Fluid Web e Componentes de Conteúdo Digital:
da visão centrada em documentos para a visão centrada em conteúdo

André Santanchè

Tese de Doutorado

Instituto de Computação
Universidade Estadual de Campinas

Fluid Web e Componentes de Conteúdo Digital: da visão centrada em documentos para a visão centrada em conteúdo

André Santanchè[1]

Junho de 2006

Banca Examinadora:

- Profa. Claudia Bauzer Medeiros (Orientadora)
- Prof. Carlos José Pereira de Lucena
  Departamento de Informática – PUC-Rio
- Prof. Marco Antonio Casanova
  Departamento de Informática – PUC-Rio
- Profa. Cecília Mary Fischer Rubira
  Instituto de Computação – UNICAMP
- Prof. Ricardo da Silva Torres
  Instituto de Computação – UNICAMP

[1]**Suporte financeiro da UNIFACS, CNPq (Proc. 140109/2005-6) e apoio parcial dos projetos SAI/PRONEX-MCT, WebMaps e AgroFlow do CNPq, e eScience da Microsoft.**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Santanchè, André
Sa59f Fluid Web e componentes de conteúdo digital: da visão centrada em documentos para a visão centrada em conteúdo / André Santanchè – Campinas, [S.P. :s.n.], 2006.

Orientadora : Claudia Bauzer Medeiros
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Componentes de software. 2. Gerenciamento de configuração de software. 3. Semântica e processamento de dados. 4. Multimídia (Computação). I. Medeiros, Claudia Bauzer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Fluid Web and digital content components: from the document-centric view to the content-centric view.

Palavras-chave em inglês (Keywords): 1. Software components. 2. Software configuration management. 3. Semantics and data processing. 4. Multimedia (Computer science).

Área de concentração: Banco de Dados

Titulação: Doutor em Ciência da Computação

Banca examinadora: Profa. Dra. Claudia Bauzer Medeiros (IC-UNICAMP), Prof. Dr. Carlos José Pereira de Lucena (DI-PUC-Rio), Prof. Dr. Marco Antonio Casanova (DI-PUC-Rio), Profa. Dra. Cecília Mary Fischer Rubira (IC-UNICAMP), Prof. Dr. Ricardo da Silva Torres (IC-UNICAMP)

Data da defesa: 10-08-2006

Programa de Pós-Graduação: Doutorado em Ciência da Computação

Fluid Web e Componentes de Conteúdo Digital: da visão centrada em documentos para a visão centrada em conteúdo

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por André Santanchè e aprovada pela Banca Examinadora.

Campinas, 10 de agosto de 2006.

Profa. Claudia Bauzer Medeiros
(Orientadora)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Substitua pela folha com a assinatura da banca

Resumo

A Web está evoluindo de um espaço para publicação/consumo de documentos para um ambiente para trabalho colaborativo, onde o conteúdo digital pode viajar e ser replicado, adaptado, decomposto, fundido e transformado. Designamos esta perspectiva por *Fluid Web*. Esta visão requer uma reformulação geral da abordagem típica orientada a documentos que permeia o gerenciamento de conteúdo na Web. Esta tese apresenta nossa solução para a Fluid Web, que permite nos deslocarmos de uma perspectiva orientada a documentos para outra orientada a conteúdo, onde "conteúdo" pode ser qualquer objeto digital. A solução é baseada em dois eixos: (i) uma unidade auto-descritiva que encapsula qualquer tipo de artefato de conteúdo – o *Componente de Conteúdo Digital* (Digital Content Component – DCC); e (ii) uma infraestrutura para a Fluid Web que permite o gerenciamento e distribuição de DCCs na Web, cujo objetivo é dar suporte à colaboração na Web.

Concebidos para serem reusados e adaptados, os DCCs encapsulam dados e software usando uma única estrutura, permitindo deste modo composição homogênea e processamento de qualquer conteúdo digital, seja este executável ou não. Estas propriedades são exploradas pela nossa infraestrutura para a Fluid Web, que engloba mecanismos de descoberta e de anotação de DCCs em múltiplos níveis, gerenciamento de configurações e controle de versões. Nosso trabalho explora padrões de Web Semântica e ontologias taxonômicas, que servem como uma ponte semântica, unificando vocabulários para gerenciamento de DCCs e facilitando as tarefas de descrição/indexação/descoberta de conteúdo. Os DCCs e sua infraestrura foram implementados e são ilustrados por meio de exemplos práticos, para aplicações científicas.

As principais contribuições desta tese são: o modelo de Digital Content Component; o projeto da infraestrutura para a Fluid Web baseada em DCCs, com suporte para armazenamento baseado em repositórios, compartilhamento, controle de versões e gerenciamento de configurações distribuídas; um algoritmo para a descoberta de conteúdo digital que explora a semântica associada aos DCCs; e a validação prática dos principais conceitos desta pesquisa, com a implementação de protótipos.

Abstract

The Web is evolving from a space for publication/consumption of documents to an environment for collaborative work, where digital content can travel and be replicated, adapted, decomposed, fusioned and transformed. We call this the *Fluid Web* perspective. This view requires a thorough revision of the typical document-oriented approach that permeates content management on the Web. This thesis presents our solution for the Fluid Web, which allows moving from the document-oriented to a content-oriented perspective, where "content" can be any digital object. The solution is based on two axes: a self-descriptive unit to encapsulate any kind of content artifact – the *Digital Content Component* (DCC); and a Fluid Web infrastructure that provides management and deployment of DCCs through the Web, and whose goal is to support collaboration on the Web.

Designed to be reused and adapted, DCCs encapsulate data and software using a single structure, thus allowing homogeneous composition and processing of any digital content, be it executable or not. These properties are exploited by our Fluid Web infrastructure, which supports DCC multilevel annotation and discovery mechanisms, configuration management and version control. Our work extensively explores Semantic Web standards and taxonomic ontologies, which serve as a semantic bridge, unifying DCC management vocabularies and improving DCC description/indexing/discovery. DCCs and infrastructure have been implemented and are illustrated by means of examples, for scientific applications.

The main contributions of this thesis are: the model of Digital Content Component; the design of the Fluid Web infrastructure based on DCCs, with support for repository-based storage, distributed sharing, version control and configuration management; an algorithm for digital content discovery that explores DCC semantics; and a practical validation of the main concepts in this research through implementation of prototypes.

Agradecimentos

À minha orientadora e mestra, Claudia Bauzer Medeiros, por acreditar no meu potencial e por dedicar tempo e paciência na orientação da minha tese e na minha formação como mestre e pesquisador.

À minha esposa Simone, que deixou emprego e casa para embarcar comigo nesta aventura. Seu amor, compreensão e paciência foram fundamentais nesta jornada.

À minha mãe Lys, que defendeu sua tese de doutorado aos setenta anos de idade e que tenho como exemplo de amor, luta e perseverança.

Ao meu pai Francesco, que sabia explicar como o mundo funciona e que plantou em mim a paixão pela ciência.

Aos meus irmãos Pepe, Giulio, Ciça e Ia, que, perto ou longe, sempre acreditaram e incentivaram o meu trabalho. Agradeço pelo apoio, amor e carinho. Agradeço, também, aos familiares e amigos pelo constante incentivo e apoio.

Aos amigos do LIS, que tiveram grande influência e contribuição no meu trabalho e com os quais dividi sonhos, conquistas, diversão e os puxões de orelha da Profa. Claudia. Agradeço ao Gilberto, que trabalhou junto comigo neste projeto e que teve grande contribuição nos resultados alcançados.

A todos os funcionários do IC, por todo suporte. Em especial, a seu Nelson, um amigo e exemplo de determinação.

Ao meu orientador de mestrado, Cesar Teixeira, por seu papel determinante no meu trajeto para o doutorado.

À UNIFACS, especialmente ao Prof. Manoel Barros, pelo crédito e apoio que deu para a minha formação e pesquisa. Agradeço, também, aos meus colegas e amigos do NUPPEAD e do NUPPERC, companheiros de batalha e amigos.

Aos meus companheiros e amigos do Volare e do Anchieta, que mesmo a distância sempre estiveram presentes.

À Eliana Brenner, pelo apoio e incentivo em meus primeiros passos de pesquisador.

Aos membros da banca, pelas sugestões e contribuições feitas ao meu trabalho.

A todos aqueles que não mencionei, talvez mereçam o maior agradecimento, pois acreditaram, participaram, colaboraram e aparecem anônimos neste agradecimento.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

O cenário atual de produção, distribuição e consumo de conteúdo digital é caracterizado por uma forte influência de dois fatores. O primeiro é a ampliação da oportunidade de compartilhamento e intercâmbio de conteúdo digital, causada principalmente pela Internet e pela difusão de padrões abertos de representação. O segundo é a crescente participação dos usuários finais no processo de produção e modificação deste conteúdo digital, como resultado da criação de ferramentas mais acessíveis a este tipo de usuário.

A combinação destes dois fatores tem progressivamente impulsionado um novo comportamento na produção, consumo e distribuição de conteúdo digital na *Web*, em que os usuários têm acesso participativo ao conteúdo, alternando papéis de autor e consumidor. Esta situação contrasta com a visão clássica de publicação/consumo, em que um restrito grupo ativo de autores publica algo a ser consumido por um vasto grupo passivo de usuários.

Este novo cenário, para o qual criamos o nome de *Fluid Web*, motivou o trabalho desta tese, que se concentra nas diferentes facetas de *reuso* de artefatos digitais. Reuso, na tese, é considerado na acepção mais geral do termo, ou seja, envolvendo os conceitos de consumo, mas também replicação, adaptação, decomposição, fusão e transformação.

Por um lado, surge a necessidade de revisão do modelo clássico da Web de publicação de conteúdo baseado em documentos, de forma a permitir o novo cenário onde o conteúdo digital pode ser *reusado*. Este tipo de demanda requer pesquisa em termos de infra-estrutura computacional – por exemplo, protocolos de comunicação, modelos de armazenamento, compartilhamento e versionamento, mecanismos de publicação, sistemas de gerenciamento, proteção e indexação do conteúdo. Por outro lado, inserido neste cenário está um usuário que progressivamente incorpora funções de autor, o usuário-autor. Neste novo contexto, atividades de autoria deixam de estar restritos a um grupo especializado

à produção de um tipo específico de conteúdo digital, e passam a ser uma característica inerente à maioria dos usuários. Tal perspectiva tem impactos em vários tipos de pesquisa, como em interfaces humano-computador, modelos de cooperação, ferramentas de autoria/gerenciamento e padrões de empacotamento de conteúdo. Isto também implica que práticas e ferramentas anteriormente concebidas para grupos e domínios específicos de autores devem ser repensadas para um contexto mais amplo que abrange o universo dos usuários finais em geral.

Como conseqüência de todas estas constatações, foi necessário abordar pesquisas em empacotamento/distribuição/reuso de conteúdo, arquitetura e reuso de software, controle de versões, gerenciamento de configurações e bibliotecas digitais. Ressaltamos que esta pesquisa, sob muitos aspectos, combina esforços independentes que têm se empenhado em resolver problemas complementares, tanto em Engenharia de Software quanto em Engenharia de Conteúdo. Muito embora a motivação seja centrada na Web, os resultados da nossa pesquisa se aplicam a qualquer cenário de trabalho colaborativo.

A motivação desta tese é resultado de observações realizadas em experiências práticas com um ambiente de autoria para a construção de aplicações educacionais desenvolvido por nós desde 1994 – o sistema Casa Mágica [71]. Este sistema tem sido aplicado em atividades práticas com alunos de ensino médio e fundamental, como também no ensino superior na cidade de Salvador/BA. Muitos dos exemplos apresentados, principalmente no Capítulo 3, tomaram como base atividades práticas realizadas.

1.2 Aspectos de Pesquisa Envolvidos

Esta seção caracteriza o cenário onde a pesquisa de tese se insere, citando alguns dos problemas e soluções existentes, a demanda em aberto e as pesquisas que estão sendo desenvolvidas para atender esta demanda. Ela traça um perfil do cenário de produção, distribuição e consumo de conteúdo digital, onde a atividade de reuso de conteúdo cumpre um papel essencial. Tal cenário é caracterizado segundo três enfoques:

- **Papel do Autor**: qual o papel atribuído ao autor na produção de conteúdo digital e como este papel tem se modificado, demandando novas soluções.
- **Modelo de Compartilhamento/Reuso de Conteúdo Digital**: quais são os modelos adotados para representar as unidades básicas de compartilhamento e reuso de conteúdo digital, dentro de um modelo de produção onde o reuso é uma prática fundamental.
- **Suporte ao Compartilhamento/Reuso de Conteúdo Digital**: quais as tarefas envolvidas no suporte a um ambiente de produção pautado no compartilhamento e reuso de conteúdo.

A tese apresenta soluções integradas – modelos, algoritmos e ferramentas implementados – para alguns dos desafios constatados nestes três enfoques. As hipóteses básicas por trás deste trabalho são:

- Abordagens para produção, compartilhamento e reuso de software executável e de conteúdo são complementares e podem ser combinadas.
- O atual modelo de produção e distribuição da Web, centrado em documentos, não é suficiente para a Fluid Web e para o reuso.

1.2.1 Papel do Autor

A caracterização do autor tem sido essencial na concepção dos modelos utilizados na produção, compartilhamento e reuso de artefatos digitais. O perfil do autor nos primórdios da computação era o de um profissional de Informática cujo foco principal de atuação era o desenvolvimento de software executável – o que denominamos *desenvolvimento centrado no processo*. Gradativamente, o conteúdo digital manipulado pelo software executável foi adquirindo importância e o perfil do autor se tornou mais diversificado. Surge então a figura do autor cujo foco de trabalho é o conteúdo – o que denominamos *desenvolvimento centrado no conteúdo*.

Como ilustra a Figura 1.1, este usuário pode ser visto como um nó de um grande hiper-espaço de conteúdo compartilhado (a Internet), consumindo conteúdo (setas de entrada) e reusando/produzindo conteúdo (setas de saída). Alguns dos nós/usuários são apenas consumidores (nó 1), enquanto outros são autores que produzem algo a partir da estaca zero (nó 2). O tipo mais comum é o daquele usuário que alterna e combina os papéis de usuário e autor, o usuário-autor. Este cenário de uso coletivo reflete a realidade de hoje em que praticamente qualquer usuário é autor de algum artefato digital (textos, planilhas, apresentações, etc.). À medida que estes artefatos trafegam entre os nós/usuários, eles sofrem evolução. Este processo de obter um conteúdo e atualizá-lo, adaptá-lo, modificá-lo e melhorá-lo é a essência do conceito de reuso. Por não serem em sua maioria profissionais de computação, estes usuários são impulsionados a se tornar "reusuários" na sua tarefa de produção, dado que não faz sentido produzir um artefato a partir da estaca zero quando eles têm em mãos material de qualidade e trabalham com limitações de recursos e tempo.

O conceito de usuário-autor se aplica indistintamente aos que criam conteúdo e aos que produzem software. A principal diferença é que se exige dos últimos sofisticação técnica enquanto que os primeiros não são necessariamente especialistas na arte de produção. Um modelo de produção e consumo de conteúdo digital e uma infraestrutura computacional que dê suporte a este modelo devem portanto estar aptos a atender a este perfil de usuário-autor, definindo as seguintes características:

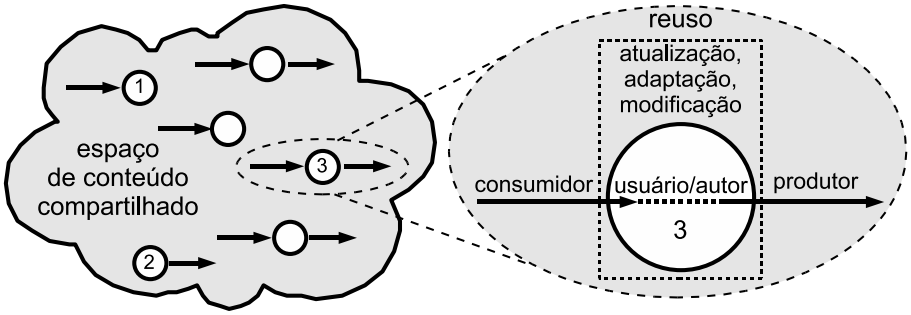

Figura 1.1: Diagrama ilustrando a nossa perspectiva do usuário nos dias de hoje.

	Desenvolvimento	
	centrado no processo	**centrado no conteúdo**
Modelo da unidade de reuso (noção de componente)	componente de software	objeto complexo
Estratégia para composição (noção de composição)	baseada em interfaces (independente de domínio)	dependente de domínio

Tabela 1.1: Comparação das noções de componente e composição nas correntes de desenvolvimento centrada no processo e no conteúdo.

- deve ser acessível a autores não-especialistas em Informática; e
- deve ser calcado na lógica do reuso.

1.2.2 Modelo de Compartilhamento/Reuso de Conteúdo Digital

Um dos elementos fundamentais de uma estratégia de produção pautada no reuso é um modelo que permita decompor o conteúdo a ser reutilizado em unidades independentes, apropriadas para distribuição e reuso. Estas unidades são usadas posteriormente para compor novos produtos. Na tarefa de *compor*, o *componente* é a parte constituinte e a *composição* é o resultado. Portanto, as noções de *componente* e *composição* sempre estão presentes em estratégias de produção que visam o reuso. Não obstante, estas noções deram origem a modelos diferentes nas correntes de desenvolvimento centrado no processo e centrado no conteúdo, como está sintetizado na Tabela 1.1.

No desenvolvimento centrado no processo predomina o modelo de componente de software da Engenharia de Software [6]. Os componentes de software têm sido estudados e aplicados há muito tempo na Engenharia de Software, atingindo por este motivo um alto nível de maturidade. Ainda que existam muitas divergências no que exatamente define um componente de software, há características básicas consensuais [10]. Entre elas, uma característica fundamental é que os componentes encapsulam módulos de software, es-

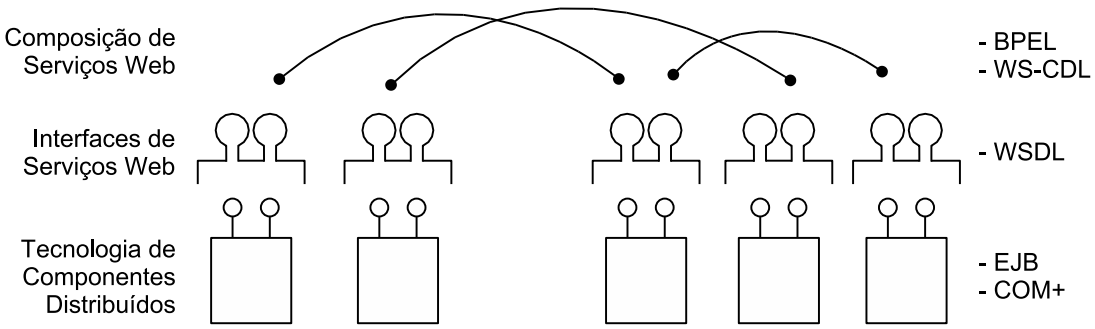

Figura 1.2: Diagrama ilustrando os componentes de software conectados usando a tecnologia de serviços Web.

condendo detalhes de implementação e publicando sua funcionalidade através de uma interface. A composição de componentes de software é baseada em suas interfaces. A separação entre interface e implementação resultou em um mecanismo genérico para explicitamente expressar como um componente pode ser conectado a outros componentes, independente dos detalhes de implementação. Por estas razões, como veremos adiante, o modelo de componente de software foi adotado como base para o nosso modelo de componente (DCC).

O advento da Internet consolidou a tecnologia dos componentes de software distribuídos e impulsionou avanços na interoperabilidade. No entanto, do ponto de vista da implementação, existem padrões diferentes e incompatíveis para a codificação de componentes, dependendo de fatores como linguagem de programação e plataforma do sistema. Isto se reflete em protocolos de comunicação incompatíveis entre os componentes. A aliança entre os componentes e os serviços Web representa um avanço promissor em direção à interoperabilidade e tem sido adotada pelos principais padrões para componentes distribuídos [22,52,75]. Os serviços Web compartilham dois aspectos com os componentes de software: a divisão clara entre implementação e interface, e a habilidade de compor unidades menores em unidades mais complexas. Como mostra a Figura 1.2, os serviços Web fornecem um mecanismo padrão para que os componentes distribuídos se intercomuniquem. As interfaces dos componentes e seus protocolos de comunicação são mapeados em interfaces e protocolos de comunicação dos serviços. Adicionalmente, como ilustrado na Figura 1.2, existem padrões abertos cujo propósito é a conexão de serviços Web para a produção de aplicações, tais como o padrão WS-Choreography [38] e a linguagem BPEL (Business Process Execution Language) [4].

Já no desenvolvimento centrado no conteúdo não existe um modelo amplamente aceito para as noções de componente e composição. Entretanto, diversas iniciativas relacionadas a domínios específicos de aplicação têm proposto padrões para o que denominaremos a partir de agora *objeto digital complexo* (ou simplesmente *objeto digital*), em que a ênfase

é na construção de conteúdo e não na execução de código. Exemplos são trabalhos em bibliotecas digitais [15,77], multimídia [35] e artefatos relacionados ao desenvolvimento de software [53]. Uma aplicação imediata desta pesquisa tem sido a Educação [2, 33, 74]. O termo "objeto digital complexo" tem origem no domínio de bibliotecas digitais [7] e será utilizado na ausência de uma terminologia comum para todos os domínios. Um objeto complexo agrega e encapsula artefatos digitais e representa algumas das relações existentes entre os artefatos encapsulados. Os padrões de representação de objetos complexos normalmente embutem na estrutura dos objetos um documento denominado *manifesto*, cuja função é declarar quais os artefatos encapsulados e as relações existentes entre eles. O formato do manifesto é especializado no domínio da aplicação que define aquele padrão para objetos complexos. Ao contrário dos componentes de software, não existe a noção de interface pública. Por este motivo, as estratégias para a composição destes objetos precisam levar em consideração detalhes da sua representação interna, sendo altamente dependentes do domínio da aplicação e dificilmente generalizáveis.

Algumas vezes temos a intenção de reusar um artefato pronto e modificá-lo/adaptá-lo. Outras vezes, no entanto, estamos interessados em reusar o *design* de uma composição, tal como um padrão de projeto (design pattern) [29] ou uma arquitetura de software. Krueger [40] analisa estas duas faces do reuso considerando que qualquer abstração de um software pode ser organizada em dois níveis: especificação da abstração e realização da abstração. Por um lado, através dos componentes, reusamos blocos de conteúdo prontos para uso; neste caso, estamos interessados na "realização da abstração". Por outro lado, a especificação da composição, independente dos componentes que ela agrega, pode ser reusada para capturar o projeto da aplicação e, neste caso, estamos interessados na "especificação da abstração". A Engenharia de Software tem pesquisado e utilizado diversas técnicas para reuso de *design*, entre elas: padrões de projeto e linguagens para descrição de arquiteturas (architecture description languages – ADLs). Já na corrente de desenvolvimento centrado no conteúdo, apesar de existirem técnicas para reuso de *design* – tal como o uso de templates de documentos – elas geralmente são dissociadas do modelo de objetos digitais complexos.

Como pode ser observado, existe uma divisão de esforços para se resolver problemas equivalentes, tanto dentro da corrente de desenvolvimento centrado no conteúdo, quanto naquela de desenvolvimento centrado no processo. A contribuição central da tese, o modelo de Digital Content Component (DCC), ou Componente de Conteúdo Digital, é um modelo genérico para o compartilhamento e reuso de conteúdo digital, aplicável a qualquer corrente e domínio. Neste modelo, a noção de interface utilizada nos componentes de software é aplicada com sucesso também no desenvolvimento centrado no conteúdo. Deste modo, a estratégia adotada para reuso e composição de conteúdo está baseada em um formato padrão de declaração de interface, que não depende de detalhes internos do

conteúdo, que são fortemente dependentes do domínio da aplicação. Além disto, o desenvolvimento de software atual pode ser visto como uma equação que envolve software executável e conteúdo, onde cada fator assume um grau de importância a depender do contexto. Por integrar modelos de produção, compartilhamento e reuso centrados no processo e no conteúdo, o modelo de DCCs dá ao autor a liberdade de decidir qual a sua ênfase no desenvolvimento.

1.2.3 Suporte ao Compartilhamento/Reuso de Conteúdo Digital

Os modelos distintos de compartilhamento e reuso adotados pelas correntes de desenvolvimento centrado no processo e centrado no conteúdo são refletidos em diferentes concepções na infraestrutura que dá suporte às atividades relacionadas com produção, gerenciamento e consumo de artefatos digitais. Estão incluídas nesta infraestrutura ferramentas para: gerenciamento de repositórios de artefatos, busca de artefatos, controle e gerenciamento de configurações e versões e distribuição de artefatos.

O gerenciamento de repositórios de objetos complexos é freqüentemente pesquisado no contexto de bibliotecas digitais, destacando-se uma proposta aberta da OAIS – Open Archival Information System [15]. A indexação e busca de objetos complexos podem ser baseadas nos metadados que descrevem estes objetos. As iniciativas nesta área têm se caracterizado por estruturas descritivas de metadados bastante detalhadas e baseadas em padrões abertos. A noção de repositórios de componentes de software explora sua interface para buscas baseadas na funcionalidade que ela descreve [55, 58, 84]. Em ambos os contextos – busca baseada em metadados e baseada em interfaces – ontologias estão sendo exploradas para definir a similaridade entre termos [51, 60, 62, 80].

Objetos digitais e software estão sujeitos ao controle de versões e à criação de configurações, tratadas como "contextos de uso". Versões e configurações são geralmente representadas como ortogonais, porém seu controle e gerenciamento são interrelacionados. No caso de objetos digitais, o gerenciamento de configurações lida com a complexa rede de relacionamentos entre estes objetos, enquanto o controle de versões lida com mudanças semanticamente significativas que acontecem ao longo do tempo [37]. Dada a sua interdependência, as pesquisas relacionadas a controle de versões geralmente estão associadas ao gerenciamento de configurações.

A área de projetos em Engenharia tem longa experiência no gerenciamento de configurações de objetos de projeto complexos [37]. Pesquisas neste domínio são geralmente denominadas *Product Data Management* (PDM) [26] ou *Engineering Data Management* (EDM) [83]. Já no domínio de desenvolvimento de software existem pesquisas em *Software Configuration Management* (SCM) [25].

Qualquer infraestrutura de reuso deve considerar o empacotamento e distribuição de artefatos digitais, uma área que tem crescido recentemente como conseqüência da Web. Por um lado estão as iniciativas para distribuição de objetos complexos, cujos padrões variam de acordo com a ênfase: educação [74], multimídia [13], desenvolvimento de software [54], bibliotecas digitais [16]. Por outro lado estão os padrões para distribuição de componentes de software distribuídos [28, 52, 75]. Tanto as iniciativas de objetos complexos quanto a de componentes de software utilizam um formato de pacote com a mesma estrutura: um *contêiner* utilizando o formato de empacotamento e compactação ZIP e um arquivo de *manifesto* em XML que mantém dados mais detalhados referentes aos artefatos empacotados e as relações existentes entre eles. Os formatos de empacotamento são confrontados em mais detalhes no Capítulo 4.

1.3 Objetivos, Contribuições e Ligação com outras Pesquisas

A seção anterior caracterizou o cenário atual de produção, distribuição e consumo de conteúdo digital sob três enfoques: papel do autor, modelo de compartilhamento/reuso de conteúdo e suporte ao compartilhamento/reuso de conteúdo. A apresentação da tese também está organizada sob estes três enfoques, conforme sintetizado na Figura 1.3.

A parte inferior da figura ilustra nosso *modelo de compartilhamento/reuso de conteúdo digital*, o Digital Content Component (DCC), bem como o modelo para a composição de DCCs. A parte superior da figura ilustra nossa perspectiva do *papel do autor* no desenvolvimento baseado em DCCs. O centro representa a Fluid Web como um panorama que envolve a produção colaborativa e consumo de conteúdo digital na Web através dos DCCs. Como ilustra a parte central da figura, nosso projeto de Fluid Web está montado sobre uma infraestrutura para *suporte ao compartilhamento/reuso de conteúdo digital*.

As subseções a seguir sintetizam os principais objetivos, características e contribuições deste trabalho. Cada uma das subseções aborda um dos enfoques do trabalho, utilizando a Figura 1.3 como pano de fundo. A seção 1.3.4 salienta algumas das pesquisas correlatas que fundamentaram nosso trabalho; estas são vistas com mais detalhes em cada capítulo.

1.3.1 Digital Content Component

Esta seção apresenta o modelo da nossa unidade de compartilhamento e reuso de conteúdo digital, o Digital Content Component (DCC), que é o fundamento para todo o restante do trabalho. O modelo do DCC parte do princípio de que os modelos de componentes de software e de objetos digitais complexos são perspectivas complementares para um problema mais amplo de reuso.

Um DCC é uma unidade de decomposição, compartilhamento, reuso e composição de conteúdo digital [66]. Este conteúdo pode ser um código executável ou um objeto digital complexo. Tal como os objetos digitais complexos, um DCC agrega e encapsula um ou mais artefatos digitais e representa internamente as suas relações. Tal como os componentes de software, um DCC esconde detalhes internos da sua representação e expõe uma interface pública. No entanto, como veremos adiante, o modelo de DCC vai muito além de uma simples combinação destas duas vertentes.

O DCC é definido a partir de um modelo abstrato, que posteriormente é implementado em diferentes formatos, conforme o que se deseje fazer com o conteúdo digital correspondente: armazenamento, distribuição e execução. O modelo abstrato de um DCC [66] é composto de quatro subdivisões distintas:

(a) conteúdo digital encapsulado;

(b) declaração de uma estrutura de gerenciamento que define como as partes dentro de um DCC estão relacionados entre si;

(c) especificação das interfaces;

(d) metadados para descrever versão, funcionalidade, aplicabilidade, restrições de uso, etc.

Existem dois tipos de DCC: os de processo e os passivos. Os DCCs de processo encapsulam software executável, que pode ser código binário, mas também pode ser codificado em representações mais apropriadas para autores não especialistas em Informática, tal como um "workflow". Os componentes passivos encapsulam qualquer conteúdo digital não executável e não contêm o software executável responsável por manipular este conteúdo.

Um DCC em seu formato de distribuição é utilizado no intercâmbio entre os autores e usuários da "Fluid Web". Por isso, ele é projetado para interoperabilidade tanto do ponto de vista sintático quanto semântico, adotando padrões de Web Semântica nas subdivisões (b), (c) e (d). A subdivisão (b) é em XML, a subdivisão (c) usa versões adaptadas de WSDL [19] e OWL-S [42] (uma ontologia OWL para serviços Web), e a subdivisão (d) usa OWL [73]. Os formatos de armazenamento e execução diferem do formato de distribuição pois são otimizados para seus respectivos contextos. Além disso, não há imposição de formato para armazenamento e execução, já que eles não impactam a interoperabilidade.

A parte inferior esquerda da Figura 1.3 ilustra parcialmente um DCC passivo em seu formato de distribuição. Neste exemplo, o DCC está encapsulando uma série temporal de imagens contendo um ano de dados de pluviosidade no estado de São Paulo. Cada imagem contém um mapa e representa a média de pluviosidade em um mês. O valor de cada pixel no mapa está relacionado ao volume de pluviosidade na respectiva região. Internamente, o DCC é organizado como um objeto complexo que além de agregar e encapsular as doze imagens, define uma `Estrutura` XML responsável por identificar, organizar e registrar as

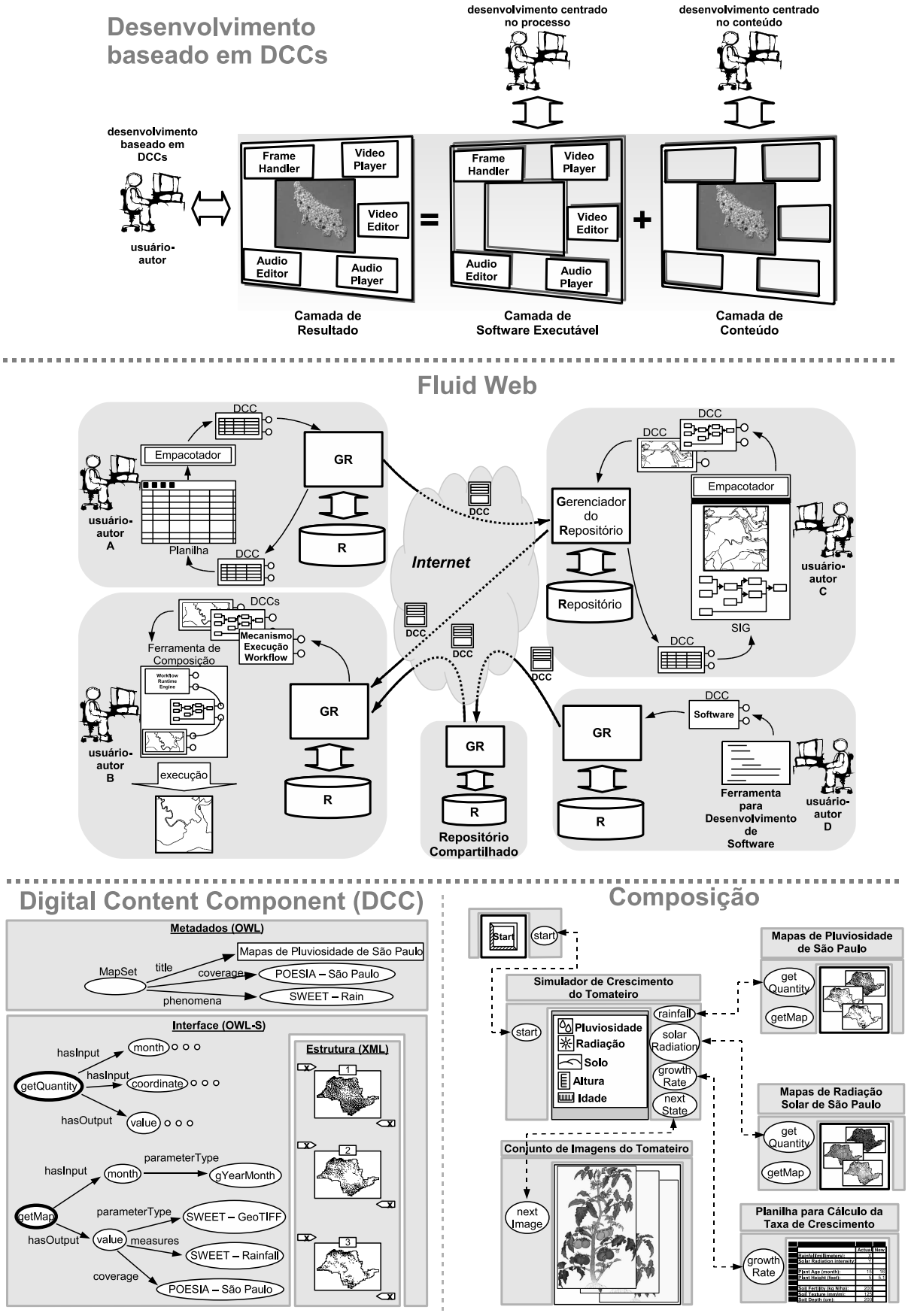

Figura 1.3: Diagrama ilustrando o processo de produção/consumo de DCCs na Fluid Web.

relações entre as imagens.

No topo do DCC está representada a subdivisão de metadados (em OWL) e em torno da estrutura de organização está a interface (OWL-S). Ambas são apresentadas utilizando-se uma versão simplificada de um grafo rotulado e direcionado (Directed Labelled Graph – DLG), em RDF [41].

Os DCCs passivos declaram uma interface com as operações que podem potencialmente ser aplicadas ao seu conteúdo (*funcionalidade potencial*). Por exemplo, o DCC apresentado na Figura 1.3 declara uma interface com duas operações que podem potencialmente ser aplicadas a seu conteúdo, `getQuantity` e `getMap`: `getQuantity` retorna a pluviosidade de uma região, dadas as coordenadas e o mês; `getMap` retorna um mapa, dado um mês. Cada DCC é classificado dentro de uma ontologia de tipos de DCC criada como parte desta pesquisa. Esta ontologia é utilizada para relacionar cada tipo de DCC passivo com um segundo DCC – denominado DCC companheiro (*companion DCC*) – capaz de manipular o conteúdo daquele tipo. O DCC companheiro implementa a interface declarada no DCC passivo, transformando em tempo de execução sua funcionalidade potencial em funcionalidade real. Retornando ao exemplo da figura, o DCC está classificado como sendo do tipo `MapSet` (vide subdivisão de metadados). A ontologia de DCCs indica um DCC companheiro para `MapSet`, que implementa as operações `getQuantity` e `getMap`. Em tempo de execução, o DCC passivo e seu companheiro são fundidos em um único DCC que contém o conteúdo e o software para manipulá-lo. A esta estratégia damos o nome de *execução dirigida pelo tipo de conteúdo*. O mesmo DCC passivo pode ser associado a diferentes DCCs companheiros, de acordo com o contexto.

O DCC é uma unidade genérica de decomposição e também de composição. Como está ilustrado no canto direito inferior da Figura 1.3, os DCCs podem ser interligados através de suas interfaces. O exemplo apresenta uma aplicação que simula o crescimento de um tomateiro em uma região específica do estado de São Paulo. Esta aplicação combina DCCs passivos e de processo. O DCC ilustrado com um botão e o DCC "`Simulador de Crescimento de um Tomateiro`" são DCCs de processo que encapsulam classes Java em formato bytecode, responsáveis pela implementação de seu comportamento. Este último é o núcleo da simulação. O DCC "`Mapas de Pluviosidade de São Paulo`" corresponde ao DCC apresentado em detalhes na esquerda da Figura 1.3, que foi explicado anteriormente. O DCC "`Mapas de Radiação Solar de São Paulo`" também tem a mesma estrutura do anterior, mas contém mapas de radiação solar ao invés de pluviosidade. O DCC "`Conjunto de Imagens do Tomateiro`" é um DCC passivo que encapsula imagens consecutivas do crescimento do tomateiro. Finalmente o DCC "`Planilha para Cálculo da Taxa de Crescimento`" contém uma planilha com as fórmulas que calculam o crescimento da planta.

Esta é uma modalidade de composição de DCCs em que eles trabalham cooperativa-

mente trocando mensagens. Ao ser executada, a aplicação inicia quando o usuário clica no DCC botão, que envia uma mensagem de início para o DCC simulador. Este DCC divide a simulação em ciclos mensais onde, para cada mês, solicita aos DCCs que contém mapas a pluviosidade e a radiação solar daquele mês, em uma região de São Paulo. O DCC simulador envia estes dados para o DCC de planilha e requisita que ele calcule o novo tamanho do tomateiro. Finalmente, o DCC simulador usa estes dados para requisitar ao DCC com imagens do tomateiro que apresente o tomateiro em seu novo tamanho. Esta composição pode, por sua vez, ser encapsulada em outro DCC. Detalhes do funcionamento desta simulação são apresentados nos Capítulos 2 e 4.

A representação de composições também pode ser explorada no sentido de se realizar o reuso do design, conforme abordado na Seção 1.2.2. Está sendo desenvolvido um trabalho neste sentido, conforme descrito em [44] e detalhado na Seção 5.3.

Em resumo, em banco de dados, objetos complexos podem ser construídos pela composição de outros objetos; no desenvolvimento de software, um produto de software pode ser construído pelo composição de outros componentes de software. Composição de dados em bancos de dados e combinação de componentes na Engenharia de Software são mecanismos distintos que foram extensivamente pesquisados. Nosso processo de composição baseado em DCCs, no entanto, é uma nova noção: usando um único mecanismo, ele permite construir objetos complexos (no sentido de bancos de dados), construir software (como em Engenharia de Software) e associar software a dados para construir artefatos mais complexos via princípio do DCC companheiro. Este mecanismo depende somente do casamento de interfaces e termos de ontologias, não precisando se preocupar com a natureza do conteúdo encapsulado.

1.3.2 Bloco de produção centrado no usuário-autor

A parte superior da Figura 1.3 contrasta a abordagem de desenvolvimento baseada em DCCs com o desenvolvimento centrado no processo e centrado no conteúdo. A Figura ilustra o desenvolvimento e execução de uma aplicação organizada em camadas. A `camada de conteúdo` engloba os artefatos de conteúdo usados na aplicação; a `camada de software` executável engloba todas as rotinas de software necessárias para manipular este conteúdo; e a `camada de resultado` corresponde à combinação em memória das camadas de software e conteúdo, sendo necessária à execução da aplicação. Como ilustrado, na abordagem de desenvolvimento centrada no conteúdo o autor atua na `camada de conteúdo`; no desenvolvimento centrado no processo, por outro lado, o autor atua na `camada de software executável`. Em qualquer dos casos, o usuário interage com a camada de resultados.

Como apresentado na seção anterior, os DCCs dispõem de um mecanismo que permite combinar o conteúdo com o software que o processa de modo transparente ao usuário-

autor. Deste modo, o autor trabalha tanto com conteúdo quanto com software executável na perspectiva da `camada de resultado`.

Além da vantagem do ponto de vista da autoria, o modelo de DCCs tem duas vantagens adicionais para o usuário-autor. Primeiro, ao contrário dos componentes de software que usualmente são codificados em uma linguagem de programação, os DCCs permitem o encapsulamento de outros tipos de especificação de software executável, cuja abordagem seja mais apropriada para autores não especialistas (por exemplo, uma especificação de workflow). Segundo, ao lidar com o modelo unificado dos DCCs o autor não precisa se preocupar com a natureza do que ele está compartilhando ou reusando, pois os detalhes são resolvidos por detrás dos bastidores automaticamente e de forma transparente para o usuário-autor.

1.3.3 Produção e consumo de DCCs

Esta seção apresenta a infraestrutura concebida para dar suporte à produção, gerenciamento e uso de DCCs. Esta infraestrutura integra contribuições do desenvolvimento centrado no processo e centrado no conteúdo para: distribuição, armazenamento e indexação, suporte a busca, controle de versões e gerenciamento de configurações. A infraestrutura não se resume a uma integração, pois articula as diversas tecnologias de produção, gerenciamento e distribuição a fim de alcançar a perspectiva da Fluid Web.

A parte central da Figura 1.3 ilustra um cenário típico de produção e consumo de conteúdo digital dentro da perspectiva de Fluid Web, utilizando DCCs. Ao contrário do modelo clássico de Web onde o documento é a peça chave, na Fluid Web o DCC é a peça chave que transita pela Internet e pode ser decomposto, armazenado, anotado em múltiplos níveis, distribuído, reutilizado e composto.

Como ilustra a Figura 1.3, a infraestrutura que dá suporte à Fluid Web se baseia no princípio de que cada nó da rede possui um repositório local de DCCs (`R`) e um gerenciador associado (`GR`). Isto permite um suporte mais efetivo à indexação dos DCCs para busca e controle de versões e gerenciamento de configurações. O gerenciador de repositório não apenas dá suporte às atividades de armazenamento e busca de DCCs, como também é responsável em converter os DCCs do formato de armazenamento para o de distribuição e vice-versa.

Ainda seguindo o diagrama da figura, pode-se observar que novos DCCs são produzidos por ferramentas de desenvolvimento de software (tipicamente DCCs contendo software executável) – usuário-autor D – ou a partir de ferramentas que geram algum tipo de conteúdo, tal como uma planilha eletrônica – usuário-autor A. Neste segundo caso, um módulo chamado empacotador é responsável por encapsular o conteúdo dentro do DCC e enviá-lo para o gerente de repositório local para armazenamento. Cada usuário-autor faz

consultas ao gerente de repositório local para buscar DCCs que irá usar. O gerente local pode se comunicar com outros gerentes distribuídos na busca de DCCs e, se necessário, requisitá-los. A transferência de DCCs entre um repositório e outro é feito usando-se o formato de distribuição. Muitos dados acompanham o DCC neste formato, tais como, identificação única, origem e versão. DCCs recuperados do repositório podem ser usados para a construção de composições, por exemplo, no site do usuário-autor B. O resultado final é executado e pode ser encapsulado dentro de um DCC de nível mais alto, que volta a ser armazenado no repositório.

O controle de versões estende o modelo *banco de dados multiversão* proposto por Cellary e Jomier [17]. A extensão atende duas importantes demandas da Fluid Web: (i) permite o versionamento e replicação através da Web; e (ii) permite a definição de configurações que relacionam DCCs em diferentes localizações da Web.

Como observado em [25], uma das fraquezas dos sistemas de SCM (*Software Configuration Management*) está no fato de que eles têm muito pouco conhecimento do produto de software que eles gerenciam. Os DCCs são um avanço neste sentido, pois não apenas representam um conjunto mais significativo de dados relacionados com o conteúdo encapsulado, podendo ser usados para unificar as abordagens de Software Configuration Management (SCM) e Product Data Management (PDM) / Engineering Data Management (EDM).

1.3.4 Alguns Trabalhos Correlatos

A tese combina diferentes vertentes e resultados de pesquisa nas áreas de componentes de software, objetos digitais complexos, padrões para da Web Semântica e serviços Web. Esta combinação unifica algumas correntes, ao identificar tratamentos complementares para um mesmo problema.

O ponto de partida deste trabalho foi a necessidade de empacotamento, distribuição e reuso de conteúdo. Muitos projetos recentes tratam estas questões seguindo caminhos paralelos para resolver problemas análogos em domínios distintos, como por exemplo: IMS Content Packaging (IMS CP) [74], em Educação, MPEG-21 [13], em multimídia, Reusable Asset Specification (RAS) [54], para desenvolvimento de software, ou OAIS XML Formated Data Unit (XFDU) [16], em bibliotecas digitais. Além disso, tecnologias de componentes de software envolvem questões semelhantes, por exemplo, relacionadas com empacotamento e distribuição de componentes [28] – e.g., Enterprise Java Beans (EJB) [75], Microsoft COM+ e CORBA Component Model (CCM) [52]. Todos esses projetos e padrões definem seus próprios formatos de pacote, para permitir a distribuição do seu conteúdo; tais padrões seguem a mesma estrutura básica, que é dividida em duas partes: (1) um contêiner de pacotes, usualmente baseado no formato ZIP; (2) um ar-

quivo de manifesto em XML, armazenado dentro do pacote. O manifesto complementa a informação fornecida pelo item (1) – e.g., indicando como o conteúdo empacotado está organizado, suas dependências ou seus metadados.

DCCs aproveitam em sua estrutura, dentre outros (ver Seção 4.3.2), a estrutura dos arquivos de manifesto de padrões como RAS, MPEG-21, METS e IMS CP. Ainda que tais estruturas de manifesto sejam diferentes, elas lidam com o mesmo problema e os mesmos conceitos básicos. Desta forma, é possível encontrar mapeamentos entre os manifestos, desde que se descarte as peculiaridades de cada tipo de conteúdo. Tal mapeamento fez uso dos princípios de Model Driven Architecture (MDA) [45].

Além disso, em contraste com outros padrões de empacotamento de conteúdo da literatura correlata, o modelo dos DCCs dá um passo além da noção de pacote, utilizando em seu lugar o conceito de componente. Neste sentido, nosso trabalho se baseia na pesquisa em componentes de software da Engenharia de Software [32], estendendo tal pesquisa para tratar qualquer tipo de conteúdo. Tal extensão não se resume em propor um novo formato de pacote; trata-se, na verdade, de um enfoque inovador para produzir aplicações ou conteúdo, sendo uma das principais contribuições da nossa pesquisa.

Ainda outras iniciativas que contribuíram para o desenvolvimento deste trabalho incluem o modelo de versões de Cellary e Jomier [17], pesquisas em arcabouços para bibliotecas digitais [15] e algoritmos de descoberta de conteúdo digital baseados em ontologias e casamento de interfaces citeZaremski1997.

Os capítulos subseqüentes deste texto contém mais detalhes sobre esses e outros trabalhos correlatos.

1.3.5 Contribuições

Resumindo, as principais contribuições da tese são:

1. O modelo de Componente de Conteúdo Digital – Digital Content Component (DCC) – capaz de encapsular software executável e outros tipos de conteúdo de modo uniforme, provendo um modo original para compô-los, eliminando do ponto de vista de gerenciamento e composição a distinção entre software executável e outros tipos de conteúdo. O modelo de DCC está aliado à estratégia de "execução dirigida pelo tipo de conteúdo", baseada no uso de ontologias, onde estão inclusas as noções de *funcionalidade potencial* e *Companion Component*.

2. Um procedimento de três passos que explora metadados associados a DCCs, combinado com a funcionalidade expressa em suas interfaces, para aprimorar o processo de busca dos DCCs, facilitando a tarefa do usuário-autor.

3. A introdução da noção de *Fluid Web* e a proposta de uma infraestrutura para dar suporte à sua efetiva materialização. A infraestrutura é baseada em repositórios locais interligados, com controle de configurações e versões, levando em conta o compartilhamento distribuído dos DCCs. O controle de versões é facilitado e enriquecido semanticamente pela adoção de uma ontologia taxonômica de relacionamentos entre versões de componentes, desenvolvida neste trabalho.

4. A validação prática de grande parte desses conceitos por meio da construção de protótipos.

1.4 Organização da Tese

Este texto reúne os principais artigos resultantes da pesquisa realizada. Cada artigo, publicado ou submetido para publicação, é apresentado sob a forma de um capítulo, com pequenas correções e adaptações no texto original (notações, erros de ortografia, etc.) a fim de tornar o texto resultante homogêneo e consistente. O Capítulo 2 define DCCs e enfatiza questões de sua especificação, reuso, composição e descoberta semi-automática. O Capítulo 3 está dirigido ao usuário-autor de artefatos multimídia e as particularidades de uso e autoria colaborativa proporcionados pela infraestrutura DCC. O Capítulo 4 introduz a Fluid Web e descreve a infraestrutura que a implementa a partir de DCCs, descrevendo, dentre outras, questões de compartilhamento e versionamento.

Em mais detalhes, os capítulos estão relacionados aos 3 aspectos de pesquisa da Seção 1.2. O Capítulo 2 se concentra no aspecto 2 (modelo de compartilhamento/reuso de conteúdo digital), apresentando o modelo de *Digital Content Component*, sua estrutura e princípios, detalhando como padrões da Web semântica e ontologias são usados na descrição de metadados e interface dos DCCs, e como isto é explorado na busca de DCCs. O Capítulo 3 explora o usuário-autor (aspecto 1), aprofundando as noções de *funcionalidade potencial* e *execução dirigida pelo tipo de conteúdo*, explicando como elas são exploradas pelos DCCs para criar um ambiente de produção/execução propício ao usuário-autor. O Capítulo 4 detalha o aspecto 3 (suporte ao compartilhamento/reuso de conteúdo digital), apresentando a noção de *Fluid Web*, bem como a infraestrutura projetada para dar suporte a ela baseada em DCCs, com recursos de armazenamento, controle de configurações e versões. O Capítulo 5 conclui a tese.

1.4.1 Capítulo 2

O Capítulo 2 (*Self Describing Components: Searching for Digital Artifacts on the Web*) foi publicado no XX Simpósio Brasileiro de Banco de Dados – SBBD 2005 [68].

O principal enfoque deste capítulo é o uso de DCCs para apoio ao reuso de conteúdo. Esta noção considera o fenômeno de usuários cada vez mais atuantes na produção e modificação de conteúdo digital, inseridos em uma rede onde o conteúdo digital disponível e circulante é sempre crescente.

O modelo de *Digital Content Component*, descrito neste capítulo, é apresentado como uma solução que permite explorar a diversidade dos conteúdos disponíveis na rede em tarefas de reuso, mantendo interoperabilidade entre diferentes sistemas. Este capítulo introduz o modelo dos DCCs e seu uso para decomposição, armazenamento, distribuição e composição de conteúdo digital. Discute soluções adotadas pelo modelo para permitir a composição e execução de DCCs independente do tipo de conteúdo, executável ou não.

Tomando a Internet sob a perspectiva de uma grande repositório mundial de conteúdo digital, o modelo de DCCs é apenas parte do desafio para o efetivo intercâmbio e reuso de conteúdo digital. Tão importante quanto viabilizar o intercâmbio é criar condições adequadas para que o usuário encontre o conteúdo desejado. O capítulo apresenta como a estrutura "auto-descritiva" dos DCCs, montada sobre padrões da Web Semântica, é usada para facilitar seu processo de descoberta.

A busca pode ser feita a partir da especificação em uma consulta do DCC desejado, ou pela busca de um DCC Y que possa ser conectado a um DCC X. O capítulo descreve estratégias para a busca de DCCs ordenados de acordo com diferentes graus de similaridade em relação ao DCC desejado, por meio do uso de ontologias, associadas às descrições dos DCCs. Estratégias de casamento de interfaces também são usadas, não apenas na busca de DCCs, como também para verificar o grau de compatibilidade entre dois DCCs.

Este capítulo é ilustrado por um exemplo prático em planejamento agro-ambiental, que começa pela descrição dos DCCs e vai até sua composição e algoritmos de busca. Todos os aspectos discutidos neste exemplo foram validados a partir da implementação de um protótipo.

1.4.2 Capítulo 3

O Capítulo 3 (*User-author centered multimedia building blocks*) foi aceito para publicação no Multimedia Systems Journal [70].

O capítulo está centrado na adoção de DCCs para o gerenciamento e construção de conteúdo e produção multimídia e discute a noção de usuário-autor em um ambiente de consumo e produção de artefatos multimídia.

A pesquisa em multimídia é um dos domínios que possuem maior tradição em lidar com o problema da diversidade de conteúdo digital e heterogeneidade de representação. Ao lidar com diversos tipos de mídia, a multimídia é essencialmente diversa e heterogênea; porém, ao integrar estas mídias surge o desafio de como relacioná-las de forma homogênea.

Este é portanto um cenário ideal onde se insere o modelo de DCCs.

Além do desafio da heterogeneidade, a área vem direcionando esforços no sentido de criar modelos e ferramentas que popularizem o acesso à produção de conteúdo multimídia. Como resultado, esta produção se tornou acessível ao usuário final, seja através de ferramentas especializadas, seja inserida em outras ferramentas de produção. Com isto se dilui a linha divisória entre usuário e autor, dando forma ao usuário-autor.

Como mencionado na Seção 1.3.2, os DCCs combinam facilidade de uso e flexibilidade de compartilhamento/reuso. Por este motivo, este capítulo apresenta o modelo de DCCs como apropriado para o perfil de usuário-autor. O capítulo mostra como o modelo de DCCs foi implementado dentro de uma versão adaptada de um sistema de autoria multimídia voltado para aplicações educacionais – o sistema Casa Mágica [71]. Através de exemplos práticos na área educacional, implementados com DCCs no sistema Casa Mágica, o capítulo ilustra como as características dos DCCs são exploradas para aproximar o processo de produção, compartilhamento e reuso do usuário final. O sistema Casa Mágica explora o modelo de DCCs para possibilitar a produção de aplicações centradas no conteúdo, centradas no processo ou combinando ambas.

Este capítulo também analisa a noção de execução dirigida pelo tipo de conteúdo em diversos contextos, como fundamento para a elaboração do modelo utilizado pelos DCCs.

1.4.3 Capítulo 4

O Capítulo 4 (*A Component Model and Infrastructure for a Fluid Web*) foi submetido ao IEEE Transactions on Knowledge and Data Engineering [69] havendo obtido parecer de "aceitação com mudanças pequenas".

Este capítulo insere os DCCs dentro do contexto da *Fluid Web*. O modelo tradicional de publicação/consumo "orientado a documentos" da Web é confrontado com o modelo "orientado a conteúdo" dos DCCs, que fornecem as bases para a criação de um ambiente de colaboração – a Fluid Web – onde o conteúdo digital pode viajar e ser replicado, adaptado, decomposto, fundido e transformado.

A partir do modelo de DCCs, o capítulo parte para uma visão mais ampla que abrange toda a infraestrutura projetada para dar suporte à nossa perspectiva de Fluid Web. Isto envolve o detalhamento da malha de repositórios interligados apresentada na Figura 1.3, que requer uma distinção entre o formato de armazenamento e de distribuição dos DCCs. Além disto, o capítulo trata de duas outras questões de grande relevância do ponto de vista do trabalho colaborativo: o controle de configurações e de versões de DCCs.

O controle de configurações tem soluções distintas nas diversas iniciativas que lidam com conteúdo digital. Tais diferenças são causadas, dentre outros, pelas especificidades de cada domínio na composição de conteúdo. Uma contribuição apresentada neste

capítulo é a separação entre aspectos do controle de configuração, que estão relacionados ao gerenciamento dos componentes (estrutura organizacional), e aqueles tratados por domínios/aplicações específicos (estrutura representacional). A estrutura organizacional pode ser generalizada por representar dados comuns a todos os domínios/aplicações.

Outra contribuição está relacionada ao aspecto distribuído da Fluid Web. Considerando que DCCs e composições de DCCs são compartilhadas na Web, torna-se necessário também compartilhar o controle de configurações e versões. O capítulo mostra como isto é feito através do uso de princípios da Web Semântica, que permitem criar ponteiros entre objetos distribuídos na Web e compartilhar a semântica associada às relações definidas entre DCCs.

O capítulo apresenta um exemplo prático na área de planejamento agro-ambiental, que é uma extensão do exemplo apresentado no Capítulo 2.

1.4.4 Outras Contribuições

A tese está organizada como uma coletânea de publicações, havendo sido escolhidos os artigos mais representativos da pesquisa desenvolvida. Outras publicações durante o doutorado foram:

- *Aplicações educacionais na Web – o papel de RDF e Metadados* [64]: este minicurso foi resultado da revisão da literatura envolvendo objetos digitais complexos no contexto da educação e padrões de metadados a eles associados. Adicionalmente, é apresentada uma crítica aos modelos de representação de metadados baseados exclusivamente em XML, e são apresentadas investigações do uso do padrão RDF (*Resource Description Framwork* – padrão usado na Web Semântica) para adicionar interoperabilidade semântica à representação dos metadados.

- *Managing Dynamic Repositories for Digital Content Components* [66]: este artigo introduz as primeiras noções dos DCCs, sua estrutura e gerenciamento em repositórios e a estratégia para composição de DCCs.

- *Managing Repositories for Digital Content Components* [67]: este artigo consiste em uma evolução do artigo anterior [66], apresentado no IV Workshop de Teses e Dissertações em Banco de Dados do SBBD 2005.

- *Geographic Digital Content Components* [65]: este artigo apresenta uma aplicação dos DCCs no contexto de Sistemas de Informação Geográficas. Um dos desafios neste contexto é o compartilhamento de projetos de SIG, que interrelacionam artefatos digitais em diferentes formatos, que podem ser tratados por diferentes aplicativos. O artigo apresenta como estes projetos podem ser representados através

de composições de DCCs. A proposta é ilustrada através de um exemplo prático, que apresenta um projeto para calcular regiões que possuem solo adequado para plantação de café.

- *WOODSS and the Web: Annotating and Reusing Scientific Workflows* [44]: neste artigo o enfoque principal é o projeto (WOrkflOw-based spatial Decision Support System) [56]. O WOODSS é um ambiente extensível que permite a captura, especificação, reuso e anotação de workflows científicos. No artigo, é apresentado como está sendo conduzido um projeto de integração entre os DCCs e o WOODSS, no qual workflows WOODSS coordenam a execução de DCCs e, por outro lado, estes workflows podem ser armazenados e distribuídos dentro de DCCs.

Capítulo 2

Self Describing Components: Searching for Digital Artifacts on the Web

2.1 Introduction

The search for efficiency in software development has prompted intensive research in reuse and documentation practices. The same goals and practices have propagated to the area of content design and management. The Web has accelerated such initiatives: IT professionals need new kinds of tools and techniques to retrieve the appropriate digital artifacts from repositories all over the world. This presents challenges both in specification and description, as well as in good searching mechanisms.

As a result of these efforts, there is an increase in the interchange of reusable artifacts (content and software), assembled inside standard "containers" – the *packages* – and stored in package libraries [15]. We define a package as a structure that delimitates, organizes and describes one or more pieces of digital content suitable for reuse. The term *digital content* is used from now on to denote any content represented digitally – e.g., pieces of software but also texts, audio, video, and so forth.

However, while the size of package libraries grows, effective reuse depends on the ability to discover artifacts for given requirements. Klischewski [39] observed that there is a variety of resources, like fine-grained information elements, multimedia items, services, or user related objects, which are meaningful for users. Therefore, they are candidates for semantic markup using Semantic Web standards, providing a more semantic way to search and use them. This observation was made in a e-Government context, but can be extended to the general reuse context.

Semantic Web efforts have addressed two directions: a common syntax and semantic

to exchange data; and a common syntactic and semantic infrastructure to provide interoperability between processes. In the former direction, the initial approach was document oriented, through annotations using RDF and OWL and pointers based on URIs connected to Web documents. As pointed out in [39], there is a wider universe of "annotatable" artifacts on the Web, formed by "annotatable" sub-parts whose organization depends on the artifact's nature. The challenge is how to associate Semantic Web annotations to artifacts and their subparts, in spite of their heterogeneity.

A similar challenge is faced by the second direction to describe different kinds of process entities, meant to inter-operate. In this case, the heterogeneity of process entities is hidden behind a standard interface. Here, Semantic Web-based standards (WSDL and OWL-S) are used to describe process functionality and details of its working activities as a composition of sub-processes.

Another challenge in this scenario is to build new products that properly combine and reuse pieces, in spite of their diversity. Developers in this new scenario will not be just computer science experts, thus requiring new models and tools [48]. Moreover, reuse requires finding the adequate pieces of digital content, and therefore for new means of describing, storing and retrieving these pieces.

WSDL and OWL-S are meant to describe process entities, and thus indirectly associate a *provided functionality* with process entities (e.g., a video player software *plays* video). However, one can also envisage the description of the *potential functionality* associated with the nature of any other digital content (e.g., a video content can be *played*). These functional annotations support finding the appropriate artifacts on the Web.

This paper contributes towards this direction. We propose a unified model to build reusable digital artifacts. It can be used both for content design (content-centric approach) or for software development (process-centric approach). In our model, each piece to be reused is encapsulated inside a unit named *Digital Content Component* (DCC). Furthermore, our model expresses both the provided and potential functionalities via an interface associated to any digital artifact. This functionality-based description provides a richer semantic way to annotate any kind of digital content, hiding its heterogeneity behind a standard interface. Semantic Web standards are adopted in many aspects of DCC specification and Web service standards are adopted for the interface specification.

More specifically, the paper focuses on the technique used to specify DCC interfaces using OWL and OWL-S respectively. As will be seen, these semantically enriched specifications enhance the possibility of reusing content. Moreover, they help discovering DCCs that are suitable for a given product construction in two ways: the functionality description is used to refine the search procedure, and the descriptions associated with DCC operations are used to discover useful DCC subproducts "hidden" within a DCC. For instance, images may provide pixels (e.g., a pixel inside a map) or videos may provide

a set of frames (e.g., a commercial from a film), and so on. The issues discussed here are presented by means of a practical example.

The remainder of the text is organized as follows. Section 2.2 details DCCs and their specification based on OWL and OWL-S. Section 2.3 presents our three step procedure for DCCs discovery, based on their OWL metadata and interface specifications. Section 2.4 presents how DCCs can be connected and assembled to form an application and the role played by the interface specification. Section 2.5 considers related work, showing how DCCs combine and generalize distinct reuse approaches. Section 2.6 presents the conclusions.

2.2 Specifying DCCs

The specification of DCC considers two issues: clear separation of content and interface specification, to support reuse; and use of ontologies for semantic annotation, to help find appropriate DCCs and to match their connections. This section presents DCCs using as background a real example for agricultural planning. Assume that experts want to forecast the evolution of a given coffee plantation under certain weather conditions. Given these as input, together with coffee plant geographic location, the output shows how the plants will evolve. This result is seen by means of an animation that simulates the growth of a plant for the input conditions provided.

2.2.1 An overview of DCC

A DCC is specified and stored as a unit composed by four distinct sections: (i) the content itself, in its original format; (ii) the declaration, in XML, of an organization structure that defines how components within a DCC relate to each other; (iii) a specification of the DCC interfaces, using adapted versions of WSDL [19] and OWL-S [42]; (iv) metadata to describe functionality, applicability, use restrictions, etc., using OWL [73].

We differentiate between two kinds of DCC – process and passive DCCs. A process DCC is process-centric: it encapsulates any kind of process description that can be executed by a computer (e.g., sequences of instructions or plans). A passive DCC is content-centric (e.g., a text or video file) and its interfaces define how its content can be accessed.

DCCs are assumed to be stored in repositories on the Web. Interface and metadata sections are used to help retrieve the appropriate DCCs from the repositories. There is furthermore a DCC infrastructure that comprises an architecture to assemble DCCs into a desired product. For more details on DCCs, see [66].

Ontologies play a fundamental part in DCC description and semantic management.

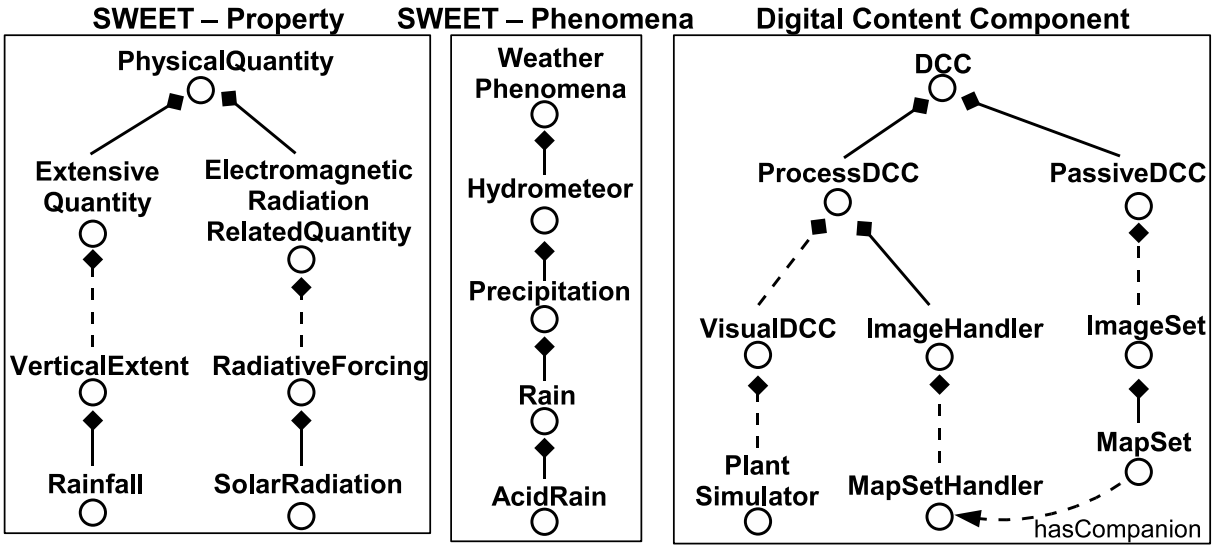

Figura 2.1: Ontologies used by rainfall map component.

According to [20], there are two main kinds of ontologies: descriptive and taxonomic. A descriptive ontology resembles database schemas. Its concepts are interconnected by many kinds of semantic associations, and its purpose is to represent the intended domain as much as possible. A taxonomic ontology is used as a basis for vocabulary alignment. Its structure organizes terms into generalization/specialization hierarchies, and semantic links to express synonymy, composition, and so on.

Taxonomic ontologies are useful in information sharing activities [20]. We adopt them in DCCs to disambiguate the meaning of DCC metadata and interface specification. More specifically, we postulate the need for specific ontologies that define valid kinds of DCC and of terms used in defining DCC interfaces. Fig. 2.1 shows diagrams that represent parts of two taxonomic ontologies, used by our examples, and whose hierarchical relations will be explored in DCC semantic relationships and search procedures. White-filled circles represent classes. Lines with a diamond in one extremity represent subclass relationships, e.g. `Rain` is subclass of `Precipitation`. Dashed lines indicate that some intervening nodes were ommitted for simplicity.

Our example concerns managing, creating and reusing content for agriculture applications. DCC discovery and reuse require domain semantics – in this case, the taxonomic ontology called SWEET – Semantic Web for Earth and Environmental Terminology [61]. Fig. 2.1 shows two fragments of SWEET. The fragment at the center concerns a taxonomic hierarchy about Rain, while the left fragment describes physical measurements. The right fragment is part of our ontology constructed to classify DCCs according their functionality. Each class in this ontology corresponds to a DCC type, and each DCC is

an instance of a class.

2.2.2 The Rainfall Map Content Component – Content Centric Approach

Our application requires combining rainfall and solar radiation data with coffee plant growth simulation. We show how this can be done by first creating the DCCs and then composing them.

Fig. 2.2 shows an example of a partial representation of a passive DCC. This component encapsulates a temporal series of images containing one year of rainfall data for São Paulo state. Each image is visualized in a map and represents the average rainfall distribution in one month (i.e., each pixel contains the average rainfall value for the corresponding region). The internal organization structure of the component, a set of twelve images (the content), is described in XML.

Both metadata (in OWL on top) and interface (in OWL-S displayed around the organization structure) are presented using a simplified version of RDF-like Directed Labelled Graph (DLG). Metadata and interface parameters are associated with ontological terms.

In the metadata section there is a reference to the DCC ontology presented in Fig. 2.1. It shows that the DCC is an instance of the *MapSet* class, with three property values: *title*, *coverage* and *phenomena*. The values of *phenomena* and *coverage* are respectively related to SWEET and to the POESIA spatial ontology [27]. The latter is a spatial ontology specific to Brazilian spatial unit organization.

The interface section presents operations using OWL-S *ServiceModel* class hierarchy [42]. It defines two operations (atomic processes in OWL-S): *getQuantity* and *getMap*. The *getQuantity* operation returns a value of a pixel inside a map image, given parameters *month* and pixel *coordinate*. The *getMap* operation returns a map image for a given *month* parameter. These atomic processes, which receive one input message (comprising all input values) and return one output message, correspond to WSDL request-response operations [42]. To simplify the explanation we will use the same names of OWL-S atomic processes to refer to WSDL related operations.

These operations illustrate how the descriptions can be connected with taxonomic ontologies. Following the OWL-S model to describe processes, each process parameter has a *parameterType* which specifies the class or datatype for that parameter [42]. Notice that many ontologies may be needed to properly specify a parameter. For instance, the *coordinate* input parameter has a type description associated with SWEET, but its domain is defined by the *coverage* property, in POESIA, here denoting that the only valid coordinates accepted are those from within the state of São Paulo. The output parameter of the *getQuantity* operation is an integer value. The additional *measures*

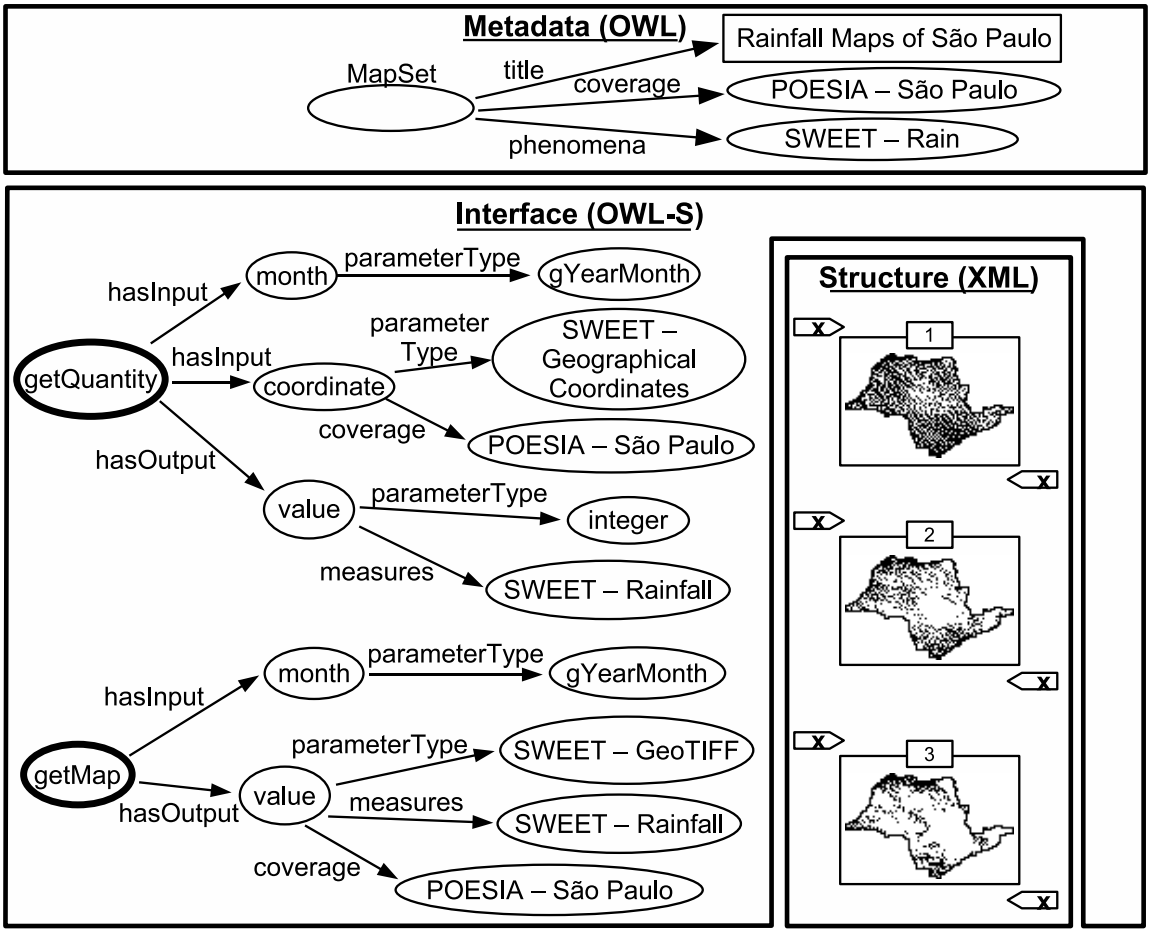

Figura 2.2: Rainfall map content component representation.

property of the SWEET ontology defines the nature of measured value. Notice that we extended the OWL-S schema to enhance parameter description with additional semantics. OWL-S specifies the need for type characterization (*parameterType*) which we improved by adding semantic parameter descriptors (e.g., coverage, measures). The parameters of the *getMap* operation work the same way.

This extension to OWL-S to organize components is similar to the *faceted* method, borrowed from library science by Prieto-Díaz [58] to classify software components. In contrast to the traditional *enumerative* method adopted by library science, which uses a classification tree to organize components in categories and sub-categories, the *faceted* method describes components by a set of attributes (named facets); each facet is specified by setting a pertinent term value. We used the RDF/OWL description approach to attach a set of descriptive property values (facets) to each parameter.

It is important to note that the DCC of Fig. 2.2 is *passive* – does not embed the

program code to execute these operations. The focus in this kind of component is in the content (i.e., the maps themselves), and the operations define how this content can be accessed. Since the program code for the operations cannot be embedded in a passive component, interface operations are implemented in a *companion component*. The association between the passive component and the companion is achieved with help of semantic information given by terms of the DCC ontology. The companion for the $MapSet$ component is the $MapSetHandler$ (see Fig. 2.1). $MapSet$ has a property value pointing to the $MapSetHandler$.

2.2.3 The Coffee Plant Simulator Process Component – Process Centric Approach

Fig. 2.3 shows an example of a partial representation of a process component. This is a software component that graphically simulates the growth of a coffee plant for a given coffee strain, and specific weather and location conditions. Its internal structure organizes Java binary code classes, which implement the simulator software, and related files.

To execute its job, the simulator DCC requests services from external DCCs. There are three processes, declared in the interface, for the requested services: $rainfall$, $solarRadiation$ and $growthRate$. They actuate in two stages, being thus composite processes. First they request a service by sending a message, containing their output parameters; next they receive the result of the solicited service in a message, whose content must match their input parameter. This kind of composite process corresponds to a WSDL solicit-response operation [42].

The simulator DCC uses $rainfall$ and $solarRadiation$ processes to request weather data, essential to estimate the coffee plant $growthRate$. In more detail, $rainfall$ provides parameters $month$ and $coordinate$ (whose semantics and types are defined ontologically) and receives back from an appropriate service a $value$ whose meaning and type are likewise defined. The same applies to the $solarRadiation$ process. Additionally, the simulator DCC declares the $start$ process, which is atomic and corresponds to a WSDL one-way operation.

We point out two further characteristics of DCC construction. First, the interface specifies processes. These may be operations implemented locally, or *other components* that have been built (reused) into the simulator. Second, parameter semantics define process (and component) semantics. Notice that in the $rainfall$ and $solarRadiation$ operations the $coverage$ of the $coordinate$ output parameter is $Campinas$: this component was built to simulate the coffee plant growth under Campinas city weather conditions, therefore it will only process values in this coverage. These constraints are used in a discovery process.

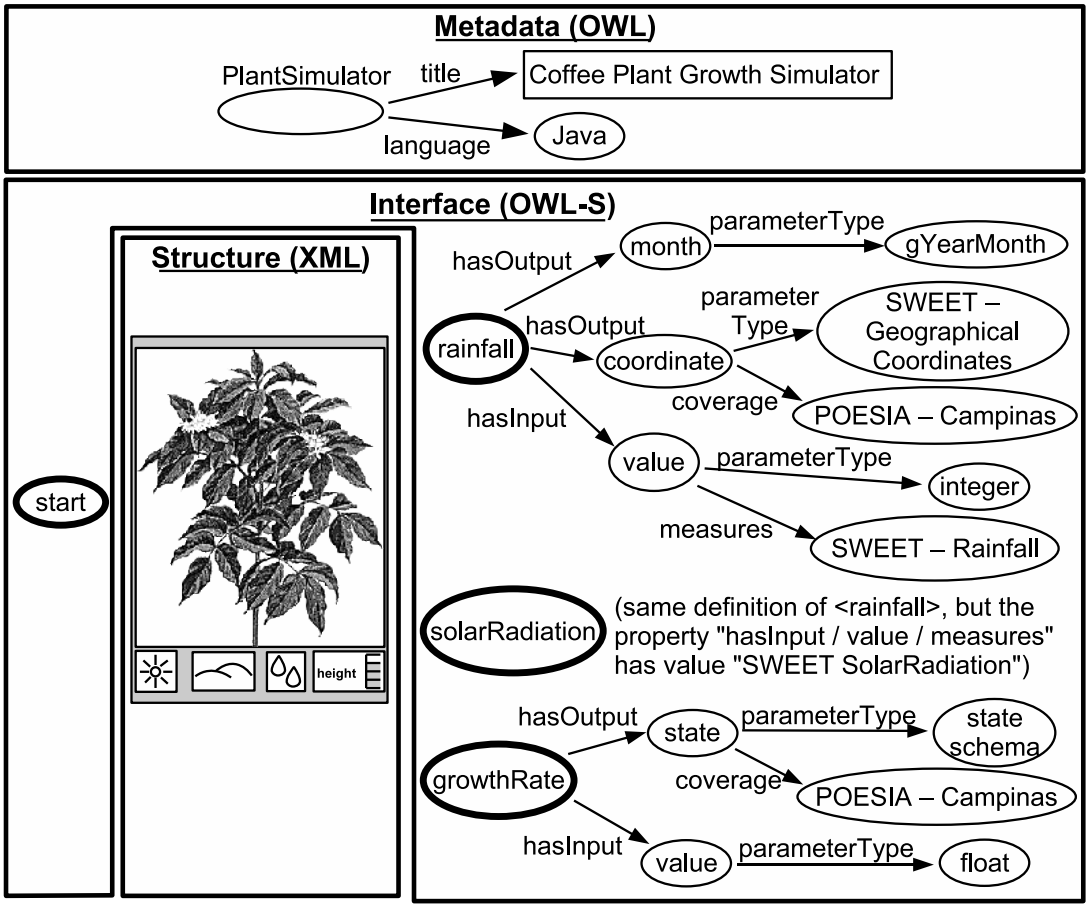

Figura 2.3: Coffee plant simulator process component representation.

2.3 Discovering DCCs

A key aspect of our proposal is how a designer discovers DCCs for reuse. DCCs' metadata and interfaces are specified in OWL/OWL-S, which can be used in component indexation and searching. Domain ontologies can help in this task in three ways: (i) they organize DCCs in taxonomic trees that can be navigated in the discovering process; (ii) ontology concepts are used to help query construction, to disambiguate terms and find synonyms; (iii) ontological relationships are used to rank DCCs based on their similarity with the searched DCC.

In DCC discovery process the designer can navigate through taxonomic trees to search for a DCC. The designer may define values of properties to characterize a desired DCC. Alternatively, the characteristics of DCCs already in a composition guide the search for the new one.

Let us consider a designer that wants to build a composition to graphically simulate

a coffee plant growth. He/she starts by the *simulator* DCC obtained navigating in the DCC taxonomic ontology and selecting a DCC instance of *PlantSimulator* class – see Fig. 2.3.

The next step is to connect the plant simulator instance to a DCC that provides the rainfall average for a given month and coordinate. Looking at the simulator specification, the designer will next query the Web looking for a DCC that supports the `Campinas` *coverage* of the POESIA ontology, and the `Rain` *phenomena* of SWEET ontology. As often occurs in this kind of searching process, maybe no DCC exactly matches with the query. The search engine can take advantage of the ontological semantic relationships to find other "most similar" DCCs and rank them depending on their similarity.

Our search procedure follows three steps: (i) metadata similarity-based searching and ranking; (ii) interface searching and ranking via inheritance relationships; (iii) interface matching-based refinement and ranking. Each of these steps will be detailed in the following three subsections.

2.3.1 Metadata similarity-based searching and ranking

Consider that the designer wants to retrieve a DCC via a query specified using the RDF-like DLG. The first step looks for metadata similarity selecting DCCs whose metadata graph is "similar to" the metadata query graph. For each query property, the search engine verifies if the same property exists in the DCC; if not, it verifies if there is a property in the DCC that is defined as an OWL subproperty of the query property.

The *rank_similarity* routine – called by the search engine – defines a value between 0 (no similarity) and 1 (equivalent concepts). *rank_similarity* uses ontologies to compare a DCC property to a query property, acting in three directions to determine: equivalent concepts, more general concepts and more specific concepts. The priority order in the ranking is: equivalent, general and specific, and can be inverted depending on the desired results. The search engine sums the ranked values of all properties.

In this comparison, two values A and B are considered equivalent if they refer to the same concept in the ontology (equal URIs), or if they point to two concepts related by OWL equality relationships (`equivalentClass` or `sameAs`). Moreover, A is said to be more general than B if A subsumes B and conversely B is more specific than A. For instance, if B is OWL `subClass` of A, or B is related with A through the $partOf$ property (B $partOf$ A), then A subsumes B. Consider A and B vertices of a graph, whose edges are properties. The subsumption relationship between A and B is a path formed by one or more edges. Therefore, the similarity rank value between A and B is inversely proportional to the number of edges which connect A and B in a subsumption relationship.

Let us return to the designer whose query is for DCCs with `Campinas` *coverage* of POESIA and `Rain` *phenomena* of SWEET. Assume that two DCCs obtained in the query response declare the following metadata: **(DCC1)** the São Paulo rainfall map, presented in Section 2.2.2; **(DCC2)** Campinas satellite images for days of acid rainfall, which declares a `Campinas` *coverage* and `AcidRain` *phenomena*.

DCC1 satisfies the search because its metadata has a *phenomena* concept equivalent to the query parameter on this concept, and because its *coverage* metadata relates to the `São Paulo` concept, that in POESIA ontologically subsumes the query predicate on `Campinas`: DCC1 covers a more general spatial surface than the one specified in the query. Thus, it includes the queried `Campinas` coverage. DCC2 metadata has an equivalence relationship on the query for the *coverage* concept (`Campinas`) and a subsumption relationship on the `AcidRain` concept (since in SWEET `Rain` is a superclass of DCC2's `AcidRain` – see Fig. 2.1). This means that this DCC produces a kind of rainfall average stricter than the one specified in the query. This result can be useful if the designer uses a generic concept to express a set of desired sub-concepts, for instance, if the designer wants to search for maps of any Brazilian state, he/she queries for `Brazil` *coverage* and expects to receive stricter coverages. For this reason the order of generalization/specialization in similarity ranking can be inverted and depends on the search context.

2.3.2 Interface searching and ranking via inheritance relationship

The query in the previous section can be refined by specifying, besides metadata, the kind of output expected from the desired DCC. In this case, this output has to match the input of the rainfall operation of the *simulator* DCC: a value with `integer` *parameterType*, whose semantics are defined by SWEET `Rainfall`. The similarity procedure to find and rank similar interface specifications is the same of the previous section.

Assume that this refined query returned another DCC – DCC3 – that will be added to the previous result (DCC1 and DCC2). **DCC3** is a software component that provides access to a remote weather repository for Campinas, and which declares an output with an `integer` *parameterType* and SWEET `PhysicalQuantity` for the *measures* property.

DCC3 satisfies the search because its output description has an equivalence relationship on the query for the *parameterType* concept (`integer`) and a subsumption relationship on the `PhysicalQuantity` concept (since in SWEET `Rainfall` is a subclass of DCC3's `PhysicalQuantity` – see Fig. 2.1). Notice that this step enhanced the searching process, finding additional DCCs based in finer-grained criteria.

2.3.3 Interface matching-based refinement and ranking

The interface specification is again used to further refine the search. For example, it is necessary to verify if the components selected in the previous queries have operations which match with the *simulator* component. The designer refines the ranking, asking which of the three DCCs have an interface with an operation that matches with the $rainfall$ operation of the *simulator* component. Here, interface matching is used to discard those DCCs whose interface does not match the request. While the previous step uses interface descriptions to increase the DCC candidates, this step can reduce the set of candidates.

Let X and Y denote inputs declared in the interface of two DCCs and consider the meaning "equivalent", "more generic" and "more specific" of Section 2.3.1. We define the follows relationships: (i) X EQ Y if for each property of X, Y has the same property with an equivalent value; (ii) X GE Y if for each property common to X and Y, the value of the X-property cannot be more specific than that of the Y-property, and conversely Y SP X. Let now `Q` be the DCC specified in a query, and `inQ` and `outQ` the set of all its inputs and outputs respectively. Let `S` be a DCC and `inS` and `outS` the set of its all inputs and outputs.

We define four levels of interface matching: **exact:** If `inQ` EQ `inS` and `outQ` EQ `outS`; **plug-in:** If `inQ` SP `inS` and `outQ` GE `outS`; **wider:** If `inQ` GE `inS` and `outQ` SP `outS`. **fail:** Not classified in the previous degrees.

The **plug-in** match is a simplification of the one proposed by Zaremski and Wing [84]. This kind of match guarantees that the `inS` input domain is a superset of the `inQ` input domain, hence, the `S` DCC can deal with any input of the domain specified in `inQ`. The `outS` output domain is subset of `outQ` output domain, hence, any output generated by the `S` DCC is within the expected results. In a nutshell, the `S` DCC can be plugged in any system where `Q` is required, and will not compromise the system functioning with an unexpected behavior. On the other hand, `S` is not equivalent to `Q`.

The **wider** match is the inverse of the plug-in match. This is the interpretation made by Paolucci et al [55] to the Zaremski and Wing plug-in match. Here, since the `outS` output domain is a superset of `outQ` output domain, it is expected that the `S` DCC can fulfill any output requested by `Q`. Analogously `inQ`, which is a superset of `inS`, can fulfill all input needs. However, it cannot be guaranteed that the `S` DCC will work properly if, for example, `inS` receives an unexpected input.

Returning to our example, the interface matching procedure will take *simulator*'s $rainfall$ operation as a basis to refine and rank the DCCs search. As illustrated in the first column of Fig. 2.4, to specify the query, to be used in the refining and ranking process, the inputs are transformed in outputs and vice-versa. The other two columns of Fig. 2.4 display a clip of the OWL-S descriptions of DCC1 *getQuantity* operation

and DCC3 `Weather Repository's` *getPhysicalQuantity* operation, both retrieved in previous steps of this search procedure.

Requested Operation	Rainfall Maps of São Paulo getQuantity operation	Campinas Weather Repository getPhysicalQuantity operation
<hasInput> <Input r:ID="coordinate"> <parameterType r:resource= "&s;GeographicalCoor..."/> <g:coverage r:resource="&p;Campinas"/> </Input> **</hasInput>** **<hasOutput>** <Output r:ID="value"> <parameterType r:resource="&x;integer"/> <g:measures r:resource="&t;Rainfall"/> </Output> **</hasOutput>**	**<hasInput>** <Input r:ID="coordinate"> <parameterType r:resource= "&s;GeographicalCoor.."/> <g:coverage r:resource="&p;SaoPaulo"/> </Input> **</hasInput>** **<hasOutput>** <Output r:ID="value"> <parameterType r:resource="&x;integer"/> <g:measures r:resource="&t;Rainfall"/> </Output> **</hasOutput>**	**<hasInput>** <Input r:ID="coordinate"> <parameterType r:resource= "&s;GeographicalCoor.."/> <g:coverage r:resource="&p;Campinas"/> </Input> **</hasInput>** **<hasOutput>** <Output r:ID="value"> <parameterType r:resource="&x;integer"/> <g:measures r:resource= "&t;PhysicalQuantity"/> </Output> **</hasOutput>**

Figura 2.4: Clips of OWL-S specifications for a queried interface and DCC1 and DCC3 interfaces.

Using the matching procedure, the DCC1 operation is classified as plug-in. Its output description has an equivalence relationship on the query for both the *parameterType* concept (`integer`) and the *measures* concept (`Rainfall`). Its input description has a subsumption relationship on the `São Paulo` *coverage* concept (since in POESIA `São Paulo` subsumes `Campinas` – see Fig. 2.1) and an equivalence relationship for the *parameterType* concept (`GeographicalCoordinates`). The DCC3 operation is classified as wider, following the same reasoning.

2.4 DCC-based Application Construction

The previous sections illustrated how to discover DCCs necessary to build a composition. This section shows the construction of an application that is built by composing these DCCs. Fig. 2.5 shows the diagram of the application. It was constructed via a composition of five DCCs, whose purpose is to simulate the growth of a coffee plant in a region of Campinas. Many aspects are omitted to simplify the example.

Roughly speaking, an application can be constructed using the following steps: (1) elicit requirements with help of experts and users; (2) determine basic data and processes needed; (3) search for appropriate process and passive DCCs to be reused using the semantic annotations provided by DCC description in metadata and interface sections; (4) construct new DCCs if needed; (5) create the application, which is materialized into

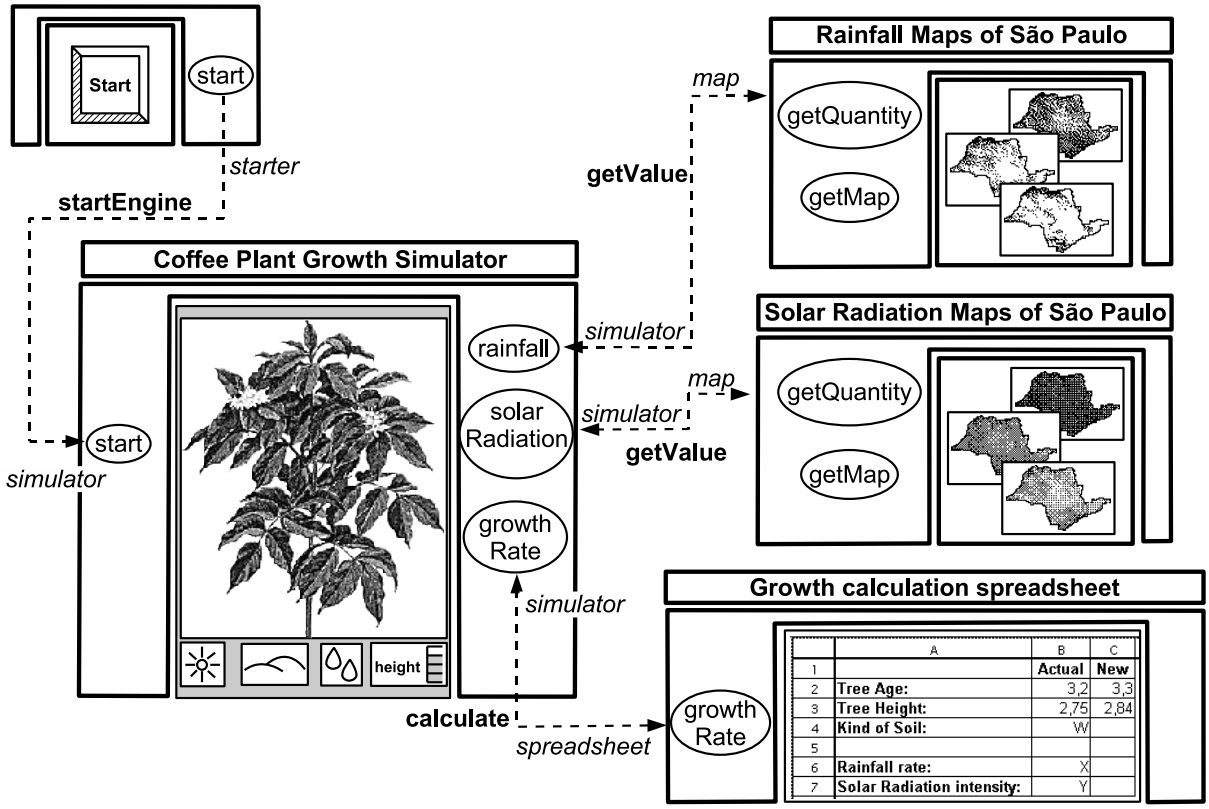

Figura 2.5: Composition to simulate coffee growth at São Paulo state.

a new DCC, by appropriate composition of reused and new DCCs.

In Fig. 2.5, DCC interfaces show the names of operations defined via OWL-S/WSDL. The Rainfall and the Solar Radiation maps are instances of the passive DCC of Section 2.2.2, and the Coffee plant simulator is an instance of the process DCC of Section 2.2.3.

The dashed lines represent the connections between components, whose format is an adaptation and simplification of WS Choreography [12]. To understand the purpose of these lines and their labels, we will summarize some key aspects of our model that adapt WS-Choreography concepts.

A composition is formed by a set of *participants*. Each participant is defined by a set of observable behaviors, which together form a *role* of this participant in the composition. A *relationship* is the association of two roles for a purpose. In Fig. 2.5 a dashed line represents a relationship between participants (DCCs), with a boldface label indicating its name. Each label in italics represents the role played by the corresponding participant in the relationship.

A given component may play several roles in a composition. In this example, each

component plays a single role. Application execution starts when the user presses the start button (*starter* component), which sends a message to the *simulator* component. The *simulator* graphically presents the growth of a coffee seedling in Campinas.

Before the execution of the simulation, the user configured the *simulator* component, selecting the number of cycles in the simulation. A cycle shows the growth of a coffee plant in a time period and runs as follows. The simulator sends requests to the map component, for a period of time and region, receiving the average rainfall and solar radiation for that period and region (here, Campinas). Next, it requests that the calculator component computes the plant's state based in these and other parameters. This response is used to feed the simulator's growth rate process, showing the plant's next stage.

2.5 Comparison to Related Work

This section analyzes related work. We focus on two relevant aspects: our choice for describing DCC functionality; and the technique to search for DCCs.

2.5.1 Associating functionality to DCCs

Many initiatives identify the importance of representing some kind of relationship between the reused content and the program code, to: guarantee correct future content interpretation in long term preservation [15], enable active interaction between an educational environment and units of educational content [43], standardize the way of how multimedia content artifacts will be accessed by software units [34], and process the content on demand to produce new transformed results [76].

More specifically, standards are being proposed in education [33, 43], digital libraries [15], multimedia [34], software development related artifacts [53], among others. Common problems to be faced include standards to: store the content, pack and deploy autonomous reusable units and define metadata standards to describe these reusable units.

Our proposal represents a step beyond, since it provides a standard ontology-based mechanism to relate units of software and content, and provides a unified reuse perspective, suitable for both software and content, which are reused together. The tight dependency between them results in a synergetic effect that increases reuse opportunity.

2.5.2 Searching DCCs

There are many kinds of proposals for searching for content on the Web. We follow the trend that concentrates on using taxonomic ontologies as basis for semantic similarity.

Strategies include: the use of minimum path length between two concepts in IS-A hierarchies [60], or the maximum value of information content achieved between a concept that subsumes two compared concepts [62]. When the search is ontology-based, many ontologies can be involved. The ontologies related to a searched content can be different from those related to content artifacts in the repository. The issue of ontology mapping is treated in semantic integration research [51]. If no mappings are available among ontologies, concepts related to them cannot be compared and a mapping discovery process may be needed [21].

Our proposal considers the existence of mappings between compared concepts. It is based on combining the work of Prieto-Díaz [58] and Paolucci et al [55], using the similarity concepts proposed by Zaremski and Wing [84].

Prieto-Díaz proposes the faceted method, borrowed from library science, to classify software components [58]. Each facet, used to describe a component, is associated with a scheme, which defines a list of terms that can be used as facet values. To enhance component searching he uses a thesaurus to disambiguate terms and find synonyms, and a conceptual distance graph to rank similar components, based on closeness of related terms. Our approach is based on the same ideas. However, it uses OWL as a unified technology for the three tasks: properties are used as facets, taxonomic ontologies are used as thesauri, and ontological relationships are used to determine component similarity instead of conceptual distance graphs.

Zaremski and Wing [84] adopt a language called Larch/ML to specify interfaces of software components and to specify queries to search required components. These specifications define pre-/postconditions for component execution using first-order predicate logic. The matching between the required component specified by a query (Q) and the provided component with interface specification (S) is based on logical relationships, like equivalence and implication. The matching between S and Q can relate pre and postconditions as separate entities, or can relate entire specification predicates S_{pred} and Q_{pred} where, for any specification X, $X_{pred} = X_{pre} \Rightarrow X_{post}$. On one hand, our approach is capable of exploring the richness of ontological relationships to compare input/output similarity, instead of logical relations. On the other hand, pre-/postconditions can detail requirements, which are not possible in our approach.

Our work is likewise related with Paolucci et al [55], which is also based on [84], and adopts DAML-S for interface matching in Web services discovery process. As mentioned before, their approach for plug-in match follows a direction distinct from that of [84]. Both approaches are contemplated in the third step of our search procedure, which is not restricted to finding software components or services, but extends this functionality-based search technique to any kind of digital content.

2.6 Concluding Remarks

This paper presented a new approach to structure digital content in order to facilitate its reuse and discovery using Web standards. Our work combines proposals to use interface specification, taxonomic relationships between concepts and interface matching, to enhance digital artifacts searching, using Semantic Web-based metadata and interface specifications in our Digital Content Component model.

One of the main challenges was the specification of the functionality of each reusable piece, which guides their discovery and combination. DCC diversity requires an expressive and flexible mechanism, equally suitable for software components, images, texts, videos, among others.

We can single out two main contributions of our work in this context: first, the extension of the interface specification to express the "potential functionality" related to any kind of digital artifact, associated to a mechanism that converts it in a "real functionality" implemented by a companion component; second, a three step procedure that explores metadata associated to DCCs, combined with the functionality expressed in their interfaces, to enhance the DCC searching process.

Traditionally, the relationship between software components and Web services is related to distributed components. Here, we propose that the same technology be applied to any kind of DCC (distributed or otherwise). Semantic Web based standards are useful to promote interoperability via components, even at the local level. Therefore, we adapt these standards, simplifying them when needed to accommodate the local context. DCC description is based on an adaption of WSDL and OWL-S to describe the interface at syntactic and semantic levels respectively. This promotes reusability and facilitates the discovering of reuse units on the Web. Ongoing work involves implementation of DCC construction and search mechanisms. We have already developed a few experiments that show the feasibility of our ideas.

Capítulo 3

User-author centered multimedia building blocks

3.1 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: (i) multimedia tools became *easier to use*, being accessible to non professional developers; (ii) the evolution of open standards combined with the Internet fostered the *sharing, reuse and adaptation of productions*; (iii) 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. 3.1.

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 non computer professionals, these users

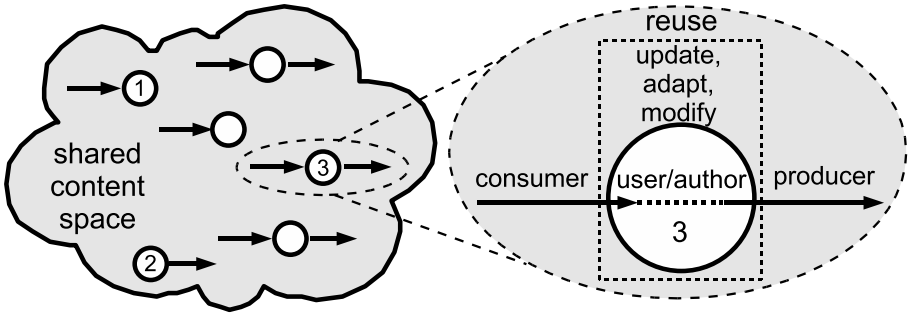

Figura 3.1: Diagram illustrating our perspective of today's user.

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-author)ing means producing any digital content involving multimedia artifacts, 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: (i) *tradeoff between ease of use and reusability*; (ii) *nature of content*; (iii) *domain of application*.

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* [7] 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 towards reuse are difficult to use. For instance, MPEG-21 [13], 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.

Content nature and Application domain barriers

Content nature (executable software versus content in general) and solutions driven by the application domain present limitations to user-authoring: (i) 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; (ii) content-centric models are designed to be used in specific application domains; (iii) 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 *Digital Content Component - 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 in 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: (i) presentation of DCCs as a user-author centered building block in the multimedia context; (ii) analysis of the notion of content-type driven execution under different guises; (iii) and 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. Sec. 3.2 presents an overview of related work. Sec. 3.3 presents the requirements that led to our user-author multimedia building block. Sec. 3.4 briefly presents our DCC model. Sec. 3.5 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. Sec. 3.6 and 3.7 discuss retrieval mechanisms and some implementation issues. Sec. 3.8 summarizes how the DCC model and infrastructure meet the requirements of Sec. 3.3. Finally, Sec. 3.9 presents conclusions and ongoing efforts.

3.2 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. Sec. 3.2.4 compiles and classifies a set of strategies used for content-type driven execution.

3.2.1 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 – for example, a video file can require a video player routine to enable its 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. 3.2a, 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

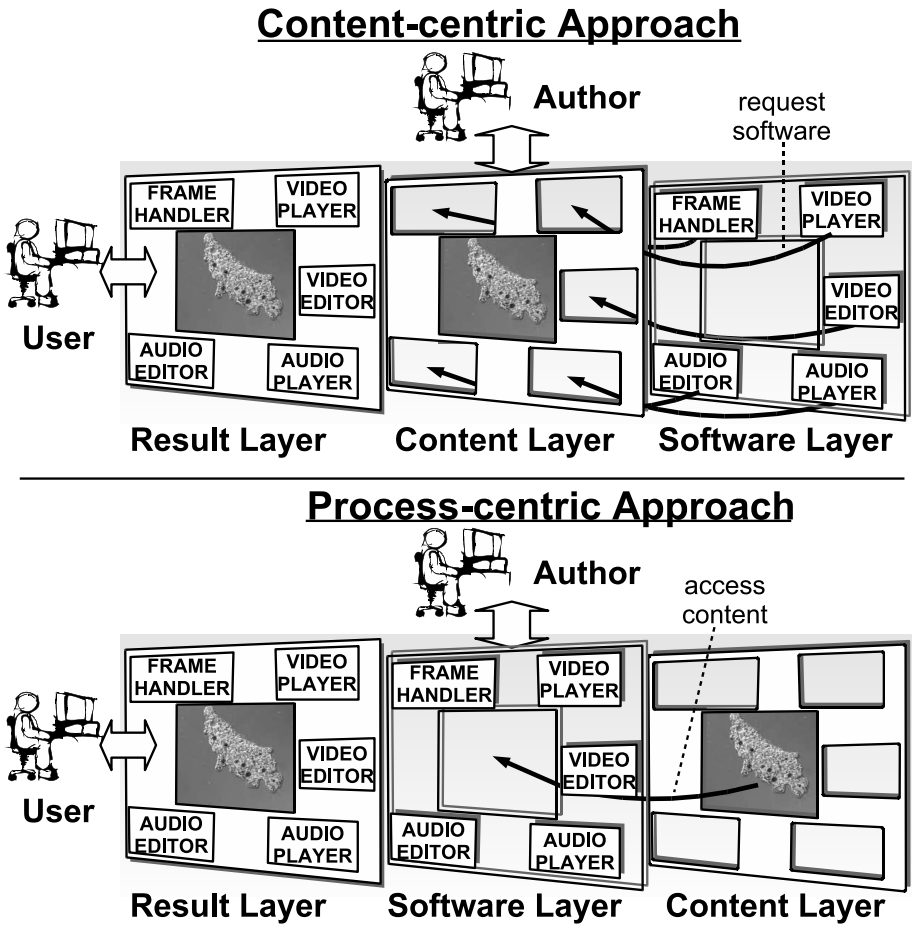

Figura 3.2: Diagrams illustrating (a) Content-centric and (b) Process-centric approaches

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 languages are the usual way to implement process-centric applications. There are however higher level approaches more suitable to the non-expert authors – e.g., composition of software components, workflow specification. As illustrated in Fig. 3.2b, 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 [23]. 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 [2,33,74], digital libraries [15,77], multimedia [35] and software development related artifacts [53].

3.2.2 Software Components (process-centric approach)

Software components have been the main unit adopted for program code reuse. There are many definitions for software component [32]. Even if they do not achieve total agreement, some characteristics are present in all definitions, or can be inferred from them: (i) a component is an entity meant to be composed; (ii) each component publishes its functionality through a well-defined and open interface; (iii) components can be nested into other components.

From a practical point-of-view, software components have additional characteristics, observed in the widespread component initiatives: (i) components contain some kind of binary code that implements the functionality declared in their interface; (ii) 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.

3.2.3 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) [7], 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 [13] is to bring these standards together. MPEG-21 [35] 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) [81], 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) [14].

The engine responsible for Digital Item processing (DIP engine [13]) 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 it to address preservation of complex digital objects over the long term, admitting impacts of changing technologies and user community [15]. 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. METS – Metadata Encoding & Transmission Standard – 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 [77]. METS specifies a behavior support associated with complex digital objects. These declarations relate items of content with a Web services API provided by Fedora [76], 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 [33], 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 – for example, 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) [43],

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.

3.2.4 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.

Software frameworks. The more similar two content types are, more closely related are 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 [57], 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. The author must adapt his/her code to properly embed the framework.

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 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 [8], Protégé plug-ins [50]. Like in software frameworks, it is necessary to adapt

programming code to port (reuse) these plug-ins to a new system.

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 [9] and Apple OpenDoc [5], 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: (i) file extension – a poor and ambiguous mechanism (many formats have the same extension); (ii) file header – not a standardized mechanism, since each content type has a distinct format to define its header; (iii) 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.

3.3 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.

I. Breaking barriers between content- and process-centric development

As presented in Sec. 3.2.1, 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).

II. 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 [40].

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 com-

prising 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.

III. 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.

IV. 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 [35]. 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 [15].

The standardization efforts discussed partially deal with this issue, as seen in Sec. 3.2.3. 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 functionality is restricted. Second, MPEG-21 DIMs work as auxiliary routines, and have no structure appropriate for sharing and reusing among authors. The Fedora [76] 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 [33] 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.

3.4 A Very Brief Overview of DCC

A *Digital Content Component* (DCC) [66] 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: (i) the content itself (data or code), in its original format or a DCC composition; (ii) the declaration, in XML, of an organization structure that defines how components within a DCC relate to each other; (iii) a specification of DCC interfaces, using adapted versions of WSDL [19] and OWL-S [42]; (iv) metadata to describe functionality, applicability, use restrictions, etc., using OWL [73].

DCCs are assumed to be stored in repositories available on the Web. Interface and metadata sections – respectively (iii) and (iv) – are used to help retrieve the appropriate DCCs from the repositories and reuse them [68]. There is furthermore 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 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 on software development; they are based on an approach where the end-user is the author, and the components are the "raw material" [48, 63].

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 Sec. 3.5.1. 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 releant to the paper, on DCCs, see [66].

3.5 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 Sec. 3.5.1. Second, how can content-type driven execution be explored to create a meaningful author-friendly authoring environment? This is treated in Sec. 3.5.2. In our implementation, the subsystem responsible for supporting authoring and execution of any composition involving DCCs is called *execution engine*, described in Sec. 3.7.

3.5.1 DCC Content-type Driven Execution

As discussed in Sec. 3.2.4, 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 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 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.

Fig. 3.3 shows an example of content-type driven execution. In the figure, a passive

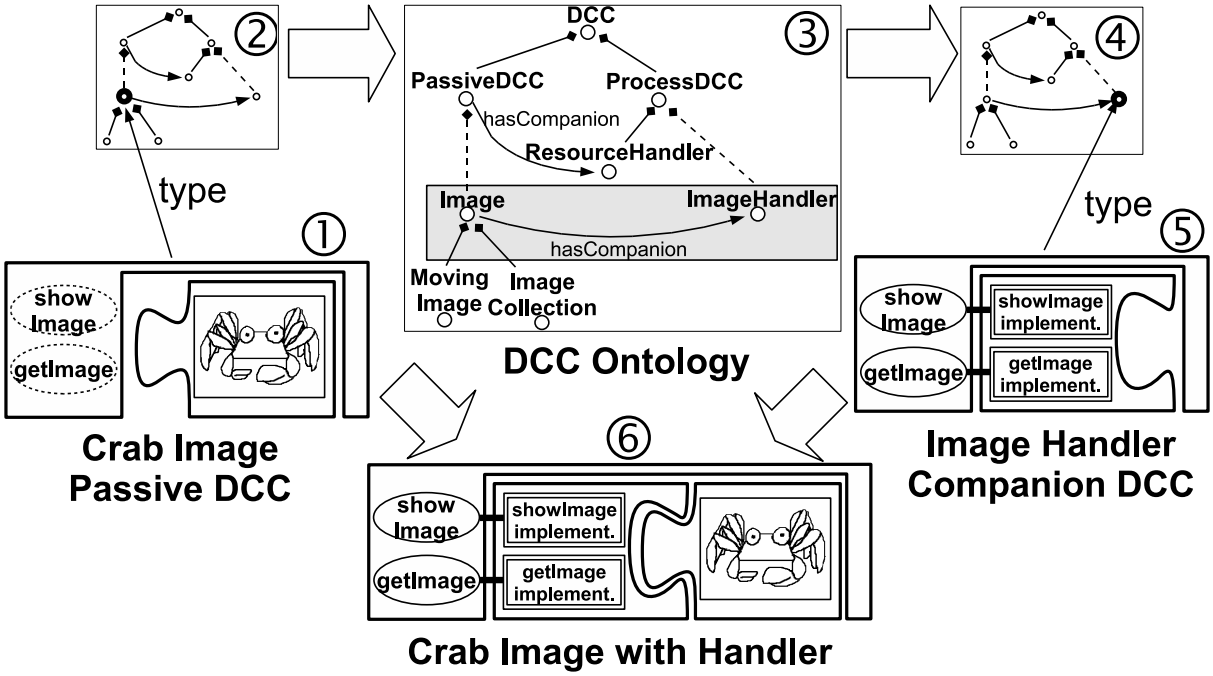

Figura 3.3: DCC content-type driven execution diagram.

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 [20] – 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. `PassiveDCC` subsumes `Image`. Dashed lines indicate that some intermediate nodes were ommitted for simplicity. Each arrow represents a property `hasCompanion` 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 (`ProcessDCC`) or content (`PassiveDCC`), 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 mi-

nimal interface of `B` extends, or is equal to, the minimal interface of `A`. If a DCC `A` defines the property `hasCompanion` 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.

Fig. 3.3 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 in 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 Sec. 3.6. 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 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: (i) implemented in Java to run in a stand-alone application, (ii) implemented in Java to run in a Web browser (applet), and (iii) 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.

3.5.2 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 a DCC-based multimedia authoring system. Magic House is based on a combination of the graph-based authoring paradigm of [11] 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.

Fig. 3.4 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

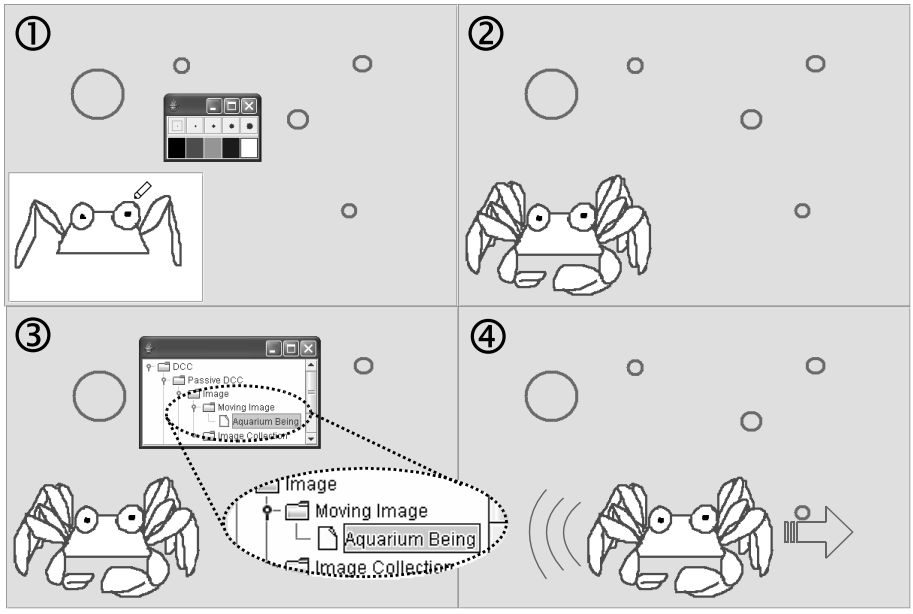

Figura 3.4: Steps followed in Magic House authoring system to produce a crab DCC.

configurable parameters that specify work conditions – e.g., language. Following the cycle described in Sec. 3.5.1, the system finds a companion DCC appropriate to handling an image DCC. The result is shown in step `1` of Fig. 3.4. 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 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. 3.4, the author requests to the Magic House environment to redefine the type of the 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.

3.6 DCC Retrieval Mechanism

We recall from the Introduction that user-author centered multimedia authoring, in our context, means: (i) ease of use in sharing, interact with and running a multimedia artifact; and (ii) 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 [68].

3.6.1 Finding/Retrieving a DCC

Fig. 3.5 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. UDDI [78] – Universal Description, Discovery and Integration – is a standard for a Web-based registry service, whose primary goal is to describe and discover Web services [3]. UDDI supports the description of entities other than Web services. Additionally, more recent versions of UDDI can accommodate and use identification taxo-

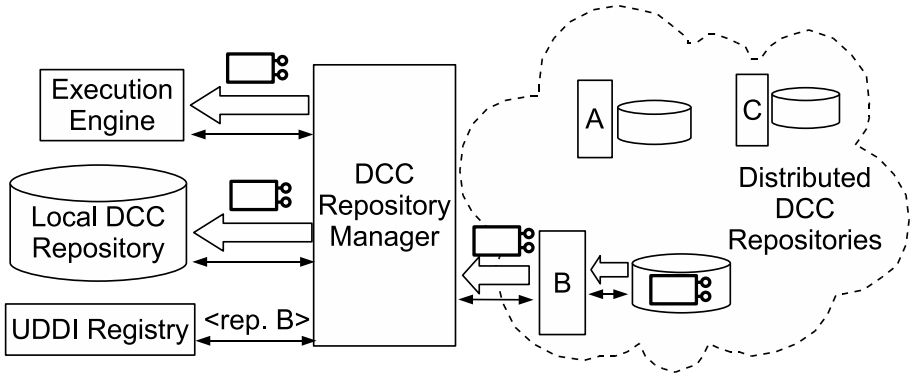

Figura 3.5: Finding and retrieving a DCC based on type matching.

nomies provided by third parties [78]. The DCC repository manager uses UDDI registry services to specify which repositories have a given DCC type, via the URI (Uniform Resource Identifier) of each type. It is important to note that the DCC type ontology works as a 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 example of Sec. 3.5.1). 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.

3.6.2 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. 3.3, the `Image` DCC type is subsumed by the `PassiveDCC` 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 with using a ResourceHandler companion.

In other words, in the DCC model, an author can define, during composition, that

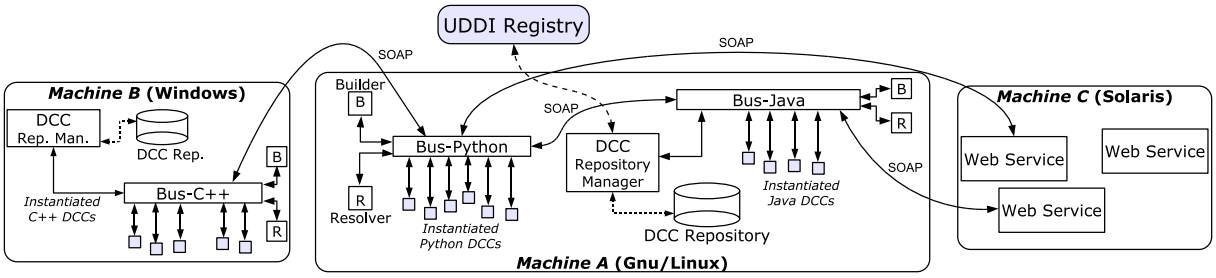

Figura 3.6: Example an arrangement with three machines adopting the Anima architecture.

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 total or partially, tailoring it to their needs.

3.7 Implementation aspects

This section presents the architecture designed to support our framework, which has evolved from previous projects named *Anima* and *Magic House* [71].

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 an educational authoring tool built over Anima. Both systems have been used to create educational tools in schools in the city of Salvador, Brazil. These tools allow authoring of multimedia products (e.g., animations employed in exploring laws of physics).

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 to 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 3.6 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. 3.6, 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: (i) among DCCs connected to the same Bus; (ii) between a DCC in a Bus and a DCC in another one, through an inter-Bus communication; (iii) 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 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: (i) a DCC in the same Bus; (ii) a DCC in another Bus; or (iii) 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 SOAP (Simple Object Access Protocol) [47] 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 in a WSDL [19] 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 Sec. 3.6.1.

As all DCCs, the Repository Manager uses the Bus for communication with DCCs, including other Repository Managers. As can be seen in Fig. 3.6, 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 Sec. 3.5.1.

3.8 Meeting the Requirements

This section summarizes how the DCC approach meets the requirements presented in Sec. 3.3. In Sec. 3.8.2 we show how DCCs bridge the gap between user interaction and reuse/sharing features. Sec. 3.8.3 shows how the author can participate in constructing executable software by composing higher-level components. Sec. 3.8.4 shows how DCCs break barriers between content and executable software.

3.8.1 How DCCs Meet the Requirements

In the diagram illustrated in Fig. 3.2, we showed that authoring tasks concentrate in the "content layer" for the content-centric approach, and in 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).

DCCs' 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 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 in 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.

Sec. 3.5.1 already showed how the DCC model and infrastructure meets 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. 3.4, 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 [49] 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. As a consequence, it meets the second requirement only partially.

3.8.2 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 a 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

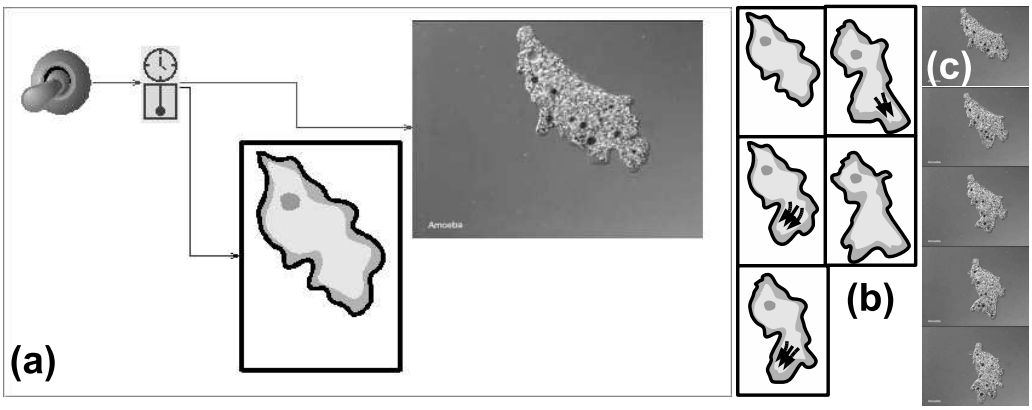

Figura 3.7: Magic House animation illustrating a cell movement. (a) Production design, (b) and (c) – passive DCCs.

blocks are selected and connected by direct manipulation using a visual tool, and customized by modifying parameters in a property sheet. A directed 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 syncronization. Once the composition is specified, it can be executed.

Fig. 3.7.a shows the design of a biology production in the Magic House environment, whose purpose is to deploy an animation that ilustrates 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 and request from it the captured 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 need to consider the nature of the 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, 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. Fig. 3.8 presents an example of this distinction. Consider a variation of the application illustrated in Fig. 3.7, 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. Now, let us consider that the author wants to implement the same presentation using two other approaches and tools: **(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; **(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.

Fig. 3.8 shows solutions (a) and (b) at execution time. Solution (c) uses a companion DCC and is divided in 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. 3.8, software plug-ins and active document

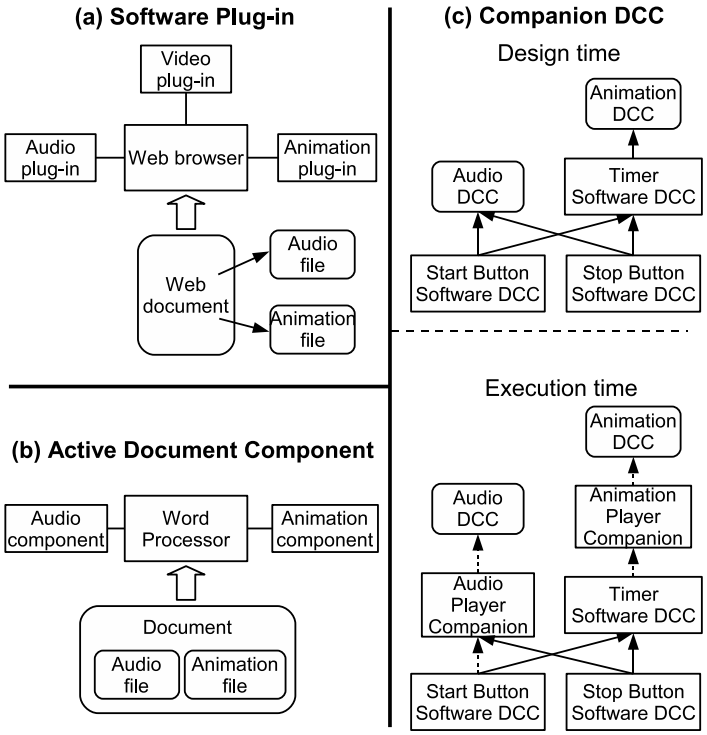

Figura 3.8: Diagram confronting connection architectures.

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 needing a central module. This is a more flexible solution and can be explored to create compositions that can be executed in a distributed way.

3.8.3 Composing Higher-level Components

Sect. 3.8.2 gave an example of authoring and interacting with a simple production. This section discusses how to create productions from composition of others.

Fig. 3.9 shows a physics project, whose purpose is to display how the values resulting from the kynematics function `f(t) = S0+V.t` affects the movement of a ball. Fig. 3.9.a 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

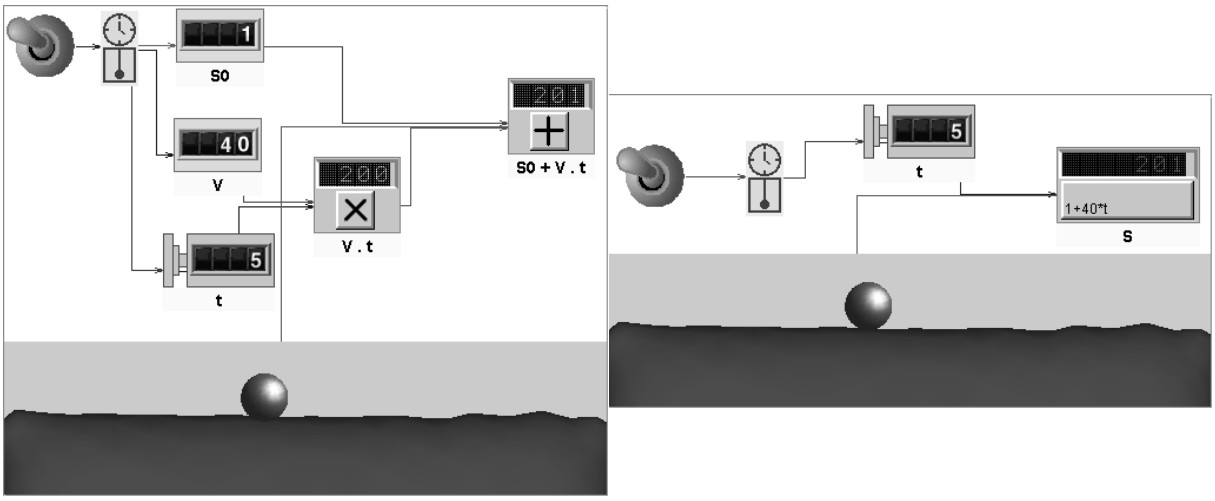

Figura 3.9: Two versions of a mathematics educational project using specialized components.

"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, schoolchildren) can change clock properties, coefficient values and even operations (e.g., S0-V.t). The net result is to allow the children to see the different ball movements and experiment with equations.

Fig. 3.9.b shows another version of the same project, where the ball position is calculated by a component, 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 to 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 [48]. 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

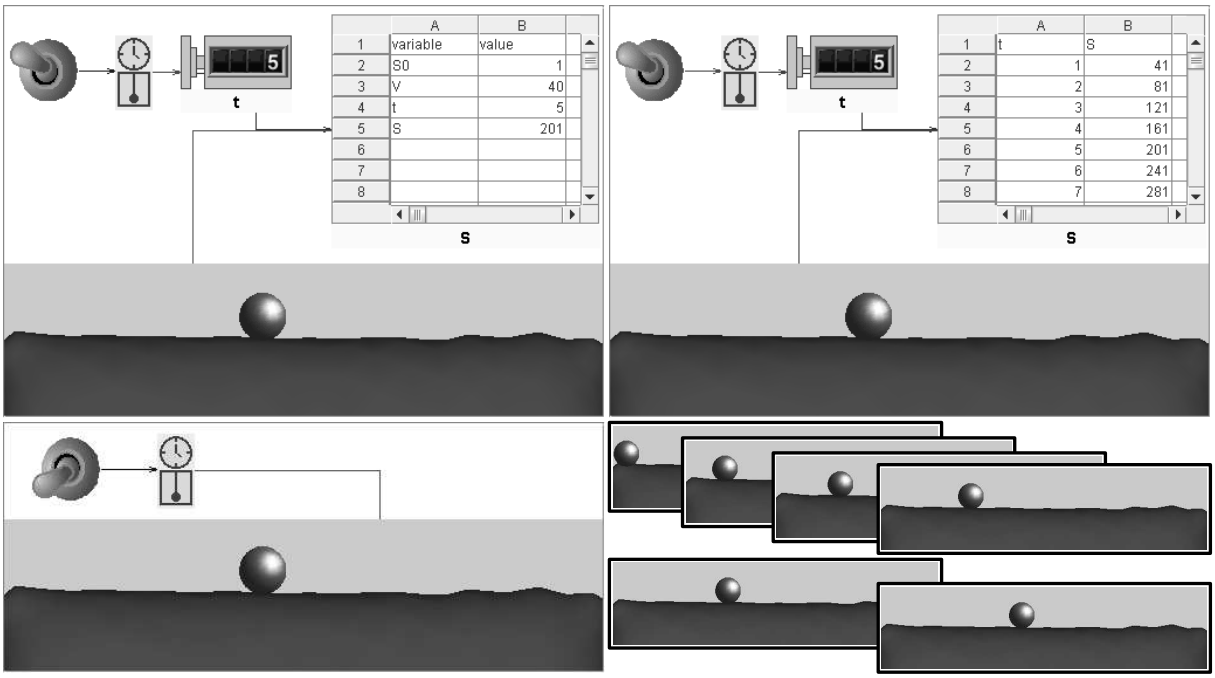

Figura 3.10: Animation of a ball: three alternatives.

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: (i) 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; (ii) have constraints on script expressiveness, e.g., MPEG-21 DIMs.

3.8.4 Breaking Barriers Between Content and Executable 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: (i) "manually" define the element position in each frame; (ii) produce a software routine to calculate the element position for each frame.

The three compositions shown in Fig. 3.10 illustrate this question. The goal is the same as the one illustrated in Fig. 3.9 – to animate a ball. The solution at the bottom of Fig. 3.10.c is the same as the one used in the cell examples (Fig. 3.7). The counter will prompt the animation, using the frames depicted on the bottom right.

In the other two solutions – Fig. 3.10.a and Fig. 3.10.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. 3.10.b, 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 new position of the ball. From an observer's point of view, all three animations are identical, moving the ball to the right.

Compositions 3.10.a and 3.10.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. 3.10.c 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 pieces or content pieces. They work in a higher level of abstraction, which is driven by the meaning conferred to each artifact.

3.9 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 (Digital Content Component). 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 [65].

The main contributions of this paper are: (i) the discussion and elaboration of the main characteristics that qualify a DCC as a user-author centered building block in the multimedia context; (ii) 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. (iii) 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.

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 [44].

Ongoing work involves version control support for DCCs and DCC compositions; the management of distributed update and alignment of the ontologies used by the framework; and and finally a SOAP-based implementation for distributed DCC management.

Capítulo 4

A Component Model and Infrastructure for a Fluid Web

4.1 Introduction

The Internet is expanding its frontiers. Tools and services are breaking domain boundaries and local limits as a consequence of progressive data and protocol interoperability, allied to an increasing connectivity. For this reason, digital content production, consumption and management has been attracting increasing attention. Communities are discovering the benefits of creating broad strategies to promote digital content distribution, reuse and customization, in order to help spatially distributed groups to work together.

Indeed, the Web is evolving from an environment for publication/consumption of documents to a distributed environment for collaborative work involving any digital content. As a consequence, the "document-oriented" approach that impelled the Web must be revised to face the increasing diversity of digital content formats and producers. The Web infrastructure must be likewise extended to support collaborative work requirements, such as content semantics, replication and modification, configuration management and version control.

The Semantic Web has appeared as an answer to these needs. However, today's Semantic Web model is strongly based on URIs as an identification and reference mechanism to build vocabularies, and to connect semantic annotations to any Web content. In the latter case this mechanism works adequately, considering that the annotated content stays in a single absolute (URL) or relative (URN) Web address. However, content subparts cannot be annotated unless they are themselves XML or HTML documents. As pointed out in [39], there is a wider universe of "annotatable" artifacts on the Web, formed by "annotatable" sub-parts whose organization depends on the artifact's nature. The challenge is how to associate Semantic Web annotations to artifacts and their subparts, in

spite of their heterogeneity.

Let us consider the scenario where an annotated content travels through nodes on the Web. Any node can replicate this content, modify it, cut it into subparts to be mixed with other contents, and so on. This is a *Fluid Web* notion, where the Web is taken as an environment suitable for distributed and collaborative work and where the partners involved in the collaboration change with time. This notion contrasts with the traditional Web scenario. It demands an extension of Web mechanisms to reference and annotate other kinds of digital content beyond Web documents, and to identify and relate replications and derivations of content objects through the Web.

Our main contribution lies in rethinking the Web to afford effective collaboration, including content packaging, deployment, identification, multilevel annotation, version control and configuration management. Our proposal to solve Fluid Web demands starts from two points: (a) the definition of a generic model based on the notion of Digital Content Component (DCC); and (b) an infrastructure to provide collaborative work through DCCs. As will be seen, this gives rise to a new approach to content production, where the distinctions between data and software management become blurred, thereby supporting the move from a document-oriented to a content-oriented Web. In particular, the following questions are tackled by the model and infrastructure:

Transparency: There exist distinct models to share, combine and manage data (passive content) and software (executable content). How can one unify these models?

Decomposability: Since digital contents can be decomposed and fusioned together, how to identify, reference and annotate content sub-parts when they are not Web documents?

Versionability: How to trace the evolution and version relationships between content artifacts through the Web?

Traceability: The Semantic Web URI-based annotation strategy provides a powerful universal reference mechanism, but the principle is that the annotated content will stay in a unique place. What happens with the annotations when a content travels through the Web and it is replicated and adapted? How can identification and annotations follow each subpart in subsequent decomposition and mixing operations?

Complex Relationships: How to express, represent and manage the diversity of relationships between digital content artifacts, and the consequences that these relationships have on the management of these artifacts (e.g., a modification in an artifact that affects other dependent artifacts)?

The DCC model allows the homogeneous treatment of data and software on the Web, via components that can be deployed, reused, versioned and fusioned regardless of the nature of their content. It introduces the notion of *companion component*, which, for a given data set, finds the appropriate software to activate it. These characteristics are supported via a uniform encapsulation and composition mechanism. Model and infras-

tructure provide a new, unified way of producing and combining software and data that relies on Semantic Web standards.

The remainder of this text presents our approach to the Fluid Web using as background a real example on the biological control of *Helicoverpa zea*. This moth larva is a pest that destroys tomato crops. A biological solution that enables to control this pest is based on the wasp of genus *Trichogramma*. This wasp lays its eggs inside the moth's eggs. As the wasp larva grows, it eats the moth's egg, thus effectively breaking the spread of this pest. In our example, the growth of tomato plants can be simulated using information on weather conditions, plant strain, moth infestation level and wasp behavior. We show how these pieces of information as well as appropriate simulation models can be stored in DCC Web repositories. A simulation of plant development in different conditions can be designed and implemented by combining the appropriate DCCs. We illustrate our solution to the Fluid Web scenario while constructing this simulation.

The rest of the paper is organized as follows. Section 4.2 gives an overview of our proposed architecture for the Fluid Web. Section 4.3 presents our product model for the Fluid Web, the Digital Content Component (DCC). Section 4.4 presents content version control and management, a key aspect in the infrastructure. Section 4.5 analyses the role played by ontologies to provide a semantic bridge across the infrastructure. Section 4.6 contains an overview of related work, contrasting our approach with other initiatives. Section 4.7 presents conclusions and ongoing efforts.

4.2 The Fluid Web

Our Fluid Web architecture is based on the notion that digital contents travel and are managed through the Internet under the guise of basic self contained units called DCC (Digital Content Components), detailed in Section 4.3. DCCs explore Semantic Web standards to provide syntactic and semantic interoperability. Fig. 4.1 displays our view of the Fluid Web: users in many sites exchange, reuse, annotate, compose, replicate, modify, store and deploy DCCs. DCCs are stored in private or shared repositories, managed by repository managers.

As will be seen, many tasks within the Fluid Web infrastructure require efficient ways to index and search for appropriate DCCs. Additionally, some characteristics of the infrastructure, like version control and configuration management, demand a more robust and consistent support than a simple file system. Consequently, the architecture is strongly based on databases: Fluid Web nodes are DCC clients and/or servers, and each node has a DCC repository and a repository manager.

A challenge in this infrastructure is how to exchange data used locally while at the same time controlling versions and relationships among DCCs. Our solution is to follow

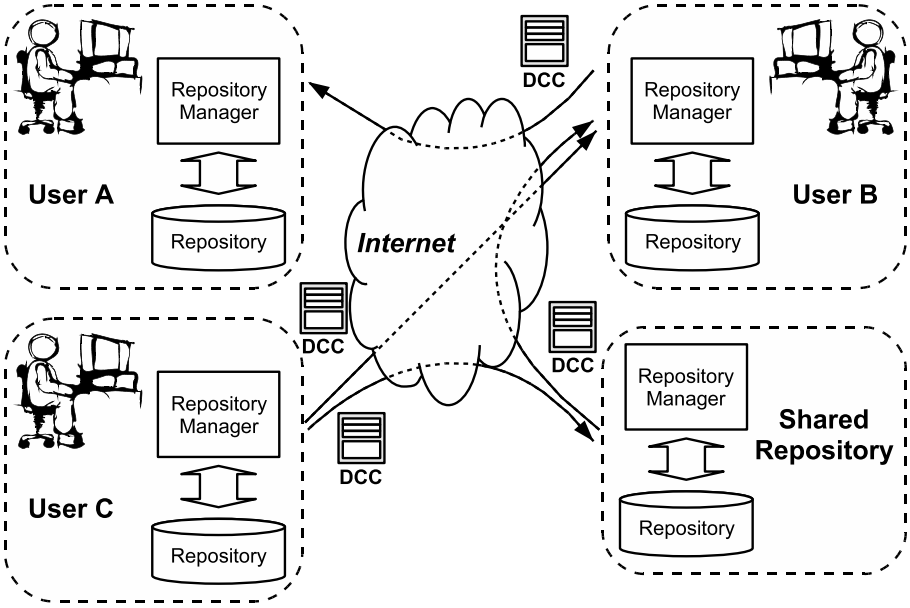

Figura 4.1: Fluid Web architecture diagram.

the strategy adopted by OAIS – Open Archival Information System [15]: we establish a clear distinction between the internal and external representations of DCCs and provide mechanisms to convert from internal representation to external and vice-versa. OAIS is a reference model whose purpose it to address preservation of digital information over the long term, admitting impacts of changing technologies and user community [15].

OAIS analyses important aspects for archiving digital content. It starts from the assumption that any content stored or retrieved from an archive will take the form of a package. Additionally, packages have different structures depending on the stage of their usage. The Submission Information Package (SIP) and Dissemination Information Package (DIP) have deployment purposes – to submit a content for archival or to receive a content respectively. The Archival Information Package (AIP) is meant to be stored in a database. The format of each package is tailored to its purpose.

The diagram in Fig. 4.2 refines the DCC repository manager displayed in Fig. 4.1. It is based in the OAIS reference model. The storage structure of our repository is divided in three main sections: (i) DCCs and their index structures; (ii) ontologies shared by the descriptions of the components; (iii) a history of component usage.

A DCC can assume two formats, illustrated in Fig. 4.2: *Deployment DCC*, which are units that flow through the Fluid Web (correspond to the OAIS SIP and DIP) and *Storage DCC*, which are internal to the repository (correspond to the OAIS AIP). As shown in the figure, `Producers` construct Deployment DCCs, which are converted into Storage DCCs for archival, and re-constructed into their deployment format for consumption by

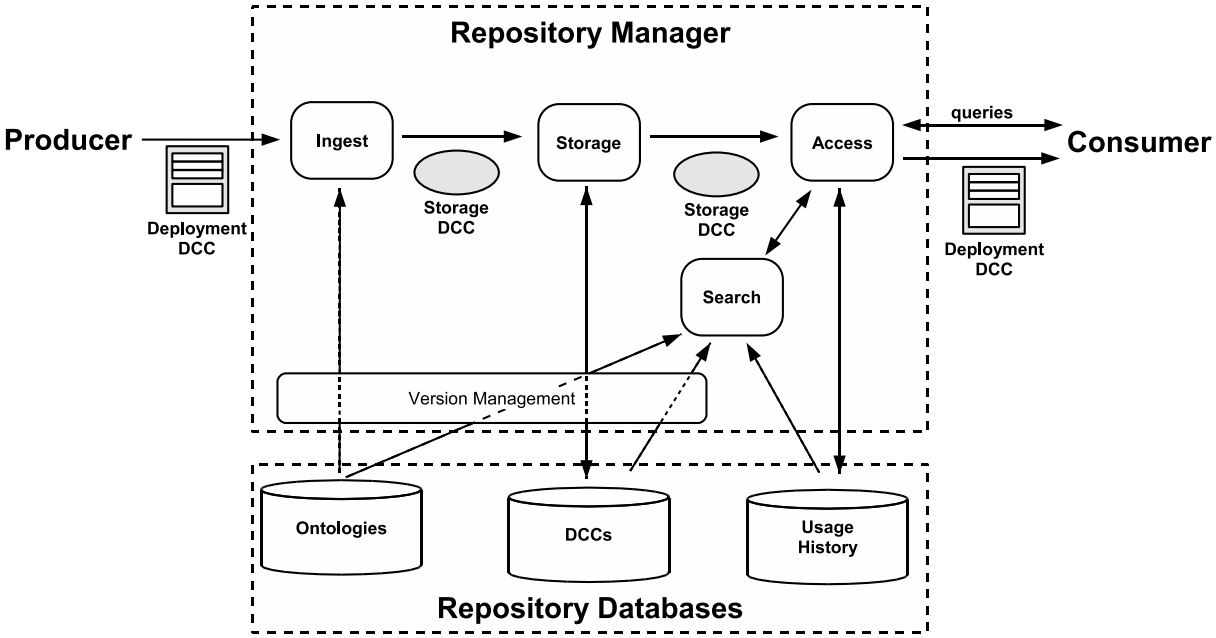

Figura 4.2: Repository management diagram.

`Consumers`. The Repository contains modules that perform these transformations, with the help of ontologies and usage history.

In more detail, the Deployment DCC format adopts open standards to represent and compress DCCs. However, this format is not adequate for storage purposes. The `Ingest` module of the repository manager has the task of processing each received Deployment DCC, and of transforming it into an internal format suitable for storage – the Storage DCC – which is then dispatched to the `Storage` module to be stored in the database. To construct the Storage DCC for subsequent discovery, the `Ingest` module must access the ontology repository to check the ontology terms used in the DCC (the role of ontologies is detailed in Section 4.5).

DCC retrieval works as follows. A `Consumer` searches for a DCC using queries through the `Access` module. This module dispatches the queries to the `Search` module, which accesses ontologies and usage history in the searching process, returning the description of possible candidates to the `Consumer`. Once the `Consumer` finds the desired DCCs, he/she/it requests them to the `Acess` module, which dispatches the request to the `Storage` module. The `Storage` module sends the appropriate Storage DCCs to the `Access` module, which converts them back into Deployment DCCs, sent to the `Consumer`.

As illustrated in Fig. 4.2, the `Version Management` module intervenes at all operations involving the DCCs and ontologies repositories. This will be detailed in Section 4.4.

From now on, in the text, the term "DCC" will be used without a qualifier (deployment

or storage) when the mention is applicable to both kinds of DCC. We will apply the "deployment" or "storage" qualification whenever such distinction is needed.

4.3 The Product Model – Digital Content Component

4.3.1 From Web Documents to Components

Initial efforts on the Semantic Web to provide a common syntax and semantics to exchange data were document-oriented. Therefore, XML *documents* are the basic syntactic support for data exchange. Annotations in RDF and OWL use pointers based on URIs connected with Web resources; the latter are not restricted to documents, but can be any resource referenced by an URI. However, to reference resource sub-parts, Web mechanisms (e.g., XPath and XPointer) are based on XML documents.

This document-oriented approach can be an obstacle when we consider the notion of *digital content* in a broad sense. To design an appropriate model that considers a plurality of schemes, formats and sizes, we will first analyse some characteristics of a "digital content" in its most general acception.

We point out two among the many possible usages of the term "content". First, content is something that is contained [1]. In this sense, a content requires a container, "something that contains". Second, content refers to the matter treated in some medium, like a document [1]. In this text we use the term content as a combination of both interpretations. More specifically, our work deals with *digital content*, which is the content represented by a stream of bytes, organized inside a structure, the "container".

A generic model to represent any digital content needs to consider a hierarchical containment structure, where content pieces can be used as components in a higher level content artifact, and so successively. This containment model has two main players: the *component* and the *composition*. Both component and composition are related with the task of composing, which derives from the Latin word *componere* that means "to put together". The component is a constituent part and the composition is the result of the composing task. The composition in one layer can be a component of the next layer.

Given these characteristics, we extend the document-oriented approach to a *component-oriented approach*. Our DCC is prepared to represent, manage and annotate this hierarchical containment structure. The DCC infrastructure is therefore based on two main players: component and composition. The component captures the necessity to decompose, individualize, pack and deploy digital artifact pieces. The composition captures the design facet, representing relationships between components and managing the unfolding of these relationships – e.g., their versioning and so on.

Two factors must be stressed concerning the originality of our work. From a component perspective, a DCC can contain any kind of digital artifact – data or code – in a transparent way. The composition perspective, on the other hand, allows creating progressively complex DCCs by composition, regardless of the content's nature. The two following subsections detail the component and composition perspectives of a DCC.

4.3.2 The Component Perspective of the DCC Model

A *Digital Content Component (DCC)* [66] is composed of four distinct subdivisions: (a) the content itself; (b) the declaration of a management structure that defines how components within a DCC relate to each other; (c) a specification of the DCC interfaces; (d) metadata to describe version, functionality, applicability, use restrictions, etc.

In the Deployment DCC format, subdivision (a) consists on the content in its original format, whereas the other three subdivisions are represented using Semantic Web standards. Subdivision (b) is in XML, subdivision (c) uses adapted versions of WSDL [19] and OWL-S [42] (an OWL Web service ontology), and subdivision (d) uses OWL [73].

A Storage DCC has no imposed format, since it is only seen within the repository, and thus does not impact interoperability. Therefore, the repository management system can represent the Storage DCC in any format that enables a reconstruction of a Deployment DCC. DCC producers and consumers deal only with Deployment DCCs, which are the Fluid Web face of DCCs. This clear assignment of responsibilities makes the infrastructure model flexible to be adapted to many contexts. Specialized systems can "export" their content encapsulating it inside Deployment DCCs on the fly: for example, an image database can be the source of Deployment DCCs containing images. This strategy can be compared to the success of Web pages, used as a mechanism by many applications to "export" their interfaces as dynamically generated pages, which results in an interoperable interface accessible by any Web client. Deployment DCCs, on the other hand, provide an interoperable Semantic Web content envelope, instead of an interoperable Web interface.

Both Deployment and Storage DCCs can be of two kinds – process and passive DCCs. A *process DCC* encapsulates any kind of process description that can be executed by a computer (e.g., workflows, sequences of instructions or plans). Non-process DCCs, named *passive DCCs*, consist of any other kind of content (e.g., a text or video file) and their interfaces define how this content can be accessed. Since passive components by definition cannot embed executable descriptions, these operations belong to a *Companion DCC* (e.g., a piece of music `M` stored in a passive DCC can be played by attaching `M` to an appropriate companion code). The association between a passive DCC and its Companion is achieved with help of semantic information given by terms related to a taxonomic ontology. Details of this mechanism are presented in Section 4.5.2.

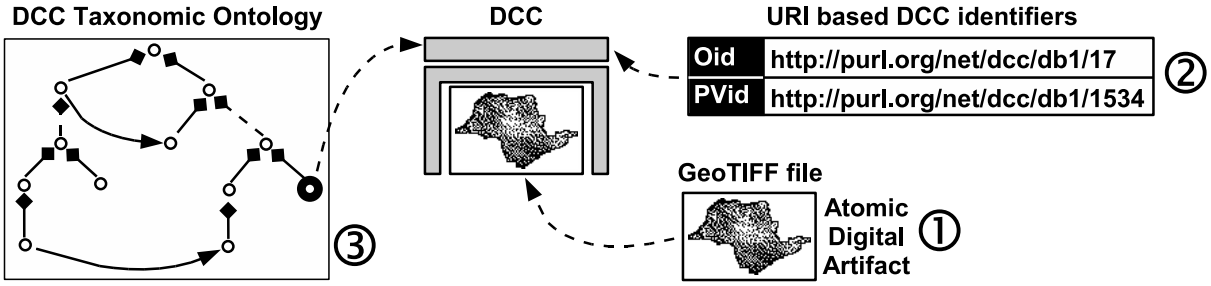

Figura 4.3: DCC minimum structure.

Any DCC can encapsulate a content, which is formed by other DCCs or by *Atomic Digital Artifacts (ADAs)*. An ADA is a piece of digital content that does not have a DCC structure. It is "atomic" under the DCC model point-of-view, since its content cannot be divided in reusable and annotatable (DCC) subparts.

To illustrate how a Deployment DCC is built, we will present a construction of a passive DCC containing a map, which will be subsequently used as part of the inputs to the wasp/moth biological control simulation. Fig. 4.3 illustrates the three basic elements to produce a Deployment DCC: ① the encapsulated content; ② two URI identifiers that support content versioning; ③ a reference to a DCC-type defined by the DCC taxonomic ontology. In the figure, the encapsulated content ① is a map containing the average rainfall distribution in one month in the state of São Paulo, Brazil (i.e., each pixel contains the average rainfall value for the corresponding region). The map is represented in GeoTIFF – a bitmap format that associates geographic coordinates to pixels. The two URI identifiers ② are responsible for identifying univocally the DCC through the Web. The first (`Oid`) denotes this DCC and the second (`PVId`) identifies its version. The role and management of both identifiers are detailed in Section 4.4. As will be seen, DCC versioning allows managing and controlling content versions in the Fluid Web.

There is a taxonomic ontology used to classify all DCC-types (e.g., `ImageDCC`, `MapDCC` – see Section 4.5.2). Each DCC references a DCC-type ③ defined in this ontology, which guides the way of how this DCC is treated by the infrastructure. Each DCC-type defined in the ontology is associated to a minimum interface specification, which is implemented by any DCC of the respective DCC-type. The DCC interface can be implicitly inferred through its DCC-type, or can be explicitly declared, if the DCC interface is an extension of the DCC-type's minimum interface.

4.3.3 The Composition Perspective of the DCC Model

As mentioned before, DCCs have a hierarchical containment structure. This enables a DCC to be a composition of ADAs and/or other DCCs. These other DCCs in turn can

also be a composition of ADAs and/or DCCs, and so on. This containment structure is controlled by two of the four DCC subdivisions mentioned in Section 4.3.2: subdivision (a) stores the ADAs and the DCCs contained in the DCC; subdivision (b) records data used to manage these ADAs and DCCs, e.g., the relationships among them. We proceed with our running example, using the DCC constructed in subsection 4.3.2 to illustrate the composition process to build a DCC application.

Roughly speaking, a Fluid Web application is developed by discovery and composition of deployment DCCs. It can be constructed using the following steps: (i) elicit requirements with help of experts and users; (ii) determine basic data and processes needed; (iii) search for appropriate process and passive DCCs to be reused/adapted using the semantic annotations in DCC metadata and interface subdivisions; (iv) construct new DCCs if needed; (v) create the application, which is materialized into a new DCC, by appropriate composition of reused and new DCCs.

Creating a DCC composition

Our biological control simulation is built in three composition layers. The first layer consists in creating a DCC that encapsulates a time series of (DCC) maps – i.e., this new DCC will contain a composition of the DCCs of Fig. 4.3. The second layer simulates the growth of a given tomato plant under certain weather conditions, where the maps of the first layer provide weather input. The third layer simulates the evolution of a tomato plantation, under the action of moth larvae and wasps, using the plant growth simulator of the second layer. Additionally, in this layer the results of simulation are confronted with ground data collected from a real tomato plantation.

The example illustrated in Figure 4.3 shows a single ADA packed inside a DCC. Now, let us consider that the user wants to pack twelve maps into a single DCC, each map representing the rainfall average of a month in a year. There are two ways to do that, as illustrated in Figure 4.4: (a) twelve GeoTIFF files (ADAs) are directly packed inside a DCC or; (b) each GeoTIFF file is packed inside a DCC that in turn will be packed inside a higher level DCC. In the latter case, each sub-DCC can be readily reused and versioned. These versions can be uniquely identified and controlled through the Web. In fact, each DCC map may come from a distinct repository. The figure also shows operations allowed in this new DCC – `getQuantity`, `getMap` – defined via OWL-S / WSDL.

The temporal map series DCC corresponds to our first composition layer. It is used as a part of the second layer, illustrated in Fig. 4.5. This second composition layer estimates how a tomato plant evolves, given certain weather conditions – defined by rainfall and solar radiation data – together with tomato plant geographic location. The result is visualized by means of an animation of the growth of a plant for the input conditions

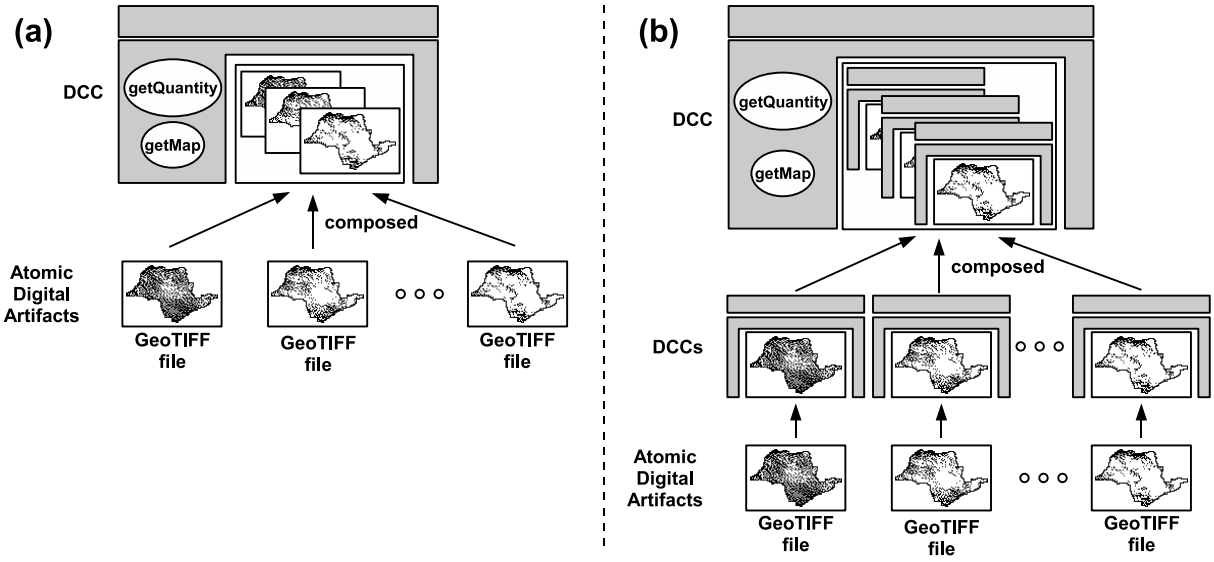

Figura 4.4: Two ways for packaging twelve maps inside a DCC.

provided. As can be seen in Fig. 4.5, this simulation was constructed via a composition of five DCCs. The figure also shows that the same kind of DCC construction (a map temporal series) can be used to envelop rainfall and solar radiation data. Many aspects are omitted to simplify the example.

Fig. 4.5 shows DCC interfaces whose operations are defined via OWL-S/WSDL. The Tomato Plant Growth Simulator DCC is the kernel of the simulation. Its internal structure organizes Java binary code classes, which implement the simulator software, and related files. Therefore, it is a process DCC. The Tomato Plant ImageSet DCC encloses a set of images representing consecutive growth stages of a tomato plant, and is a passive DCC. The Growth Calculation Spreadsheet contains a spreadsheet with equations to calculate the plant growth.

In order to configure and connect components, we used an extended version of the Magic House software [71] developed by us. Magic House is an educational authoring tool built over a framework named Anima, which is an infrastructure for managing and executing DCCs and their compositions. This infrastructure determines how DCCs will interact within a composition. Both systems are being used to support the teaching of sciences in schools in the city of Salvador, Brazil. In order to fully support DCC composition and communication, we have extended Magic House authoring capabilities beyond the educational domain.

We now summarize some key aspects of our model, to help understand our composition process, and the way we connect components (i.e., what researchers in software engineering call a software's architectural style [72]).

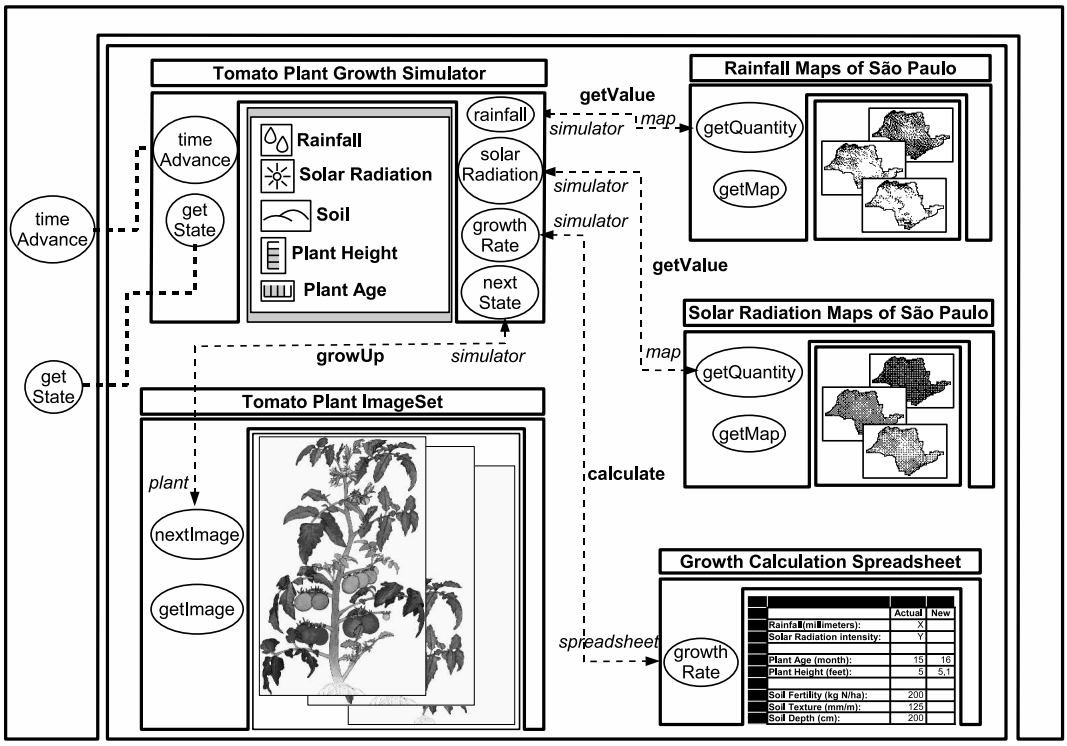

Figura 4.5: Composition to simulate tomato growth in São Paulo state.

In our architectural style, an application is formed from a composition of DCCs that work cooperatively exchanging messages, as a *choreography*, creating what we call choreography-based composition. A composition is formed by a set of *participants*. Each participant is defined by a set of observable behaviors, which together form a *role* of this participant in the composition. A *relationship* is the association of two roles for a purpose. In Fig. 4.5 a dashed line represents a relationship between participants (DCCs), with a boldface label indicating its name. Each label in italics represents the role played by the corresponding participant in the relationship. A given component may play several roles in a composition.In this example, each component plays a single role.

The composition is packed inside a higher level DCC whose interface operations connect with interface operations of its component DCCs. For instance, two operations of the `simulator` DCC – `timeAdvance` and `getState` – are connected to two operations of the higher level DCC. This means that, when one of these operations of the higher level DCC is invoked, the higher level DCC forwards their execution to the `simulator` DCC.

Executing a DCC Composition

In the example, application execution starts the first time the operation `timeAdvance` is executed. The `simulator` initializes its internal variables to start the process of calculating the growth of a tomato seedling in São Paulo state. Time in the simulation is divided in discrete units defined by cycles. An external application determines the next cycle sending a message to the `timeAdvance` operation of the higher level DCC. A cycle determines the growth of a tomato plant in a time period and runs as follows. The `simulator` sends requests to the `map` component, for a period of time and region, receiving the average rainfall and solar radiation for that period and region (here, an area within São Paulo state). Next, it requests that the `calculator` component computes the plant's state based on these and other parameters. This computation is used to feed the `simulator`'s `growthRate` process. Finally, the `simulator` sends a request to the `Plant image set` DCC to show the image corresponding to the calculated plant age/height/health state.

The composition of Fig. 4.5 corresponds to the second layer and was packed inside a higher level DCC. It is used as the component labeled ③ inside the third and last level composition, illustrated in Fig. 4.6. This final composition is a simulation of a tomato plantation in the São Paulo region, including moth larva attack and wasp biological control. Roughly speaking, the final result (labeled ⑧) shows a map that compares a map created from actual collected data on a plantation (labeled ⑥) with a map created by simulation (②) of tomato (③), moths (④) and wasps (⑤). The geographic space covered by the maps corresponds to a tomato plantation in São Paulo state.

Our simulators are based on a cellular automaton. A cellular automaton is a kind of system that is discrete in three dimensions: time, space and state [24]. Often used to produce simulations, the state of the next cycle is determined by two elements: the state of the previous cycle and the state of its neighbors. Our simulation adopts a deterministic cellular automaton model, where the space is divided in a fixed lattice [24], as can be seen in DCC labelled ②. Each cell in the lattice has an associated state, which in our simulation can be: an empty ground space, a tomato plant (with a state for each plant age), a tomato plant with moth eggs, a tomato plant with moth eggs parasitized by wasp eggs, a wasp or a moth. The behavior of each cell in the automaton ② is defined by an appropriate DCC, according to the cell's state. For example, if a cell state represents a tomato plant, then its behavior is defined by the tomato plant simulator DCC ③. There is a DCC to simulate the moth behavior ④ at all stages (egg, larva and adult) and another to simulate wasp behavior ⑤ in all stages. The behavior of a cell can be determined by a combination of DCCs – for example, the behavior of a cell whose state represents a tomato plant with moth eggs cell is determined by interaction between two DCCs.

The DCC model can accept different design strategies to build a Fluid Web appli-

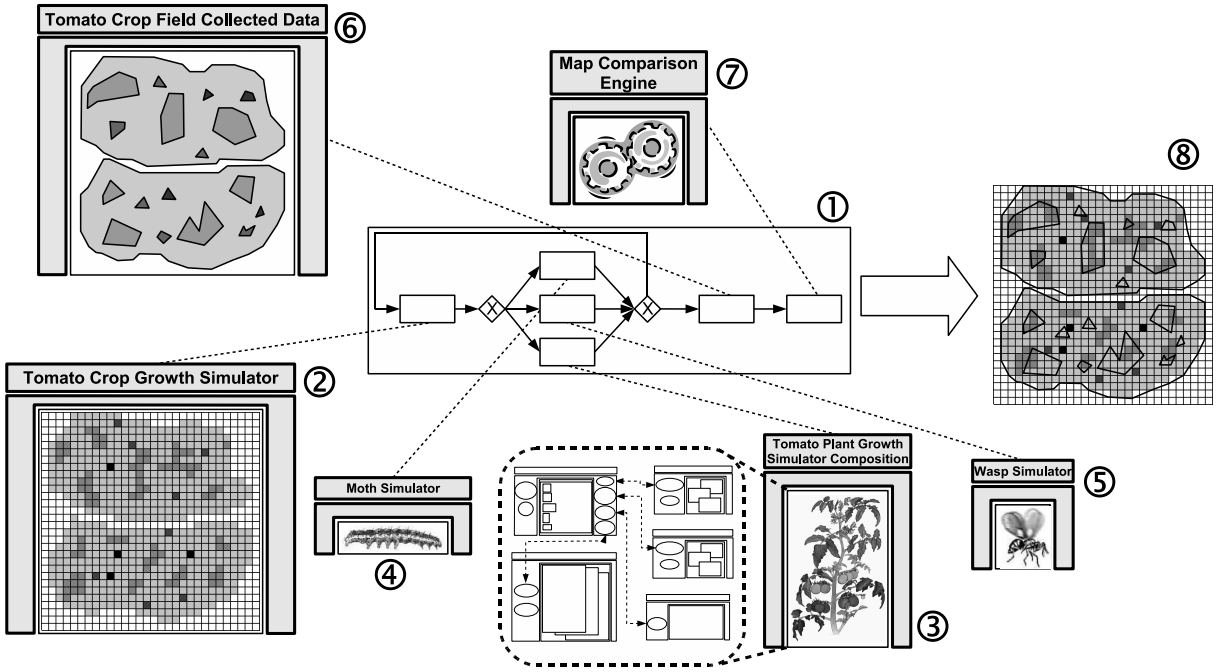

Figura 4.6: Composition to simulate wasp biological control in a tomato crop at São Paulo state.

cation. The simulation presented in Fig. 4.5 was based on the collaboration of DCCs, without a central coordinator of the whole process (Magic House architectural style). The application of Fig. 4.6, on the other hand, uses a style that relies on a workflow ① to act as a central coordinator. In this case, we assume that the WOODSS system was used. Developed by the Laboratory of Information Systems (LIS) at UNICAMP, WOODSS enables the capture of activities in agro-environmental planning to be stored as scientific workflows, which can be later edited, composed and re-executed. WOODSS has evolved to an extensible environment that supports specification, reuse and annotation of scientific workflows. These workflows can coordinate DCCs and can also be stored inside a DCC [44].

Here, the workflow is a part of the DCC that can be executed. Each activity in the workflow activates a DCC (see dashed lines). This workflow has a loop, where it visits each cell in ② and requests to the specific DCCs (③, ④ and ⑤) to provide data that will allow computing the next state of this cell, according to its state and its neighbors' states. The new state of the cell is then updated in DCC ②. When the simulation achieves its final state, the workflow leaves the loop and retrieves a map from a DCC ⑥ that represents data collected from a real tomato plantation. Each colored area in the map represents an observed feature in the tomato plantation: tomato plants, presence of moth eggs (infected or not by wasps eggs) and larvae, wasps and so on. The workflow engine dispatches this

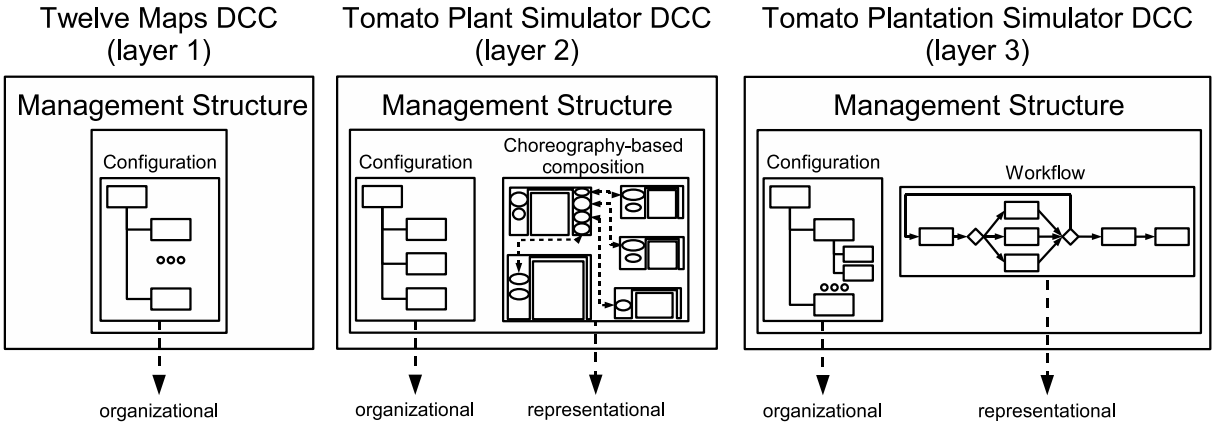

Figura 4.7: Diagram illustrating organizational/representational structures of three example DCCs.

map and the map that resulted from the cellular automaton simulation to a DCC that compares maps ⑦. The final result is a map that graphically presents the differences between the simulation and the field collected data.

4.3.4 Organizational × Representational Structure

The examples illustrated in the previous section show three strategies to compose DCCs to build higher level components and Web applications. In this section we will analyse how the DCC model deals with the differences between the strategies.

We use two kinds of data in the compositions – organizational and representational data. The former are related to the management of DCCs inside a composition, and are represented in the same way for any composition; representational data are related with specific design and execution tools, like Magic House and WOODSS – see Katz [37], where the same data distinctions appear. Organizational and representational data structures are stored inside DCC management subdivision (b), see Section 4.3.2. Fig. 4.7 shows how these structures are represented in the three example compositions.

The organizational structure is used by DCC management tools, e.g., DCC repository managers. This structure controls: (1) which ADAs and sub-DCCs compose a DCC; (2) how each ADA/sub-DCC is stored inside a DCC (e.g., as an internal stored resource or pointer to an external resource); (3) the relationships among sub-DCCs and ADAs that are essential to DCC management, such as dependencies. Additionally, ADAs and DCCs can be organized in a hierarchical structure similar to a file directory structure. Contextualized metadata can be associated to each ADA and sub-DCC.

The representational structure is used by specialized design and execution tools. The DCC that comprises twelve maps (left of Fig. 4.7) has only an organizational structure,

since it does not require specific concerns for design tools and execution environments. The tomato plant simulator DCC (center of the figure) stores in the representational structure the choreography-based composition, which is managed by the Magic House authoring tool. The tomato plantation simulator DCC (right of the figure) stores in the representational structure the workflow managed by WOODSS.

This model guarantees that the DCC infrastructure can manage any DCC composition, clearly separating the generic organizational structure from specificities of specialized design tools. It also supports transparent combinatin of any kind of content to produce a more complex content structure, that can be a passive or process DCC.

4.3.5 Handling Annotations in a Content-Oriented Approach

Ours is a content-oriented approach, as opposed to the typical Semantic Web document-oriented approach. For all content artifacts, even for those that are not Web documents, the DCC content-oriented approach provides: a *self describing structure* – associating semantic annotations to the content and transmitting both together through the Web; and a *multilevel annotation mechanism* – enabling annotation of digital content subparts.

In the typical Web document model, there are two approaches to associate semantic annotations to resources. In the first, the annotations are inserted inside the document (the content). The standard way to do that is restricted to XML and HTML documents. In the second approach, the annotations are stored in an independent resource and point to the annotated content. A problem here is how to know which annotations are associated to a given content, since the content does not point to the annotations. In a Fluid Web perspective, where the content travels through the Web and is replicated and adapted, it is essential to have annotations traveling together with the content. This is a principle of the DCC *self describing structure*, where annotations are a constituent part of a DCC (both in storage and deployment states) and always travel with it.

The DCC model defines two strategies to associate annotations to DCCs and ADAs: individually to a DCC in its metadata subdivision, and to sub-DCCs/ADAs inside a DCC using contextualized annotations in the organizational structure. We say that the second strategy is contextualized since the annotations do not pertain to the annotated sub-DCC/ADA, but are related to it in the context of the DCC in which they are inserted. These two strategies are part of the DCC *multilevel annotation* strategy. Compared to the typical Semantic Web strategy, which requires an XML or HTML document to reference and annotate its subparts, the DCC strategy enables to encapsulate other kinds of digital content composed by subparts, and to represent and annotate these subparts. Additionally, sub-DCC identifications and annotations can follow it throughout Fluid Web decomposition and mixing operations.

4.4 Version and Configuration Control and Management

Any system that manages a complex set of digital objects that evolves with time needs to consider two perspectives: version control and configuration management. Version control is not restricted to registering the evolution of each single object along time. A version is associated with a significant change that is semantically meaningful to the system [37] and any version change of an object must be related with version changes of associated objects. The control and management of *configurations* deals with the complex network of relationships among the objects. Versions and configurations are usually represented using orthogonal structures; however, their control and management are interlaced. In the following subsections we will first treat version control and then configuration management within and across DCC repositories.

4.4.1 Version Control: Objects and Configurations

Object versions often evolve together. This means that a new version of an object-A probably may be better combined with a new version of an object-B that evolved with it, rather than with the original object-B version. To guarantee the consistency between object versions, databases that support objects with multiple versions are organized in virtual sections of the database called *contexts*, or *configurations*. A configuration is a consistent subset of the database formed by one version of each object.

The Multiversion Database Model

Several versioning solutions have been proposed. The DCC version mechanism is based on the *multiversion database model* proposed by Cellary and Jomier [17] to manage DCC versions within and across repositories. A multiversion database manages a set of multiversion objects. As illustrated in Fig. 4.8, `DCC_X` is a multiversion object with three versions `v1`, `v1.1` and `v1.2`. It has an identification (`Oid`) that is independent of its version, as well as its physical version identification (`PVid`) [30]. Both identifiers are unique in the database.

The multiversion database can be analysed under two operational levels: logical level – presented to the client application – and physical level – managed by the DBMS [30]. At the logical level, the database is seen as a set of Database Versions (DBVs). Each DBV corresponds to a logical, nonversioned, view of the entire database, and is composed by logical versions of objects. This is illustrated in Fig. 4.9 – at any instant, a client application will have a single (monoversion) view of the database, corresponding to a given

Oid	
http://purl.org/net/dcc/db1/17	
PVid	**DCC**
http://purl.org/net/dcc/db1/1534	DCC_X_v1
http://purl.org/net/dcc/db1/2167	DCC_X_v1.1
http://lis.ic.unicamp.br/dccdb/1867	DCC_X_v1.2

Figura 4.8: Example of a multiversion object.

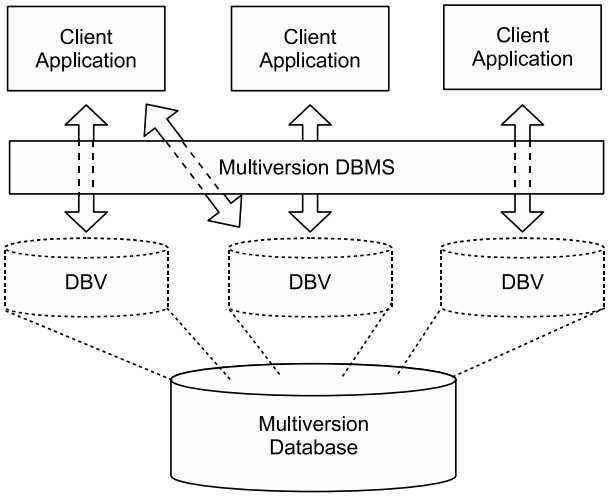

Figura 4.9: Diagram relating a multiversion database and its database versions.

DBV, and DBVs are handled independent from each other. The figure also shows that two client applications can share the same DBV. In particular, the leftmost application can alternate between two DBVs. However, at any time it will just handle one DBV (e.g., by explicitly defining its working database context). In this sense, a DBV can be considered to be itself a configuration. We adopt this flexible definition of a Database Version (DBV): it is a *monoversion partition* of the database, a workspace seen by an application or a user.

DBVs and versioned objects are managed in the DBMS by means of identifiers. Each DBV has a unique identifier, a `DBVid`, and its logical object versions have logical identifiers. Different from an object's physical version ID (`PVid`), the logical version ID (`LVid`) of an object is related to a specific DBV. A `LVid` is identified by the tuple (`DBVid`, `Oid`) – i.e., an object as "seen" through different monoversion workspaces. Each `LVid` is related to a `PVid`, and many `LVids` can share the same `PVid`. Applications work within a monoversion context and thus within a DBV, and deal with `LVids`.

Two kinds of update transactions exist: versioning and non-versioning [18, 36]. In a versioning transaction, object updates create new object versions (and thus new DBVs).

In a non-versioning transaction, updates do not create DBVs, and objects are updated in place. Section 4.4.2 has more details on these transactions.

Versions are created by deriving new DBVs from existing ones. To save space and increase the database management efficiency, even if a DBV is presented to the client as an independent set of objects (logical level), at the physical level a DBV–1.1, which derives from a DBV–1, stores only the differences against the original DBV–1. When DBV–1.1 is created, it is equal to DBV–1, thus all of its contents are implicitly derived from DBV–1. Each versioning update operation on DBV–1.1 stores only the differences compared to DBV–1. In other words, whenever an object is versioned in DBV–1.1, a new physical version is created. All other objects in DBV–1.1 can be retrieved via their `PVids` from DBV–1.

The multiversion model of [17] presumes a centralized version control, in a single multiversion database. The DCC infrastructure, on the other hand, is prepared to control and manage DCCs that are distributed and replicated through the Fluid Web. In many cases, it is important to control version information and relationships, even among distinct databases. To do that, the DCC model extended some aspects of the multiversion database model, adopting Semantic Web strategies, thereby enabling to share DCC version information through the Web.

Extending DBVs to the Web: ID becomes URI

Each DCC contains three identifiers: `PVid`, `Oid` and `DBVid`. Two DCCs containing the same `PVid` value represent exactly the same DCC. The `Oid` is shared by all versions of a DCC.

Since a DCC is designed to be transferred and replicated in DCC repositories in the Web, the `Oid` and `PVid` associated to the DCC become URIs. `DBVids` are also identified by URIs to enable to share and control context information between databases. URIs guarantee that the identifier will be unique not only inside a repository, but for all repositories, allowing to relate DCCs across the Web. The pair `(DBVid, Oid)` that corresponds to a logical identifier in a single database thus becomes `(DBV-URI, Oid-URI)`, a logical identifier for Web repositories.

The identifiers are constructed as follows. A URI ID prefix identifies the address of the DBMS that created it. For example, a DBMS–db1 accessed through the Web by the address "`http://purl.org/net/dcc/db1`" will use this to prefix any URI ID created inside it. Returning to Fig. 4.8, consider that it illustrates a multiversion object that represents a DCC X. It was created by DBMS–db1 (see its `Oid`) and its versions `v1` and `v1.1` were also created by this DBMS. Version `v1.2` is created by a DBMS–dccdb addressed by the URI "`http://lis.ic.unicamp.br/dccdb`". Notice that the goal of

the URI-prefix mechanism is to guarantee the uniqueness of identifiers through the Web. Even if it is possible to explore this mechanism to track DCC provenance, this would imply additional controls that are beyond the scope of this article.

We also point out that the triple `(Oid, DBVid, PVid)` is maintained in DCC replication operations. This follows the principle that replication occurs for performance reasons, but should not affect the contents of a DCC, and identifiers are needed for semantic version control. As will be seen in section 4.4.3, this is fundamental to maintain the consistency of DCC versions across the Web.

Our extension of the multiversion database mechanism meets two important needs of the Fluid Web: (i) it allows DCC versioning and replication across the Web; and (ii) it supports construction of configurations across multiple sites, since each DCC is identified uniquely via its URI, and furthermore it carries within it the information necessary to organize the configurations. The next subsection expands on these issues.

4.4.2 Configuration Management – Relationships within and across DCCs

In the DCC model, configurations can be seen as a graph where DCCs/ADAs are vertices and relationships among them are edges. The relationships are organized in a taxonomic ontology, detailed in Section 4.5.2. The following types of relationship are used in configuration management: *Version* – relates a DCC to its derived version; *Aggregation* – relates an aggregation with its sub-parts; *Exclusive aggregation* – if an ADA/DCC is `part-of` an exclusive aggregation, it cannot be part of other aggregations; *Equivalence* – relates two DCCs that represent the same real world object in distinct ways; *Abstraction* – defines a DCC as an abstraction of another; *Connection* – defines that two DCCs are connected by their interface; *Annotation* – defines that a DCC annotates another DCC; *Dependent* – any other kind of dependency between two DCCs and/or ADAs.

Configurations are managed in the DCC infrastructure in two scopes: inside DCCs and outside/across DCCs. Configuration management inside DCCs can be seen as an instance of DCC composition using versions, and was detailed in Section 4.3.3. It allows mixing of DCCs and ADAs, and is controlled in the DCC management subdivision. Configurations across DCCs relate independent DCCs without encapsulating them in a higher level composition. They correspond to an auxiliary structure stored in the repository together with the storage DCCs. Some types of relationships are used inside DCC configurations, others externally, and others in both scopes. *Aggregation*, *Exclusive aggregation* and *Connection* are used to represent DCC compositions inside a higher level DCC. *Version*, *Equivalence* and *Abstraction* are used in configuration management across DCCs. *Annotation* and *Dependent* are used in both scopes.

Equivalence and *Abstraction* relationships help to control the DCC production cycle and consecutive modifications. Indeed, the construction of a DCC passes through many phases before its implementation, such as requirements elicitation and design. Such phases also can produce digital content artifacts (e.g., diagrams and tables), which can be related with artifacts in the next phase. For example, a diagram represented inside a DCC can be an *Abstraction* of a software component represented inside another (process) DCC, or of a data file represented inside another (passive) DCC. The *Equivalence* relationship can be used in a similar way. For example, a DCC containing a source code of a software component can be *Equivalent* to a DCC containing this software in binary format.

The configuration management infrastructure is used to determine actions to be taken when a DCC version changes. For example, if the user modifies a DCC containing a diagram that is an abstraction of another DCC, the design application can report to the user that the related "DCC needs to be updated". Some actions can be automatically executed – e.g., a modification in a DCC containing source code triggers its compilation and production of a new related DCC, containing the corresponding binary code.

4.4.3 Version and Configuration Management in the Fluid Web

This section illustrates, continuing our running example, how DCC version and configuration strategies are explored in a Fluid Web collaborative work. The example uses simple numbers instead of URIs to represent version ids to simplify the discussion.

This example is developed in two stages. In the first, collaboration occurs only in the beginning: a scientist starts an experiment by *reusing* another's DCCs. Afterwards, they proceed independently, modifying and versioning their objects through different DBVs. The second stage concerns a durable collaboration, in which scientists *share a workspace*. This requires *version synchronization*, which is not dealt with in the original DBV model. In each case, we start by considering a local (one repository) work environment, and proceed to a multiple repository situation.

Collaboration by version reuse

Suppose that a scientist (John) has launched the DCC application presented in Section 4.3.3, illustrated in Fig. 4.5, whose purpose is to simulate the growth of a tomato plant. Consider now that this DCC is stored in a multiversion DCC repository and has a single version, associated to a `DBV1`. Another scientist (Mary) sharing the same multiversion repository wants to create a new version of this application, to examine alternative scenarios, without affecting the existing DCCs. This is done by a two step operation named *versioning transaction*, which implies first copying `DBV1` into a new `DBV2`, and then

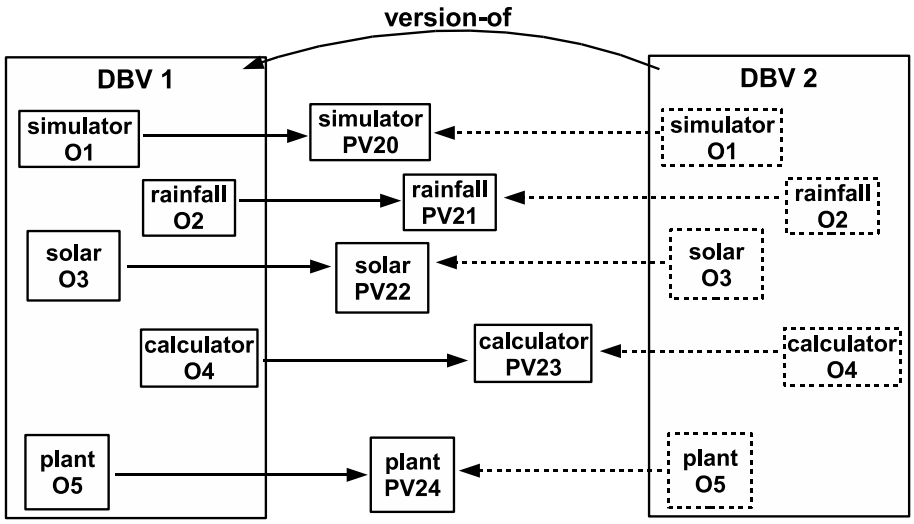

Figura 4.10: Two DBVs presenting DCCs of the plant simulator composition.

changing `DBV2` (the second DBV is said to be derived from the first).

Fig. 4.10 presents a diagram illustrating the first step, the initial derivation. Many aspects are ommitted to simplify the example. The figure shows the physical DCC versions in the middle, with one DBV at each side. Each physical DCC version has a `PVid` that is represented by a number prefixed by `PV`. `DBV1` represents logical DCC versions pointing to the corresponding physical DCC version. Each logical DCC version has an `Oid` that is symbolically represented by a number prefixed by `O`. In `DBV2` these references are represented using dashed lines, to indicate that `DBV2` inherits these references from `DBV1`.

After creating `DBV2`, the second step in the versioning transaction is to apply the modifications in the new DBV. In the example, these modifications will enable to simulate a coffee plant growth instead of a tomato plant. To do that, Mary modifies the `calculator` spreadsheet DCC adapting the formulae to simulate coffee plant growth and the `plant` image set DCC, replacing the tomato plant images by coffee plant images. These modifications produce new DCC versions derived from the previous ones. Fig. 4.11 illustrates the changes in the database. Two new physical DCC versions of `calculator` and `plant` are created, derived from their previous versions, and two corresponding logical DCC versions appear in `DBV2`, pointing to the appropriate physical versions. For these two DCCs, dashed lines are replaced by continuous lines in `DBV2` to indicate that the implicitly inherited references are replaced by explicit references to new physical versions.

Now, consider that John, who created the original DCC application, also made some modifications in the `simulator` DCC, using `DBV1`. Unlike other versioning models, the multiversion database model allows updates without forcing versioning – the so-called *non-*

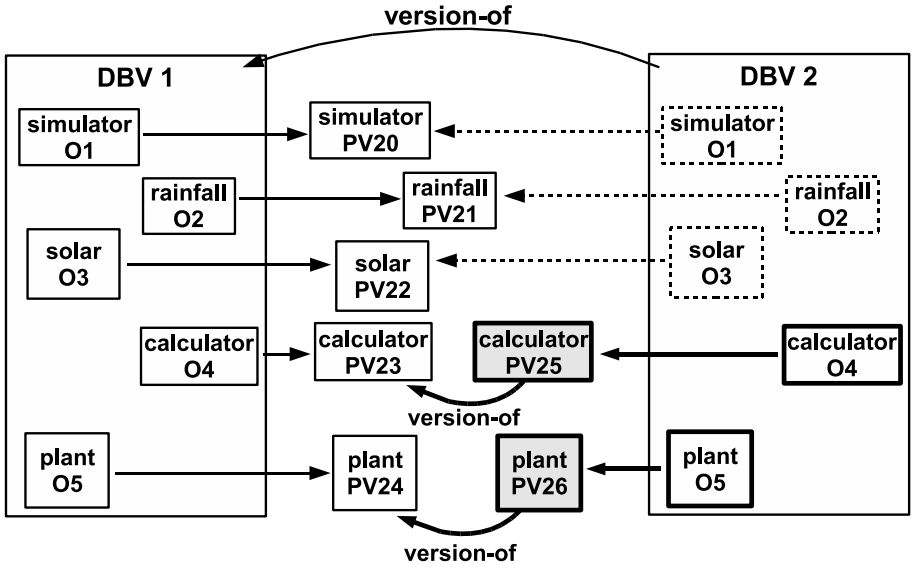

Figura 4.11: New calculator and plant DCCs are created in DBV2.

versioning transactions. In this case, however, all other DBVs that share the same physical version would be aware of this update, and lose the previous state. In order to avoid this problem, the multiversion database mechanism allows the creation of a new physical version of the updated entity – in our example, the `simulator` DCC – independent from the original physical entity. In other words, `DBV2` will not share the modifications in this DCC, and each DBV will now explicitly point to different independent physical storage units, as illustrated in Fig. 4.12.

If, instead, John had wanted to do a *versioning transaction update* to the `simulator` DCC, he would have to follow the steps already explained when `DBV2` was created. First, `DBV1` would be copied into a new database version `DBV3`; and next, the appropriate objects in `DBV3` would be updated, with versioning links established when appropriate.

Figures 4.11 and 4.12 informally illustrate the differences between versioning and non-versioning transactions. In the first case, the version mechanism requires a new DBV to be created (e.g., `DBV2` is a version of `DBV1`). In the second case, this does not occur – i.e., `DBV1` is not versioned, but its state has changed.

The example has a single local repository. Consider now a Fluid Web environment, where a third scientist, George, is located in another site with its own local repository. George wants to reuse Mary's DCCs. In this case, he starts exactly as in the standard multiversion mechanism – i.e., he executes a versioning transaction that creates a new `DBV3` from `DBV2`. However, instead of being located at site S1, `DBV3` is created at site S2 by copying all DCCs in `DBV2`; this copy maintains the same physical `PVids`. The assumption here is that the advantages of implicit references (savings in storage) disappear when

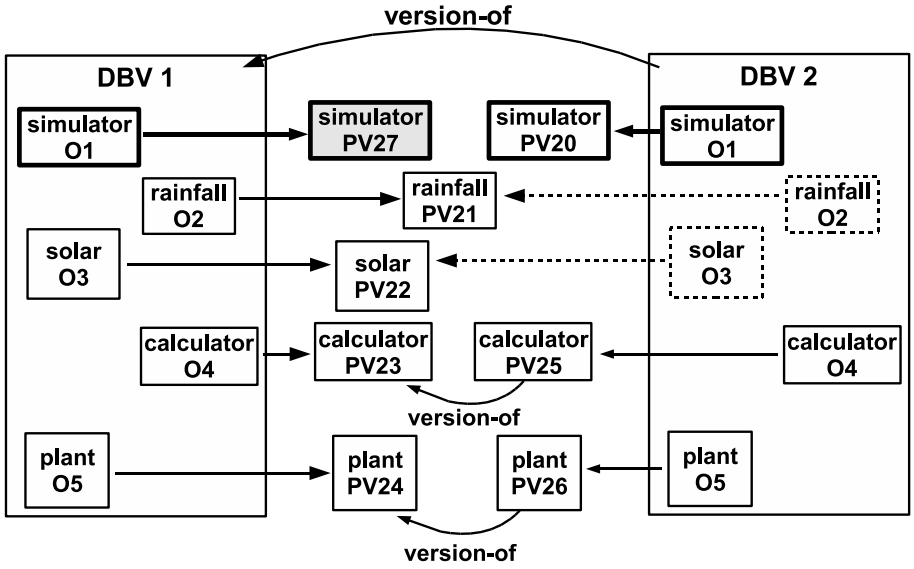

Figura 4.12: The simulator DCC is updated in `DBV1`.

multiple sites are involved and no durable cooperation is intended. George does not want to share DCCs with Mary, just to start from the same DCC configuration. This scenario is illustrated in Figure 4.13.a.

There is no need for version synchronization, because the scientists will work independently, and the versions are handled exactly as in the single repository scenario, thanks to the URI properties. We point out that several other alternatives exist – e.g., copying a DCC only upon its versioning – but full DBV replication is the solution with the smallest cost in version integrity control.

Durable collaboration via version synchronization

Up to now, we have discussed the first collaboration scenario (by reuse). In the second collaboration scenario, the scientists will work together, exchanging and comparing versions. For a local (single) repository, this is very simple: at any time, the scientists just have to declare which DBV they want to work with. This is in fact the standard collaborative scenario in the multiversion database model (see the two client applications sharing a DBV in Fig. 4.9).

Let us expand this example to a Fluid Web environment. Consider again Mary and John at site S1, and George at S2, but now George wants a long term durable cooperation with Mary. This is achieved by sharing the same DBV (i.e., the same workspace). Mary and George can each create their own versions, but to cooperate they must explicitly state

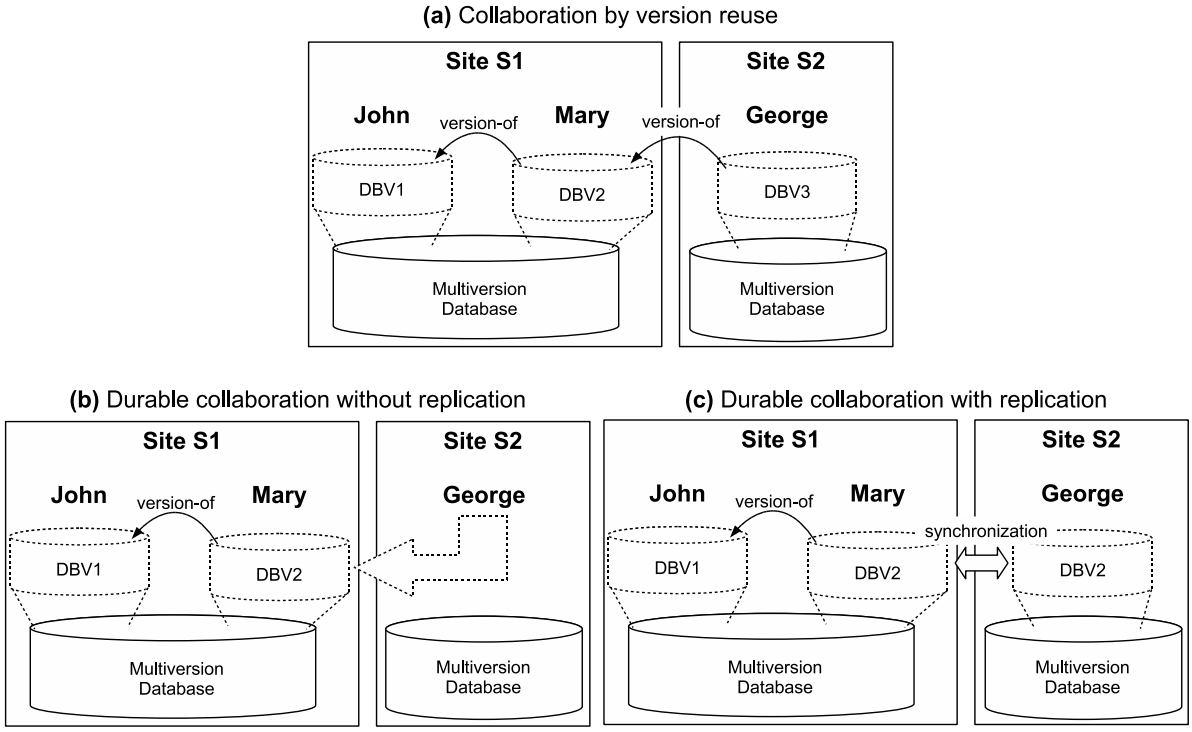

Figura 4.13: Collaboration between two sites.

the DBV they want to maintain in common.

This kind of collaboration implies version sharing. We distinguish two implementation policies: without any replication or with full replication. In the first case, all DCCs in `DBV2` will remain in the repository at site S1, and no real distribution of DCCs will occur. George will always need to access Mary's site to work at `DBV2`. This scenario is depicted in Figure 4.13.b.

Full replication, instead, implies that George replicates `DBV2` into site S2 for performance reasons. However, since this must be *the same DBV*, it does not create a new version. Rather, all version identifiers (DBVid and PVid) are copied into site S2. This means, however, that non-versioning transactions on site S1 must be reflected at S2 and vice-versa. This is performed via synchronization and is illustrated in Figure 4.13.c.

We consider only one-way synchronizations, where a database `B` periodically checks for updates in a database `A` and downloads the updated DCCs. Since the synchronization is one-way, database `B` does not upload its own updates to database `A`. Our goal is not to develop a full distributed versioning mechanism. Rather, we are only interested in providing basic support for versioning of DCCs on the Web. In the future, this can be extended to more complex protocols.

Synchronization using our one-way protocol occurs as follows. Suppose that George's

LVid		PVid	DCC
DBVid	Oid		
DBV2	O1	PV20	simulator
	O2	PV21	rainfall
	O3	PV22	solar
	O4	PV23	calculator
	O5	PV24	plant

Figura 4.14: DCCs replicated by George from DBV2, before versioning.

copy of DBV2 is the one right before Mary started creating DCC versions – i.e., the one depicted in Figure 4.10. Figure 4.14 shows George's table of the replicated DCCs. George requests a synchronization operation. His local multiversion manager starts a message exchange with Mary's multiversion manager. It sends to the remote manager a list containing the pairs (DBV2, Oid) identifiers of George's five DCC logical versions, and asks for the associated physical version identifiers. The answer will be: *simulator*({DBV2,O1},{PV20}); *rainfall*({DBV2,O2},{PV21}); *solar*({DBV2,O3},{PV22}); **calculator**({DBV2,O4},{PV25}); **plant**({DBV2,O5},{PV26}).

Site S2's multiversion manager compares the received PVids with the PVids stored in the local database (i.e., those depicted in Figure 4.14) and verifies that the calculator and the plant DCCs have been modified. Thus, it requests copies of these two DCCs and replicates them in the local database.

4.5 The Role of Ontologies

Ontologies can be classified, according to their focus, in two kinds [20]: descriptive and taxonomic. The former resemble database schemas. The concepts are interconnected by many kinds of semantic associations, and the purpose is to represent the intended domain as much as possible. The latter are used as a referential vocabulary. Their structure organizes terms into generalization/specialization hierarchies, and semantic links to express synonymy, composition, and so on. In this section we point out the importance of both kinds of ontology to provide a semantic bridge across the database nodes of the Fluid Web infrastructure.

4.5.1 Descriptive Ontologies

In the Deployment DCC format, descriptive ontologies work as a standard schema to define metadata and interfaces. Since these schemas are defined using OWL, they can be

extended and the semantics of the schemas and their extensions can be shared through the Web. Even if in OWL there is not a clear distinction of a schema and an instance that follows the schema, we will use the term OWL schema to refer to the definition of classes and properties. Classes and properties serve as a schema for DCC metadata and interface subdivisions, whose descriptions are instances of these classes. The role played by an OWL schema is the same as that of RDF schema.

Fig. 4.15 shows a partial representation of the passive DCC containing the map time series, whose construction was illustrated in Fig. 4.4. Both metadata (in OWL, on top) and interface (in OWL-S, displayed around the organization structure) are presented using a simplified version of RDF-like Directed Labelled Graph (DLG). Metadata and interface parameters are associated with ontological terms. There is a standard OWL schema to define the minimum DCC metadata properties – `Oid` and `PVid` – and other usual properties, like title and description. The OWL schema of the interface is defined using OWL-S.

The interface subdivision presents operations using the OWL-S `ServiceModel` class hierarchy [42]. In the example, the interface defines two operations (atomic processes in OWL-S): `getQuantity` and `getMap`. The `getQuantity` operation returns the value of a pixel inside a map image, given parameters `month` and pixel `coordinate`. The `getMap` operation returns a map image for a given `month` parameter. These atomic processes, which receive one input message (comprising all input values) and return one output message, correspond to WSDL request-response operations [42]. To simplify the explanation we will use the same names of OWL-S atomic processes to refer to WSDL related operations.

An interface in passive DCCs denotes which operations can be applied over it (e.g., a video content can be *played*). We call the set of operations associated with a passive content type to be its "*potential functionality*", in the sense that their implementation is not part of the content. In a process DCC, instead, we have a "*provided functionality*" inherent to any process description module (e.g., a software component, a workflow specification) – since a piece of software is by definition executable.

Functionality declaration (potential or provided) in DCCs is used to help find DCCs that meet specific requirements – e.g., in a composition process [68]. This functionality-based search complements a metadata-based search in two ways: it can find additional DCCs – not discovered by the metadata-based search – for a given necessity, and it produces more consistent search results, since interfaces can be checked to verify DCC adequacy to given requirements.

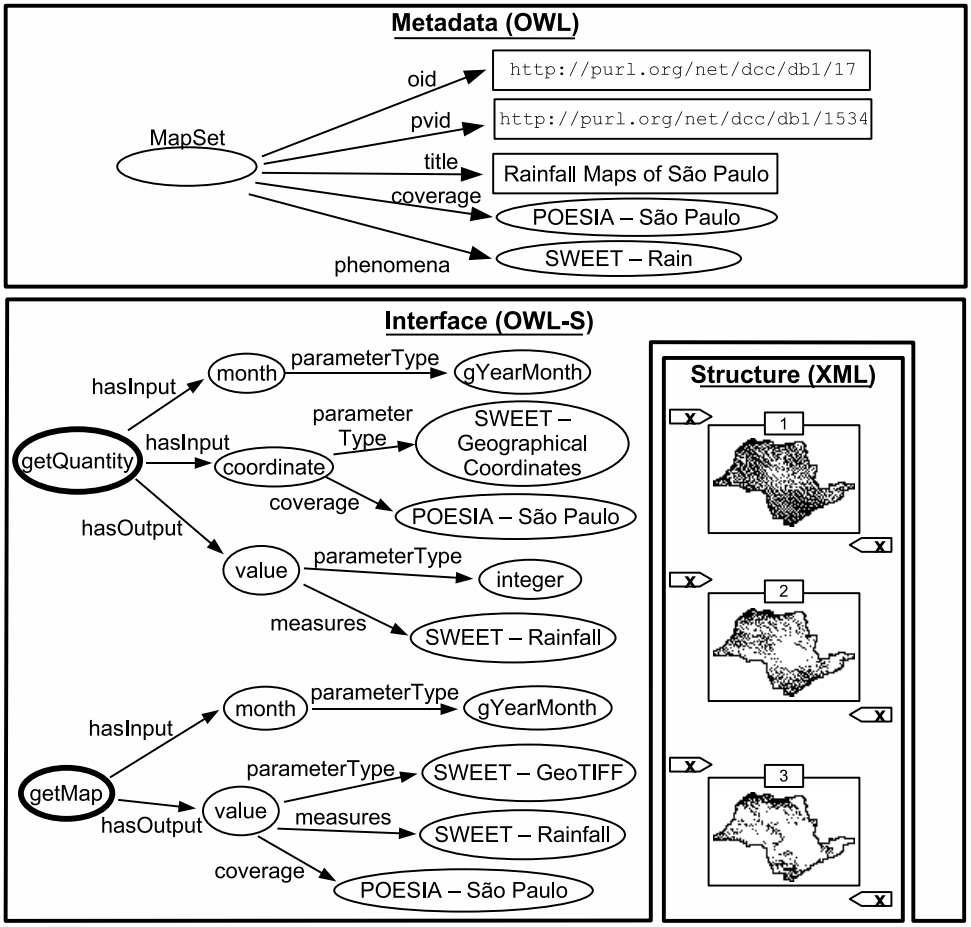

Figura 4.15: Deployment rainfall map content component representation.

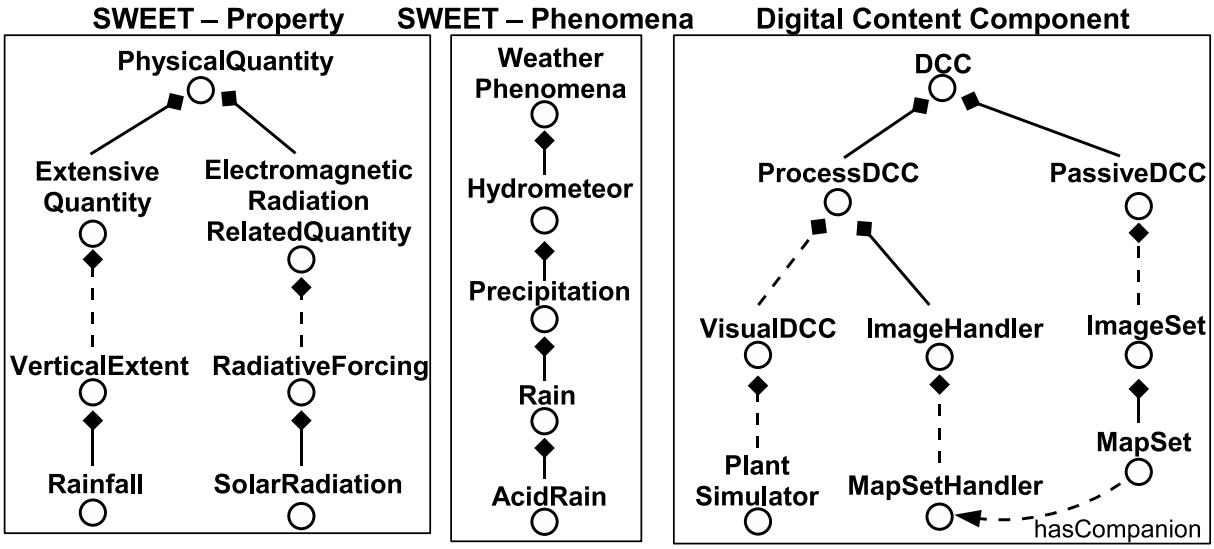

Figura 4.16: Ontologies used by rainfall map component.

4.5.2 Taxonomic Ontologies

DCC Metadata and Interface Specification

Taxonomic ontologies are useful in information sharing activities [20]. We adopt them in DCCs to disambiguate the meaning of DCC metadata and interface specification. More specifically, we postulate the need for specific ontologies that define valid kinds of DCC and of terms used in defining DCC interfaces. Fig. 4.16 shows diagrams that represent parts of two taxonomic ontologies, used by our examples, and whose hierarchical relations are explored in DCC semantic relationships and search procedures. White-filled circles represent classes. Lines with a diamond in one extremity represent subclass relationships, e.g. `Rain` is subclass of `Precipitation`. Dashed lines indicate that some intervening nodes were ommitted for simplicity.

Our running example concerns managing, creating and reusing content for applications that involve geographically related data. DCC discovery and reuse require domain semantics – in this case, the taxonomic ontology called SWEET – Semantic Web for Earth and Environmental Terminology [61]. Fig. 4.16 shows two fragments of SWEET. The fragment at the center concerns a taxonomic hierarchy for the Rain phenomenon, while the left fragment describes physical measurements. The right fragment is part of an ontology we constructed to classify DCCs according to their functionality. Each class in this ontology corresponds to a DCC type, and each DCC is an instance of a class.

The metadata subdivision of the DCC presented in Fig. 4.15 references the DCC

ontology presented in Fig. 4.16. It shows that the DCC is an instance of the `MapSet` class, with five property values: `oid`, `pvid`, `title`, `coverage` and `phenomena`. The values of `phenomena` and `coverage` are respectively related to SWEET and to the POESIA spatial ontology [27]. The latter is a spatial ontology specific to Brazilian territorial organization.

The operations of the rainfall DCC illustrate how the interface descriptions can be connected with taxonomic ontologies. Let us consider operation `getQuantity`; the parameters of `getMap` can be analysed the same way. Following the OWL-S model to describe processes, each process parameter has a `parameterType` which specifies the class or datatype for that parameter [42]. Notice that many ontologies may be needed to properly specify a parameter. For instance, the `coordinate` input parameter has a type description associated with SWEET, but its domain is defined by the `coverage` property, in POESIA, here denoting that the only valid coordinates accepted are those from within the state of São Paulo. The output parameter of the `getQuantity` operation is an integer value. The additional `measures` property of the SWEET ontology defines the nature of the measured value. Notice that we extended the OWL-S schema to enhance parameter description with additional semantics. OWL-S specifies the need for type characterization (`parameterType`), which we improved by adding semantic parameter descriptors (e.g., coverage, measures).

The use of ontologies to describe DCC metadata and interface can be explored in many ways in DCC discovery and composition. Some of the benefits, detailed in [68] are: (i) they organize DCCs in taxonomic trees that can be navigated in the DCC discovery process; (ii) ontology concepts are used to help query processing, to disambiguate terms and find synonyms; (iii) ontological relationships are used to rank DCCs based on their similarity with the DCC being handled; (iv) they enable a more semantical interface matching.

Associating Passive DCCs to their Companions

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*, already mentioned in Section 4.3.2. The Companion DCC lends its operations to a passive DCC in a way that is transparent to composition designers. The choice of the appropriate Companion for a passive DCC is context sensitive, and is determined by an execution engine, when this passive DCC is used. This allows a homogeneous treatment of passive and process DCCs from the user's perspective.

Fig. 4.17 shows an example of a Passive DCC association with a Companion DCC based on its content type (content-type driven execution). *Taxonomic ontologies* play a

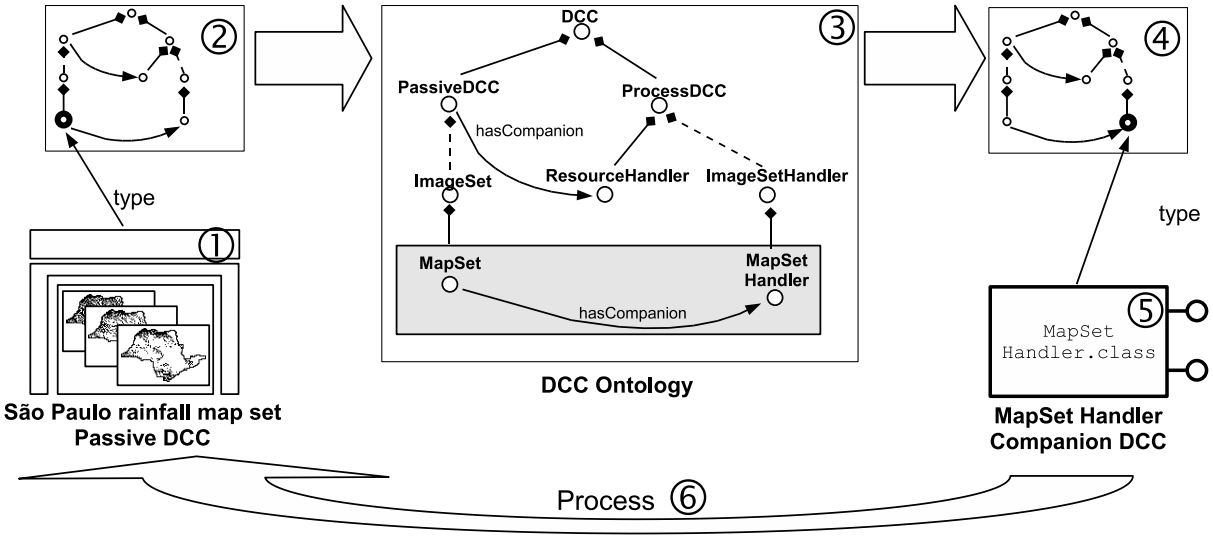

Figura 4.17: DCC content-type driven execution diagram.

central role in this matching process. The taxonomic ontology illustrated in the center of the figure is repeated from Fig. 4.16. As shown in the figure, any DCC is associated with a DCC type defined in the ontology. The association is carried out through an explicit reference in a DCC's metadata subdivision, coded in OWL. Arrows that represent a property `hasCompanion` relate two nodes, indicating that a given DCC type is processed by the indicated Companion DCC.

In order to associate a passive DCC with its companion, an execution engine follows a cycle indicated by the numbers ① to ⑥, illustrated in Fig. 4.17. Consider the passive DCC of Fig. 4.15, here reproduced as ① in the figure. It is related to the `MapSet` DCC type in the ontology ②; the ontology records that the companion to a `MapSet` DCC is a `MapSetHandler` DCC ③. Suppose that the execution engine asks the DCC repository manager for a DCC of this type. The selected DCC is retrieved ⑤ and connected to the passive DCC, which it will process ⑥.

This taxonomic ontology-based execution enables to explore some advantages provided by the DCC type inheritance hierarchy. In many cases, there is not a Companion DCC to process a specific Passive DCC type. Then, the engine retrieves the companion that is most suitable, given encompass relationships.

Classifying Relationships

As presented in Section 4.4.2, some relationships among DCCs are represented outside the DCCs. The challenge is to represent these relationships in such a way that they are

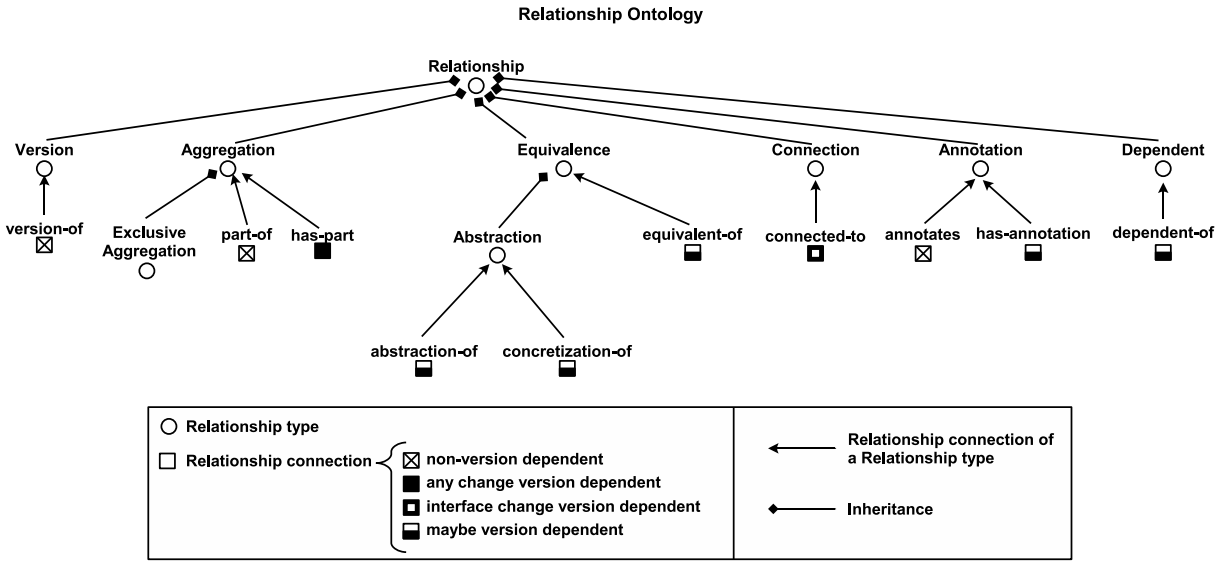

Figura 4.18: Diagram of the Relationship Taxonomy.

maintained when DCCs migrate. Since DCCs are identified by URIs, the relationships persist regardless of which node requested a DCC. We defined a DCC relationship taxonomic ontology, illustrated in Fig. 4.18, to represent the variety of possible relationships between DCCs in a unified and extensible way, This relationship taxonomy is used for all relationships within and across DCCs.

In the ontology, white-filled circles represent relationship classes. Lines with a diamond in one extremity represent subclass relationships. Squares represent properties and arrows connect properties to their classes. An instance of a relationship class defines a relationship among two or more ADAs or DCCs. The values of the properties associated to the class, which are depicted in this figure, are pointers to the DCCs that participate in the relationship. For example, if a DCC-x aggregates a DCC-y, then there is an instance of the class `Aggregation`, whose property `has-part` points to DCC-y and whose property `part-of` points to DCC-x.

The properties in the taxonomy are classified according to four types: (i) non-version dependent, (ii) any change version dependent, (iii) interface change version dependent and (iv) maybe version dependent. This classification defines what happens with a DCC-y, which has a relationship with a DCC-x, when DCC-x changes its version. DCC-y can be affected in two ways: if any change has been made in the DCC-x (type ii), or if any change has been made in the DCC-x interface (type iii) – this case does not consider any other DCC change. The classification (iv) declares that is not possible to determine previously whether DCC-y is affected. This classification is useful when the DCC management tool needs to evaluate the impact of a DCC version change in a DBV.

4.6 Comparison to Related Work

In many aspects our work relates disjoint research efforts aimed to solve complementary problems, combining them using a Semantic Web "glue". It addresses research on: content packing/deployment/reuse, software architecture and reuse, version control, configuration management and digital libraries.

4.6.1 Content Packing/Deployment/Reuse

Content packaging, deployment and reuse was the start point of this work. Many recent projects deal with these questions following parallel tracks to solve analogous problems in different domains, like: IMS Content Packaging (IMS CP) [74], for education, MPEG-21 [13], for multimedia, Reusable Asset Specification (RAS) [54], for software development, and OAIS XML Formated Data Unit (XFDU) [16], for digital libraries. Moreover, distributed software component technologies have similar concerns related with packaging and deployment of software components [28] – e.g., Enterprise Java Beans (EJB) [75], Microsoft COM+ and CORBA Component Model (CCM) [52]. All of these projects/standards define package formats, to deploy their content, following the same basic structure divided in two parts: (1) a package container, usually based in the popular ZIP compressed file format; (2) a XML based manifest file stored inside the package, which complements the information about the packed content with data not provided by the package container (ZIP) – e.g., related to organization, dependencies or metadata.

Fig. 4.19 graphically synthesizes the basis of each package format. Shaded elements represent reuse package formats comprising the container and manifest files. Dashed circles represent formats without a final specification. Arrows from top to bottom represent schema derivations used in manifest files, and arrows from bottom to top represent container format derivations.

As can be seen in the figure, all reuse initiatives define a package container based on ZIP. MPEG21 has not defined its package container yet. RAS accepts an alternative format based on a CVS directory structure. OAIS XFDU is still under development, but is analysing the possibility to use other optional formats, a content inserted inside an XML document (XML inline) and a format based in Multipurpose Internet Mail Extensions (MIME), a format that enables packing and attaching files to mail. All reuse initiatives use XML to define their manifest structure. CCM and Microsoft COM+ are derived from the the Open Software Description Format (OSD) [79] and OAIS XFDU is considering the Metadata Encoding & Transmission Standard (METS) [77]. MPEG-21 defines a specific section, the Digital Item Declaration (DID) [14], which works as a manifest structure.

The Deployment DCC format has its conception derived from content package and reuse standards, since program code reuse proposals are neither concerned with detailed

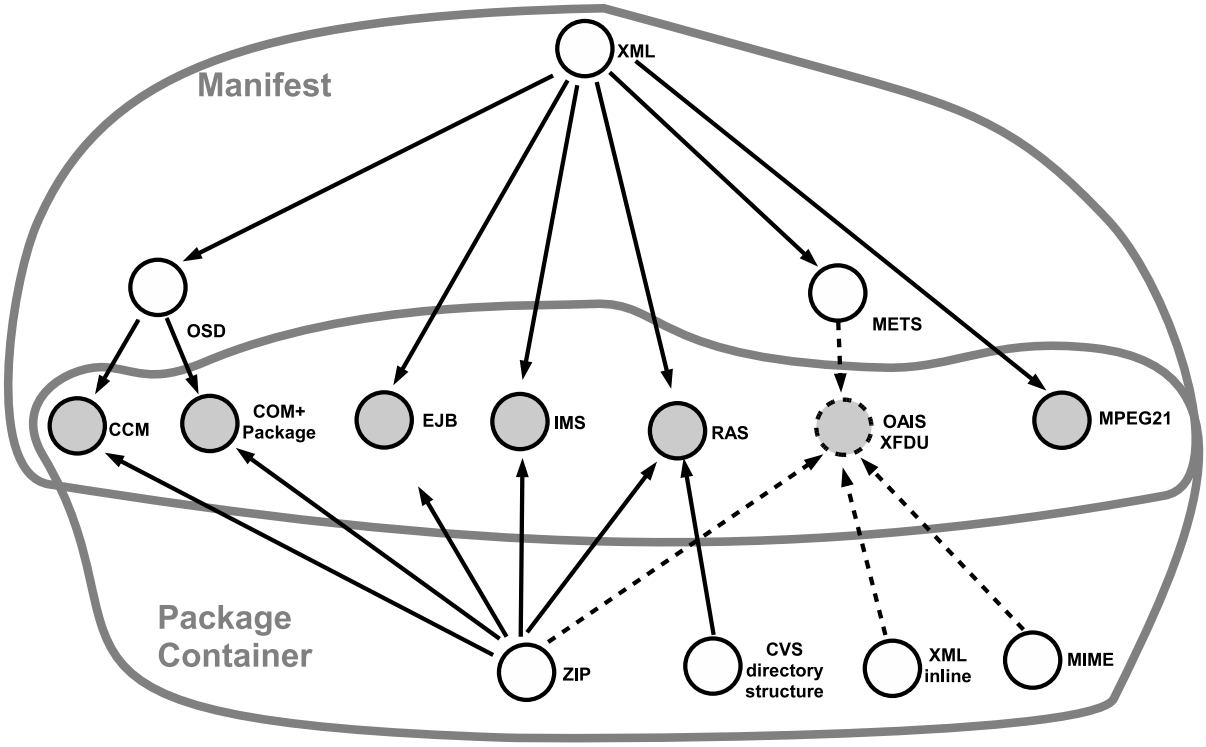

Figura 4.19: Diagram tracing basis and derivations of package formats for reuse approaches.

structure representation, nor with metadata. Each Deployment DCC is stored inside a ZIP file. The DCC management structure subdivision (ii) (see Section 4.3.2) is a generalization of the content reuse manifest file structures (RAS, MPEG-21 DID, METS and IMS CP). Even if the manifest structures are organized in different ways, they deal with the same problem and the same basic concepts. Therefore, it is possible to map the main structural elements of one manifest to another, if the particularities of each reused content are disregarded. The UML model behind each one served to help comparing and defining this unifying model. RAS and MPEG-21 DID [7] structures are derived from an UML model, and the Model Driven Architecture (MDA) [45] principles were used to extract an UML model from METS and IMS CP.

The reused standards manifest structure mix data used to manage content artifacts with domain specific data, constraining their usage to the original domain. The DCC management structure is divided in the organizational structure – which maintains generic data applicable to any content – and representational structure – which represents domain specific considerations (see Section 4.3.4). The relationship taxonomic ontology (see Section 4.5.2), used by the DCC management structure, incorporates and classifies the relationship types found in the manifest structures.

In contrast with the other proposed standards, the DCC model upgrades the "*package*" structure to a "*component*" structure. In fact, our work takes advantage of the mature software engineering research on software components, extending it to any kind of digital content. This upgrade is not just a new package format, but supports a novel approach to produce applications/content, and is one of the main contributions of this work. In databases, complex objects can be constructed by composition of other objects; in software development, software can be constructed by composition of software components. Data composition in databases and component combination in software engineering are separate mechanisms that have been extensively researched. Our DCC composition process, however, is a new notion: using one single mechanism, it allows constructing complex objects (in a database sense), constructing software (as in software engineering) and attaching software to data to construct a more complex artifact via the companion principle. This mechanism depends only in interface and ontology matching and does not need to concern itself with the nature of the encapsulated contents. The content, the software that handles the content and other software routines adopt the same component model and are combined following a single procedure.

In contrast with the other proposed standards, the DCC model considers interfaces an essential part of the reusable artifact representation. In the Fluid Web environment, many kinds of digital content artifacts can go around. DCC interfaces are a key characteristic to provide to DCCs a *self describing* structure [68], encapsulating the plurality of digital content kinds. The interface is a key player that guides DCC usage and composition. It

is a relevant characteristic considering that developers in this new scenario will not be just computer science experts [48].

Interfaces are also essential to relate a given type of digital content with a software component enabled to deal with it. The main digital content reuse initiatives point out the importance of relating the appropriate program code to the reused content. In the multimedia context, MPEG-21 stresses the need for specifying not only a standard for media exchange, but also a complete framework, including the software dimension [35]. Educational initiatives point out the necessity for 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 preserved in the long term [15]. Still related to OAIS, the Fedora repository shows how service objects (disseminators) can produce multiple (dynamically processed) versions of the same content, adapted to specific needs [76].

MPEG-21 defines a software framework, extensible with software plug-ins, to deal with multimedia contents. However, compared to the DCC model, this software framework has two limitations, from a Fluid Web perspective. There are two kinds of MPEG-21 software modules enabled to deal with specific media types: software plug-ins and methods attached to media artifacts. A software plug-in has generally a central producer and many clients. It contrasts with the collaborative work perspective of software components inside DCCs, which can be replicated, modified/adapted and redeployed. Methods attached to media artifacts are constrained to deal with specific MPEG-21 libraries (DIBO); their functionality is restricted. Additionally, MPEG-21 defines a specific model to represent methods' functionality, instead of adopting Web standards.

The educational initiatives of content-centric reuse 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, for instance, the interaction of a student with an educational object, 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 of the Aviation Industry CBT Committee (AICC) [43], which is based on the assumption that any educational content will be Web-based. AICC defined an API that is responsible for defining what services can be requested to the RTE and what information can be delivered to it. Unlike the DCC model, which defines a generic model to encode process DCCs, the AICC standard is highly specialized in some tasks envisaged in the Web activities for education.

Fedora [76] is a general purpose repository service, which supports complex content 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. However, unlike the DCC model, there is a lack of a strategy to deploy and reuse such objects, and their related disseminators outside the repository.

As illustrated in Fig. 4.19, some important content reuse initiatives developed domain specific XML-based metadata schemas. Compared to the DCC OWL-based metadata strategy, XML-based metadata strategies have two limitations. First, XML only provides a standard mechanism to syntactical extensions of the schema, and there is no mechanism to share semantics associated with these extensions. Second, the XML metadata schema is isolated from other parallel metadata schemas and vocabularies, since there is not a standard way to relate XML metadata schemas. For this reason, initiatives like IEEE Learning Object Metadata (LOM) [33] – used by the IMS CP packaging format – started to study an RDF approach to represent the metadata.

A natural consequence in the implementation of the content packing/deployment/reuse standards discussed here is a scenario where people increasingly exchange their productions. However, none of these standards expanded their view to consider a broader outlook, which will require much more functionalities to support this content exchange. Beyond content packaging and deployment, some initiatives address additional requirements: OAIS considers the content storage, RAS defines a version id creation policy to their manifests (but does not address the version control and management infrastructure) and the MPEG-21 group defines an XML language to express rights and permissions (Rights Expression Language) that uses terms defined in a Rights Data Dictionary [81], which is based on the XrML – eXtensible rights Markup Language [82]. Our Fluid Web approach, on the other hand, expanded the perspective from a content artifact that is deployed through the Web to a complete infrastructure to support collaborative work. Our model will include in future extensions rights and permissions management. However, it will use a OWL-based rights model as defined by OREL – Ontology-based Rights Expression Language [59].

4.6.2 Version Control and Management

The control and management of versions and relationships among software and data objects is subject of research in many Computer Science domains. Many systems represent versionable artifacts as generic objects in a database. This approach is convenient for design systems, specially those related with CAD and CAE systems, which work with design objects [37]. Design systems represent complex objects using a hierarchical containment structure, also used in DCCs. The control of versions and configurations are orthogonal. Object derivations are controlled through explicit links among object versions. Configurations are defined using explicit links to specific object versions. A complex object version

is composed via referencing specific versions of its component objects. A problem arises when an object evolves to a new version. This triggers a change propagation procedure, since the new version of an object `X` implies in a new version of the complex objects that have `X` as a component and cascades can reach very many objects [37]. This procedure automatically generates a lot of new, often useless, configurations [30].

The multiversion database approach, adopted in the DCC model and infrastructure, takes a distinct approach that does not produce this side effect on each new object version. The connections are made among objects in a specific context (a DBV) and they are linked to the object `Oid` instead a specific object version (`PVid`). As a consequence, any modification in the object automatically repercusses on the entire DBV, without any new link creation.

Notice that the DBV is the basic unit to control the impact of object changes in the related objects and consequently is also a unit that represent a consistent view of the database. The complete process of a new DBV creation and the modifications inside it is controlled by the user. A user's awareness and control of the scenario where version modifications will have an impact is a benefit of the multiversion database approach, when confronted with traditional approaches whose consistency control mechanism is task oriented [31]. In a task-oriented approach, a context representing a consistent state of the database is created by a transaction – which comprises a set of coordinated modifications on object versions – applied over a previous context.

Version management mechanisms are usually based on a central (local) database. In distributed environments, they require explicit check-in/check-out operations, usually with configuration replication (e.g., as discussed in [37]). We extended the multiversion database mechanism to use URIs to support versioning on the Fluid Web. Moreover, we consider two kinds of collaboration over versions – by reuse and by sharing DBVs, where synchronization mechanisms take advantage of DBV identifiers to find the differences between two copies of the same DBV.

4.6.3 Configuration Management

As mentioned in Section 4.4, version and configuration control/management are often represented using orthogonal but interlaced structures. Engineering design systems – usually CAD and CAE systems – manage their complex design objects as configurations [37]. Research in this domain is often called *Product Data Management* (PDM) [26] or *Engineering Data Management* (EDM) [83]. In parallel, the software development process also involves the management of a complex set of interrelated pieces, which have many characteristics in common with design objects, such as the hierarchical containment structure (a software module can be composed by sub-modules, which in turn can be

composed by sub-modules) and version control. Research in this domain is called *Software Configuration Management* (SCM) [25]. Even when dealing with equivalent problems, research on PDM/EDM and SCM are often treated as distinct disciplines [26] [83]. The DCC model can be considered a confluence of both (PDM/EDM and SCM), since it supports design objects and software artifacts alike.

Software design and development tools deal with a progressive diversity of digital artifacts, as a consequence of: (i) the expansion of their functionality to automate software development phases not covered before, and (ii) the diversification of digital artifacts involved in a software product (e.g., multimedia files, XML documents, etc.). However, as observed by Estublier [25], a weakness of the SCM tools is to have too little knowledge of the managed software product. The data model used by many of these tools is similar to a file system plus a few attributes [25]. Our Fluid Web infrastructure, on the other hand, is founded on the DCC as its basic product model. The DCC structure combines a semantically rich representation with a homogeneous model.

4.7 Concluding Remarks

Behind the title *Fluid Web* we are constructing an extended scenario for Semantic Web initiatives, to immerse the user in an environment where any digital artifact can be shared, replicated, decomposed, versioned, mixed and adapted. To achieve this goal our work combines research in databases, digital content and software management/reuse, version/configuration control and management, Semantic Web and Web Service standards.

It is based on two contributions: the concept of Digital Content Component (DCC), an infrastructure to support storage and management of DCCs, both of which imply the migration from a document-oriented to a component-oriented Web. The infrastructure supports new annotation and versioning mechanisms, that allow flexibility in digital artifact production and sharing. Digital content components encapsulate software and data uniformly and provide an original way to compose them, which eliminates the distinction between processes (active) and data (passive) digital content. An important part of our solution is the notion of *companion component*, which, for a given data set, finds the appropriate software to activate it. Deployment DCCs, the components that can travel on the Web, can be semantically annotated at any level, by using ontologies, and the annotations are carried within the component. These characteristics enable the Fluid Web to be component-oriented, evolving from the present document-oriented paradigm. Both DCC and infrastructure rely on Semantic Web standards. Semantic Web ontologies play a central role as a syntactic and semantic bridge, enabling to define interoperable self describing DCC structures and extensible vocabularies based on taxonomic ontologies.

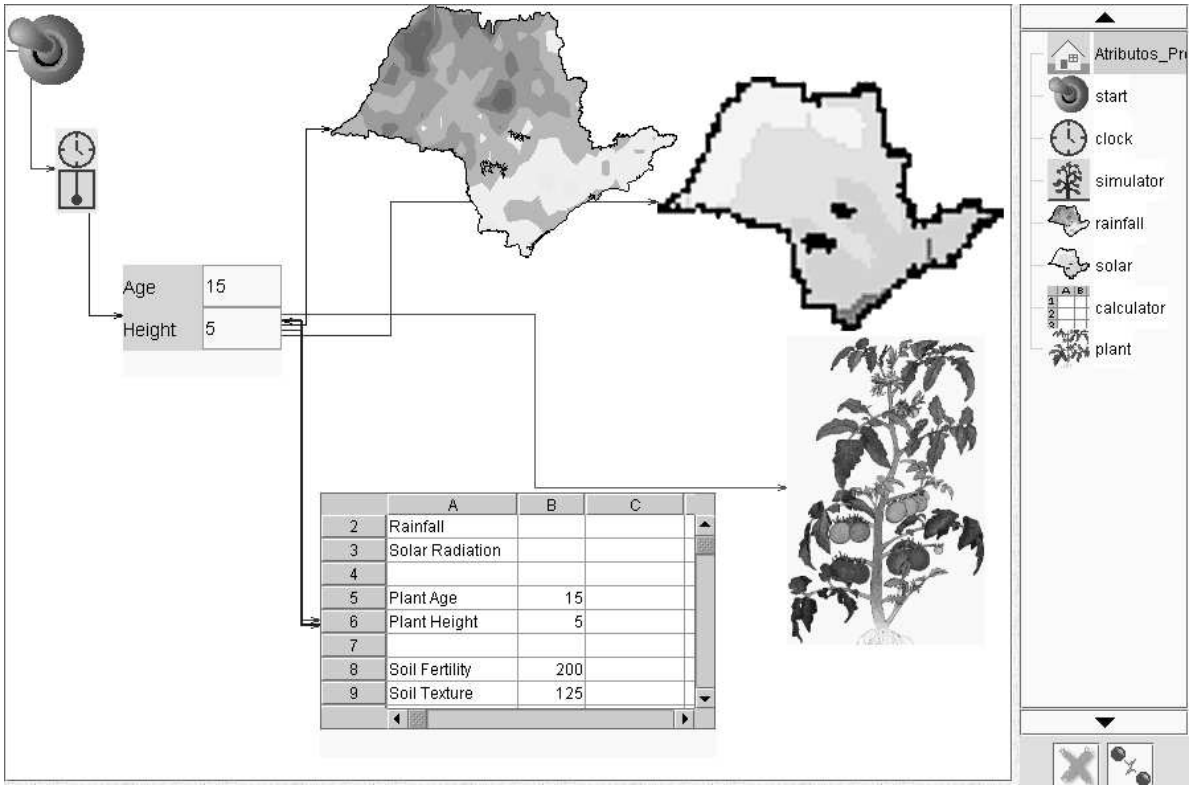

Figura 4.20: Tomato plant simulation produced in Magic House environment.

The current implementation extends a stable version of Anima, which is being used in conjunction with Magic House to build and execute educational digital objects. This current version is fully functional in a local environment, and is implemented in the Java language. The implemented framework can deal with process and passive DCCs, and uses an OWL ontology to match passive DCCs with their Companions, fully supporting the content-type driven execution described in Section 4.5.2. Fig. 4.20 shows a screenshot of the example presented in Section 4.3.3, which simulates a single tomato plant growth.

We are now working to transform the Anima file based storage structure in the multiversion database structure and to connect local environments to exchange DCCs and related data. Additional ongoing work involves offering more support to design activities through the integration of the DCC model with the WOODSS model [44]. Among other benefits, this will enable the reuse of process descriptions (represented as high level workflow specifications), which can be subsequently refined and implemented through the specification of the DCCs that will execute the tasks. This will allow interrelating design DCCs representing different abstraction levels of a process.

Finally, our model can handle DCC versions distributed over different repositories, synchronizing them at request. However, this is not yet a full-fledged distributed mana-

gement system, since synchronization is not automatic, works as one-way synchronization, and requires user intervention. Future work thus includes extending this mechanism to be automatically supported.

Capítulo 5

Conclusões

5.1 Contribuições

Este trabalho se situa em uma intersecção da Engenharia de Software e da "Engenharia de Conteúdo". Da Engenharia de Software herdamos principalmente o modelo de componente de software e outras tecnologias que giram em torno da produção e reuso de software, tais como controle de versões e configurações, frameworks de software.

Ainda que não consensual, o que chamamos de "Engenharia de Conteúdo" envolve diversos domínios que englobam gerenciamento de conteúdo, objetos digitais complexos, controle de configurações e versões. De um certo modo, é um espelho da Engenharia de Software do ponto de vista do conteúdo.

O desafio deste trabalho foi demonstrar que é possível integrar ambas as abordagens aproveitando o melhor de cada uma e resolver problemas decorrentes da interdependência entre software executável e conteúdo. Para atingir este objetivo, o trabalho combina pesquisa em bancos de dados, reuso e gerenciamento de conteúdo digital e software, controle e gerenciamento de configurações/versões, padrões de serviços Web e de Web Semântica.

As principais contribuições foram, desta forma:

1. O modelo de Componente de Conteúdo Digital – Digital Content Component (DCC) – capaz de encapsular software executável e outros tipos de conteúdo de modo uniforme, provendo um modo original para compô-los, eliminando do ponto de vista de gerenciamento e composição a distinção entre software executável e outros tipos de conteúdo. O modelo de DCC está aliado à estratégia de "execução dirigida pelo tipo de conteúdo", baseada no uso de ontologias, onde estão inclusas as noções de *funcionalidade potencial* e *Companion Component*.

2. Um procedimento de três passos que explora metadados associados a DCCs, combinado com a funcionalidade expressa em suas interfaces, para aprimorar o processo

de busca dos DCCs, facilitando a tarefa do usuário-autor.

3. A introdução da noção de *Fluid Web* e a proposta de uma infraestrutura para dar suporte à sua efetiva materialização. A infraestrutura é baseada em repositórios locais interligados, com controle de configurações e versões, levando em conta o compartilhamento distribuído dos DCCs. O controle de versões é facilitado e enriquecido semanticamente pela adoção de uma ontologia taxonômica de relacionamentos entre versões de componentes, desenvolvida neste trabalho.

4. A validação prática de grande parte desses conceitos por meio da construção de protótipos.

Estas contribuições atacam pontos em aberto em 3 frentes – discutidas no Capítulo 1 – apoio ao usuário, modelagem de conteúdo e infraestrutura computacional:

- a necessidade de uma abordagem de produção/consumo de conteúdo adequada ao novo *papel de usuário-autor*;
- o *modelo de compartilhamento/reuso de conteúdo digital* para a Fluid Web;
- a infraestrutura que dá *suporte ao compartilhamento/reuso deste conteúdo digital* na Fluid Web.

5.2 Estágio Atual da Implementação

A implementação pode ser discutida sob três aspectos: extensão do sistema Casa Mágica, que originou esta proposta; framework para suporte a DCCs e Fluid Web; e protótipo do mecanismo de busca. O estágio atual da implantação de cada um deles é o seguinte:

- **Sistema Casa Mágica adaptado para DCCs:** o sistema Casa Mágica [71], desenvolvido a partir de 1994, é ambiente de autoria para a construção de aplicações educacionais. Diversos aspectos desse sistema foram adaptados para trabalhar com DCCs. A interface foi adaptada para que o autor possa manipular e combinar DCCs passivos e de processo indistintamente, sem perceber a diferença entre ambos.
- **Framework para suporte a DCCs e Fluid Web:** este framework é derivado de um framework anterior implementado por nós denominado Anima [71], utilizado como base no sistema Casa Mágica. As seguintes funcionalidades do Anima foram estendidas:

  - **Manipulação e execução de DCCs:** Anima visava a manipulação e execução de componentes de software no sistema Casa Mágica. O framework foi completamente adaptado para manipular e gerenciar a execução de DCCs. Isto incluiu a implementação da *execução dirigida pelo tipo de conteúdo*.
  - **Biblioteca para "Deployment DCC":** foi inteiramente implementada uma biblioteca responsável por exportar um DCC de um modelo de objetos em memória para o formato de distribuição e vice-versa. Este processo envolve a manipulação das seções do DCC descritas em XML, OWL e OWL-S.
  - **Repositório:** Anima armazenava os componentes de software como arquivos independentes no sistema de arquivos. A primeira adaptação realizada para gerenciar DCCs manteve esta abordagem. Uma nova versão do framework está em processo de desenvolvimento, visando armazenar os DCCs em um banco de dados relacional. A modelagem e o esquema do banco de dados já estão prontos. Atualmente o framework está sendo adaptado para interagir com um SGBD Postgres.

- **Mecanismo de busca para DCCs:** Foi implementado um mecanismo que implementa as estratégias de busca descritas no Capítulo 2. Este mecanismo funciona com uma interface Web implementada em Java Server Pages (JSP) e interage com a versão parcialmente implementada do repositório de DCCs.

5.3 Extensões

Há inúmeras extensões possíveis ao trabalho da tese, tanto teóricas quanto de implementação. Algumas delas se enquadram no modelo de DCCs, outras na infraestrutura para Fluid Web. Dentre elas podem ser citadas:

- **Integração entre DCCs e WOODSS:** como descrito em [44] e parcialmente mencionado no Capítulo 4, está sendo desenvolvido um projeto de integração entre os DCCs e workflows especificados dentro do projeto WOODSS (WOrkflOw-based spatial Decision Support System) [56]. O WOODSS é um ambiente extensível que permite a captura, especificação, reuso e anotação de workflows científicos. Ele vem sendo desenvolvido no Laboratório de Sistemas de Informação (LIS) da UNICAMP – onde também está sendo desenvolvido este projeto. No projeto de integração, os workflows em WOODSS são utilizados para coordenar DCCs e também podem ser armazenados dentro deles para reuso. Isto exige, dentre outros, estudo de problemas de especificação, compartilhamento e execução de workflows e acesso a serviços.

- **Reuso de *design*:** no processo de construção de um workflow em WOODSS é possível se trabalhar em diferentes níveis de abstração. Os níveis mais abstratos funcionam como um *design* do processo e os níveis mais concretos especificam detalhes de execução. O modelo WOODSS permite ainda mesclar partes em diferentes níveis de abstração. Como conseqüência da integração dos DCCs com o WOODSS, surgiu a necessidade de se representar outra categoria de DCC denominada Design DCC. Este tipo de DCC encapsula especificações abstratas de processos e, em especial, workflows abstratos [44]. Uma característica distintiva deste tipo de DCC é a possibilidade de se realizar o reuso não apenas do executável mas também do projeto de uma aplicação, conforme abordado na Seção 1.2.2. Esta extensão envolve desafios em reuso de esquemas de software [40, 46] e em representação de diferentes estilos arquiteturais [28].

- **Controle de configurações e versões distribuído:** atualmente o modelo de controle de versões e configurações permite compartilhar dados de repositórios distribuídos e realizar sincronização, tal como foi descrito no Capítulo 4. Este modelo deve ser estendido para permitir um controle distribuído de configurações e versões, onde sincronizações entre repositórios possam ser realizadas automaticamente. Um outro aspecto relacionado são as questões de autoria e segurança dos conteúdos.

- **Novas estratégias para avaliar similaridade:** a descoberta de conteúdo digital explora ontologias para o ranqueamento de DCCs baseando-se nos relacionamentos ontológicos de equivalência, generalização e especialização. Outra extensão é usar outras estratégias para a busca de termos similares via ontologias que vão além da generalização e especialização, tais como as discutidas em [60, 62]. Estas técnicas utilizam distância entre termos na ontologia ou técnicas estatísticas na árvore para avaliar a similaridade entre termos.

- **Framework para execução distribuída de composições:** conforme descrito no Capítulo 3, foi projetado um framework que permite que os DCCs distribuídos se comuniquem. Um passo subseqüente é a possibilidade de se criar composições ligando componentes distribuídos pela rede, que possam ser executadas de forma distribuída, sem a necessidade de se carregar todos os componentes de uma mesma composição para um mesmo computador.

- **Avaliação do uso da proposta:** vários aspectos da proposta foram validados em ambientes educacionais, com o uso de Casa Mágica e Anima em escolas secundárias em Salvador e na UNIFACS, para cursos de Engenharia de Software e Compiladores. No entanto, é necessário fazer uma avaliação da proposta e implementação de Fluid Web, inclusive em cooperação à distância. Isto envolve inúmeras frentes de pesquisa

em Computação, tanto do ponto de vista de projeto e implementação de software quanto de avaliação por parte do usuário.

Referências Bibliográficas

[1] Webster's new international dictionary of the english language, 1952.

[2] ADL. Sharable Content Object Reference Model (SCORM) 2004 – 2nd Edition – Overview, July 2004. `www.adlnet.org/screens/shares/dsp_displayfile.cfm?fileid=992`, accessed on 11/2004.

[3] G. Alonso et al. *Web Services – Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[4] T. Andrews et al. Specification: Business Process Execution Language for Web Services Version 1.1, May 2003. `http://www-106.ibm.com/developerworks/library/ws-bpel/`, accessed on 12/2004.

[5] Apple Computer Inc. *OpenDoc Programmer's Guide for the Mac OS*. Apple Press, 1996.

[6] F. Bachman et al. Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon University, July 2000.

[7] J. Bekaert, E. De Kooning, and R. Van de Walle. Packaging models for the storage and distribution of complex digital objects in archival information systems: a review of MPEG-21 DID principles. *Multimedia Systems*, 10(4):286–301, October 2005.

[8] A. Bolour. Notes on the eclipse plug-in architecture, July 2003. `www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html`, accessed on 06/2005.

[9] K. Brockschmidt. *Inside OLE*. Microsoft Press, 2 edition, 1995.

[10] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plášil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski. What characterizes a (software) component? *Software – Concepts & Tools*, 19(1):49–56, June 1998.

[11] D. Bulterman and L. Hardman. Structured multimedia authoring. *ACM Trans. Multimedia Comput. Commun. Appl.*, 1(1):89–109, 2005.

[12] D. Burdett and N. Kavantzas. WS Choreography Model Overview – W3C Working Draft 24 March 2004, March 2004. `http://www.w3.org/TR/2004/WD-ws-chor-model-20040324/`, accessed on 12/2004.

[13] I. Burnett, R. Van de Walle, K. Hill, J. Bormans, and F. Pereira. MPEG-21: goals and achievements. *Multimedia*, 10:60–70, Oct–Dec 2003.

[14] I.S. Burnett, S.J. Davis, and G.M. Drury. MPEG-21 digital item declaration and Identification-principles and compression. *IEEE Trans. on Multimedia*, 7(3):400–407, June 2005.

[15] CCSDS. Reference Model for an Open Archival Information System (OAIS) – Blue Book. Technical Report CCSDS 650.0-B-1, January 2002. `www.ccsds.org/CCSDS/documents/650x0b1.pdf`, accessed on 11/2004.

[16] CCSDS. XML Formatted Data Unit (XFDU) Structure and Construction Rules, September 2004. `http://www.ccsds.org/docu/dscgi/ds.py/Get/File-1913/IPRWBv2a.2.pdf`, accessed on 12/2004.

[17] W. Cellary and G. Jomier. Consistency of versions in object-oriented databases. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 432–441. Morgan Kaufmann, 1990.

[18] W. Cellary, G. Jomier, S. Gançarski, and M. Manouvrier. *Bases de données et internet Modèles, langages et système*, chapter 8- Les Versions, pages 235–255. Hermès Science, 2001. Traité IC2 Information - Commande - Communication, sous la direction de A. Doucet et G. Jomier, Edition Lavoisier.

[19] R. Chinnici et al. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language – W3C Working Draft 3 August 2004, August 2004. `www.w3.org/TR/2004/WD-wsdl20-20040803/`, accessed on 11/2004.

[20] N. Cullot, C. Parent, S. Spaccapietra, and C. Vangenot. Ontologies: A contribution to the DL/DB debate. In *Proc. of the 1st International Workshop on the Semantic Web and Databases, 29th International Conf. on Very Large Data Bases*, pages 109–129, September 2003.

[21] A. Doan, J. Madhavan, R. Dhamankar, P. Domingos, and A. Halevy. Learning to match ontologies on the semantic web. *The VLDB Journal*, 12(4):303–319, 2003.

[22] G. Eddon. COM+: the evolution of component services. *Computer*, 32(7):104–106, July 1999.

[23] W. Emmerich. Distributed component technologies and their software engineering implications. In *ICSE '02: Proc. of the 24th International Conf. on Software Engineering*, pages 537–546. ACM Press, 2002.

[24] G. Bard Ermentrout and Leah Edelstein-Keshet. Cellular automata approaches to biological modeling. *Journal of Theoretical Biology*, 160(1):97–133, January 1993.

[25] J. Estublier. Software configuration management: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 279–289, New York, NY, USA, 2000. ACM Press.

[26] J. Estublier, J. Favre, and P. Morat. Toward scm / pdm integration? In *ECOOP '98: Proceedings of the SCM-8 Symposium on System Configuration Management*, pages 75–94, London, UK, 1998. Springer-Verlag.

[27] R. Fileto, L. Liu, C. Pu, E. D. Assad, and C. B. Medeiros. POESIA: An ontological workflow approach for composing Web services in agriculture. *The VLDB Journal*, 12(4):352–367, 2003.

[28] W. B. Frakes and K. Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[29] E. Gamma, R. Helm, and J. M. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[30] S. Gançarski and G. Jomier. Managing entity versions within their contexts: A formal approach. In *DEXA '94: Proc. of the 5th Int. Conf. on Database and Expert Systems Applications*, pages 400–409, London, UK, 1994. Springer-Verlag.

[31] S. Gançarski, G. Jomier, and M. Zamfiroiu. A Framework for the Manipulation of a Multiversion Database. In *Workshop Proc. of Database and Expert Systems Applications Conference (DEXA'95)*, pages 247–256, Londres (U.K.), 1995.

[32] Jon Hopkins. Component primer. *Communications ACM*, 43(10):27–30, 2000.

[33] IEEE L.T.S.C. Draft Standard for Learning Object Metadata – IEEE 1484.12.1-2002, July 2002. `http://ltsc.ieee.org/doc/wg12/LOM_1484_12_1_v1_Final_Draft.pdf`, accessed on 10/2003.

[34] ISO/IEC. ISO/IEC TR 2100-1 – Information technology – Multimedia framework (MPEG-21) – Part 1: Vision, Technologies and Strategy, December 2001.

[35] ISO/IEC. ISO/IEC TR 2100-1 – Information technology – Multimedia framework (MPEG-21) – Part 1: Vision, Technologies and Strategy – 2nd Ed. Technical report, 2004.

[36] G. Jomier and W. Cellary. The Database Version Approach. *Networking and Information Systems Journal*, 3(1/2000):177–214, 2000. Hermes, Paris, France.

[37] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):375–409, 1990.

[38] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web Services Choreography Description Language Version 1.0 – W3C Candidate Recommendation 9 November 2005, November 2005. `http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/`, accessed on 06/2006.

[39] R. Klischewski. Semantic web for e-government. *Lecture Notes in Computer Science*, 2739:288–295, January 2003.

[40] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[41] F. Manola and E. Miller. RDF Primer – W3C Recommendation 10 February 2004, February 2004. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`, accessed on 11/2004.

[42] D. Martin et al. OWL-S: Semantic Markup for Web Services, November 2004. `www.daml.org/services/owl-s/1.1/overview/`, accessed on 12/2004.

[43] W. A. McDonald, J. Hyde, and A. Montgomery. CMI Guidelines for Interoperability AICC, August 2004. `www.aicc.org/docs/tech/cmi001v4.pdf`, accessed on 11/2004.

[44] C. B. Medeiros, J. Perez-Alcazar, L. Digiampietri, G. Z. Pastorello Jr, A. Santanchè, R. S. Torres, E. Madeira, and E. Bacarin. WOODSS and the Web: Annotating and reusing scientific workflows. *SIGMOD Record*, 34(3):18–23, September 2005.

[45] T. O. Meservy and K. D. Fenstermacher. Transforming software development: An MDA road map. *Computer*, 38(9):52–58, September 2005.

[46] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[47] N. Mitra. SOAP Version 1.2 Part 0: Primer – W3C Recommendation 24 June, June 2003. `www.w3.org/TR/2003/REC-soap12-part0-20030624/`, accessed on 12/2004.

[48] A. I. Mørch, G. Stevens, M. Won, M. Klann, Y. Dittrich, and V. Wulf. Component-based technologies for end-user development. *Commun. ACM*, 47(9):59–62, 2004.

[49] M. Nanard, J. Nanard, and P. King. IUHM: a hypermedia-based model for integrating open services, data and metadata. In *HYPERTEXT '03: Proc of the 14th ACM Conf on Hypertext and Hypermedia*, pages 128–137, New York, NY, USA, 2003.

[50] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson, and M. A. Musen. Creating semantic web contents with protégé-2000. *IEEE Intelligent Systems*, 16(2):60–71, 2001.

[51] Natalya F. Noy. Semantic integration: a survey of ontology-based approaches. *SIGMOD Record*, 33(4):65–70, 2004.

[52] Object Management Group. CORBA Components – Version 3.0 – formal/02-06-65, 2002. `www.omg.org/cgi-bin/doc?formal/02-06-65`, accessed on 10/2003.

[53] OMG. Reusable Asset Specification – final adopted specification, June 2004. `http://www.omg.org/cgi-bin/doc?ptc/2004-06-06`, accessed on 10/2004.

[54] OMG. Reusable Asset Specification – final specification – version 2.2, April 2005. `www.omg.org/cgi-bin/doc?ptc/2005-04-02`, accessed on 09/2005.

[55] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic Matching of Web Services Capabilities. In *ISWC '02: Proc. of the First International Semantic Web Conf. on The Semantic Web*, pages 333–347. Springer-Verlag, 2002.

[56] G. Z. Pastorello Jr., C. B. Medeiros, S. M. Resende, and H. A. Rocha. Interoperability for GIS Document Management in Environmental Planning. *Journal of Data Semantics*, 3(LNCS 3534):100–124, 2005.

[57] R. Potts. Mozilla Developer Documentation – NGLayout Embedding APIs, 10 1998. `www.mozilla.org/newlayout/doc/webwidget.html`, accessed on 06/2005.

[58] R. Prieto-Díaz. Classification of reusable modules. In T. J. Biggerstaff and A. J. Perlis, editors, *Software reusability: vol. 1, concepts and models*, pages 99–123. ACM Press, 1989.

[59] Y. Qu, X. Zhang, and H. Li. Orel: an ontology-based rights expression language. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 324–325, New York, NY, USA, 2004. ACM Press.

[60] R. Rada, H. Mili, E. Bicknell, and M. Blettner. Development and application of a metric on semantic nets. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(1):17–30, Jan–Feb 1989.

[61] R. Raskin and M. Pan. Semantic Web for Earth and Environmental Terminology (SWEET). In *Proc. of the Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data*, October 2003.

[62] P. Resnik. Using information content to evaluate semantic similarity in a taxonomy. In C. S. Mellish, editor, *Proc. of the Fourteenth International Joint Conf. on Artificial Intelligence*, pages 448–453, August 1995.

[63] J. Roschelle, C. DiGiano, M. Koutlis, A. Repenning, J. Phillips, N. Jackiw, and D. Suthers. Developing educational software components. *Computer*, 32(9):50–58, 1999.

[64] A. Santanchè. Aplicações educacionais na Web – o papel de RDF e Metadados. In *XIV Simpósio Brasileiro de Informática na Educação – Mini-Cursos*, pages 127–152, 2003.

[65] A. Santanchè and C. B. Medeiros. Geographic Digital Content Components. In *Proc. of VI Brazilian Symposium on GeoInformatics*, pages 281–290, November 2004.

[66] A. Santanchè and C. B. Medeiros. Managing Dynamic Repositories for Digital Content Components. In *Current Trends in Database Technology – EDBT 2004 Workshops*, volume LNCS 3268, pages 66–77, 2004.

[67] A. Santanchè and C. B. Medeiros. Managing Repositories for Digital Content Components. In *Proc. of III Workshop of Thesis on Databases of XIX Brazilian Symposium on Databases*, pages 12–19, 2004.

[68] A. Santanchè and C. B. Medeiros. Self Describing Components: Searching for Digital Artifacts on the Web. In *Proc. of XX Brazilian Symposium on Databases*, pages 10–24, 2005.

[69] A. Santanchè and C. B. Medeiros. A Component Model and Infrastructure for a Fluid Web. *IEEE Transactions on Knowledge and Data Engineering*, 2006. Submitted.

[70] A. Santanchè and C. B. Medeiros. User-author centered multimedia building blocks. *Multimedia Systems*, 2006. To be published.

[71] A. Santanchè and C. A. C. Teixeira. Anima: Promoting component integration in the web. In *Proc. of 7th Brasilian Symposium on Multimedia and Hypermedia Systems*, pages 261–268, October 2001. (in portuguese).

[72] M. Shaw and P. C. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *COMPSAC '97: Proc. of the 21st International Computer Software and Applications Conference*, pages 6–13. IEEE Computer Society, 1997.

[73] M. K. Smith, C. Welty, and D. L. McGuinness. OWL Web Ontology Language Guide – W3C Recommendation 10 February 2004, February 2004. `www.w3.org/TR/2004/REC-owl-guide-20040210/`, accessed on 11/2004.

[74] C. Smythe and A. Jackl. IMS Content Packaging Information Model. Specification, IMS Global Learning Consortium, Inc., October 2004. `www.imsglobal.org/content/packaging/cpv1p1p4/imscp_infov1p1p4.html`, accessed on 11/2004.

[75] Sun Microsystems. Enterprise JavaBeans^{TM} Specification, Version 2.1, November 2003.

[76] The Fedora Project team. Mellon Fedora Technical Specification, December 2002. `www.fedora.info/documents/master-spec-12.20.02.pdf`, accessed on 12/2004.

[77] The Library of Congress. METS: An Overview & Tutorial, September 2004. `www.loc.gov/standards/mets/METSOverview.v2.html`, accessed on 11/2004.

[78] UDDI Committee Specification. Uddi version 2.04 api specification, July 2002. `uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm`, accessed on 06/2005.

[79] A. van Hoff, H. Partovi, and T. Thai. The Open Software Description Format (OSD), August 1997. `http://www.w3.org/TR/NOTE-OSD`, accessed on 10/2004.

[80] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-based integration of information – A survey of existing approaches. In *Proc. of the IJCAI-01*, pages 108–117, 2001.

[81] X. Wang, T. DeMartini, B. Wragg, M. Paramasivam, and C. Barlas. The MPEG-21 rights expression language and rights data dictionary. *IEEE Transactions on Multimedia*, 7(3):408–417, June 2005.

[82] X. Wang, G. Lao, T. DeMartini, H. Reddy, M. Nguyen, and E. Valenzuela. XrML – eXtensible rights Markup Language. In *XMLSEC '02: Proceedings of the 2002 ACM workshop on XML security*, pages 71–79, New York, NY, USA, 2002. ACM Press.

[83] B. Westfechtel and R. Conradi. Software configuration management and engineering data management: Differences and similarities. In *ECOOP '98: Proceedings of the SCM-8 Symposium on System Configuration Management*, pages 95–106, London, UK, 1998. Springer-Verlag.

[84] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Trans. Softw. Eng. Methodol.*, 6(4):333–369, 1997.