potential for composition reusability. It is important to notice that not only DCCs will be reused, but the compositions too, and that compositions can also become DCCs that are stored in repositories. Authors can thus reuse a composition, totally or partially, tailoring it to their needs.

7 Implementation aspects

This section presents the architecture designed to support our framework, which has evolved from previous projects named *Anima* and *Magic House* [31].

Anima is an infrastructure for managing and executing DCCs and their compositions. This infrastructure determines a communication model for DCCs specifying how they will interact within a composition. The execution of a composition may have a centralized coordinator or may be a result of a cooperation among independent DCCs. All communication among DCCs is performed through a software managed bus. Magic House is built over Anima.

In this section, we focus on the part of the Anima infrastructure responsible for the execution of authored products formed by connected DCCs, and which supports the authoring process. Even though the Anima infrastructure has many other attributions related to DCC management and authoring tasks, we stress execution aspects to clarify the main concepts treated here.

The architecture has been designed to support local and distributed component management and execution. It is independent of specific programming languages, supporting the interaction of DCCs implemented in different languages. Figure 6 shows a possible configuration of the architecture considering three different machines that run distinct operating systems, and a UDDI service that offers the publishing/discovering mechanism for DCCs and web services.

As pictured in Fig. 6, an environment to support DCC execution is a combination of hardware, operating system and programming language. For each environment,

there is a specialized *Bus* that is responsible for the communication: (1) among DCCs connected to the same Bus; (2) between a DCC in a Bus and a DCC in another one, through an inter-Bus communication; (3) between a DCC in the Bus and a web service.

The minimum infrastructure that must be available to support any execution task is defined by the Bus and three specialized DCCs (explained further): the *Builder*, the *Repository Manager* and the *Resolver*. The execution of a composition requires the existence of a primary DCC responsible for starting the process and invoking the execution of the other DCCs.

When any DCC is first invoked, it is retrieved from a DCC repository, then it is loaded to memory and then prepared to be executed. We call this procedure DCC *instantiation*. The process of DCC instantiation is delegated to a specialized DCC called *Builder*, which carries out all the above instantiation steps. It also defines and associates a *runtime URI* to each new instantiated DCC. The runtime URI is used to univocally identify an executing instance of a DCC, and is valid only during that execution of the DCC. Instances of the same DCC will receive distinct URIs.

When a DCC sends a message to another DCC, it does not know exactly the destination of the message. This source DCC sends the message through the Bus addressed to a runtime URI . A specialized DCC, called *Resolver*, intercepts the message and decides whether the message should be sent to: (1) a DCC in the same Bus; (2) a DCC in another Bus or (3) a web service.

There are three main scopes for message exchanging. First, DCCs within the same Bus communicate using native programming language schemes. A second form is the communication between DCCs attached to different Buses (either in a local or remote machine). The third form involves communication between a DCC and web services. The last two forms use simple object access protocol (SOAP) [21] XML messages. Whenever a message is meant to leave the Bus, the Resolver converts it from the internal format to SOAP. This SOAP message is based on a WSDL [11] specification. Since both DCCs and web services are described using WSDL, there is no need to make a distinction between them. If the receiver is a web service, the message is already adequately formatted. However, if the receiver is another Bus, the message must be converted back to the internal format.

The *Repository Manager* is also a specialized DCC that works as described on Sect. 6.1. As all DCCs, the Repository Manager uses the Bus for communication with DCCs, including other Repository Managers. As can be seen in Fig. 6, the Bus-Python of machine A does not have a Repository Manager directly attached to it. It makes use of the Repository Manager attached to Bus-Java.

The current version of this architecture is fully functional in a local environment, and is implemented in the Java language. It implements the Bus-based communication, and can deal with process and passive DCCs. Furthermore, this implemented framework uses an OWL ontology to match passive DCCs with their companions, fully supporting the content-type driven execution described in Sect. 5.1.

8 Meeting the requirements

This section summarizes how the DCC approach meets the requirements presented in Sect. 3. In Sect. 8.2, we show how DCCs bridge the gap between user interaction and reuse/sharing features. Section 8.3 shows how the author can participate in constructing executable software by composing components. Section 8.4 shows how DCCs break barriers between content and executable software.

8.1 How DCCs meet the requirements

In the diagram illustrated in Fig. 2, we showed that authoring tasks concentrate on the “content layer” for the content-centric approach, and the “software layer” for the process-centric approach. The DCC infrastructure, on the other hand, works behind the scenes and uses DCC semantic annotations to appropriately combine software and content pieces and present to the author, in a transparent way, a “result layer” perspective. Moreover, authors’ shareable contributions are not limited to content or to software: they produce, reuse and share both indistinctly. This meets the first requirement (unify content- and process-centric models).

Digital content components’ functionality also meets the second and third requirements (single abstraction and homogeneous interface, respectively), providing a single mechanism for consuming and authoring any multimedia artifact. In the content-centric approaches, complex digital objects work as content aggregators. They are prepared to be plugged to a client platform, which will be used to consume the content. The software necessary to handle the content is concentrated in the client platform. This platform architecture varies according to domain standards. In MPEG-21, for example, the client platform can be the player that will run the media inside the complex digital objects. Usually the client platform knows each type of supported content, and the ways to relate this content with other content types. For

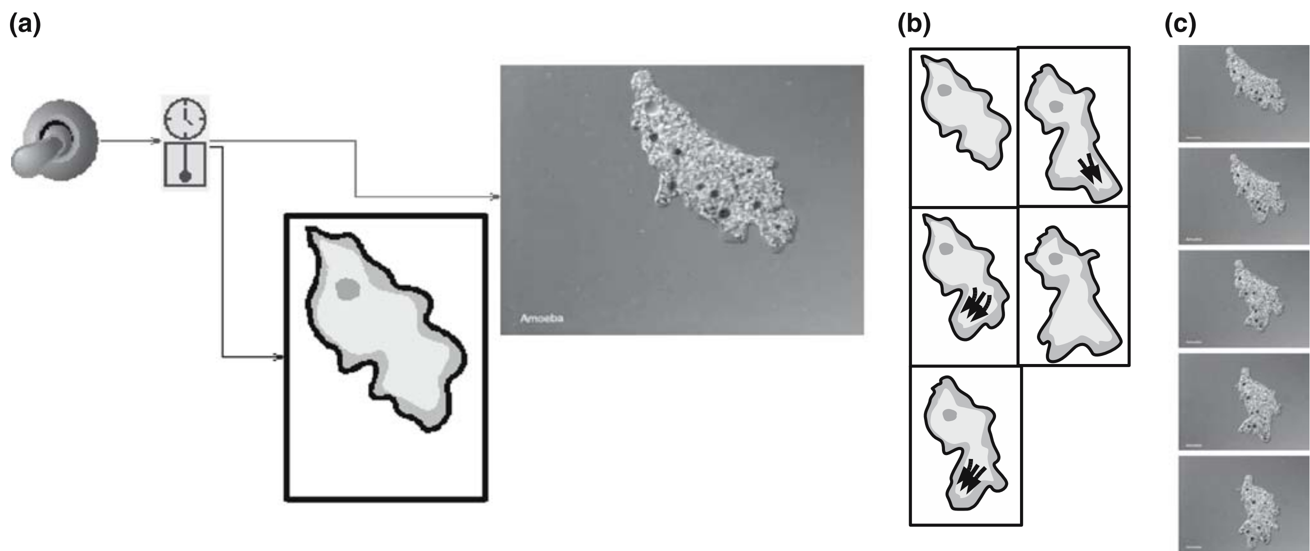

Fig. 7 Magic House animation illustrating a cell movement. (**a**) Production design, (**b**) and (**c**) passive DCCs

this reason, the combinations among content artifacts are constrained to those pre-specified by the client platform. To handle new kinds of content, not supported by the client platform, some content-centric infrastructures accept software extensions in the client platform.

In the DCC model, on the other hand, the interface works like an explicit platform-neutral contract that specifies how DCCs can be connected. The client platform does not need to know beforehand how a DCC can be connected to another, since it is explicitly declared. Instead of defining client platform built-in content handlers, the DCC approach defines a specialized software DCC (companion DCC), whose purpose is to handle the content of another DCC. In contrast with content-centric approaches, the DCC infrastructure is based on a thin client platform, which decentralizes content handling tasks. So, authors are free to create both content or software expansions inside DCCs; this task does not need to be delegated to software development specialists that implement the client platform.

Section 5.1 already showed how the DCC model and infrastructure meet the fourth requirement (content-type driven execution). Compared with the approaches to invoke software routines according to the type of the content to be handled (software frameworks, active document components and software plug-ins), the use of a taxonomic ontology to associate a companion with a passive DCC enables to express not only “how the content is stored”, but also “how the content must be interpreted”. Returning to the example illustrated in Fig. 4, the same content—the drawing of a crab—can be interpreted as a static image, or as a moving aquatic being.

The IUHM hypermedia model of [23] addresses problems similar to ours. It encapsulates executable code and other kinds of content in homogeneous units, meeting the first requirement. IUHM provides a strategy for content-type driven execution, and thus meets the fourth requirement, also supporting a notion similar to that of our companion component. The main distinction between IUHM and DCC lies in the “component” approach adopted by DCCs, where the interface plays a major role. The IUHM model does not meet the third requirement, since it does not define an explicit interface.

8.2 Swapping content and program code

This section shows examples of multimedia DCC productions, comparing them with other approaches in terms of facilitating user interaction × reuse. These examples emphasize the importance of an homogeneous model in situations where the author can use a content or a program code to perform equivalent tasks.

Consider the following general context. In our Magic House environment, production authors are offered a choice of (DCC) blocks to be composed to design animations. These blocks are selected and connected by direct manipulation using a visual tool, and customized by modifying parameters in a property sheet. A direct connection (arrow) between two DCCs indicates that the first DCC will send a message to the second every time a selected event occurs in the first DCC. A switch + clock indicates that the animation will start by pressing the switch, and that the clock will control

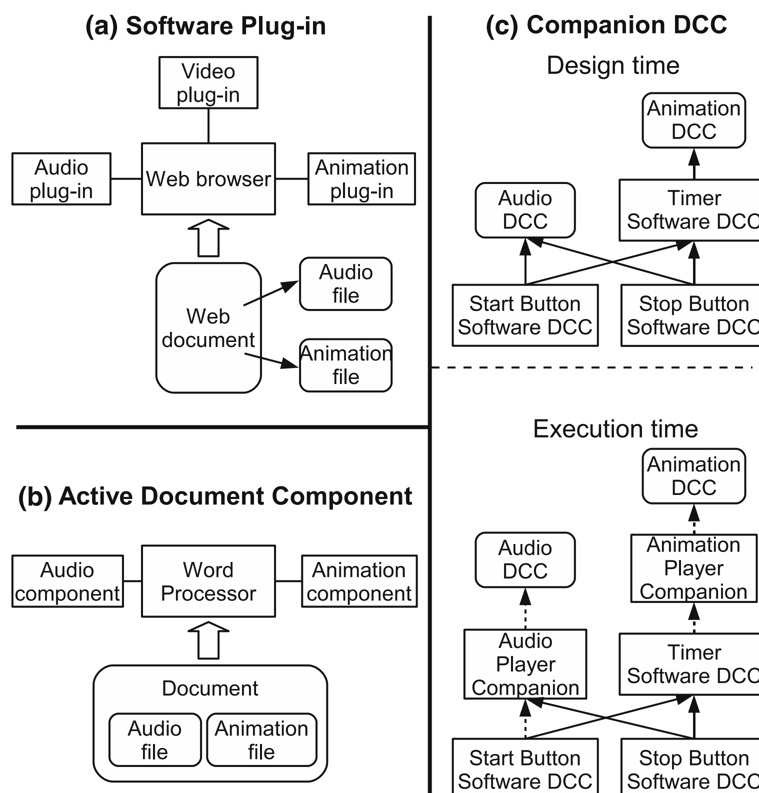

Fig. 8 Diagram confronting connection architectures

synchronization. Once the composition is specified, it can be executed.

Figure 7a shows the design of a biology production in the Magic House environment, whose purpose is to deploy an animation that illustrates how a cell moves. The animation synchronizes two sequences of images (i.e., DCCs that encapsulate sequences of image frames): shots taken from an electronic microscope showing cell movement and diagrams that depict the movement dynamics. The execution of this production is a movie that synchronously shows cell movement and corresponding dynamics. During execution, the clock sends messages (ticks) to both frame sequences at a given rate. Users can interact with this animation at any given time, for instance, changing tick frequency, editing clock parameters, congealing frames, etc. Here, a given author (e.g., a scientist or a biology teacher) can design the production in the environment using the plug and play paradigm. Another user (e.g., another scientist or biology students) can not only execute (consume) the production, but also interact with it and change it by customizing its blocks.

Continuing with this example, suppose that instead of using a video encapsulated inside a passive DCC, the author wants to show images streamed on-line from the electronic microscope. Here, the author will replace the passive video DCC by a process DCC, containing software routines to access the microscope's images.

This example points out three important issues in the DCC model. The first issue concerns its user-centered nature, where the distinction between multimedia authoring and execution/consumption becomes fuzzy. Since people playing the production can easily interact and change it, the "audience" becomes a partner in the authorship, by experimenting with the DCC building blocks.

The other two issues concern the unified implementation philosophy behind passive and process DCCs. First, the absence of a distinction between software reuse (process-centric) and content reuse (content-centric) gives the author the freedom to combine software and content in a production, without the need to consider the nature of DCC. Second, thanks to content-type driven execution, the video and animation DCCs are dynamically associated with the respective companion DCCs, which will show their content in the screen. At a first glance, this may seem similar to the mechanism used by software plug-ins to display videos and animations inside a web browser. However, plug-ins work inside the browser as isolated routines; unlike DCCs, they do not expose explicit interfaces to be connected with other content objects of a web document. In some cases,

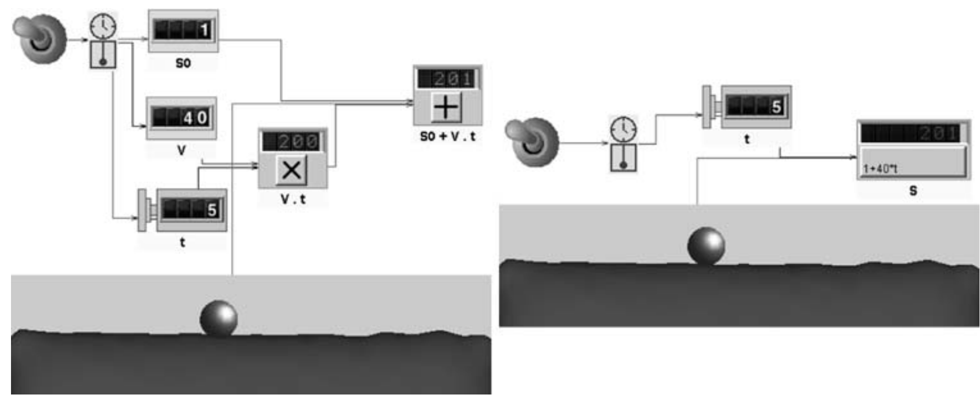

Fig. 9 Two versions of a mathematics educational project using specialized components

software development experts can use script routines inside web pages (e.g., Javascript routines) to interact with plug-ins, but this procedure involves hard programming and is limited to the plug-in predefined functionalities. A companion DCC, on the other hand, can be tailored to each content-type, exposing a specific interface.

This key difference between DCCs and software plug-ins/active document components is related to the way they are associated with the content. Figure 8 presents an example of this distinction. Consider a variation of the application illustrated in Fig. 7, where the author added another button DCC directly connected to the video and animation DCCs. The role of this button is to request to the video and animation DCCs to advance just one frame at a time. Suppose the author wants to implement the same presentation using two other approaches: (a) an HTML page referring to the animation and the video files, which will be submitted to a web browser containing plug-ins to deal with animation and video and (b) a text document embedding the animation and the video files, in a text processor that can access active document components to deal with animation and video.

Figure 8 shows solutions (a) and (b) at execution time. Solution (c) uses a companion DCC and is divided into design time and execution time. At design time, process DCCs are directly connected to passive DCCs. At execution time, companion DCCs are dynamically invoked to access the passive DCCs, i.e., authors do not need to concern themselves with these execution time details. Dashed arrows show connections that appear at execution time, based on content-type driven execution.

Software plug-ins and active document components are based on a connection architecture where the plug-ins/components are attached to a host system. In the companion DCC approach, each companion can be connected with any other component that has a compatible interface. As shown in Fig. 8, software plug-ins and active document components approaches adopt a star connection architecture, while the companion DCCs approach uses a network connection architecture. The DCC connection architecture enables the author to connect, for example, the animation and video handler DCCs with other DCCs, to provide synchronization. In the architectures of software plug-ins and active document components, this will require specialized programming from software development specialists. The DCC connection architecture provides content-type driven execution support without the need of a central module, and can be explored to create distributed compositions.

8.3 Composing higher-level components

Section 8.2 gave an example of authoring and interacting with a simple production. This section discusses how to create productions from composition of others.

Figure 9 shows a physics project, whose purpose is to display how the values resulting from the kinematics function `f(t) = S0+V.t` affects the movement of a ball. Figure 9a shows a composition that represents the equation `1+40.t` that will animate a ball (passive DCC). The execution of this composition works as follows. Again, a switch DCC starts the clock. The clock sends regular messages to DCCs labeled by "`S0`", "`V`" and "`t`". The "`t`" DCC is a counter, which increments its value and dispatches this value as a message to the "`V.t`" DCC. The "`V`" and "`S0`" DCCs are constants, and dispatch their values to the "`V.t`" and "`S0+V.t`" DCCs, respectively. The "`V.t`" DCC multiplies the values received and dispatches the result to the "`S0+V.t`" DCC, which sums the received values. Finally, the values of "`S0+V.t`" are dispatched to the ball DCC, whose position is defined by the received value. Authors (here, school children) can change clock properties, coefficient values and even operations (e.g., `S0-V.t`). The net result is to allow the children to experiment with equations.

Figure 9b shows another version of the same project, where the ball position is calculated by a compo-

nent, labeled with "S", previously built and stored in the DCC repository. Again, authors can edit this component properties. This "S" component has a property named function, which contains the mathematical expression used to calculate the component's output value, based on an input value. The expression is displayed at the bottom of the component (1+40*t). The contrast between these two versions illustrates that fine-grained components give the end-user more control in application development than coarse-grained ones.

We can thus consider two extremes of flexibility: in one, the author combines and customizes existing components using DCC interfaces; at the other, a developer creates a component by writing a program code, and the author cannot modify the program code. Assembling fine-grained components to devise a solution lies somewhere in between [22]. Since this assembly can produce a higher-level component, authors can produce and share their components without writing program code.

The DCC model enables the creation of higher level components via composition of lower level components. This example shows that user-author collaboration is not restricted to non-executable content, but also to software, using a single mechanism and environment. In contrast, complex digital object initiatives accept some kinds of scripts attached to content. However, these strategies: (1) do not have suitable mechanisms to reuse the script code and adapt it to new contexts, since they do not define software reuse strategies, like those used by software components and DCCs and (2) have constraints on script expressiveness, e.g., MPEG-21 DIMs.

8.4 Breaking barriers: content and software

Our last example concerns a traditional question in computer graphics animation. When authors want to move elements in animations they have two possible ways: (1) "manually" define the element position in each frame and (2) produce a software routine to calculate the element position for each frame.

The three compositions shown in Fig. 10 illustrate this question. The goal is the same as the one illustrated in Fig. 9, to animate a ball. The solution at the bottom of Fig. 10c is the same as the one used in the cell examples (Fig. 7). The counter will prompt the animation, using the frames depicted on the bottom right.

In the other two solutions, Fig. 10a, b, a spreadsheet DCC, fed by the counter, will direct the ball's movement, computing the function 1+40.t. The counter updates t (cell B4) and the result (cell B5) sends a message to the ball DCC, updating the ball's position. In Fig. 10b, the spreadsheet DCC encapsulates a set of (t, x) position values. Every time the counter updates t (column A), the x-value (column B) indicates the ball's new position. From an observer's point of view, all three animations are identical, moving the ball to the right.

Compositions 10a, b use a spreadsheet, but for different purposes. In the first case, the spreadsheet is used to compute a function, and thus acts like a process DCC. In the second case, it contains a list of states (positions) for the ball, and acts like a passive DCC. Depending on the solution, the same artifact (spreadsheet) can have passive content as well as executable routines. This kind of situation can be handled neither by process-centric nor by content-centric approaches. Finally, the composition of Fig. 10c shows that "passive" digital content can replace program code.

This final example shows that, using the DCC model and infrastructure, authors do not need to concern themselves with whether they are connecting software or content pieces. They work in a higher level of abstraction, which is driven by the meaning conferred to each artifact.

9 Concluding remarks

Our work presented an user-author centered multimedia building block in a scenario where the gap between the roles of author and end-user is being closed.

It combines the support to users' ease of use to the author's need for content adaptation, share and reuse. Our work is based on the notion of DCC. It involves both content and software reuse, adopting a model that unifies advances in content-centric and process-centric approaches. On the one side, it takes advantage of the "interface as functionality abstraction" paradigm of the software component approach to provide content with functionality description. On the other hand, it brings results of complex digital object research into the field of software sharing and reuse. This model is being used to implement content compositions in several domains within our DCC composition and discovery framework, e.g., geographic data management [28].

The main contributions of this paper are: (1) the discussion and elaboration of the main characteristics that qualify DCC as a user-author centered building block in the multimedia context; (2) the analysis of the notion of content-type driven execution, under different guises (e.g., software frameworks, plug-ins and active document components), thereby unifying their study under a single set of criteria and (3) the application of content-type driven execution to multimedia user-authoring, in which a given artifact "searches for" appropriate software to execute it, based on a taxonomic ontology.

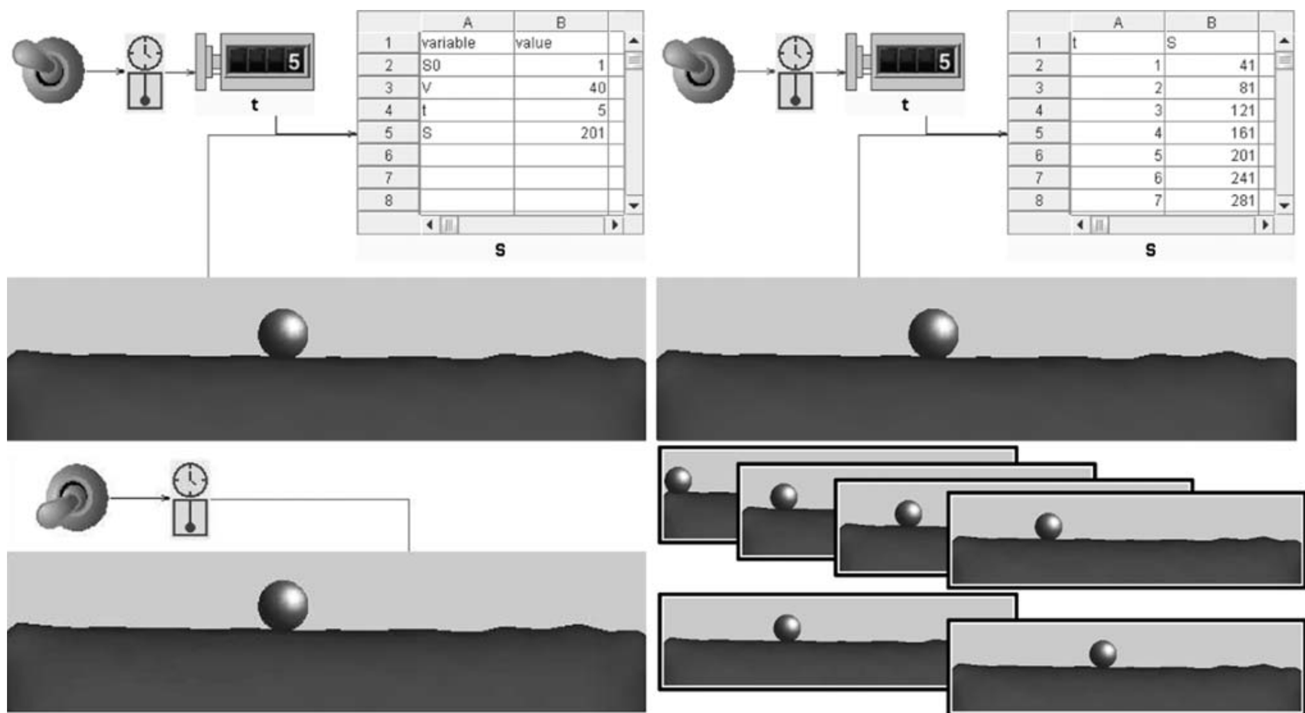

Fig. 10 Animation of a ball: three alternatives

Besides the strategies presented in the paper, the model is designed to accept other kinds of composition. In particular, we are working to enable the use of workflows to describe a process [20]. Ongoing work involves version control support for DCCs and DCC compositions; the management of distributed update and alignment of the framework's ontologies.

Acknowledgements This paper was partially supported by grants from CNPq, CAPES, FAPESP and UNIFACS and CNPq projects WebMaps and Agroflow, as well as Microsoft WeBios e-science project. We also thank Maria Cecília Baranauskas for her insightful comments.

References

1. ADL: Sharable content object reference model (SCORM) 2004—overview, 2nd edn. www.adlnet.org/screens/shares/dsp_displayfile.cfm?fileid=992, accessed on 11/2004 (2004)
2. Alonso, G., et al.: Web Services—Concepts, Architectures and Applications. Springer, Berlin Heidelberg New York (2004)
3. Apple Computer Inc.: OpenDoc Programmer's Guide for the Mac OS. Apple Press (1996)
4. Bekaert, J., De Kooning, E., Van de Walle, R.: Packaging models for the storage and distribution of complex digital objects in archival information systems: a review of MPEG-21 DID principles. Multimedia Syst. **10**(4), 286–301 (2005)
5. Bolour, A.: Notes on the eclipse plug-in architecture. www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, accessed on 06/2005 (2003)
6. Brockschmidt, K.: Inside OLE, 2nd edn. Microsoft Press (1995)
7. Bulterman, D., Hardman, L.: Structured multimedia authoring. ACM Trans. Multimedia Comput. Commun. Appl. **1**(1), 89–109 (2005)
8. Burnett, I., Davis, S., Drury, G.: MPEG-21 digital item declaration and identification-principles and compression. IEEE Trans. Multimedia **7**(3), 400–407 (2005)
9. Burnett, I., Van de Walle, R., Hill, K., Bormans, J., Pereira, F.: MPEG-21: goals and achievements. Multimedia **10**, 60–70 (2003)
10. CCSDS: Reference model for an open archival information system (OAIS)—blue book. Technical Report CCSDS 650.0-B-1. www.ccsds.org/CCSDS/documents/650x0b1.pdf, accessed on 11/2004 (2002)
11. Chinnici, R., et al.: Web services description language (WSDL) version 2.0 part 1: core language—W3C working draft, 3 August 2004. www.w3.org/TR/2004/WD-wsdl20-20040803/, accessed on 11/2004 (2004)
12. Cullot, N., Parent, C., Spaccapietra, S., Vangenot, C.: Ontologies: a contribution to the DL/DB debate. In: Proceedings of the 1st International Workshop on the Semantic Web and Databases, 29th International Conference on Very Large Data Bases, pp. 109–129 (2003)
13. Emmerich, W.: Distributed component technologies and their software engineering implications. In: ICSE '02, Proceedngs of the 24th International Conference on Software Engineering, pp. 537–546. ACM Press (2002)
14. Hopkins, J.: Component primer. Commun. ACM **43**(10), 27–30 (2000)

15. IEEE L.T.S.C.: Draft standard for learning object metadata—IEEE 1484.12.1-2002. http://ltsc.ieee.org/doc/wg12/LOM_1484_12_1_v1_Final_Draft.pdf, access on 10/2003 (2002)
16. ISO/IEC: ISO/IEC TR 2100-1—information technology—multimedia framework (MPEG-21), part 1: vision, technologies and strategy, 2nd edn. Technical Report (2004)
17. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131–183 (1992)
18. Martin, D., et al.: OWL-S: semantic markup for web services. www.daml.org/services/owl-s/1.1/overview/, accessed on 12/2004 (2004)
19. McDonald, W.A., Hyde, J., Montgomery, A.: CMI guidelines for interoperability AICC. www.aicc.org/docs/tech/cmi001v4.pdf, accessed on 11/2004 (2004)
20. Medeiros, C.B., Perez-Alcazar, J., Digiampietri, L., Pastorello Jr, G.Z., Santanchè, A., Torres, R.S., Madeira, E., Bacarin, E.: WOODS and the web: annotating and reusing scientific workflows. SIGMOD Record **34**(3), 18–23 (2005)
21. Mitra, N.: SOAP version 1.2, part 0: primer—W3C recommendation 24 June. www.w3.org/TR/2003/REC-soap12-part0-20030624/, accessed on 12/2004 (2003)
22. Mrch, A.I., Stevens, G., Won, M., Klann, M., Dittrich, Y., Wulf, V.: Component-based technologies for end-user development. Commun. ACM **47**(9), 59–62 (2004)
23. Nanard, M., Nanard, J., King, P.: IUHM: a hypermedia-based model for integrating open services, data and metadata. In: HYPERTEXT '03: Proceedings of the 14th ACM Conference on Hypertext and Hypermedia, New York, pp. 128–137 (2003)
24. Noy, N.F., Sintek, M., Decker, S., Crubézy, M., Fergerson, R.W., Musen, M.A.: Creating semantic web contents with protégé-2000. IEEE Intell Syst **16**(2), 60–71 (2001)
25. OMG: Reusable asset specification—final adopted specification. http://www.omg.org/cgi-bin/doc?ptc/2004-06-06, accessed on 10/2004 (2004)
26. Potts, R.: Mozilla developer documentation—NG layout embedding APIs. www.mozilla.org/newlayout/doc/webwidget.html, accessed on 06/2005 (1998)
27. Roschelle, J., DiGiano, C., Koutlis, M., Repenning, A., Phillips, J., Jackiw, N., Suthers, D.: Developing educational software components. Computer **32**(9), 50–58 (1999)
28. Santanchè, A., Medeiros, C.B.: Geographic digital content components. In: Proceeings of VI Brazilian Symposium on GeoInformatics, pp. 281–290 (2004)
29. Santanchè, A., Medeiros, C.B.: Managing dynamic repositories for digital content components. In: Current Trends in Database Technology—EDBT 2004 Workshops, vol. LNCS 3268, pp. 66–77 (2004)
30. Santanchè, A., Medeiros, C.B.: Self describing components: searching for digital artifacts on the web. In: Proceedngs of XX Brazilian Symposium on Databases, pp. 10–24 (2005)
31. Santanchè, A., Teixeira, C.A.C.: Anima: promoting component integration in the web (in Portuguese). In: Proceedings of 7th Brasilian Symposium on Multimedia and Hypermedia Systems, pp. 261–268 (2001).
32. Smith, M.K., Welty, C., McGuinness, D.L.: OWL web ontology language guide—W3C recommendation 10 February 2004. www.w3.org/TR/2004/REC-owl-guide-20040210/, accessed on 11/2004 (2004)
33. Smythe, C., Jackl, A.: IMS content packaging information model. Specification, IMS Global Learning Consortium, Inc. www.imsglobal.org/content/packaging/cpv1p1p4/imscp_infov1p1p4.html, accessed on 11/2004 (2004)
34. The Fedora Project team: Mellon Fedora Technical Specification. www.fedora.info/documents/master-spec-12.20.02.pdf, accessed on 12/2004 (2002)
35. The Library of Congress: METS: an overview & tutorial. www.loc.gov/standards/mets/METSOverview.v2.html, accessed on 11/2004 (2004)
36. UDDI Committee Specification: Uddi version 2.04 API specification. uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm, accessed on 06/2005 (2002)
37. Wang, X., DeMartini, T., Wragg, B., Paramasivam, M., Barlas, C.: The MPEG-21 rights expression language and rights data dictionary. IEEE Trans. Multimedia **7**(3), 408–417 (2005)