Universidade Federal do Rio de Janeiro
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia
Programa de Engenharia de Sistemas e Computação

Exame de Qualificação

ODYSSEY-SCM: UMA ABORDAGEM DE GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE PARA O DESENVOLVIMENTO BASEADO EM COMPONENTES

Leonardo Gresta Paulino Murta

Orientadora: Cláudia Maria Lima Werner

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 2004

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1 - Introdução

1.1 Contexto

Durante os primeiros anos de pesquisa nas áreas de Gerência de Configuração de Software (GCS) e de Reutilização de Software, o foco era sobre código-fonte, armazenado em arquivos do sistema operacional. Uma grande infra-estrutura foi criada nas últimas décadas para apoiar a evolução de código-fonte utilizando técnicas de GCS. Durante esse mesmo período, a Reutilização de Software foi aplicada ao nível de código-fonte através de linguagens orientadas a objeto, idiomas, bibliotecas de ligação dinâmica, dentre outras.

Com o passar do tempo, novos paradigmas foram definidos para possibilitar a uniformização dos conceitos usados para representar as entidades do mundo real, passando pelos diversos níveis de abstração abordados pelas atividades de análise, projeto e codificação. Apesar da orientação a objetos atender, de certa forma, a essa demanda, a granularidade de encapsulamento utilizada é muito fina, dificultando a sistematização da substituição ou reutilização das partes que constituem os sistemas.

O paradigma de Desenvolvimento Baseado em Componentes (DBC) contorna algumas das deficiências detectadas na orientação a objetos, no que tange a reutilização de software. Além disso, o DBC fornece uma estrutura bem definida, composta por componentes, interfaces e conectores, que visa permitir o tratamento da complexidade crescente do desenvolvimento de software.

1.2 Motivação

No contexto de DBC, podem existir diversas equipes de desenvolvimento de componentes e diversas equipes de desenvolvimento com componentes trabalhando dentro de uma mesma organização. Além disso, as equipes de desenvolvimento de componentes podem fazer uso de componentes providos por outras equipes de desenvolvimento de componentes, caracterizando equipes híbridas, que atuam concomitantemente nos dois papéis.

Essas diferentes equipes estão intimamente relacionadas através do processo de reutilização, onde as equipes de desenvolvimento de componentes atuam como produtoras e as equipes de desenvolvimento com componentes atuam como

consumidoras. Nesse processo, as equipes produtoras constroem e mantêm componentes reutilizáveis, enquanto as equipes consumidoras adaptam e reutilizam esses componentes para construir aplicações específicas.

Devido à necessidade intransponível de manter o software após a sua liberação, a reutilização deve ocorrer de forma controlada, possibilitando que a evolução dos componentes reutilizáveis não dificulte o trabalho já complexo de desenvolvimento de componentes e de desenvolvimento com componentes. Esse controle pode ser obtido através do uso de técnicas de GCS.

Apesar de existir um forte apelo para o uso da GCS durante a etapa de manutenção, a sua aplicação não se restringe somente a essa etapa do ciclo de vida do software (IEEE, 1987; LEON, 2000). Durante o desenvolvimento, os sistemas de GCS são fundamentais para prover controle sobre os artefatos produzidos e modificados por diferentes desenvolvedores. Além disso, esses sistemas permitem um acompanhamento minucioso do andamento das tarefas de desenvolvimento, possibilitando que diversas métricas sejam coletadas e analisadas.

1.3 Problema

O problema em questão está relacionado com a complexidade da evolução de artefatos reutilizáveis. Essa complexidade ocorre em virtude da necessidade freqüente de adaptar o artefato antes da sua efetiva reutilização. Contudo, as várias instâncias adaptadas de um artefato são profundamente semelhantes, e possivelmente os mesmos defeitos ou necessidades de melhorias acontecerão em todas essas instâncias. O re-trabalho de manutenção pode ser evitado se forem definidas técnicas de GCS para apoiar a evolução desses artefatos de forma controlada.

Este problema pode ser dividido em cinco partes mais específicas, que são: (1) os processos, normas, procedimentos, políticas e padrões que guiam a GCS dentro de um contexto de DBC, (2) o controle de requisições de modificação sobre os artefatos, (3) o controle de versões de artefatos reutilizáveis no nível de abstração do próprio artefato, (4) o controle das construções e liberações, que determinam como os artefatos se relacionam para estruturar o sistema, e (5) a integração dos espaços de trabalho de GCS e DBC.

1.4 Hipótese

A hipótese desse trabalho consiste na possibilidade de aplicação da GCS no contexto de DBC para prover um maior controle na evolução de artefatos reutilizáveis. Apesar do controle sobre as atividades de DBC introduzir uma certa sobrecarga no trabalho original, devido a atividades até então inexistentes, acreditamos que essas atividades trazem benefícios à evolução do software, aumentando a produtividade e diminuindo a quantidade de erros, na medida em que o desenvolvimento se dá em um ambiente controlado e organizado.

No cenário de desenvolvimento convencional, usualmente de menor complexidade que o cenário de DBC, é possível detectar aumentos substanciais de qualidade e produtividade devido à adoção de GCS (LEON, 2000). Por essa razão, acreditamos que os benefícios da GCS serão ainda maiores em situações como a do DBC, onde a complexidade envolvida dificulta a atuação do desenvolvedor no desenvolvimento de software propriamente dito, devido ao aumento de atividades de suporte não automatizadas.

1.5 Objetivos

A meta deste trabalho consiste em definir uma solução de GCS, que envolva processos e ferramental de apoio, para atender os requisitos específicos do DBC, de acordo com a hipótese apresentada na seção 1.4. Essa meta pode ser decomposta em cinco objetivos, em função dos problemas definidos na seção 1.3, que são: (1) processos, normas, procedimentos, políticas e padrões, (2) controle de modificações, (3) controle de versões, (4) controle de construções e liberações e (5) integração dos espaços de trabalho.

No que se refere aos processos, normas, procedimentos, políticas e padrões, o principal objetivo consiste em estabelecer um processo de GCS voltado para as questões específicas de DBC, tendo como base as principais normas existentes em GCS. O objetivo referente ao controle de modificações é fornecer um sistema configurável, que permita a modelagem de processos de GCS e a adaptação desses processos sempre que necessário. Quanto ao controle de versões, o objetivo consiste em apoiar o versionamento de artefatos de alto nível de abstração, utilizando políticas configuráveis. Em relação ao controle de construções e liberações, o objetivo é mapear os conceitos existentes na GCS dentro da metáfora do DBC, possibilitando que as funções de GCS

possam ser propagadas dentro da hierarquia de componentes. Finalmente, o objetivo referente à integração dos espaços de trabalho consiste na interação semitransparente entre GCS e DBC, evitando modificar as atividades existentes de DBC, contudo, provendo troca de conhecimento entre as duas áreas.

A contribuição esperada desse trabalho refere-se principalmente à área de reutilização de software, mais especificamente, ao DBC. A GCS está sendo utilizada como meio para possibilitar o aumento de controle nas atividades de DBC.

1.6 Contexto da solução

A solução proposta está contextualizada no Projeto Odyssey (ODYSSEY, 2004), que visa prover uma infra-estrutura de suporte à reutilização baseada em modelos de domínio, linha de produtos[1] e DBC. A reutilização de software no Ambiente Odyssey (WERNER et al., 2003) ocorre através do processo de engenharia de domínio Odyssey-DE (BRAGA, 2000), que tem por objetivo construir componentes reutilizáveis voltados para domínios de conhecimento específicos, e do processo de engenharia da aplicação Odyssey-AE (MILER, 2000), que tem por objetivo construir aplicações em um determinado domínio, reutilizando os componentes existentes. O Ambiente Odyssey utiliza modelos para representar o conhecimento de um domínio, e permite que esses modelos sejam reutilizados para facilitar a construção de aplicações.

1.7 Organização

Este documento está organizado em outros quatro capítulos, além deste primeiro capítulo de introdução. O Capítulo 2 apresenta uma introdução à área de GCS e fornece uma revisão da literatura de GCS, organizada através dos cinco objetivos apresentados na seção 1.5. O Capítulo 3 exibe uma revisão da literatura da aplicação de GCS no DBC, também organizada através dos cinco objetivos apresentados na seção 1.5. O Capítulo 4, referente à abordagem proposta, detalha possíveis caminhos para contemplar cada um dos cinco objetivos apresentados na seção 1.5, com o intuito de viabilizar o alcance da hipótese apresentada na seção 1.4. Finalmente, o Capítulo 5 conclui o documento apresentando o cronograma proposto.

---

[1] Linha de produtos é um conjunto de sistemas compartilhando características gerenciáveis comuns, que satisfazem às necessidades específicas de um seguimento de mercado em particular e que são desenvolvidos sistematicamente, a partir de um conjunto comum de artefatos (CLEMENTS et al., 2001).

Capítulo 2 - Gerência de Configuração de Software

2.1 Introdução

A Gerência de Configuração (GC) surgiu nos anos 50 devido à necessidade da indústria aeroespacial norte-americana controlar as modificações na documentação referente a produção de aviões de guerra e naves espaciais (LEON, 2000; ESTUBLIER et al., 2002; HASS, 2003). Posteriormente, nos anos 60 e 70, a GC passou a abranger artefatos de software, indo além dos artefatos de hardware já estabelecidos e desencadeando o surgimento da Gerência de Configuração de Software (GCS) (CHRISTENSEN et al., 2002).

Apesar do surgimento da GCS nos anos 70, o seu foco era muito restrito às aplicações militares e aeroespaciais, e somente no final dos anos 80, com o surgimento do padrão IEEE Std 1042 (IEEE, 1987), e em meados dos anos 90, com o surgimento da norma ISO 10007 (ISO, 1995a), a GCS foi finalmente assimilada no processo de desenvolvimento de software de organizações não militares (LEON, 2000).

Como toda nova área de pesquisa, existem diversas definições para GCS. Contudo, a definição mais aceita e utilizada caracteriza a GCS como "uma disciplina que visa identificar e documentar as características de itens de configuração[2], controlar as suas alterações, armazenar e relatar as modificações aos interessados e garantir que foi feito o que deveria ter sido feito" (IEEE, 1990). Desta forma, a GCS não se propõe a definir quando e como devem ser executadas as modificações nos artefatos de software, papel este reservado ao próprio processo de desenvolvimento de software. A sua atuação ocorre como processo auxiliar de controle e acompanhamento dessas atividades.

A GCS pode ser tratada sob diferentes perspectivas em função do papel exercido pelo participante do processo de desenvolvimento de software (ASKLUND et al., 2002). Na perspectiva gerencial, a GCS é dividida em quatro funções, que são (IEEE, 1990; ISO, 1995a): identificação da configuração, controle da configuração, acompanhamento da configuração e auditoria da configuração.

---

[2] O termo item de configuração (*configuration item*) representa a agregação de hardware, software ou ambos, tratada pela GCS como um elemento único (IEEE, 1990).

A **função de identificação da configuração** tem por objetivo possibilitar: (1) a seleção dos itens de configuração (IC) que são os elementos passíveis de GCS; (2) a definição do esquema de nomes e números, que possibilite a identificação inequívoca dos ICs no grafo de versões[3] e variantes[4]; e (3) a descrição dos ICs, tanto física quanto funcionalmente.

A **função de controle da configuração** é designada para o acompanhamento da evolução dos ICs selecionados e descritos pela função de identificação. Para que os ICs possam evoluir de forma controlada, esta função estabelece as seguintes atividades: (1) requisição de modificação, iniciando um ciclo da função de controle dado um pedido de manutenção, que pode ser corretiva, evolutiva, adaptativa ou preventiva (PRESSMAN, 1997); (2) classificação da modificação, que estabelece a prioridade do pedido em relação aos demais pedidos efetuados anteriormente; (3) análise da modificação, que visa relatar os impactos em esforço, cronograma e custo e definir uma proposta de implementação da manutenção; (4) avaliação da modificação pelo Comitê de Controle da Configuração[5] (CCC), que estabelece se o pedido será implementado, rejeitado ou postergado, em função do laudo fornecido pela análise da modificação; (5) implementação da modificação, caso o pedido tenha sido aprovado pela avaliação da modificação; (6) verificação da modificação, aplicando uma bateria de testes de sistema e validando com a proposta de implementação levantada na análise da modificação; e (7) geração de configuração de referência[6], que pode ou não ser liberada[7] para o cliente em função da sua importância e questões de marketing associadas.

A **função de acompanhamento da configuração** visa: (1) armazenar as informações geradas pelas demais funções; e (2) permitir que essas informações possam ser acessadas em função de necessidades específicas. Essas necessidades específicas

---

[3] O termo versão representa o estado de um IC em um determinado momento do desenvolvimento de software (LEON, 2000).

[4] O termo variante representa uma versão funcionalmente equivalente a outra, mas projetada para ambiente de hardware ou software distintos (LEON, 2000).

[5] Do inglês, *configuration control board* (IEEE, 1990).

[6] O termo configuração de referência (*baseline*) representa um conjunto de ICs formalmente aprovados que serve de base para as etapas seguintes de desenvolvimento (IEEE, 1990).

[7] O termo liberação (*release*) representa a notificação e liberação formal de uma versão aprovada do software para o cliente (IEEE, 1998).

abrangem o uso de métricas para a melhoria do processo, a estimativa de custos futuros e a geração de relatórios gerenciais.

A **função de auditoria da configuração** ocorre quando a configuração de referência, gerada na função de controle da configuração, é selecionada para ser liberada para o cliente. Suas atividades compreendem: (1) verificação funcional da configuração de referência, através da revisão dos planos, dados, metodologia e resultados dos testes, assegurando que a mesma cumpra corretamente o que foi especificado; e (2) verificação física da configuração de referência, com o objetivo de certificar que a mesma é completa em relação ao que foi acertado em cláusulas contratuais.

Contudo, sob a perspectiva de desenvolvimento, a GCS é dividida em três sistemas principais, que são: controle de modificações, controle de versões e controle de construções[8] e liberações.

O **sistema de controle de modificações** é encarregado de executar a função de controle da configuração de forma sistemática, armazenando todas as informações geradas durante o andamento das requisições de modificação e relatando essas informações aos participantes interessados e autorizados, assim como estabelecido pela função de acompanhamento da configuração.

O **sistema de controle de versões** permite que os ICs sejam identificados, segundo estabelecido pela função de identificação da configuração, e que eles evoluam de forma distribuída e concorrente, porém disciplinada. Essa característica é necessária para que diversas requisições de modificação efetuadas através da função de controle da configuração possam ser tratadas em paralelo, sem corromper o sistema de GCS como um todo.

O **sistema de controle de construções e liberações** automatiza o complexo processo de transformação dos diversos artefatos de software que compõem um projeto no sistema executável propriamente dito, de forma aderente aos processos, normas, procedimentos, políticas e padrões definidos para o projeto. Além disso, esse sistema estrutura as configurações de referência selecionadas para liberação, conforme necessário para a execução da função de auditoria da configuração.

Apesar de existirem essas diferentes perspectivas para abordar a GCS, elas não se relacionam de forma complementar, mas sim, de forma sobreposta. As quatro

[8] O termo construção (*building*) representa o procedimento de geração do sistema para uma configuração alvo (LEON, 2000).

funções descritas na perspectiva gerencial são implementadas através dos três sistemas descritos na perspectiva de desenvolvimento, acrescidos de processos, normas, procedimentos, políticas e padrões. Além disso, para que esses recursos sejam efetivamente utilizados, alguns serviços especializados são definidos na integração dos espaços de trabalho de GCS e Ambientes de Desenvolvimento de Software (ADS). A Figura 1 exibe essa interação entre as perspectivas.

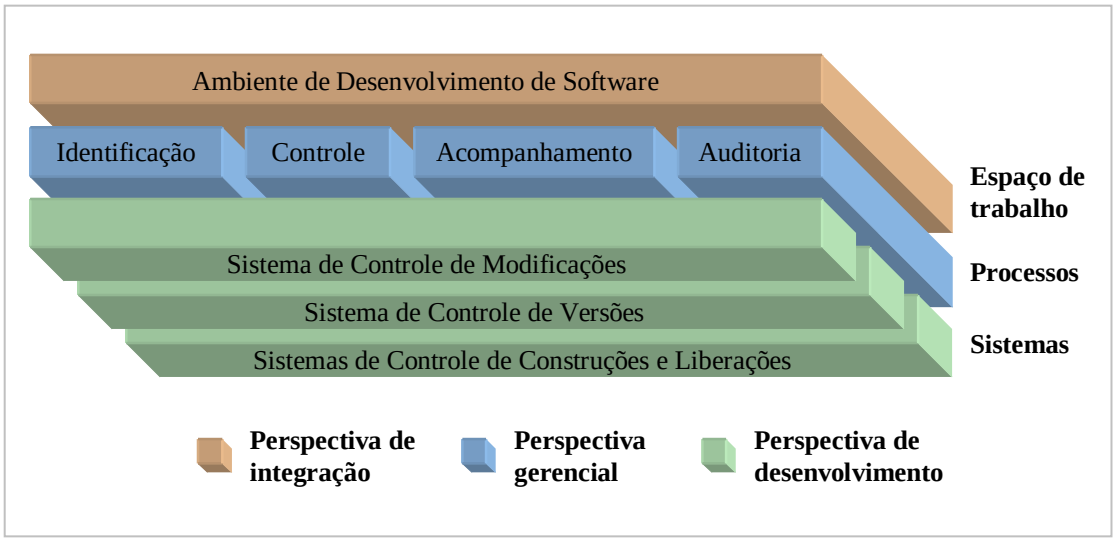

Figura 1: Perspectivas de GCS.

Apesar da GCS ter processos, normas, procedimentos, políticas e padrões gerais, cada sistema descrito na perspectiva de desenvolvimento tem os seus próprios processos, normas, procedimentos, políticas e padrões para atender às funções descritas na perspectiva gerencial. Por exemplo: requisições de modificação podem seguir fluxos díspares em diferentes projetos, numerações e rotulação de versões podem ocorrer de diversas formas em função das necessidades específicas de cada organização, e a geração das versões de produção pode depender de fatores como decisões de marketing e grau de qualidade desejada (LEON, 2000).

Desta forma, será feito uso de uma taxonomia baseada em cinco elementos para caracterizar a literatura de GCS, que cobre as diferentes perspectivas levantadas na Figura 1. Esses elementos são:

- Processos, normas, procedimentos, políticas e padrões;
- Sistemas de controle de modificação;
- Sistemas de controle de versões;
- Sistemas de controle de construções e liberações;
- Integração dos espaços de trabalho.

Na seção 2.2, são descritos os processos, normas, procedimentos, políticas e padrões que regem a aplicação da GCS nos diversos cenários e contextos da engenharia de software. Posteriormente, as abordagens descritas nas seções 2.3, 2.4 e 2.5 preocupam-se com os problemas específicos dos subsistemas de controle de modificações, controle de versões e controle de construções e liberações, respectivamente, mesmo que esses problemas envolvam mais de uma das funções da GCS segundo a perspectiva gerencial. Cada subsistema atua em partes ortogonais das funções de GCS, segundo a perspectiva gerencial, e fornece funcionalidades independentes dos demais subsistemas. Por esta razão, as abordagens descritas nessas seções, apesar de não serem abrangentes para toda a GCS, são completas e úteis individualmente dentro do subsistema a que se propõem a atuar. Finalmente, na seção 2.6, são apresentadas as abordagens para a integração dos ambientes de trabalho de GCS e ADS.

2.2 Processos, normas, procedimentos, políticas e padrões de GCS

A GCS, por ser uma área fortemente calcada em controle, é celeiro de uma vasta gama de processos, normas, procedimentos, políticas e padrões. Inicialmente, em 1962, a Força Aérea dos Estados Unidos publicou a primeira norma relacionada com a gerência de configuração, a AFSCM 375-1 (DOD, 1962). Apesar dessa norma separar claramente o processo de engenharia do processo de controle sobre a evolução do produto, apenas produtos físicos (*hardware*) eram levados em consideração até então.

Somente em 1971 foi publicada, também pela Força Aérea dos Estados Unidos, uma norma que relacionava software como possível IC, a MIL Std 483 (DOD, 1971). A partir daí, um grande esforço foi feito para consolidar as diversas iniciativas de estruturação da área de GCS (OLIVEIRA et al., 2001), culminando na norma DOD Std 2167A (DOD, 1985).

Em 1988, o governo dos Estados Unidos indicou que estaria se afastando da responsabilidade de criação de normas, passando essa tarefa para organizações como EIA (*Electronics Industries Association*), IEEE (*Institute of Electrical and Electronics Engineers*), SAE (*Society of Automotive Engineers*), ANSI (*American National Standards Institute*) e ISO (*International Organization for Standardization*).

A partir desta indicação de afastamento do governo dos Estados Unidos da tarefa de produzir normas, as principais referências internacionais no contexto de GCS passaram a ser a ANSI/IEEE e a ISO. Dentre as diversas normas produzidas por essas organizações, que de alguma forma tocam no tema de GCS, as mais importantes são:

- IEEE Std 1042 (IEEE, 1987), que é considerada a norma internacional mais completa sobre GCS;
- IEEE Std 828 (IEEE, 1998), que trata da confecção de planos de GCS e é uma reedição da versão inicial que data de 1983;
- ISO 10007 (ISO, 1995a), que fornece diretrizes para a utilização de GCS na indústria e define a interface da GCS com as demais áreas de gerência; e
- CMM (JALOTE, 1999) e CMMI (CHRISSIS et al., 2003), que fornecem modelos multi-níveis para a classificação de maturidade de empresas de desenvolvimento de software;

Além dessas normas estabelecidas por órgãos internacionais, existem várias publicações relacionadas ao estabelecimento de procedimentos e padrões para uso de sistemas específicos de GCS, como, por exemplo, CVS (FOGEL et al., 2001; VENUGOPALAN, 2002) e Bugzilla (GOLDBERG, 2002).

No restante desta seção, são apresentadas, em maior detalhe, algumas abordagens referentes a processos, normas, procedimentos, políticas ou padrões de GCS (IEEE, 1987; ISO, 1995a; IEEE, 1998; JALOTE, 1999; ESTUBLIER, 2001; CHRISSIS et al., 2003; FIGUEIREDO, 2004).

2.2.1 IEEE Std 1042

A norma IEEE Std 1042 tem como principal objetivo descrever a aplicação da disciplina de GC ao desenvolvimento de software. Essa aplicação é dividida em duas partes: planejamento e implementação. Para apoiar o planejamento, são apresentadas sugestões referentes aos vários aspectos da GCS. Para apoiar a implementação, são descritos quatro exemplos de planos de GCS elaborados segundo a versão de 1983 da norma IEEE Std 828. Esses exemplos consistem em: (1) projeto complexo e crítico; (2) projeto pequeno; (3) projeto somente de manutenção; e (4) projeto de software embutido em hardware.

Nas seções iniciais do plano, são apresentadas algumas siglas comumente usadas na área, como, por exemplo, as de CCC e de IC. São também definidos alguns termos, como, por exemplo, o de configuração de referência, como sendo uma configuração da especificação ou do produto, revisada e aprovada, que serve como base para desenvolvimento futuro. Segundo a norma, o formalismo aplicado às configurações de referencia pode variar em função da flexibilidade que determinados processos de desenvolvimento de software necessitam. A configuração de referência pode ser promovida internamente através dos níveis de análise (configuração de referência funcional), de projeto (configuração de referência alocada) e implementação (configuração de referência de produto). Quando a configuração de referência passa por um processo de auditoria e é entregue ao cliente, recebe o nome de liberação. O mecanismo de rótulo nas versões de um conjunto de artefatos, implementado por diversos sistemas de controle de versões, é sugerido para implementar o conceito de configuração de referência.

Nessa norma, os termos versão e revisão, que são rotineiramente utilizados na literatura como sinônimos, recebem conotações diferentes. As versões de ICs representam evoluções funcionais, enquanto as revisões em ICs representam correções de erros ou melhorias que não modifiquem as funcionalidades dos ICs.

A importância da GCS como processo de apoio ao processo de desenvolvimento de software é enfatizada. A sua efetividade nessa função só pode ser obtida caso ela esteja incorporada ao dia-a-dia do projeto, desde o desenvolvimento até a manutenção. As etapas iniciais de modelagem, que normalmente recebem menor suporte de GCS, devem ser tratadas com maior atenção, por causa da dificuldade de divisão das atividades em tarefas mais granulares, devido à complexidade das mesmas.

Uma característica importante levantada pela norma é o fato da GCS diferenciar da GC de Hardware pela forma que o produto é tratado. No caso do hardware, somente a documentação reside nos repositórios[9]. Contudo, no caso de software, o próprio produto, além da documentação, reside nos repositórios de versão. Essa característica possibilita a introdução de uma maior automação na função de controle da GCS.

O processo de gerência de modificações descrito na norma consiste na identificação das configurações de referência e no acompanhamento das modificações

[9] O termo repositório representa o local de acesso controlado onde as versões dos ICs são armazenadas (LEON, 2000).

aplicadas a essas configurações de referência. Dependendo da etapa do processo de desenvolvimento, determinados artefatos terão maior importância, e constituirão diferentes configurações de referência. Por exemplo, no momento da implementação, os artefatos de projeto, que constituem a configuração de referência alocada, são os de maior importância e devem receber maior atenção por parte da GCS. Isso ocorre porque o código-fonte que está sendo construído em função do projeto pode ficar inconsistente, caso alguma modificação não relatada ocorrer nesses ICs de projeto.

2.2.2 IEEE Std 828

A norma IEEE Std 828, que foi criada em 1983 e sofreu revisões em 1990 e 1998, se preocupa com a atividade de planejamento da GCS. A aplicabilidade dessa norma não está relacionada com o tipo do sistema a ser construído, podendo ser utilizada desde sistemas de informação a sistemas de tempo-real, críticos ou não. O público alvo dessa norma engloba os responsáveis pela confecção do plano de GCS ou pelas auditorias de GCS. A norma ressalta que a GCS é sempre utilizada em projetos de software, seja de forma planejada ou *ad-hoc*. Contudo, o planejamento dessas atividades contribui para o aumento da eficiência das mesmas.

A norma determina que os planos de GCS devem conter, ao menos, as seções de introdução, gerenciamento, atividades, cronograma, recursos e manutenção do plano. Essas seções, apesar de obrigatórias, podem aparecer em qualquer ordem no documento, podem ter partes internas omitidas, desde que justificado na seção de introdução, e podem ser acrescidas de outras seções, caso constatado necessário para o projeto em questão.

A **seção de introdução** deve fornecer uma visão geral do projeto, identificar os artefatos de software e hardware que participarão direta ou indiretamente da GCS, determinar o grau de formalismo e controle em que a GCS será aplicada, estipular sobre quais atividades do ciclo de vida do software a GCS será adotada e descrever quais aspectos geram restrições em custo, cronograma ou na própria possibilidade de execução da GCS.

A **seção de gerenciamento** deve determinar quais organizações participarão das atividades de GCS do projeto, estipulando seus papéis e relacionamentos. Além disso, devem ser claramente descritos, para cada comitê criado pelas organizações, o seu propósito, membros, período de vigência, escopo de autoridade e procedimentos operacionais.

A **seção de atividades**, que usualmente é a mais extensa de todas, é subdividida em outras seis subseções: identificação, controle, acompanhamento, auditoria, interfaces e terceiros. As quatro primeiras subseções, que são equivalentes às funções de GCS, devem estabelecer como essas funções devem ser executadas no contexto do projeto em questão. As últimas duas subseções devem estabelecer como artefatos e equipes fora do escopo do sistema em questão devem ser tratados.

A **seção de cronograma** deve estipular a seqüência e a coordenação das atividades identificadas no plano. O cronograma pode ser descrito usando datas absolutas ou relativas às demais atividades do projeto. O uso de representação gráfica é indicado para esse tipo de informação.

A **seção de recursos** deve determinar as ferramentas, técnicas, equipamentos, pessoal e treinamento necessário para implementar o plano de GCS. Além disso, deve ser estabelecido como cada recurso será obtido.

A **seção de manutenção** deve estipular quem é responsável pela manutenção do plano, com qual freqüência o plano será atualizado e como as modificações no plano serão verificadas, aprovadas e relatadas.

Como todo artefato que evolui durante o ciclo de vida do software, os planos gerenciais do projeto, incluindo o plano de GCS, devem ser tratados como ICs, e pertencer a uma configuração de referência. É desejável a criação de uma configuração de referência gerencial, para permitir o controle individualizado sobre esses artefatos, minimizando o impacto que a evolução desses artefatos pode ter sobre a atividade de análise. Contudo, a primeira configuração de referência definida na norma IEEE Std 1042 é a funcional. Neste caso, esses artefatos gerenciais serão controlados pela GCS somente depois do término da atividade de análise.

Um maior detalhamento sobre as partes que compõem planos de GCS segundo a norma IEEE Std 828, juntamente com exemplos de suas aplicações, pode ser obtido na seção 3 e nos apêndices da norma IEEE Std 1042.

2.2.3 ISO 10007

A norma ISO 10007, que fornece diretrizes para a GC, tem como propósito aumentar o entendimento comum sobre o assunto e incentivar o uso de GC, alinhando a abordagem na indústria. Essa norma visa atender os requisitos de GC apresentados nas normas da família ISO 9000. Uma característica interessante da norma é sua abrangência. Ela não foca somente na GCS, mas em GC de qualquer produto. Contudo,

em algumas partes do seu corpo são levantadas questões especificamente relacionadas a software.

Nas suas seções iniciais são definidas terminologias. As maiores diferenças de terminologia em relação aos demais trabalhos da literatura são: o uso do termo configuração básica no lugar do termo configuração de referência, o uso do termo conselho de controle de configuração no lugar do termo comitê de controle de configuração, o uso do termo gestão de configuração no lugar de gerência de configuração e o uso do termo contabilização da situação de configuração no lugar do termo acompanhamento da configuração.

Nas seções seguintes da norma é apresentada a visão geral sobre o assunto, onde questões semelhantes às levantadas pelas normas IEEE Std 1042 e IEEE Std 828 são relatadas. Em especial, é discutida a necessidade de auditoria sobre os próprios planos e procedimentos de GC, a seleção de IC usando técnicas de decomposição e o uso de rastros entre as alterações e suas configurações de referência.

Assim como apontado pela norma IEEE Std 1042, essa norma salienta que a GC pode e deve ser aplicada sobre o próprio produto quando o mesmo é constituído de software, além do controle sobre a documentação do produto. Uma discussão introduzida nessa norma, que não é tocada nas demais, se refere ao grão de seleção de ICs. A norma indica que caso o grão seja fino, o número de ICs será grande, e isso poderá afetar a visibilidade do produto, dificultar o gerenciamento e aumentar o custo de operação. Por outro lado, se o grão for grosso, o número de ICs será pequeno, e isso poderá gerar dificuldades de logística e manutenção, limitando as possibilidades de gerência. Esse problema é contornado pelas ferramentas atuais através do uso de granularidade dinâmica de ICs em função das requisições de modificação.

São também descritos na norma os critérios para seleção dos ICs, a função do CCC, as informações necessárias para pedidos de modificação, os critérios para a avaliação dos pedidos de modificação e os tipos de relatório mais comuns da função de acompanhamento. Um fato colocado explicitamente nessa norma é a existência de dependência entre a função de acompanhamento e as funções de identificação e controle. A função de acompanhamento só será executada de forma correta se as funções de identificação e controle forem capazes de coletar as informações apropriadas.

No anexo A dessa norma, a estrutura desejada de um plano de GC é apresentada. Essa estrutura diverge da estrutura definida na norma IEEE Std 828. Contudo, o

conteúdo é basicamente o mesmo, organizado de forma diferente. No anexo B, é apresentada uma tabela de referência cruzada entre as seções dessa norma e as seções das normas ISO 9001, ISO 9002, ISO 9003 e ISO 9004-1.

2.2.4 CMM e CMMI

A versão 1.1 do CMM, lançada em 1993, é considerada o modelo de maturidade mais utilizado mundialmente (HASS, 2003). Esse modelo define cinco níveis de maturidade: (1) Inicial; (2) Repetível; (3) Definido; (4) Gerenciável; e (5) Otimizado. Para cada nível de maturidade, são estipuladas áreas chave de processo que devem ser atendidas, possibilitando que esse nível seja alcançado.

A GCS pode contribuir indiretamente para atender a áreas chave de processo existentes em vários níveis de maturidade, como, por exemplo, prevenção de defeitos (nível 4) e gerenciamento quantitativo do processo (nível 5). Contudo, sua maior contribuição está no nível 2 de maturidade, em uma área chave de processo batizada com o seu nome.

Os objetivos da área chave de processo de GCS são: (1) planejamento das atividades de GCS; (2) identificação e controle dos ICs; (3) controle dos pedidos de modificação; e (4) propagação das informações aos indivíduos afetados. Para que esses objetivos possam ser atendidos, um conjunto de atividades é definido: (1) preparação do plano de GCS para cada projeto; (2) aprovação do plano de GCS para guiar as demais atividades; (3) estabelecimento de um repositório para as configurações de referência; (4) identificação dos ICs; (5) acompanhamento dos pedidos de modificação segundo processo previamente acordado; (6) controle sobre modificações em configurações de referência; (7) controle sobre liberação das configurações de referência; (8) armazenamento das informações necessárias; (9) relato do andamento das atividades de GCS e do estado atual das configurações de referência; e (10) auditoria sobre as configurações de referências.

Um fato que contribuiu com a propagação da disciplina de GCS nas grandes empresas de software foi a determinação, feita pelo departamento de defesa dos Estados Unidos, onde os principais contratos de software obrigavam às empresas interessadas a estarem ao menos no nível 3 de maturidade do CMM (CHRISTENSEN et al., 2002). Como a GCS é área chave de processo para o nível 2, todas essas empresas foram obrigadas indiretamente a adotar GCS.

O modelo CMMI-SE/SW/IPPD/SS teve a sua versão 1.1 lançada em 2002. Essa versão se baseia nos rascunhos da versão 2.0 do CMM. O CMMI define duas representações de modelos de maturidade: em estágios (SEI, 2002b) e contínua (SEI, 2002a). A representação em estágios é similar ao modelo de maturidade definido pelo CMM. A representação contínua introduz o conceito de nível de capacidade, que difere do conceito de nível de maturidade usado pela representação em estágios por focar de forma individual nas áreas chave de processo, ao invés de lidar com o processo organizacional como um todo.

Dentro do CMMI, a GCS é área chave do grupo de processos de suporte. Essa área chave é composta por três objetivos específicos: (1) estabelecimento de configurações de referência; (2) acompanhamento e controle de modificações; e (3) manutenção da integridade das configurações de referência. Para cada objetivo da área chave de GCS, são identificadas práticas específicas: (1.1) identificação dos ICs; (1.2) estabelecimento de sistema de GCS; (1.3) criação e liberação de configurações de referência; (2.1) acompanhamento de modificações; (2.2) controle das modificações; (3.1) registro das informações; e (3.2) auditoria da configuração.

A representação contínua do CMMI é composta por seis níveis de capacidade: (0) Incompleto; (1) Executável; (2) Gerenciável; (3) Definido; (4) Quantitativamente Gerenciável; e (5) Otimizado. Cada um desses níveis de capacidade, a partir do nível 1, tem objetivos genéricos associados. Para cada objetivo genérico, são definidas práticas genéricas que devem ser atendidas para que a área de processo cumpra o objetivo genérico e possa ser classificada no nível de capacidade que esse objetivo genérico está relacionado. A GCS é a prática genérica número 6 do objetivo genérico número 2, relacionado ao nível de capacidade Gerenciável. Desta forma, a GCS é requisito para que qualquer área chave de processo possa atingir o nível 2 de capacidade.

HASS (2003) fornece um guia incremental detalhado para a adoção da GCS através dos níveis de capacidade CMMI. Esse guia estabelece, no nível 1, os passos necessários para atender a todos os objetivos de GCS definidos no CMMI. No nível 2, os passos necessários para planejar, executar, monitorar e controlar a GCS em projetos individuais. No nível 3, os passos necessários para instanciar processos a partir de adaptações do processo padrão da organização. No nível 4, os passos necessários para fazer uso de técnicas quantitativas para medir o processo. Finalmente, no nível 5, os passos necessários para melhorar o processo em função da interpretação das estatísticas obtidas.

2.2.5 Políticas de Controle de Objetos

Em um trabalho prático, ESTUBLIER (2001) discute a falta de escalabilidade das soluções existentes para controle de versões e acrescenta que nenhuma solução atual, baseada em repositório central, pode suportar grandes projetos. O exemplo citado consiste no desenvolvimento de um software de quatro milhões de linhas de código por uma equipe de 800 desenvolvedores. Neste contexto, nos horários de pico, que ocorrem no início do dia e no final do dia, as atualizações do ambiente local de cada desenvolvedor demandavam da rede e do servidor uma carga que não era possível de ser atendida a contento.

Ainda neste contexto, o uso de modelos de concorrência pessimistas torna-se impraticável, pois em média três desenvolvedores atuam sobre o mesmo artefato em paralelo, com picos de até trinta desenvolvedores. Como as transações de desenvolvimento são longas, políticas de bloqueio, como as adotadas em sistemas de gerenciamento de banco de dados, se tornam inviáveis.

Para contornar o problema de escalabilidade, é proposta a criação de grupos onde um ambiente de trabalho serve como integrador para os demais ambientes de trabalho do grupo. Em um segundo nível, todos os ambientes de trabalho integradores dos grupos são servidos por um outro ambiente integrador, e assim sucessivamente. Esse modelo em níveis, que pode ser encontrado em outros trabalhos da literatura (LINGEN et al., 2004), diminui a sobrecarga sobre um único servidor central.

Para estabelecer a mecânica da colaboração entre os ambientes de trabalho dentro de um grupo, foram definidas seis ações atômicas, que são: modificar, propor, integrar, sincronizar, reservar e liberar. Além disso, foram definidas três políticas, que são: exclusão, reserva tardia e reserva tardia com integração. Cada política é descrita através das ações atômicas. Por exemplo, a política de exclusão é descrita por: (1) sincronizar, (2) reservar, (3) modificar, (4) propor, (5) integrar e (6) liberar.

2.2.6 GConf

FIGUEIREDO (2004) apresenta um processo de GCS para ambientes de desenvolvimento de software orientados a organização (ADSOrg) baseado na norma ISO/IEC 12207 (ISO, 1995b), no processo de GCS do SWEBOK (SCOTT et al., 2001) e no modelo CMM. Os ADSOrgs enfatizam a necessidade da gestão do conhecimento relacionado com a produção de software de uma determinada organização. Esse conhecimento, obtido em projetos anteriores, é utilizado como diferencial no apoio à

execução das atividades de engenharia de software, dentre elas as relacionadas com GCS.

O processo proposto faz uso da infra-estrutura existente, provida pela estação TABA (ROCHA et al., 1990; TRAVASSOS, 1994), que permite a propagação de conhecimento entre os participantes das atividades de GCS. Esses participantes, segundo o GConf, são agrupados em dois papéis: gerentes de projeto e desenvolvedores. Esses papéis colaboram na execução de três atividades principais, que são: planejar GC, controlar liberação e distribuição e controlar configuração.

A **atividade de planejamento da GC** abrange as atividades de definição e implementação do processo, definida nos três processos em que esse processo se baseia, e de identificação da configuração, descrita tanto na ISO/IEC 12207 quanto no SWEBOK. Essa atividade é restrita aos gerentes de projeto, e tem por objetivo preparar a infra-estrutura necessária, tanto no que diz respeito a processos quanto no que diz respeito à definição de produtos que irão compor os itens de configuração, para que o desenvolvimento de software possa ser iniciado.

A **atividade de controle de liberação e distribuição** permite que determinadas versões de ICs sejam obtidas pelos desenvolvedores. Segundo o autor, essa atividade está provendo serviços semelhantes às atividades de acompanhamento da situação, definida nos três processos base, e de gerência de liberação e entrega, definida nos processos ISO/IEC 12207 e SWEBOK. Contudo, a semântica para liberação descrita nos processos base está fortemente relacionada com a entrega de versão fechada ao usuário final, e não com a entrega de versão em desenvolvimento aos desenvolvedores.

Finalmente, a **atividade de controle da configuração** define um processo de controle de pedidos de modificação composto pelas sub-atividades de requisição da alteração, análise do pedido, implementação da alteração e verificação dos ICs alterados. Devido à simplicidade do processo sugerido, que exclui a atividade de classificação e junta a atividade de análise com a atividade de avaliação em uma única atividade de análise exercida pelo gerente do projeto, o papel de gerente de projeto fica sobrecarregado com atribuições referentes ao desenvolvimento. A criação de um novo papel, referente ao analista de impacto, poderia aliviar a carga de trabalho sobre o gerente do projeto. Além disso, seria interessante possibilitar que o próprio cliente pudesse solicitar modificações diretamente no sistema, como determinado pela norma ISO 10007. Da forma que foi concebida, essa atividade abrange a atividade de controle

da configuração, definida nos três processos base, e auditoria da configuração, definida nos processos ISO/IEC 12207 e SWEBOK.

2.2.7 Considerações sobre processos, normas, procedimentos, políticas e padrões

A norma IEEE Std 1042, que é considerada a norma mais completa de GCS, fornece exemplos de utilização da norma IEEE Std 828, que enfatiza a construção do plano de GCS. Além disso, apresenta um modelo evolutivo para a classificação das ferramentas de GCS. Contudo, a primeira configuração de referência considerada é a funcional, o que implica na falta de controle sobre ICs gerenciais antes do término da atividade de análise.

As normas ISO 10007 e IEEE Std 1042 levantam considerações especiais referentes ao contexto de GCS. Quando o IC é *hardware*, somente a documentação do IC pode ser controlada. Entretanto, no caso de software, o IC propriamente dito pode ficar sob controle dos sistemas de GCS. Além disso, apesar das seções propostas pela ISO 10007 para o plano de GCS serem diferentes das seções propostas pelo IEEE Std 828, ambos os planos contém as mesmas informações, só que organizadas de forma diferente.

Diferentemente das normas, os modelos CMM e CMMI apresentam um mecanismo evolutivo para adoção das funções de GCS. Já o processo proposto pelo GConf relaciona características do modelo CMM com características existentes na norma ISO/IEC 12207 e com o SWEBOK.

2.3 Sistemas de controle de modificações

A GCS é fortemente calcada em processos e normas (IEEE, 1987; ISO, 1995a), que definem os procedimentos necessários para permitir uma evolução controlada do desenvolvimento de software. Para automatizar a execução desses processos, são utilizados sistemas de controle de modificações.

Inicialmente, os sistemas de controle de modificações tinham seu foco principal em modificações corretivas, mas, com o amadurecimento da área, esses sistemas passaram a ser utilizados para qualquer tipo de modificação. Em alguns casos, quando o processo de desenvolvimento prioriza a geração precoce de liberações, os sistemas de controle de modificações passam a ser utilizados desde o início do desenvolvimento,

exercendo a função de máquina de processo e recebendo o nome de sistema de controle de requisições.

Um dos sistemas de controle de modificações mais conhecidos e utilizados em projetos de software livre é o Bugzilla (BARNSON et al., 2003). O Bugzilla provê suporte à busca detalhada de modificações, criação de vínculos entre modificações, controle do estado das modificações e relacionamento entre modificações e os artefatos alterados propriamente ditos. Além disso, uma preocupação constante no projeto é a necessidade de um alto desempenho da ferramenta. Para atingir esse objetivo, o Bugzilla faz uso de banco de dados relacional e interfaces HTML (*Hypertext Markup Language*) (W3C, 1999).

Contudo, existe atualmente um número relativamente grande de sistemas com características semelhantes ao Bugzilla (CMTODAY, 2004a; CMTODAY, 2004b), como, por exemplo, o Scarab (TIGRIS, 2004), que apresenta como principal diferencial o uso da linguagem Java, e o ClearQuest (WHITE, 2000), que faz parte da iniciativa da IBM Rational de prover um ambiente integrado de GCS, conhecido como UCM (*Unified Change Management*).

Apesar dessa diversidade de sistemas de controle de modificações, essa área ainda é carente de pesquisa que faça uso da infra-estrutura existente com o objetivo de atingir maior qualidade e produtividade no desenvolvimento de software. Por exemplo, a capacidade de configuração do sistema de controle de modificações em relação aos processos de GCS é uma característica desejável.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a sistemas de controle de modificações (SCHNEIDER, 2001; ESTRADA, 2003; BRIAND et al., 2003; KNETHEN et al., 2003).

2.3.1 GCCS

Com o intuito de aumentar o grau de colaboração durante a GCS, foram criados um método e um ferramental de apoio à Gerência Cooperativa de Configuração de Software (GCCS) (SCHNEIDER, 2001). Esse método tem como pontos chave o controle de versões, a gerência de modificações, o controle sobre o processo e a parametrização do nível de colaboração desejado.

O modelo de controle de modificações adotado pelo GCCS herda as atividades definidas no contexto do ClearQuest, que são: (1) requisição; (2) análise; (3) avaliação; (4) implementação; (5) verificação; e (6) fechamento. Para armazenar as informações

geradas durante a execução do processo de controle de modificações, foi utilizado o sistema de gerenciamento de banco de dados InterBase (BORLAND, 2004a). Para prover controle de versões à atividade de implementação foi utilizado o CVS (CEDERQVIST, 2003).

O GCCS pode ser acessado, através de um *plug-in*, diretamente do ambiente de modelagem ou de programação que o usuário está habituado a utilizar. Além disso, um agente, situado na barra de ferramenta do sistema operacional, permite a execução automática de tarefas rotineiras relacionadas à coordenação e comunicação entre os membros da equipe e a base de dados. As principais tarefas desse agente são: sincronizar os dados dos usuários, registrar usuários ativos nos ambientes dos demais usuários, procurar por usuários ativos, permitir a comunicação síncrona e assíncrona entre os usuários, controlar as seções de trabalho ou de revisão, de forma síncrona ou assíncrona, e avisar a ocorrência de eventos de reuniões agendadas.

Dentre os parâmetros configuráveis do GCCS estão: a periodicidade de controle, que determina de quanto em quanto tempo cada usuário deve relatar o andamento das suas tarefas, a data de execução de controle, que estabelece a execução periódica de controle, e o tipo do projeto, que determina quais artefatos devem ser armazenados no repositório.

2.3.2 ConfigCASE e SoSoft

ESTRADA (2003) detectou a ineficiência dos processos e normas de GCS existentes no contexto de pequenas e médias empresas. O seu principal argumento é que esses processos e normas foram criados pensando em cenários mais complexos, existentes em grandes empresas, o que torna sua aplicação inviável em empresas que não podem arcar com o nível de burocracia associada. Um exemplo é o questionamento da efetividade de um CCC neste contexto. Como resultado desse cenário, é apresentada uma estatística que aponta 73% do universo das empresas de software como não utilizadoras de GCS.

Para fornecer uma solução que atendesse à pequena e média empresa, com até 25 empregados, representando 8% do total das empresas de software, foram definidos cinco processos, que abrangem a identificação de itens de configuração, o controle de modificações, o controle de versões, o planejamento e a geração de informações referentes ao estado da configuração.

Juntamente com os processos, foram definidas 17 métricas, sendo que 8 delas são uni-processos e 9 são bi-processos. Uma métrica uni-processo é definida pela dependência dos dados gerados por um, e somente um, dos cinco processos definidos. Já as métricas bi-processo correlacionam dados produzidos por dois dos cinco processos definidos.

Como ferramental, foram construídos os sistemas ConfigCASE e SoSoft para a automatização dos processos e da coleta de métricas. Além disso, é proposta a utilização do Visual SourceSafe (ROCHE et al., 2001) para apoiar o processo de controle de versões.

A abordagem, apesar de diferenciar pequenas e médias empresas das grandes empresas, e ressaltar a necessidade de utilização de métricas durante os processos de GCS, deixa a desejar exatamente no emprego dessas métricas para obter benefícios tangíveis no desenvolvimento de software das pequenas e médias empresas. Questões como a utilização dos resultados obtidos pela aplicação das métricas para a construção de rastros, aprendizado com projetos similares e análise de impacto das modificações são colocadas como trabalhos futuros.

2.3.3 Análise de Impacto

Uma das atividades mais importantes do processo de GCS é a análise de impacto. É a partir dos laudos elaborados durante essa atividade que o CCC pode tomar as decisões referentes à aprovação ou não dos pedidos de modificação. Devido a essa importância, a atividade de análise de impacto é responsável por grande parte do tempo gasto durante o ciclo de vida de uma modificação.

Podem ser detectadas diferentes linhas de trabalho para automatizar, ao menos de forma parcial, a atividade de Análise de Impacto. BRIAND et al. (2003) definem uma abordagem, implementada através da ferramenta iACMTool, para a análise da propagação do impacto de modificações já efetuadas. Para atingir esse objetivo, o modelo UML (*Unified Modeling Language*) (OMG, 2003b) construído para contemplar a modificação é comparado com o modelo UML original, fazendo uso de regras descritas em OCL (*Object Constraint Language*) (OMG, 2003b).

As regras OCL para auxílio à análise de impacto são agrupadas em três categorias: regras de consistência, regras de classificação e regras de impacto. As 120 regras de consistência têm por objetivo verificar se os modelos estão consistentes com a especificação da UML. Sem essa avaliação não seria possível confiar na qualidade da

análise de impacto gerada. As 97 regras de classificação têm por objetivo detectar o tipo da modificação em questão. E, finalmente, outras 97 regras de impacto têm por objetivo identificar, para cada tipo possível de modificação, os impactos existentes.

Com objetivos ligeiramente diferentes, KNETHEN et al. (2003) apresentam uma ferramenta denominada QuaTrace que fornece suporte semi-automático para a análise de impacto, fazendo uso dos rastros entre artefatos. A principal diferença de objetivos entre a iACMTool e a QuaTrace é que a QuaTrace atua apoiando a análise de impacto referente à própria modificação e às modificações que são propagadas em função dessa modificação principal. Já a iACMTool atua somente na análise de impacto referente às modificações propagadas, sem apoiar a análise de impacto da modificação que originou as propagações.

A abordagem QuaTrace introduz um novo papel no desenvolvimento de software, o de gerente de requisitos. O gerente de requisitos é responsável por preparar as atividades necessárias para a coleta de rastros no contexto da organização. Além desse papel, os papéis já existentes de engenheiro de requisitos, analista de impacto e mantenedor também são afetados pela abordagem. O engenheiro de requisitos faz uso da QuaTrace durante a construção dos rastros entre os diversos artefatos em desenvolvimento. O analista de impacto faz uso da QuaTrace para analisar, de forma semi-automática, os impactos de uma modificação. Essa análise ocorre através da navegação pelos rastros e possibilita uma estimativa mais precisa dos custos. O mantenedor faz uso da QuaTrace para detectar possíveis propagações da modificação original.

2.3.4 Considerações sobre sistemas de controle de modificações

As abordagens de controle de modificações existentes têm, em sua maioria, um processo predefinido, implementado diretamente no seu código fonte. Essa limitação pode ser encontrada, por exemplo, nas abordagens GCCS, ConfigCASE e SoSoft. Entretanto, essas abordagens apresentam inovações úteis para a GCS. O GCCS introduz novas funcionalidades referentes ao estimulo à colaboração entre os participantes do processo e as abordagens ConfigCASE e SoSoft enfatizam o uso de métricas configuráveis sobre esse processo. O ClearQuest, diferentemente da maioria das abordagens, possibilita alguma configuração sobre o processo e sobre as informações necessárias a cada atividade.

Outras abordagens mais pontuais foram apresentadas. Essas abordagens dão suporte a atividades específicas, sem tratar o processo como um todo. Por essa razão, elas poderiam trabalhar em conjunto com as demais abordagens. Dentre essas abordagens específicas, a iACMTool e a QuaTrace apóiam a análise de impacto utilizando técnicas diferentes, mas com efeitos semelhantes para o usuário.

2.4 Sistemas de controle de versões

O subsistema de GCS que mais recebeu contribuições tanto de pesquisa quanto comercialmente é o de controle de versões. Várias soluções comerciais, como, por exemplo, ClearCase (WHITE, 2000) e Visual SourceSafe, fornecem características satisfatórias para o problema quando os ICs são artefatos simples, definidos através de arquivos do sistema operacional. Além disso, outras tantas soluções livres estão disponíveis, como, por exemplo, CVS e RCS (TICHY, 1985).

Geralmente, essas soluções provêem infra-estrutura para a definição de quais artefatos necessitam de controle de versões e permitem que esses artefatos sejam obtidos, através de um processo conhecido como *check-out*[10], modificados dentro do espaço de trabalho do desenvolvedor e retornados ao repositório, através de um processo conhecido como *check-in*[11]. Além disso, essas soluções normalmente suportam a definição de diferentes políticas de trabalho. Dentre essas políticas, podemos citar a política pessimista, que enfatiza o uso de *check-out* reservado, fazendo bloqueio (*lock*) e inibindo o paralelismo do desenvolvimento sobre o mesmo artefato. Outra política amplamente utilizada é a otimista, que assume que a quantidade de conflitos é naturalmente baixa e que é mais fácil tratar cada conflito individualmente, caso eles venham a ocorrer. A política otimista usa um mecanismo de tratamento de conflito conhecido como junção (*merge*), que une os trabalhos efetuados em paralelo sobre um mesmo IC e produz uma nova versão do IC que contém a soma desses trabalhos.

Existem situações onde uma determinada política é mais indicada do que as demais. Nos casos onde a junção dos trabalhos tende a ser complexa, quando, por exemplo, os ICs não são textuais e a ferramenta que conhece a representação binária dos

[10] O termo *check-out* representa o processo de requisição, aprovação e cópia de ICs do repositório para o espaço de trabalho do desenvolvedor (LEON, 2000).

[11] O termo *check-in* representa o processo de revisão, aprovação e cópia de ICs do espaço de trabalho do desenvolvedor para o repositório (LEON, 2000).

ICs não dá suporte automatizado para junções, é mais indicado trabalhar usando políticas pessimistas. Contudo, na grande maioria das situações referentes ao desenvolvimento de software, as políticas otimistas atendem satisfatoriamente (ESTUBLIER, 2001).

Além desses recursos fundamentais de sistemas de controle de versão, outros recursos um pouco mais elaborados também são encontrados com freqüência. Dentre eles, os mais difundidos são a automação no acesso a configurações de referência, que são representadas por determinadas versões de ICs e identificadas na grande maioria das vezes através do uso de rótulos (*tags*), e o controle de desenvolvimento paralelo com isolamento através de ramos[12], que podem ser unidos novamente à linha de desenvolvimento principal através do processo de junção.

Apesar da variedade de soluções disponíveis, um conjunto de novas características é desejável e novas pesquisas estão sendo feitas com o intuito de atender a essa demanda (ESTUBLIER, 2000). Dentre as características desejadas, podemos citar:

- Metáfora uniforme para o versionamento de itens de configuração primitivos e compostos;
- Mecanismo e versionamento não intrusivo nas tarefas já complexas de desenvolvimento;
- Necessidade de um conceito de IC que não traga as heranças indesejáveis existentes em arquivos do sistema operacional;
- Suporte aos diferentes níveis de abstração existentes no desenvolvimento de software, contemplando de modelos a código;
- Mecanismos de armazenamento eficientes e confiáveis, fazendo uso de banco de dados dentro do paradigma adotado;
- Escalabilidade e disponibilidade compatível com a importância do problema em questão.

[12] O termo ramo (*branch*) representa versões temporárias que não seguem a linha principal do desenvolvimento (LEON, 2000).

Além dessas características técnicas, fatores como custo reduzido de aquisição e manutenção e facilidade na adição de novas funcionalidades também devem ser levados em conta.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a sistemas de controle de versões (MORO et al., 2001; CHIEN et al., 2001; COBENA et al., 2002; WANG et al., 2003; LINGEN et al., 2004; HNETYNKA et al., 2004; EL-JAICK, 2004).

2.4.1 TVM

Com o objetivo de desvincular o controle de versões do conceito de arquivos de sistema operacional, SILVA et al. (2003) fizeram uso do modelo temporal versionado TVM (MORO et al., 2001) aplicado a uma perspectiva de controle de versões. Nesse modelo, o IC passa a ser um objeto, em contraposição às soluções do estado da prática, que fazem uso de arquivos do sistema operacional. Apesar dessa prática ser positiva (ESTUBLIER, 2000), a implementação do modelo ocorre sobre um banco de dados relacional. Para acessar as versões de forma transparente, é utilizada a linguagem TVQL (MORO et al., 2002), que é uma extensão da sintaxe de SQL. Para armazenar os objetos, o modelo faz uso de versionamento baseado em estados, onde cada versão armazenada contém toda a informação necessária para representar a versão. Essa modalidade de versionamento oferece menores tempos de acesso às versões se comparada com a de versionamento baseado em diferenças[13], mas consome mais espaço em disco e banda de rede.

2.4.2 DVM

HNETYNKA et al. (2004) argumentam a necessidade de versionamento para modelos armazenados em repositórios MOF (*Meta Object Facility*) (OMG, 2002a) e a insuficiência das soluções tradicionais para esse fim, como, por exemplo, CVS. Mais ainda, é ressaltado que as respostas enviadas para a requisição de propostas (RFP – *Request for Proposals*) de versionamento MOF, publicada pela OMG, não são suficientes para tratar as questões de distribuição envolvidas no problema.

---

[13] O termo diferença (*delta*) representa o que mudou entre duas versões consecutivas (LEON, 2000). A técnica de diferença para frente (*forward delta*) armazena a versão mais antiga do IC e as diferenças para as versões posteriores. Já a técnica de diferença para trás (*reverse delta*) armazena a versão mais recente do IC e as diferenças para as versões anteriores.

Para contornar essas deficiências, foi definido um conjunto de regras que fazem uso de identificadores de localidade na construção do identificador de versão dos elementos armazenados em repositórios MOF distribuídos. Essas regras permitem a criação de versões na linha corrente de desenvolvimento somente no mesmo repositório, mas possibilitam a criação de ramos em repositórios distribuídos desde que esses ramos tenham identificadores de versão que encadeiem os nomes dos seus nós originais. A implementação de um repositório MOF com suporte a essas regras está em andamento.

2.4.3 MIMIX

Com a adoção do MOF como meta-modelo da UML, e da especificação XMI (*XML Metadata Interchange*) (OMG, 2003c) como mecanismo para externar modelos (M1) criados no contexto de meta-modelos MOF (M2), tornou-se possível compartilhar modelos UML criados em diferentes ferramentas CASE.

O MIMIX (EL-JAICK, 2004) foi criado com o intuito de controlar a evolução desses modelos. As ferramentas CASE podem acessar o MIMIX, internamente ou externamente ao seu ambiente, usando *Web Services* (W3C, 2001a). O MIMIX é capaz de armazenar diversas versões de modelos UML contidas em documentos XMI e executar junção entre essas versões, caso necessário.

Apesar do versionamento de modelos UML ser extremamente importante, a solução adotada pelo MIMIX é deficiente no que diz respeito ao grão de versionamento utilizado. Como o modelo é versionado como um todo, não é possível acompanhar a evolução individual de cada elemento de modelagem da UML, como, por exemplo, classes e casos de uso. Em sistemas grandes, compostos por centenas de casos de uso e milhares de classes, esse requisito é fundamental para possibilitar uma evolução controlada do modelo e facilitar a coleta de métricas sobre essa evolução.

2.4.4 Versionamento de XML

Os sistemas de controle de versões necessitam de algoritmos para a comparação de artefatos e detecção dos pontos em que esses artefatos foram modificados. O algoritmo LCS (*Longest Common Subsequence*) (MYERS, 1986) foi criado com o intuito de detectar diferenças entre documentos de texto. Esse algoritmo está implementado na ferramenta GNU diff (FSF, 2002), e é usado pela maioria dos sistemas de controle de versões atuais, como, por exemplo, o RCS e o CVS.

Contudo, quando o artefato que está sendo manipulado contém uma estruturação própria, apesar de ser um documento de texto, esse tipo de algoritmo passa a gerar resultados não desejados. No caso dos documentos XML (*Extensible Markup Language*) (W3C, 2004), o conceito de linha como grão de comparação não é o ideal, já que o XML define estruturas próprias que seriam recomendadas para esse fim, como, por exemplo, atributos e elementos. Na literatura existem diversas propostas para a comparação e versionamento de documentos XML (CHIEN et al., 2001; COBENA et al., 2002; WANG et al., 2003). Dentre os sistemas comerciais de controle de versões, alguns tratam de forma integrada alguns tipos específicos de arquivos ou permitem o acoplamento de tratadores externos. O ClearCase é um exemplo de sistema de controle de versões com tratamento especial para XML, DOC (MICROSOFT, 2004b) e MDL (IBM RATIONAL, 2004).

2.4.5 MCCM

O MCCM (LINGEN et al., 2004), continuação de um trabalho desenvolvido por HOEK et al. (2002) intitulado NUCM, é um sistema de controle de versões baseado na composição de políticas. A motivação desse trabalho está na complexidade de se construir do zero um sistema de controle de versões, juntamente com a necessidade constante de criar novos sistemas de controle de versões que atendam a políticas particulares de desenvolvimento. Assim sendo, com o uso do MCCM, é possível instanciar conjuntos de políticas que definam de que forma um sistema de controle de versões deve se portar e implementar em poucas linhas de código a interação entre essas políticas.

Dois tipos de políticas são definidos: políticas de restrição e políticas de ação. As políticas de restrição são definidas no contexto do servidor do sistema de controle de versões e as políticas de ação são definidas no contexto dos clientes. O efeito do uso das políticas de restrição é diminuir o espaço de opções do usuário, e o efeito do uso das políticas de ação é definir quais caminhos seguir dentro desse espaço de opções já imposto pelas políticas de restrição. Desta forma, semelhantemente ao trabalho de ESTUBLIER (2001), este trabalho permite a combinação de políticas específicas para a construção da política geral do sistema de controle de versões. Além disso, novas políticas podem ser especificadas e implementadas para atender a requisitos específicos de sistemas de controle de versão futuros.

Cinco políticas de restrição já foram especificadas e implementadas. Essas políticas atendem às seguintes necessidades: (1) armazenamento, que possibilita a definição das situações onde é necessário persistir o IC completo ou somente a diferença entre versões consecutivas do IC; (2) composição, que permite a definição de como os ICs se compõem e como as modificações em ICs compostos podem ser representadas através de modificações em ICs primitivos; (3) concorrência, que define se os ICs poderão ser modificados de forma otimista ou pessimista; (4) distribuição, que estipula se os ICs estarão em somente um servidor ou em múltiplos servidores para contemplar requisitos de alta disponibilidade e desempenho; e (5) seleção, que permite a definição das possibilidades de construção da área de trabalho sem tornar a mesma inconsistente.

Além dessas políticas localizadas no servidor, outras cinco políticas, localizadas no cliente, foram definidas: (1) evolução, que estabelece como serão representadas as diversas versões dos artefatos e como serão os relacionamentos de dependência entre essas versões; (2) hierarquia, que especifica de que forma devem ser propagadas as operações como, por exemplo, a de bloqueio, dentro de uma estrutura de composição de ICs; (3) bloqueio, que determina quais outros elementos, como, por exemplo, ICs e configurações de referência, devem ser bloqueados juntamente com os ICs selecionados; (4) colocação, que estabelece quando um IC deve ou não ser replicado em outros servidores; e (5) população, que possibilita a seleção consistente de dependências entre ICs.

Experimentalmente, essas políticas foram compostas para construir comportamentos semelhantes aos sistemas RCS, CVS, Subversion (COLLINS-SUSSMAN B. et al., 2004) e Aide de Camp (SMDS, 1994). Como resultado, foi constatado que, com menos de 700 linhas de código extras, essas políticas puderam ser combinadas para emular cada um desses sistemas. Apesar de nenhum desses sistemas emulados ser completamente funcional, os indícios obtidos levantam a possibilidade de reutilização efetiva na construção de sistemas de controle de versão através do uso de meta-sistemas baseados em políticas.

Outra característica interessante do MCCM é a sua arquitetura distribuída em servidores ligados através de conexões *peer to peer*, obtidas através da tecnologia de chamadas remotas de métodos RMI (*Remote Method Invocation*).

2.4.6 Considerações sobre sistemas de controle de versões

Das abordagens apresentadas, o TVM, o DVM e o MIMIX têm como principal contribuição o apoio ao versionamento de objetos, rompendo com a metáfora de sistema de arquivos. Contudo, o TVM faz uso de sistemas de banco de dados relacional para armazenar os objetos, além de se basear em um meta-modelo proprietário. O DVM, por sua vez, introduz características de distribuição sobre ICs descritos via meta-modelo MOF, mas não tem nenhuma versão implementada até o momento.

O MIMIX, apesar de se propor a versionar elementos MOF transportados via XMI, se baseia em XML para fazer comparações entre documentos e não possibilita que a versão individual de cada elemento MOF possa ser controlada. Esta solução apresenta similaridade com as demais soluções de versionamento de XML apresentadas nesta seção.

Finalmente, o MCCM fornece um ambiente que possibilita a configuração de políticas de GCS. Todavia, a sua abordagem se baseia na metáfora de sistema de arquivos e os sistemas de versionamento, obtidos através da aplicação das políticas, não são completamente funcionais.

2.5 Sistemas de controle de construções e liberações

Os procedimentos para a construção e liberação de sistemas complexos podem tomar grande parte do tempo de desenvolvimento, especialmente quando essas atividades se repetem com freqüência.

Em processos ágeis de desenvolvimento, como, por exemplo, o de eXtreme Programming (XP) (BECK, 1999), é aplicado o conceito de integração contínua (FOWLER et al., 2004), que visa compilar e testar o sistema várias vezes durante o dia para evitar surpresas futuras. Esse tipo de procedimento só se torna viável com suporte de ferramental apropriado, automatizando a tarefa.

A construção do sistema, que consiste na geração de ICs derivados[14] para uma configuração alvo a partir de um conjunto de ICs fonte[15], já é apoiada em parte por ferramentas de compilação e link-edição. Contudo, em sistemas grandes, esse

[14] O termo IC derivado representa um IC que pode ser obtido a partir de outros ICs pela aplicação de um procedimento de construção, não sendo necessário o seu armazenamento no repositório.

[15] O termo IC fonte representa um IC que serve como origem para a geração de ICs derivados via um procedimento de construção.

procedimento não consiste somente nessas tarefas. Funcionalidades como a definição de variáveis de ambiente, seleção de módulos que devem fazer parte do programa gerado segundo as variáveis de ambiente definidas e a ordenação da compilação dos ICs fonte são exemplos de aplicações para sistemas tradicionais de controle de construção, como, por exemplo, o Make (FELDMAN, 1979).

Os sistemas de controle de construção evoluíram muito desde suas primeiras implementações. Novas funcionalidades, que incluem a execução independente de plataforma, o acesso ao repositório de versões para obter os ICs fonte, a automação da execução de testes e publicação dos resultados, o empacotamento de liberações temporárias, conhecidas como *nightly build*, e a cópia das liberações que passaram nos testes para servidores visíveis aos responsáveis de testes beta passaram a ser requisitos necessários aos sistemas de construção e liberação modernos, como, por exemplo, o Ant (HATCHER et al., 2004).

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a sistemas de controle de construções e liberações (HATCHER, 2001; FOWLER et al., 2004; MOZILLA, 2004b).

2.5.1 Tinderbox

O Tinderbox (MOZILLA, 2004b) é uma ferramenta de controle da qualidade que visa evitar que sistemas em desenvolvimento fiquem corrompidos por um longo período de tempo. O seu funcionamento na versão 1.0 depende do Bonsai (MOZILLA, 2004a), que é um sistema que possibilita a execução de uma grande variedade de consultas relacionando as informações coletadas pelo CVS. Todavia, a versão 2.0 do Tinderbox não necessita mais do Bonsai.

A atuação do Tinderbox consiste na verificação constante da compilação dos sistemas em desenvolvimento. Caso algum dos sistemas não compile, é feito um conjunto de consultas ao Bonsai para detectar quais são os desenvolvedores suspeitos da sua quebra[16]. Essas consultas procuram por desenvolvedores que fizeram modificações desde a última vez que o sistema foi compilado com sucesso. Os desenvolvedores

[16] O termo quebra representa a situação onde uma falha de compilação ocorre após a implementação de alguma modificação no sistema. As quebras são nocivas ao desenvolvimento em equipe, pois outros integrantes da equipe estarão impedidos de efetuar testes sobre as suas modificações devido à impossibilidade de compilar o código-fonte.

suspeitos da quebra são notificados de imediato, para que o código seja revisto e o problema que originou a quebra seja sanado.

2.5.2 Automação de testes

Após o processo de construção, normalmente ocorre a execução de baterias de testes para assegurar que o produto liberado esteja correto. Esses testes são capazes de detectar, em um grau limitado de precisão, a ocorrência de quebras lógicas[17]. Diversas técnicas são utilizadas para a automação da execução dos testes em ambientes de integração contínua (HATCHER, 2001; FOWLER et al., 2004). Geralmente, essas técnicas fazem uso da combinação de sistemas de construção e *frameworks* de testes.

O sistema definido por FOWLER et al. (2004) roda em uma máquina dedicada, com quatro processadores, que compila o código-fonte dos sistemas em desenvolvimento de forma contínua, sempre que algum desenvolvedor faz *check-in*. Após a construção, que é apoiada pela ferramenta Ant, o sistema é implantado[18] em um ambiente de desenvolvimento para a execução dos testes. Caso os testes rodem de forma correta, a versão que foi testada é rotulada no repositório de versões. Um e-mail é enviado, logo após a execução dos testes, para todos os desenvolvedores que fizeram *check-in* desde a última verificação. Esse e-mail notifica os resultados dos testes, transferindo para os desenvolvedores suspeitos de quebra, ou quebra lógica, a responsabilidade de consertá-las.

Os ambientes que fazem uso desse tipo de procedimento normalmente impõem novos requisitos aos desenvolvedores de software. Nesse contexto, um requisito novo é verificar a caixa de e-mail, após fazer *check-in*, para ver se o seu trabalho introduziu quebra no sistema. Há alguns anos esse tipo de requisito seria inviável, devido à demora dos procedimentos de compilação. Contudo, atualmente, a compilação e a execução de testes sobre um sistema de 200.000 linhas de código pode durar menos de quinze minutos, dependendo da máquina a ser utilizada para essa tarefa, o que torna os procedimentos de testes automáticos viáveis.

---

[17] O termo quebra lógica representa a situação onde o código-fonte compila de forma correta, mas o sistema não funciona por um erro na lógica de execução. As quebras lógicas são mais difíceis de serem detectadas, pois passam despercebidas pela ferramenta de análise sintática do processo de compilação.

[18] O termo implantação (*deploy*) representa a arrumação estratégica do sistema no ambiente de desenvolvimento ou produção (TECHTARGET, 2004).

2.5.3 Considerações sobre sistemas de controle de construções e liberações

A ferramenta Tinderbox, apresentada nesta seção, fornece avanços consideráveis para o apoio ao desenvolvimento distribuído de sistemas em comparação à utilização de ferramentas de controle de versões isoladamente. Com a sua característica de integração contínua, é possível detectar, além dos conflitos, as quebras de compilação do sistema.

Para complementar esse cenário de construção contínua de sistemas, são apresentadas técnicas de automação de testes. Essas técnicas, que podem ser utilizadas para a detecção de quebras lógicas, hoje são viáveis devido aos avanços tecnológicos de hardware e de ferramentas de compilação.

2.6 Integração dos espaços de trabalho de GCS

As atividades de Engenharia de Software lidam com problemas complexos que podem ser agravados se acrescentadas preocupações referentes ao controle de modificações, controle de versões e controle de construções e liberações. Para lidar com essa complexidade, diferentes técnicas de integração dos espaços de trabalho são utilizadas. Essas técnicas visam prover ao desenvolvedor de software um ambiente que forneça recursos de GCS sem afetar de forma profunda a rotina de trabalho.

Alguns ambientes de programação (IDE – *Integrated Development Environment*), como, por exemplo, o Eclipse (ECLIPSE FOUNDATION, 2004), o NetBeans (NETBEANS COMMUNITY, 2004) e o JBuilder (BORLAND, 2004b), fornecem recursos de controle de versões integrados. Esses recursos permitem que o programador acesse as versões compatíveis de arquivos de código-fonte sem ter que sair do IDE. Usualmente, o acesso a essas versões compatíveis é feito usando o conceito de projeto, que agrupa esses artefatos, facilitando a manutenção da consistência dos mesmos. Contudo, existem iniciativas para a introdução desse conceito na camada de GCS, como, por exemplo, o TCCS (*Trivial Configuration Control System*) (BOLINGER et al., 1995), que consiste em uma camada implementada sobre o SCCS (ROCHKIND, 1975) ou sobre o RCS, com o objetivo de explorar o conceito de projeto para o versionamento de código fonte.

Além de controle de versões, esses IDEs também permitem o acesso integrado ao controle de construções. Em alguns casos, o controle de construções é definido e gerenciado por mecanismos proprietários do IDE. Contudo, o uso de descritores de

construção não proprietários está ficando cada vez mais comum. O Ant é um exemplo desses mecanismos de construção não proprietários amplamente adotados nos IDEs voltados para a programação Java.

Com a adoção dessas infra-estruturas integradas, porém não proprietárias, é possível permitir que diferentes desenvolvedores de um mesmo projeto façam uso de diferentes IDEs para a programação, sem interferir no procedimento de compilação e execução de testes. Por exemplo, cada IDE tem um comando específico para iniciar a compilação, contudo, a execução real desse comando ocorre através da interpretação de um descritor de construção. Desta forma, o usuário do JBuilder usará o seu comando habitual de compilação, que é diferente do comando habitual de compilação do Eclipse, mas o efeito será o mesmo, pois ambos os comandos estarão acionando o Ant para a leitura do descritor de construção anexado ao projeto. Nesse cenário, o conhecimento referente à organização dos diretórios do projeto deixa de pertencer aos IDEs e passa a pertencer ao descritor de construção, que é um IC do projeto que não depende de IDEs específicos.

No caso de controle de modificações, a integração mais comum ocorre diretamente no produto. O *browser* Netscape (NETSCAPE, 2004) e o sistema operacional Windows XP (MICROSOFT, 2004c) são exemplos dessa integração. No caso do Windows XP, relatórios de erros são automaticamente enviados ao fabricante caso algo aconteça fora do previsto com algum programa em execução. No caso do Netscape, é possível reportar erros ou sugestões de melhoria através da opção "*Feedback Center*". Nesse contexto, o próprio usuário está informando indiretamente ao sistema de GCS a necessidade de modificações para corrigir o erro relatado. Contudo, existem ambientes de desenvolvimento de software que fornecem esse recurso de pedido de modificações integrado, possibilitando que o desenvolvedor faça o requerimento (FIGUEIREDO, 2004).

Outros casos mais avançados de integração se referem à análise das informações referentes a GCS, com o intuito de dar subsídios para a melhoria das atividades de Engenharia de Software. Para atingir esse objetivo, as informações coletadas pelos diferentes sistemas de GCS devem ser organizadas e relacionadas, provendo conhecimento indireto referente ao desenvolvimento de software.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a integração dos espaços de trabalho de GCS (BALL et al., 1997; DRAHEIM et al., 2003).

2.6.1 Mineração de informações

BALL et al. (1997) propuseram que elementos que sofrem modificações em conjunto podem estar relacionados semanticamente. A partir da análise em um repositório existente, foi possível determinar a afinidade entre os arquivos (classes em C++). Inicialmente, cada uma das classes foi classificada em função do seu papel dentro da arquitetura utilizada. A partir dessa classificação, surgiram diferentes tipos de classe. Cada tipo de classe recebeu uma representação gráfica simbolizada por uma forma geométrica e uma cor. Um dos resultados da aplicação da técnica proposta gerou um grafo, onde o tamanho de cada nó, que simboliza uma classe em C++, indica a quantidade de modificações que a classe sofreu.

Um outro resultado ainda mais interessante, também explicitado através de um grafo, usa a distância entre os nós para indicar o grau de afinidade existente entre classes em C++. Para construir esse grafo, foi necessária a utilização de uma fórmula que determinasse o tamanho de cada aresta existente entre os nós. Essa fórmula, $CD_{mr} \div \sqrt{C_{mr} \times D_{mr}}$, define que a distância entre duas classes C e D é igual à quantidade de pedidos de modificações que afetarem C e D em conjunto dividido pela raiz quadrada da multiplicação da quantidade de modificações que atingiram C e D isoladamente.

Assim sendo, é possível averiguar se a arquitetura proposta para a aplicação distribui as responsabilidades nas classes de forma a proporcionar maior coesão e menor acoplamento durante a manutenção. Esse tipo de resultado também pode ser útil tanto durante a análise de impacto, etapa do processo de controle de modificações, quanto na definição de um novo projeto para a aplicação de manutenção preventiva.

2.6.2 Bloof

Usualmente, os sistemas de controle de versões guardam uma grande gama de informações. Contudo, o suporte para a análise dessas informações não é provido a contento pelo próprio sistema. Com o intuito de contornar esse problema, foi desenvolvido o Bloof (DRAHEIM et al., 2003), que é uma interface para a análise dos dados coletados pelo CVS.

O Bloof se conecta a um repositório CVS e extrai as informações necessárias para possibilitar a análise. Essas informações passam a povoar um banco de dados cujo modelo inclui os desenvolvedores, os arquivos e as revisões, além dos relacionamentos

entre essas entidades. Todas as informações relacionadas com as revisões também são persistidas nesse repositório. Dentre essas informações estão o comentário que o desenvolvedor forneceu ao fazer a revisão, o número de linhas adicionadas e removidas, a versão gerada e o momento em que a revisão ocorreu.

Utilizando a interface de usuário Bloof Browser, é possível executar consultas pré-definidas sobre esse modelo, que são denominadas de métricas. Essas métricas, relatadas na forma de gráficos, mostram a evolução do desenvolvimento referente ao número de linhas de código, número de arquivos, contribuição dos desenvolvedores, impacto das modificações, etc.

Atualmente, o Bloof atua sobre somente um projeto de cada vez, correlacionando as informações referentes ao desenvolvimento desse projeto. Contudo, uma aplicação futura identificada por DRAHEIM et al. (2003) englobaria o acesso a um grande conjunto de projetos e a aplicação de métricas sobre todos esses projetos, permitindo uma posterior análise dos resultados com o objetivo de detectar semelhanças na evolução dos seus desenvolvimentos. Essa detecção de semelhanças em projetos de software já foi tratada por LEHMAN et al. (1997), em um trabalho que constatou que ao menos 5 das 8 leis de Lehman criadas nos anos 70 ainda eram válidas nos anos 90.

2.6.3 Considerações sobre integração dos espaços de trabalho

Foram apresentadas duas abordagens para a extração e análise de informações existentes nos repositórios de versão. Enquanto o Bloof permite que métricas sejam estabelecidas e executadas sobre o histórico de versões do projeto, a abordagem proposta por BALL et al. (1997) visa detectar automaticamente relações semânticas entre artefatos através da análise das modificações. O uso de mineração de dados neste contexto aparenta ser mais promissor, pois permite ir além das informações explicitas existentes no repositório.

2.7 Conclusão

Neste Capítulo foram apresentadas várias abordagens de GCS no estado da prática e da arte. Essas abordagens foram agrupadas segundo o tipo principal de contribuição para a GCS. Os cinco grupos definidos foram (1) processos, normas, procedimentos, políticas e padrões; (2) controle de modificações; (3) controle de versões; (4) controle de construções e liberações; e (5) integração dos espaços de trabalho.

O uso de cada uma dessas abordagens e o formalismo adotado nesse uso podem variar em função do tipo de aplicação a ser desenvolvida. No caso de projetos de software livre, por exemplo, não existe nem interesse e nem recurso para a adoção de processos e sistemas que aumentem a burocracia com a introdução de um maior formalismo (FOGEL et al., 2001; ASKLUND et al., 2002).

Apesar da existência de processos e ferramental de GCS adequado para todos os níveis de formalismo desejados, ainda existem projetos de desenvolvimento de software que ignoram completamente a automação da GCS. Um exemplo clássico dessa situação é o Linux (TORVALDS, 2004), que não faz uso de sistema algum de controle de versões. O seu código-fonte é organizado em diretórios em função da versão e somente os moderadores podem aplicar remendos (*patches*) nessas versões.

No início deste milênio, o mercado de GCS estava movimentando anualmente em torno de um bilhão de dólares, e cada vez mais as empresas têm interesse em introduzir processos de GCS em algum nível nos seus projetos (ESTUBLIER et al., 2002). Contudo, a introdução gradativa em projetos sem nenhum suporte de GCS é vista como a melhor solução (LEON, 2000). Caso uma equipe que não esteja habituada a trabalhar usando GCS passe a fazer uso de todos os sistemas e funções concomitantemente, o nível de burocracia associado pode se elevar rapidamente, sem que haja tempo suficiente para avanços significativos em produtividade e qualidade.

Das abordagens existentes para GCS, a grande maioria tem enfoque em código-fonte. Por essa razão, o suporte às etapas iniciais do desenvolvimento de software, que englobam análise e projeto, e às etapas finais do desenvolvimento de software, que englobam implantação e evolução dinâmica, é visto como um campo promissor de pesquisas em GCS (ESTUBLIER et al., 2002).

Independentemente do grau de formalismo adotado, a GCS deve ser vista como uma disciplina útil tanto para clientes quanto para gerentes e desenvolvedores. A GCS possibilita o aumento da transparência na relação entre o cliente e o fornecedor de software ao permitir o acompanhamento detalhado do andamento das requisições de modificação. Além disso, a GCS fornece os subsídios necessários para que gerentes possam ter conhecimento sobre o andamento do projeto, facilitando a tomada de decisões. Do ponto de vista de desenvolvimento, a GCS é uma disciplina indispensável para controlar a complexidade existente quando equipes numerosas manipulam, concomitantemente, um conjunto de artefatos, evitando erros e retrabalho e, conseqüentemente, aumentando a qualidade e a produtividade.

Capítulo 3 - Gerência de Configuração de Software no Desenvolvimento Baseado em Componentes

3.1 Introdução

No capítulo 2 foram apresentados sistemas da GCS para o desenvolvimento convencional de software. Contudo, no contexto de DBC, é necessária a adoção de novos processos, normas, procedimentos, políticas e padrões para reger a interação entre os sistemas de GCS das equipes produtoras e consumidoras de artefatos reutilizáveis (LARSSON, 2000). Além disso, novos problemas referentes a GCS surgem com a mudança do foco de desenvolvimento (LEBSACK et al., 2001), tornando necessária a criação de novos sistemas ou a customização dos sistemas existentes ao novo cenário.

O DBC faz uso de componentes, interfaces e conectores como elementos de estruturação de sistemas. Um componente é visto como parte não trivial, independente e substituível de sistemas (KRUCHTEN P., 2001). Componentes, que são partes reutilizáveis de software (D'SOUZA et al., 1998), fazem uso de interfaces descritas de forma contratual para interagir com os demais elementos de software (PAGE-JONES, 1999; SZYPERSKI, 2002). A ligação propriamente dita entre os componentes, que pode ser simples ou complexa dependendo de questões como distribuição, adaptação e coordenação, é obtida através dos conectores (SHAW et al., 1996; LARSSON, 2000; OMG, 2002b).

Existem diferentes métodos para apoiar o DBC. Dentre os mais conhecidos estão Catalysis (D'SOUZA et al., 1998), UML Components (CHEESMAN et al., 2000) e KobrA (ATKINSON et al., 2000). Um maior detalhamento sobre os conceitos ou os métodos referentes ao DBC pode ser obtido através de diversos trabalhos existentes na literatura (HERZUM et al., 1999; BROWN, 2000; HEINEMAN et al., 2001; WALLNAU et al., 2001), ou em alguns trabalhos realizados na própria COPPE/UFRJ (BRAGA, 2000; TEIXEIRA, 2003; BLOIS, 2004).

Apesar da existência desses métodos de DBC, a carência de processos, padrões e ferramental de suporte ainda é grande. Com o uso de DBC, a quantidade de artefatos produzidos durante o processo de desenvolvimento aumenta substancialmente em comparação ao desenvolvimento convencional. Além disso, esses artefatos, que se situam em diferentes níveis de abstração, estão intimamente relacionados através do

conceito de componente. A reutilização desses componentes, precedida de adaptações, implica em novas dimensões de relacionamento entre a versão reutilizável e a versão reutilizada do componente. Essas novas dimensões de relacionamento devem ser rastreadas pelos sistemas de GCS. Contudo, os componentes originais também evoluem com o tempo, e essa evolução deve ser propagada para as suas diversas instâncias reutilizadas de forma consistente, através dos rastros existentes (ATKINSON et al., 2001).

Tanto a GCS quanto o DBC têm como principais características o aumento da produtividade, o aumento da qualidade e a redução de custos (KWON et al., 1999). Entretanto, para potencializar essas características, é necessário que ambas as técnicas sejam adotadas de forma consistente e integrada.

Neste capítulo, são apresentadas abordagens de GCS voltadas para esse contexto específico de desenvolvimento de componentes reutilizáveis e as suas adoções posteriores em diferentes aplicações. Neste cenário, onde a GCS se torna ainda mais importante, faltam soluções e sobram problemas (ZHANG et al., 2001). Desta forma, são apresentadas, na seção 3.2, as abordagens que lidam com processos, normas, procedimentos, políticas e padrões para a adoção efetiva de GCS em organizações calcadas no DBC. Posteriormente, são apresentadas, nas seções 3.3, 3.4 e 3.5, as abordagens focadas nos subsistemas específicos de GCS, que são, respectivamente, controle de modificações, controle de versões e controle de construções e liberações. Finalmente, são apresentadas, na seção 3.6, abordagens para a integração dos espaços de trabalho de DBC e GCS.

3.2 Processos, normas, procedimentos, políticas e padrões de GCS no DBC

O processo de DBC difere do processo de desenvolvimento convencional devido à adição de atividades relacionadas à reutilização e substituição de componentes existentes. Enquanto um ciclo de desenvolvimento convencional é composto pelas macro-atividades de (1) análise, (2) projeto, (3) codificação, (4) testes e (5) implantação, um ciclo de desenvolvimento com reutilização de componentes é composto pelas macro-atividades de (1) busca por componentes existentes, (2) seleção dos componentes encontrados, (3) criação de componentes, caso necessário, (4) adaptação dos

componentes, (5) implantação dos componentes em ambiente de produção e (6) substituição dos componentes (LARSSON, 2000).

Devido a essas modificações no processo de engenharia, o processo de evolução também deverá ser adaptado à nova realidade. Até então, toda requisição de modificação era avaliada e tratada por completo pela equipe de desenvolvimento. Contudo, nesse novo cenário, pode ser necessário delegar parte da avaliação à equipe que implementou determinados componentes utilizados na aplicação. Além disso, decisões referentes a continuar utilizando esses componentes, trocar de fornecedor ou implementar as funcionalidades dentro da própria organização passam a fazer parte das atribuições do CCC.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a processos, normas, procedimentos, políticas e padrões de GCS no DBC (KWON et al., 1999; ATKINSON et al., 2001).

3.2.1 MwR

KWON et al. (1999) fornecem processos integrados de GCS, de reutilização e de manutenção para apoiar a evolução de sistemas legados e de bibliotecas de software reutilizáveis. Esses processos são divididos em duas principais perspectivas: (1) DwR, que é um processo de desenvolvimento de aplicações utilizando componentes reutilizáveis, e (2) MwR, que é um processo de manutenção dos componentes reutilizáveis.

No MwR, são discutidas questões referentes à decisão de adaptar os componentes reutilizados através de técnicas de caixa-preta, ou modificar esses componentes através de técnicas de caixa-branca. Segundo KWON et al. (KWON et al., 1999), é preferível a adoção de reutilização caixa-preta, a não ser que o esforço para a reutilização caixa-branca seja inferior, o que é pouco provável de ocorrer. Além disso, é realçada a importância dos testes de integração e de regressão dos componentes no contexto das aplicações, principalmente quando esses componentes tiverem sido modificados durante o processo de reutilização.

O MwR assume que os pedidos de modificação ocorrerão somente na etapa de manutenção, não considerando a etapa de desenvolvimento. O processo de manutenção descrito faz uso das atividades previstas nas normas IEEE Std 1042 (1987) e ISO 10007 (1995a) para a função de controle da configuração.

Para apoiar o processo MwR, são definidos dois papéis: (1) reutilizador e (2) mantenedor. O reutilizador é responsável por povoar o repositório de componentes. Para povoar o repositório de componentes, pode ser necessário adquirir esses componentes de terceiros ou solicitar a construção de novos componentes à equipe de desenvolvimento de componentes, que fará uso, por sua vez, do processo DwR. Já o mantenedor é responsável por aprovar, implementar e propagar as modificações nos componentes existentes no repositório. O mantenedor também é responsável por coletar informações sobre as modificações. Essas informações são necessárias para responder a perguntas como, por exemplo, quem implementou uma determinada modificação, quais modificações já foram implementadas, quando essas modificações foram implementadas e por que elas foram implementadas.

No MwR, os pedidos de modificação são propagados tanto para o reutilizador quanto para o mantenedor. Em paralelo, o reutilizador procura por componentes que possam atender às necessidades do pedido de modificação e o mantenedor verifica quais modificações seriam necessárias nos componentes já existentes. Após a apresentação dos dois laudos, o CCC decide se serão utilizados novos componentes ou se serão modificados os componentes já em uso.

A estratégia proposta pelo MwR faz uso das técnicas de reutilização como recurso no apoio à manutenção. Essa postura difere das demais, que tratam a reutilização como processo principal e acoplam processos auxiliares de GCS com o objetivo de aumentar o nível de controle sobre esse processo principal.

Outra característica do MwR é explicitar a possibilidade de construção de componentes a partir de partes de uma aplicação. Existem situações onde é decidido, inicialmente, não construir um componente para atender as funcionalidades requeridas. Por esse motivo, o desenvolvimento acontece no âmbito da aplicação específica que necessita das funcionalidades. Todavia, com o passar do tempo, pode ser constatado que essas funcionalidades também são interessantes para outras aplicações, iniciando um processo de fatoração para transformar as funcionalidades em um componente e reutilizar esse componente em todas as aplicações, inclusive na inicial.

O tratamento de testes descrito no MwR não contempla testes de integração e de regressão para os componentes no ambiente de desenvolvimento de componentes, antes das suas liberações. Esses testes são executados somente no ambiente de desenvolvimento com componentes, depois que os componentes foram inseridos no contexto de uma determinada aplicação. Entretanto, com o uso de rastros entre os dois

ambientes de desenvolvimento e com a especificação de interfaces na forma de contratos (MEYER, 1992), seria possível executar testes de integração e regressão sobre os componentes antes de efetuar a liberação propriamente dita.

3.2.2 KobrA

O método KobrA (ATKINSON et al., 2001) define procedimentos e processos para permitir a evolução de componentes reutilizáveis e reutilizados. Diferentemente do desenvolvimento convencional, é detectada a existência de dois contextos para a requisição de modificação: (1) nos clientes, referente às aplicações, e (2) nos desenvolvedores de aplicações, referente aos componentes reutilizáveis.

Para apoiar a integração de modificações durante o processo de GCS, são definidas quatro estratégias: (1) no contexto de desenvolvimento de componentes, sempre que um artefato é modificado, essa modificação é propagada para os demais artefatos para manter a consistência; do mesmo modo, (2) no contexto de desenvolvimento com componentes, sempre que um artefato é modificado, essa modificação também é propagada para os demais artefatos; além disso, (3) sempre que um artefato reutilizável é modificado, essa modificação é propagada para os artefatos reutilizados; e, da mesma forma, (4) sempre que um artefato reutilizado é modificado, essa modificação também é propagada para os artefatos reutilizáveis. Por exemplo, nos dois primeiros casos, se o modelo UML de análise pertencente a um componente for modificado, os modelos de projeto e o código fonte desse componente também deverão refletir essa modificação. Por outro lado, nos dois últimos casos, se um erro for corrigido em uma ocorrência de um componente, todas as demais ocorrências desse componente devem refletir essa correção de erro.

Essas estratégias representam os quatro processos principais de manutenção no DBC (ATKINSON et al., 2001): (1) modificação nos componentes reutilizáveis, (2) modificação nos produtos baseados em componentes, (3) propagação das modificações dos componentes reutilizáveis para os produtos e (4) propagação das modificações dos produtos para os componentes reutilizáveis. Cada um desses processos é detalhado, segundo a perspectiva de GCS, por três atividades genéricas: (1) identificação da modificação, responsável por (1.1) estabelecer quais artefatos devem ser modificados, (1.2) determinar o tipo da modificação, e (1.3) armazenar a modificação dentro do conjunto de modificações (*Change Set*) pertinente; (2) análise de impacto, responsável por (2.1) estabelecer quão profundo será o impacto em cada artefato afetado e (2.2)

determinar se outras modificações serão necessárias; e (3) propagação da modificação, responsável por alastrar a modificação entre o desenvolvimento de componentes e o desenvolvimento com componentes, visando obter a consistência mútua dos artefatos reutilizáveis e reutilizados.

Esse processo de reutilização de componentes sugere que quando os artefatos reutilizáveis sofrem modificação, essas modificações devem ser propagadas para os artefatos reutilizados contidos em aplicações previamente geradas. Contudo, segundo o método KobrA, caso os artefatos reutilizados tenham sido modificados desde a sua reutilização, será necessário gerar uma aplicação temporária a partir das novas versões dos artefatos reutilizados e efetuar procedimentos de junção entre a aplicação original e a temporária. Um procedimento análogo é sugerido para o processo de fatoração de componentes.

Entretanto, enquanto as rotinas de construção de aplicações não forem totalmente automatizadas, esse tipo de procedimento não será razoável, tendo em vista a complexidade envolvida na geração de aplicações a partir de artefatos reutilizáveis. Visto que algumas modificações nunca são propagadas das aplicações para os artefatos reutilizáveis, a geração de aplicações temporárias deve levar em conta essas modificações, tornando o processo complexo o suficiente para acarretar no abandono dos procedimentos de integração por parte dos desenvolvedores e na posterior inconsistência entre os ambientes de desenvolvimento de componentes e desenvolvimento de aplicações. Uma abordagem com maior possibilidade de sucesso é sugerida por VENUGOPALAN (2002) para o desenvolvimento convencional, onde correções de erros são efetuadas em ramos de desenvolvimento auxiliares e posteriormente integrados no ramo principal.

Outra característica questionável da abordagem KobrA se refere ao procedimento de sempre incorporar as modificações específicas de aplicações na linha de produtos, mesmo quando elas não são de interesse para nenhuma outra aplicação. É sugerido que essas características específicas sejam incorporadas como elementos opcionais, mas o tamanho e a complexidade da linha de produtos aumentaria em função do número de aplicações que fossem geradas a partir dela. Esse cenário se opõe ao cenário em que a linha de produtos tende à estabilidade com o aumento do número de aplicações geradas. Essas aplicações geradas deveriam agregar conhecimento referente aos elementos que são realmente variantes e opcionais, obtidos a partir de processos de fatoração que levam em consideração as suas ocorrências em várias aplicações

existentes no domínio (NEIGHBORS, 1980; ARANGO, 1988; PRIETO-DIAZ, 1990; ARANGO, 1994).

3.2.3 Considerações sobre processos, normas, procedimentos, políticas e padrões no DBC

Os processos apresentados nesta seção fornecem características específicas para a GCS aplicada ao DBC. Contudo, o MWR, apesar de fazer uso das atividades descritas nas normas IEEE Std 1042 e ISO 10007, não tem ênfase na reutilização. A reutilização é somente uma técnica utilizada para a manutenção no contexto de DBC. O KobrA, por sua vez, é concebido especialmente para a reutilização de componentes. Todavia, a abordagem faz uso de propagação imediata e obrigatória de modificações, e assume como passo necessário para a incorporação das modificações a regeração da aplicação. Esse tipo de solução somente será viável quando existirem rotinas automatizadas para a construção no DBC.

3.3 Sistemas de controle de modificações no DBC

Os sistemas de controle de modificações existentes para o desenvolvimento de software convencional geralmente implementam um processo específico de GCS, não possibilitando a customização desse processo para novas situações. Por esse motivo, o uso desses sistemas se torna impróprio ao contexto de DBC, que é regido por novos processos, a não ser que sejam introduzidas adaptações.

Mesmo que adaptados para atender aos processos de DBC, é importante que os sistemas possibilitem customizações futuras, devido à imaturidade desses processos. Como não existem normas que definam processos estáveis e amplamente utilizados para a GCS aplicada ao DBC, é provável que esses processos evoluam até atingir uma maturidade satisfatória. Os sistemas de controle de modificações que implementam esses processos devem estar preparados para possibilitar que essas adaptações ocorram sem prejudicar os processos em execução.

Além disso, o DBC introduz uma série de características até então não exploradas no desenvolvimento convencional. Várias dessas características afetam a forma em que a GCS deve ser utilizada. Dentre elas está o desenvolvimento de software em várias camadas, onde a equipe de desenvolvimento de componentes situada na camada N fornece componentes para equipes de desenvolvimento com componentes

situadas na camada N-1, mas que faz uso de componentes providos por equipes situadas na camada N+1, como exibido na Figura 2.

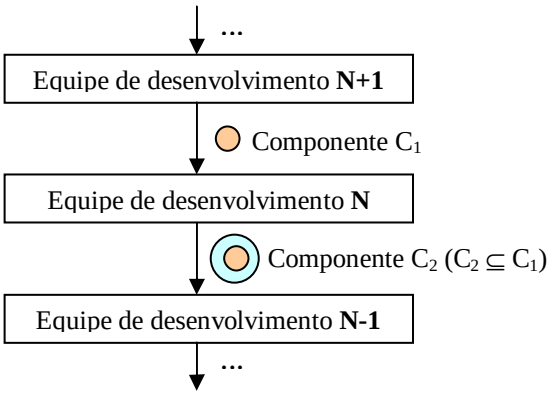

Figura 2: Desenvolvimento de software em várias camadas

Nesse cenário, é fundamental que o sistema de controle de modificações de cada equipe de desenvolvimento seja capaz de apoiar a evolução não somente dos ICs que estão sendo construídos ou mantidos, mas também dos componentes adquiridos de terceiros. Para representar esses componentes externos dentro do sistema de controle de modificações, pode ser necessária a utilização de documentos específicos, que possibilitem o armazenamento e posterior consulta sobre os dados de contato do fabricante do componente (KWON et al., 1999; LARSSON, 2000).

No restante desta seção, são apresentadas, em maior detalhe, algumas abordagens referentes a sistemas de controle de modificações no DBC (KWON et al., 1999; ATKINSON et al., 2001).

3.3.1 TERRA

O sistema TERRA (KWON et al., 1999) é um protótipo que implementa o processo MwR de manutenção de componentes e sistemas legados, descrito na seção 3.2.1. Ele possibilita o registro de novos componentes, a requisição de modificações sobre os componentes e o acesso e reutilização dos componentes. Todas essas atividades são efetuadas através da Internet.

O registro de componente ocorre através do preenchimento de um formulário com as seguintes informações obrigatórias: identificador do componente, nome do componente, nome do autor, data de criação, data de registro, nome do mantenedor, sistemas operacionais compatíveis, linguagem de programação utilizada, formato (modelo, código, binário, etc.), domínios relacionados, métodos e técnicas relacionados e palavras-chave.

A etapa de requisição de modificações, conforme implementada no sistema TERRA, engloba a requisição, a classificação, o armazenamento e a recuperação dos pedidos de modificação. O formulário de requisição de modificações necessita obrigatoriamente das seguintes informações: identificador do pedido de modificação, nome do requerente, data da requisição, tipo da requisição (manutenção em componentes em produção, manutenção em componentes em desenvolvimento, etc.) e situação da modificação. As seguintes informações são opcionais: sistemas legados relacionados, identificadores dos componentes relacionados, versão dos componentes relacionados, tipo da manutenção (corretiva, evolutiva, preventiva ou adaptativa), descrição da modificação e razão da modificação.

O CCC faz uso de um terceiro formulário para expedir pareceres sobre a aprovação dos pedidos de modificação. Segundo KWON et al. (1999), uma aprovação pode estar associada a vários pedidos de modificação e pode afetar diferentes versões de diversos componentes. As informações contidas nesse formulário são: identificador da aprovação, nome do expedidor, data da aprovação, identificador dos pedidos de modificação relacionados, sistemas legados relacionados, identificadores dos componentes relacionados, versão dos componentes relacionados, identificadores das aplicações relacionadas, linhas de produtos relacionadas, tipo da manutenção, situação do pedido de modificação, especificação da modificação, data prevista para a implementação e estimativas em custo e esforço.

Estranhamente, o registro do componente não possibilita a entrada de informações referentes a versões, nem a descrição dos rastros entre as diversas versões de um componente e as aplicações que reutilizam esse componente. Além disso, as informações de estimativas de custo e esforço, conforme proposto, são fornecidas pelo CCC. Porém, essa responsabilidade não é adequada, pois o CCC consiste de um comitê essencialmente gerencial, que faz uso de laudos técnicos para avaliar custo e esforço. Esses laudos técnicos deveriam ser de responsabilidade dos analistas de impacto, segundo as normas IEEE Std 1042 (1987) e ISO 10007 (1995a).

Da forma em que o TERRA está implementado, o processo MwR é definido diretamente no código-fonte em PERL, comprometendo a sua evolução. Por esse motivo, seria necessária a modificação do código-fonte do TERRA para que melhorias no processo pudessem ser incorporadas, comprometendo todos os processos em execução. Mais ainda, o TERRA não fornece uma integração satisfatória entre os diversos formulários, deixando a cargo da pessoa responsável pelo preenchimento a

responsabilidade de localizar as informações necessárias. Várias dessas informações poderiam ser preenchidas automaticamente pelo sistema. Além disso, o TERRA atua de forma semelhante ao Bugzilla, fazendo uso da edição de um campo de situação para designar o novo estado dos pedidos de modificação.

3.3.2 KobrA

A abordagem sugerida pelo método KobrA (ATKINSON et al., 2001) para apoiar o sistema de controle de modificações se baseia no uso da técnica de conjuntos de modificações (SMDS, 1994). A técnica de conjuntos de modificações consiste na definição explícita do conceito de modificação e na associação desse conceito a todas as diferenças geradas através da comparação das versões dos artefatos antes e depois da modificação ser implementada. Desta forma, é possível aplicar uma determinada modificação sobre qualquer configuração de referência. A aplicação da modificação é feita através da junção de cada uma das diferenças relacionadas com os artefatos respectivos da configuração de referência.

Essas modificações são agrupadas, formando então um conjunto de modificações. Cada conjunto de modificações representa uma evolução que agrega valor suficiente para ser, por exemplo, considerada a geração de novas liberações. Esses conjuntos de modificações podem ser aplicados sobre configurações que evoluíram separadamente, permitindo que essas incorporem as novas funcionalidades. Por exemplo: A configuração C_1 foi entregue a um cliente, incluindo os modelos e códigos-fonte. Esse cliente efetuou adaptações para customizar o software para o seu problema, transformando a configuração em C_1'. Contudo, em paralelo, a equipe de desenvolvimento corrigiu defeitos e incluiu funcionalidades importantes em C_1, gerando a configuração C_2. O cliente, que tem muito interesse nessas melhorias, pode solicitar o conjunto de modificações $M_{C1\text{-}C2}$ que represente a diferença entre as duas configurações C_1 e C_2 ($M_{C1\text{-}C2} = C_2 - C_1$). Ao aplicar $M_{C1\text{-}C2}$ sobre C_1', e efetuar as verificações de consistência necessárias, será obtida a configuração C_2', que inclui tanto as correções de defeito e novas funcionalidades quanto as customizações feitas pelo próprio cliente, como exibido na Figura 3.

O principal argumento para a utilização desse tipo de abordagem no DBC é a necessidade de transferir modificações entre os contextos de desenvolvimento de componentes e desenvolvimento com componentes. Assim, adaptações feitas pela equipe de desenvolvimento de um produto específico podem ser facilmente transferidas

para a linha de produtos associada caso seja útil para os demais produtos. Além disso, o uso de descrições de modificação em conjunto com a identificação dos artefatos afetados possibilita um maior controle sobre os rastros de modificação, permitindo que questões referentes a quem, quando, como, onde, o quê e por quê sejam respondidas com facilidade, através de consultas sobre o sistema de controle de modificações e sobre o sistema de controle de versões. Uma outra sugestão importante é o uso de associações de causa no modelo de modificações, viabilizando a adoção de técnicas de análise causal no caso de modificações corretivas (LEON, 2000).

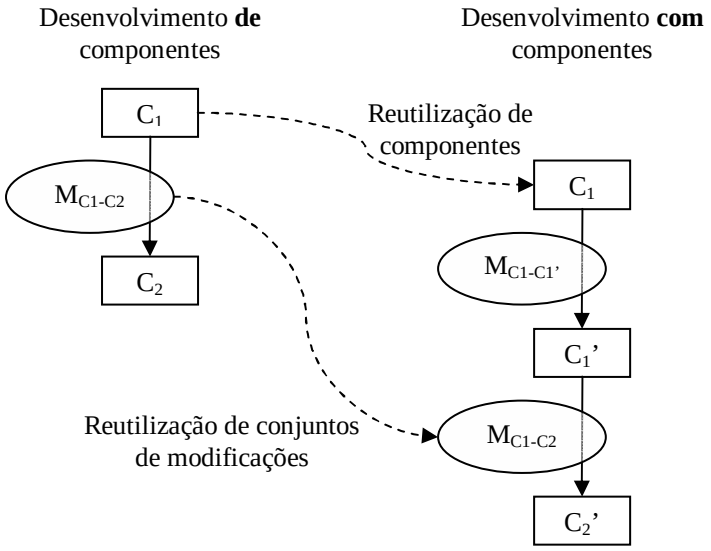

Figura 3: Reutilização de conjuntos de modificações

Para que a técnica de conjuntos de modificações atinja o seu objetivo, é importante fazer uso de mecanismos de detecção de diferenças que possibilite o armazenamento relativo das diferenças, para que a posterior junção dessas diferenças faça sentido. Por exemplo, quando um código-fonte CF é modificado na configuração C_1, não é desejável armazenar que a linha 123 foi editada, pois a linha 123 do código-fonte CF na configuração C_2 pode ser outra ou, em algumas situações, nem existir. O correto seria detectar o contexto relativo da modificação, de forma que fosse possível encontrar a linha editada mesmo que ela estivesse em outra posição. Só assim a técnica de conjuntos de modificações poderia ser utilizada com todo o seu potencial.

3.3.3 Considerações sobre sistemas de controle de modificações no DBC

A maior deficiência dos sistemas de controle de modificações no contexto de DBC é referente à dificuldade de adaptação do processo de GCS sem afetar os projetos em execução. Além disso, existe uma necessidade especial relativa à manutenção das informações sobre fornecedores de componentes para a propagação das requisições de modificação, caso necessário. O sistema TERRA, apesar de não contribuir para a questão da customização e evolução do processo, atua no problema de registro de componentes, fazendo uso de um conjunto fixo de informações.

O KobrA, apesar de não atuar diretamente em nenhum desses dois problemas, apresenta outras duas características importantes para sistemas de controle de modificações no contexto de DBC: conjunto de modificações e análise de causa. Essas características viabilizam, respectivamente, a transferência de modificações entre os ambientes de desenvolvimento de componentes e de desenvolvimento com componentes e a detecção do real motivo da ocorrência de defeitos.

3.4 Sistemas de controle de versões no DBC

Com a adoção do DBC, se torna necessária a utilização de novos tipos de artefato para permitir a representação de componentes, interfaces e conectores em diferentes níveis de abstração. Esses novos tipos de artefato, que englobam modelos de componentes e descritores de implantação, necessitam de mecanismos especializados para o controle de versões.

Apesar dos mecanismos convencionais de controle de versões poderem ser utilizados para apoiar a evolução desses novos tipos de artefato, a perda semântica é significativa, pois esses sistemas convencionais não têm conhecimento referente aos conceitos existentes no domínio de DBC. Em algumas situações onde o conhecimento referente aos conceitos de DCB existe, as soluções para o problema de evolução não são satisfatórias, como, por exemplo, o acúmulo de interfaces utilizado pelo COM (MICROSOFT, 2004a) devido à impossibilidade de modificar uma interface existente (LARSSON, 2000).

Outra característica diferenciada, introduzida pelo DBC, é o alto grau de encapsulamento que o conceito de componente apresenta. Como os componentes não têm relacionamentos diretos com outros componentes, é possível isolar o impacto da

modificação de um componente desde que as suas interfaces permaneçam intactas (LARSSON, 2000). Em razão disso, o versionamento de componentes está fortemente dependente do versionamento das interfaces e essa dependência contribui para a diminuição da complexidade associada à construção e manutenção dos diversos componentes de uma arquitetura.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a sistemas de controle de versões no DBC (CHRISTENSEN, 1999a; DASHOFY et al., 2001; ATKINSON et al., 2001; CHEN et al., 2003). Diferentemente do que pode ser constatado para o desenvolvimento convencional, o suporte de controle de versões para o DBC é deficiente e necessita maior maturidade, que só será obtida através do aumento das pesquisas na área.

3.4.1 RCM

A grande maioria dos sistemas de controle de versões trata arquivos do sistema operacional como unidade de abstração para o versionamento. Essa postura se torna especialmente inadequada no contexto de DBC. Para possibilitar o controle de versões de arquiteturas de componentes, foi proposto o RCM (CHRISTENSEN, 1999a), criando uma camada de abstração em relação aos arquivos que armazenam os modelos e código-fonte desses componentes.

O RCM, que é implementado no contexto do ambiente de desenvolvimento de software Ragnarok (CHRISTENSEN, 1999b), atua no nível lógico, controlando a versão de componentes. Esse nível lógico mapeia, através de atributos, o nível físico, englobando os artefatos que implementam o componente, como, por exemplo, modelos e código-fonte.

Um componente é representado através da tupla (CID,VID,S_{sub},S_{rel}), onde CID representa o identificador do componente, VID representa a versão do componente, S_{sub} contém as dependências para artefatos no nível físico que implementam o componente e S_{rel} contém as dependências para outros componentes no nível lógico.

Utilizado essa estrutura, as funcionalidades de GCS foram remodeladas para atender às necessidades de controle de versões de componentes. A funcionalidade de *check-in* foi concebida de forma a atuar recursivamente, criando uma nova versão para cada componente relacionado que tenha sido modificado. De forma semelhante, a funcionalidade de *check-out* atua recursivamente, buscando todos os componentes que dependem em algum nível do componente solicitado.

Além disso, são fornecidos recursos de ramos e diferença arquitetural entre componentes, que é o fundamento necessário para a funcionalidade de junção. A junção ocorre através da união dos conjuntos S_{sub} e S_{rel} entre os respectivos componentes. Apesar dessa abordagem ser suscetível a efeitos colaterais, o autor argumenta que para casos onde as junções ocorrem com razoável freqüência, tudo acontece dentro do esperado. Todavia, para possibilitar a junção de artefatos no nível físico, é necessário fazer uso dos algoritmos de junção referentes a cada tipo de artefato. Além disso, embora seja fornecido um mecanismo automático para a junção de componentes, é salientada a necessidade de intervenção manual nas situações onde os componentes se diferem substancialmente.

Apesar de trazer características desejáveis de GCS para o domínio de DBC, a abordagem não permite que, a partir de uma determinada configuração, existam dependências para mais de uma versão de um mesmo componente. Essa restrição é questionável em cenários reais, onde, por exemplo, um componente C_1 desenvolvido em Java depende dos componentes C_2 na versão 1.0 e Log4j na versão 1.2.8. Contudo, o componente C_2, por ser mais antigo, depende do Log4j na versão 1.1.3. Desta forma, o componente C_1 tem dependências em diferentes níveis para diferentes versões de um mesmo componente, o Log4j. Esta Situação não é permitida pelo RCM.

Mais ainda, o tratamento dos artefatos pertencentes ao nível físico necessita do uso dos atuais sistemas de GCS, impossibilitando o RCM de atuar em níveis mais granulares, que envolvem, no caso particular da orientação a objetos, os conceitos de classes, atributos e métodos.

3.4.2 KobrA

A abordagem sugerida pelo KobrA (ATKINSON et al., 2001) para o sistema de controle de versões não faz uso de configuração como um elemento explícito. Para o KobrA, os artefatos são descritos através de um conjunto de dependência com outros artefatos. Desta forma, uma configuração pode ser obtida quando, a partir de um artefato qualquer, são percorridas todas as dependências e encontrados todos os artefatos compatíveis.

Essas dependências são descritas através de diferentes perspectivas. Um componente, por exemplo, está relacionado com versões compatíveis de especificações, realizações e implementações através de dependências do tipo <<*contains*>>. Dentro do componente existem informações sobre quais combinações desses elementos geram

configurações consistentes. Além disso, um componente também pode se relacionar com outros componentes, determinando estruturas de decomposição. Esse relacionamento ocorre no contexto de um IC de maior granularidade, que mantém as informações referentes a suas partes sem a necessidade do uso do conceito explícito de configuração. Desta forma, cada IC armazena a configuração dos outros ICs que o compõem, recursivamente. Uma grande vantagem dessa recursão é possibilitar que, com uma metáfora homogênea, recursos de versionamento possam ser aplicados uniformemente, abrangendo desde arquiteturas completas de componentes até elementos individuais de modelagem UML.

Um outro mecanismo de dependências utilizado, do tipo <<*derived*>>, consiste no relacionamento existente entre as ocorrências de um mesmo componente no ambiente de desenvolvimento de componentes e no ambiente de desenvolvimento com componentes. Esse tipo de relacionamento é fundamental para possibilitar a notificação de modificação e posterior intercâmbio dos conjuntos de modificações entre essas ocorrências do componente.

Para controlar a versão dos ICs, é sugerido o uso de um vetor com quatro informações de versionamento: (1) revisão, tratada pelo KobrA como modificação semântica, (2) variante, (3) edição, tratada pelo KobrA como modificação cosmética, e (4) estado de qualidade. Além disso, a própria dependência entre os ICs também armazena a informação do estado de qualidade. Assim, é possível definir mecanismos automáticos com linguagens como OCL para evoluir os artefatos e atribuir estados de qualidade que indiquem a necessidade de uma posterior verificação manual. Por exemplo, no caso da revisão da especificação de um componente, novas versões de realização, implementação, componentes relacionados e do próprio componente poderiam ser criadas e associadas à nova especificação. Contudo, o atributo de qualidade dessa associação deveria relatar a necessidade de inspeção.

3.4.3 Versionamento de modelos xADL

DASHOFY et al. (2001) apresentam uma linguagem extensível para a descrição de arquiteturas denominada xADL, que possibilita, entre outros recursos, o versionamento dos elementos que compõem a arquitetura. A xADL é considerada extensível pois todos os seus recursos são mapeados em módulos individuais, implementados através de XML Schema (W3C, 2001b). Um desses módulos é utilizado para prover recursos de grafos de versões para componentes, interfaces e conectores.

Esse recurso de versionamento provido pela xADL permite que diferentes versões de um componente possam ser utilizadas em diferentes locais de uma mesma arquitetura. Além disso, o grafo de versões pode relatar tanto a evolução de um único componente quanto a evolução de uma arquitetura completa.

Outros módulos providos pela xADL que se relacionam com a GCS permitem a descrição dos elementos variantes e opcionais das arquiteturas. Com o módulo de condições booleanas ativo, é possível definir fórmulas lógicas para estabelecer as dependências entre os elementos variantes, opcionais e os demais elementos.

Além disso, existe um módulo responsável pela representação das diferenças entre arquiteturas. Com o uso desse módulo juntamente com as ferramentas apropriadas, também pertencentes a xADL, é possível comparar arquiteturas com o intuito de obter a diferença ($D = diff[A_1, A_2]$) ou a arquitetura resultado da aplicação de uma diferença sobre uma arquitetura base ($A_2 = merge[A_1, D]$).

3.4.4 Comparação e junção de linhas de produtos

Uma das aplicações da xADL consiste no suporte à representação de linhas de produtos. Essas linhas de produtos podem ser refinadas de diferentes maneiras, encadeando na geração de diferentes produtos. CHEN et al. (2003) apresentam uma abordagem para a comparação entre duas versões de produtos oriundas de uma determinada linha de produtos e a posterior junção do resultado dessa comparação na linha de produtos.

Os algoritmos utilizados para efetuar comparação e junção, que são baseados em algoritmos definidos anteriormente por WESTHUIZEN et al. (2002), são úteis tanto para as arquiteturas de componentes voltadas para um produto específico ou para as arquiteturas de componentes de linhas de produtos. Esses algoritmos diferem dos anteriores em relação a características necessárias introduzidas para suportar linhas de produtos, como, por exemplo, o uso de similaridade ao invés de identificadores e o uso de granularidade fina para comparar os elementos das linhas de produtos.

Apesar do algoritmo de comparação e junção não fazer uso de identificadores, o seu principal recurso para a detecção de similaridade entre componentes é o nome do componente. Esse tipo de técnica pode gerar resultados indesejáveis em domínios novos ou instáveis, onde os desenvolvedores de componentes não têm certeza sobre a nomenclatura mais adequada. Além disso, não é fornecido nenhum suporte para a

comparação e junção de diferentes linhas de produtos. Esse suporte seria interessante para possibilitar a detecção de outras linhas de produtos mais abrangentes.

3.4.5 Considerações sobre sistemas de controle de versões no DBC

Das abordagens apresentadas, a RCM e a KobrA atuam principalmente no versionamento de componentes. A RCM define o mapeamento entre as versões de componentes e as versões dos demais artefatos, incluindo código-fonte. Contudo, obriga que somente uma versão do componente seja utilizada por projeto. O Kobra, por sua vez, faz uso do conceito de configuração através dos próprios relacionamentos entre artefatos e fornece suporte explícito para reutilização. Foram também apresentadas abordagens para o controle de versões sobre arquiteturas de linha de produtos.

Independentemente da abordagem, o foco principal é no controle de versões dos componentes em alto nível de abstração, propagando as ações de GCS para as demais ferramentas de controle de versões nos níveis de abstração inferiores. Todavia, o suporte ao controle de versões nos níveis de abstração intermediários, que lidam com modelos de análise e projeto, continua deficiente.

3.5 Sistemas de controle de construções e liberações no DBC

De forma semelhante ao desenvolvimento de software convencional, o DBC necessita de mecanismos que possibilitem a estruturação dos ICs para construir um ambiente de trabalho condizente com o paradigma utilizado. Além dos problemas já conhecidos, novos problemas surgem quando o objetivo deixa de ser a construção de uma única aplicação e passa a ser a construção de componentes para atender a famílias de aplicações. Dentre esses problemas estão o tratamento da recursividade existente na dependência entre componentes e a necessidade de definição de partes opcionais e variantes dentro de um componente.

Apesar de autocontidos, componentes se caracterizam fortemente pela reutilização de serviços providos por outros componentes. Desta forma, a seleção de uma versão de um componente dentro do repositório de GCS implica na seleção de versões de outros componentes. Este encadeamento de dependência entre componentes é normalmente descrito através de um modelo de sistema (*system model*) (KWON et al., 1999; ESTUBLIER et al., 2002). Contudo, o procedimento de seleção ocorre de forma recursiva, limitando o espaço de versões a cada iteração. Esse tipo de comportamento, que pode levar a necessidade de substituição de alguns componentes da seleção inicial

por motivos de inconsistência, difere do que é habitual no desenvolvimento convencional de sistemas, e conseqüentemente requer um tratamento diferenciado.

Além da dimensão das versões, os componentes podem ser selecionados segundo dimensões de obrigatoriedade e variabilidade, ou a combinação de ambas. O tratamento de obrigatoriedade e variabilidade, que consiste na troca de comportamento em determinados pontos do ciclo de vida do software (SVAHNBERG et al., 2002), também é diferenciado em relação ao desenvolvimento convencional de software. Apesar de existirem várias propostas conhecidas no assunto, dentre elas o uso de diretivas de compilação, *plug-ins* de carga dinâmica e arquivos de configuração, a preocupação dessas propostas é referente ao problema da construção de um único sistema, não levando em conta as características existentes em coleções de sistemas semelhantes.

A liberação de sistemas construídos através do DBC também diferencia da liberação de sistemas construídos através do desenvolvimento tradicional. A principal característica é a separação do papel do desenvolvedor em dois novos papéis: o de desenvolvedor de componentes e o de desenvolvedor de aplicações utilizando componentes. Mais ainda, na maioria dos cenários, o próprio desenvolvedor de componentes exerce o papel de utilizador de componentes quando parte do componente em desenvolvimento depende de serviços providos por componentes já existentes.

Nesse novo contexto, a liberação de configurações de referência passa a ocorrer tanto quando os componentes recém construídos são providos às equipes de desenvolvimento de aplicações, quanto quando as aplicações são fornecidas aos clientes propriamente ditos. Esse processo de liberação deve ser acompanhado de forma minuciosa, catalogando informações sobre os provedores e os usuários das configurações de referência em questão, para possibilitar a notificação futura no caso de surgimento de erro ou novas versões, respectivamente.

As equipes de desenvolvimento de componentes devem executar testes sobre os componentes a serem liberados, e fornecer, sempre que possível, os dados, planos e resultados dos testes junto com os componentes. Apesar dos componentes já serem testados no ambiente produtor, é fundamental que também sejam testados no ambiente consumidor, para assegurar que a qualidade declarada pelo produtor realmente existe fora do ambiente controlado de desenvolvimento do produtor (LARSSON, 2000).

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a sistemas de controle de construções e liberações no DBC (LARSSON, 2000; ZHANG et al., 2001; LEBSACK et al., 2001; HOEK, 2004).

3.5.1 Dependency Browser

Em arquiteturas de componentes, é crucial a obtenção de configurações consistentes, compostas por versões de componentes compatíveis. Para apoiar essa tarefa de manutenção da consistência das configurações, LARSSON et al. (2000) propõem um modelo para a verificação de dependências entre componentes. Para permitir o controle das dependências, é introduzida uma representação baseada em grafo. Essa representação é armazenada em um repositório de versões para possibilitar a evolução controlada da mesma.

O principal objetivo desse trabalho é responder a cinco perguntas, que são (LARSSON, 2000): (1) Quais componentes foram adicionados ou removidos depois de uma modificação na configuração? (2) Quais dependências foram adicionadas, removidas ou afetadas por uma determinada modificação na configuração? (3) Se um componente é modificado, quais outros componentes no sistema são afetados? (4) Qual é o efeito no sistema quando um novo componente é implantado? (5) Qual é a diferença entre duas configurações determinadas?

Para atender a essas necessidades, é proposto um esquema de identificação de componentes que utiliza o nome, o momento de criação, o tamanho e um número gerado pelo compilador como identificador composto. Além disso, são formalizadas operações de busca por dependências em grafos e detectadas duas estruturas possíveis para a representação dos grafos: matrizes e listas.

A partir da representação matricial de dependências, composta por binários que indicam se existe dependência entre dois elementos, são definidos algoritmos que possibilitam a detecção das diferenças. Para isso, o tamanho das matrizes é inicialmente igualado e, posteriormente, uma matriz é subtraída da outra, resultando na matriz diferença.

Através de uma ferramenta denominada *Dependency Browser*, é feita uma análise das duas matrizes que foram comparadas e da matriz resultado da comparação, visando responder, mesmo que superficialmente, às perguntas propostas como objetivo do trabalho. A superficialidade das respostas é devido ao pouco conhecimento inserido no modelo de dependências. Como esse modelo lida somente com componentes em

tempo de execução, não é possível tratar questões referentes ao projeto interno dos componentes.

3.5.2 CMPro

Em um trabalho semelhante ao *Dependency Browser*, LEBSACK et al. (2001) apresentam a ferramenta CMPro, que possibilita o controle de dependências de componentes COTS (*Commercial Off-The-Shelf*). A principal motivação do CMPro é permitir a identificação e a propagação das informações referentes a componentes COTS através de diversos projetos de desenvolvimento com componentes numa mesma organização.

Mesmo existindo a possibilidade de construir componentes dentro da organização, LEBSACK et al. (2001) argumentam que componentes COTS podem reduzir drasticamente o tempo e o custo necessários para povoar um repositório de componentes. Apesar desse benefício, existem algumas desvantagens, como, por exemplo, a inexistência de informações referentes aos componentes COTS nos sistemas de controle de versões da organização, pois o desenvolvimento e a manutenção dos componentes COTS ocorrem fora da organização. Além disso, os componentes COTS não são usualmente distribuídos com documentação completa que inclua modelos e código-fonte, forçando a reutilização do tipo caixa preta.

Devido à norma IEEE Std 828 (1998) tratar superficialmente questões referentes à gerência de componentes COTS através de sub-contratos, alguns problemas ainda necessitam de solução, como, por exemplo, o acompanhamento do conhecimento referente aos diversos pontos em que componentes COTS são reutilizados.

O CMPro se propõe a solucionar esses problemas através do uso de um banco de dados de GCS que contenha a representação lógica de componentes COTS armazenada como ICs. O componente propriamente dito não é controlado, mas é criado um IC que representa esse componente logicamente e possibilita a definição de dependências entre os componentes e as aplicações que fazem uso desse componente.

As informações necessárias para povoar esse banco de dados de GCS são, entre outras: nome do projeto, número da versão, data de modificação, localização, componentes dependentes e versões anteriores do mesmo componente. Essas informações podem ser utilizadas tanto para análises de impacto quanto para auditorias.

Com o uso do conceito de projeto, é possível tratar um componente de duas formas distintas: (1) componente caixa preta, que é considerado um IC, faz parte de um

projeto e interage com outros componentes, ou (2) componente caixa branca, que é considerado uma configuração composta por diversos ICs e representada no CMPro através do conceito de projeto. Essa segunda opção, apesar de citada, não é utilizada no CMPro, que lida principalmente com o desenvolvimento com reutilização calcado no uso de componentes COTS, deixando em segundo plano preocupações referentes a componentes caixa branca.

Para que o CMPro forneça os serviços de geração de relatórios descritos como essenciais para a análise de impacto e auditoria, é necessária a aquisição e o armazenamento das informações referentes às dependências entre componentes COTS. O processo manual para a aquisição e manutenção dessas informações pode tornar o banco de dados de GCS desatualizado devido à complexidade envolvida. Seria desejável que mecanismos automatizados acessassem informações já existentes nos sistemas de controle de versões e no sistema de controle de modificações para povoar essa base de dados de GCS.

3.5.3 JBCM

ZHANG et al. (2001) propõem um sistema de controle de versões voltado para o DBC baseado em um modelo proposto por MEI et al. (2001). Esse sistema, nomeado JBCM, se baseia na existência de dois tipos de componentes: (1) constituintes primitivos e (2) constituintes compostos. Os constituintes primitivos são implementados através de linguagens de programação e os constituintes compostos são obtidos através da conexão de constituintes primitivos via linguagens de descrição de componentes ou linguagens de descrição de arquiteturas. Desta forma, os constituintes compostos são tratados como configurações de referência de constituintes primitivos.

A filosofia adotada por ZHANG et al. (2001) em relação à metáfora de controle de versões de componentes condiz com os requisitos de uniformidade levantados por ESTUBLIER (2000). Essa filosofia consiste em fornecer uma única metáfora para tratar elementos primitivos ou compostos. Logo, constituintes primitivos evoluem através de versões e constituintes compostos evoluem através de configurações de referência, sempre que seus constituintes primitivos evoluem. Portanto, o conceito de versão está para o constituinte primitivo assim como o conceito de configurações de referência está para o constituinte composto, fazendo com que um constituinte composto seja uma configuração formada por outros constituintes, sejam eles primitivos ou compostos.

Apesar da abordagem tratar do controle de versões de componentes, a implementação utiliza algoritmos baseados no RCS (TICHY, 1985). Esses algoritmos fazem uso de um arquivo do sistema operacional para simular as versões de constituintes primitivos. Os arquivos que fazem parte funcional dos constituintes primitivos também são tratados através desses algoritmos.

3.5.4 Any-time Variability

O processo de seleção de variabilidade em sistemas depende usualmente de técnicas específicas, que se aplicam a fases determinadas do ciclo de vida do software. Por exemplo: durante o início da especificação de aplicações, no contexto da engenharia de domínio (NEIGHBORS, 1980; ARANGO, 1988; PRIETO-DIAZ, 1990; ARANGO, 1994), é possível selecionar as características desejadas utilizando técnicas de recorte, como a proposta por MILER (2000); durante a codificação de aplicações, é possível usar diretivas de compilação; durante a instalação de aplicações, é possível fazer uso de sistemas de seleção de módulos; durante a execução de aplicações, é possível utilizar mecanismos de carga dinâmica através de *plug-ins*.

Contudo, devido à especificidade dessas técnicas, um mesmo projeto necessita do uso de cada uma delas em fases específicas, aumentando a sua complexidade. Para contornar esse problema, HOEK (2004) propôs o uso de uma infra-estrutura genérica para modelar as obrigatoriedades e variabilidades através de linhas de produtos (CLEMENTS et al., 2001) e estruturas específicas que selecionam os artefatos modelados nas diversas fases do ciclo de vida.

A estrutura genérica compreende um formalismo de representação de variabilidade, implementado através da xADL 2.0 (DASHOFY et al., 2002), e uma ferramenta para especificação e seleção de linhas de produtos, implementada através do Ménage (GARG et al., 2003). A estrutura específica compreende um conjunto de diversas ferramentas, uma para cada fase do ciclo de vida, que interpretam a linha de produtos resultante de sucessivas seleções e aplicam os cortes estipulados nos artefatos de suas competências.

Para mostrar a viabilidade da solução, diferentes aplicações tiveram suas arquiteturas modeladas segundo essa técnica e foram alvo de seleções de variabilidade em momentos distintos: projeto, invocação e execução. Dois tipos de seleção foram tratados nesse experimento, que são: variações e versões. Como resultado, foi detectada a necessidade de uma maior automação no suporte ao controle de versões de

componentes de software, pois o mapeamento manual entre as tecnologias de DBC e a infra-estrutura existente de GCS é altamente suscetível a erro.

3.5.5 Considerações sobre sistemas de controle de construções e liberações no DBC

Tanto a abordagem *Dependency Browser*, quanto a abordagem CMPro, atuam na manutenção de dependências entre componentes, com enfoque em componentes em execução e componentes COTS, respectivamente. O *Dependency Browser*, por atuar em componentes em execução, faz uso de um modelo simples, o que dificulta a inferência de relações complexas entre os componentes.

O JBCM, em contrapartida ao CMPro, possibilita o controle de construção sobre componentes caixa-branca, atribuindo o conceito de configuração de referência a componentes compostos e associando mecanismos existentes de controle de versões aos componentes primitivos. Em ambas as abordagens, existem mapeamentos entre componentes físicos e elementos lógicos, usando arquivos do sistema operacional.

A dificuldade de controlar a construção e liberação de componentes é relatada tanto no CMPro quanto no *Any-time Variability*, indicando que mecanismos automatizados são necessários para apoiar a seleção de componentes e povoar o espaço de trabalho com a configuração correta dos artefatos que descrevem esses componentes.

3.6 Integração dos espaços de trabalho de GCS no DBC

De forma análoga à necessidade de integração dos espaços de trabalho de GCS e desenvolvimento convencional, o DBC necessita que os conceitos de GCS sejam encapsulados nos conceitos já existentes de componente, interface e conector. Contudo, apesar dessa necessidade existir, o DBC não dispõe da mesma infra-estrutura de GCS existente para o desenvolvimento convencional.

Para que possa existir uma integração real entre GCS e DBC, é necessário que os sistemas de controle de versões, controle de modificações e controle de construções e liberações passem a atuar no nível de abstração requerido pelos conceitos existentes no DBC. Para atingir esse objetivo, existem duas possíveis abordagens: (1) mascaramento dos sistemas de base ou (2) redefinição dos sistemas de base. A abordagem de mascaramento do sistema de base consiste em fornecer uma metáfora de DBC para o desenvolvedor, acoplada ao mapeamento dessa metáfora para os sistemas de GCS convencionais. Já a abordagem de redefinição dos sistemas de base consiste na

reconstrução dos sistemas de GCS, para que eles passem a fornecer de forma nativa a metáfora de DBC.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes à integração dos espaços de trabalho de GCS no DBC (GARG et al., 2003; XDOCLET TEAM, 2004; BORLAND, 2004b).

3.6.1 JBuilder

O ambiente de desenvolvimento JBuilder, na sua edição para empresas, permite a construção de componentes EJB (SUN, 2004a) através de diagramas com notação semelhante à UML. O código desses componentes é gerado automaticamente e diversos descritores são produzidos, permitindo que a implantação do componente possa ser otimizada para a grande maioria dos servidores de aplicação existentes no mercado.

Independentemente do sistema de controle de versões utilizado, é possível obter recursos de GCS sobre os componentes modelados. A plataforma do JBuilder permite que diversos sistemas de controle de versões possam ser integrados, como, por exemplo, o CVS e o ClearCase. Esses sistemas têm seus comandos mapeados para um conjunto de comandos genéricos definidos pelo JBuilder. Desta forma, a troca do sistema de controle de versões não afeta a maneira com que o desenvolvedor interage com os recursos de GCS sobre os componentes. Contudo, devido às características complexas associadas à modelagem concorrente e distribuída, o JBuilder faz uso da política pessimista (bloqueio) sempre que algum desenvolvedor necessita editar o modelo de componentes.

A construção dos componentes ocorre a partir do modelo, passando pela geração do código fonte e posterior criação do pacote contendo os componentes compilados e os descritores de implantação. Esse processo é totalmente automatizado, possibilitando inclusive a automação do próprio processo de implantação através do uso de *plug-ins* específicos dos servidores de aplicação.

Apesar de existir uma boa integração do sistema de controle de versões e do sistema de controle de construções e liberações com o JBuilder, não existe nenhum suporte em relação ao sistema de controle de modificações. Além disso, a modelagem de componentes dentro do JBuilder é fortemente dependente da tecnologia EJB, não possibilitando a separação entre a especificação e a realização do componente.

Finalmente, o modelo de trabalho sugerido pelo JBuilder utiliza componentes principalmente como estruturas de controle de complexidade. As preocupações

referentes à reutilização ocorrem somente dentro da própria aplicação, pois não existe nenhum suporte para o controle e versionamento da biblioteca de componentes em produção. Outra deficiência acontece no apoio à substituição dos componentes, pois devido à forma automática em que as interfaces são criadas, com relacionamento um-para-um com os componentes, o uso de interfaces como contratos de serviços prestados por diversos componentes fica prejudicado.

3.6.2 XDoclet

Uma outra forma de construir componentes EJB é através do uso de programação orientada a atributos. O XDoclet (XDOCLET TEAM, 2004) é uma implementação em Java, com licença de código livre, para a programação orientada a atributos.

A filosofia por trás do XDoclet consiste em integrar no código de uma única classe todas as informações necessárias para gerar um componente. Essas informações são declaradas usando comentários do tipo Javadoc (SUN, 2004b) e pré-processadas pela ferramenta XDoclet, através de rotinas de construção descritas utilizando Ant. Esse pré-processamento analisa tanto o código da classe quanto os atributos colocados propositalmente nas seções de comentários da declaração da classe, dos métodos e dos atributos. Como resultado, são geradas as interfaces do componente, os descritores de implantação, o código completo do componente e as classes de apoio.

Desta forma, o código da classe com os comentários XDoclets pode ser visto como um IC fonte para o componente EJB, que é um IC derivado. Esse IC fonte pode ser armazenado em qualquer repositório de controle de versões e desenvolvido de forma paralela e concorrente por equipes distribuídas, característica essa que não é possível no caso do JBuilder devido a complexidade existente em efetuar junções sobre diagramas editados concorrentemente.

Apesar das vantagens aparentes, o XDoclet vai no sentido oposto ao das outras abordagens quando acumula em um único elemento as informações referentes a interfaces, componentes, descritores e classes de apoio. Esse tipo de abordagem tende a gerar problemas relacionados com escalabilidade. Além disso, de forma análoga ao JBuilder, as interfaces são relacionadas em um-para-um com os componentes, dificultando a substituição do componente e inibindo o uso de interfaces como elementos contratuais.

3.6.3 Ménage

O ambiente Ménage (GARG et al., 2003) foi construído para permitir a evolução de arquiteturas de linha de produtos descritas via xADL. Um dos objetivos almejados pelo autor é prover transparência nas atividades de GCS relacionadas ao versionamento dos componentes. O principal argumento defendido é que a atividade de construção de arquiteturas já é, por si só, demasiadamente complexa, e que a inclusão das atividades de GCS poderiam dificultar ainda mais o desenvolvimento.

Para atender a esse objetivo, o conceito de versionamento foi incorporado aos conceitos de componente, interface e conectores, possibilitando a criação de novas versões desses elementos através do uso dos comandos *check-in* e *check-out* integrados ao ambiente. Além disso, o conceito de configuração de referência pode ser obtido através da aplicação do conceito de versão para os componentes que definem subarquiteturas.

Um vasto conjunto de ferramentas integradas ao Ménage apóia tanto a construção quanto a evolução de arquiteturas xADL. Dentre essas ferramentas, a ferramenta de crítica traz contribuições especiais para a GCS através de críticas específicas para verificar a corretude sintática dos grafos de versões de componentes, interfaces e conectores e as condições de guarda de variantes e opções.

Outra ferramenta importante permite a seleção de componentes de forma intuitiva, através do preenchimento de propriedades de corte. Quando são atribuídos valores a um conjunto de propriedades de cortes, as condições booleanas descritas na seção 3.4.3 são calculadas e é determinado se os elementos devem ser removidos da arquitetura. Ao final desse processamento, a arquitetura original da linha de produtos pode ser transformada em uma arquitetura específica de um produto ou em uma arquitetura de linha de produtos simplificada. Esse segundo caso ocorre quando, mesmo após a seleção, ainda restam condições booleanas não determinadas.

Apesar de prover uma solução satisfatória de integração de espaço de trabalho, o Ménage lida somente com as questões referentes ao sistema de controle de versões, deixando a desejar nos aspectos relacionados ao controle de modificações. Por exemplo, não são definidos processos de controle para as atividades de criação da linha de produtos e de obtenção dos produtos. Sem esses processos, torna-se difícil a manutenção dos rastros dos pedidos de modificação que implicaram na evolução da própria linha de produtos, ou de um determinado produto.

3.6.4 Considerações sobre integração dos espaços de trabalho no DBC

As abordagens JBuilder e XDoclet, apresentadas nesta seção, apóiam o desenvolvimento de componentes EJB, possibilitando a construção automatizada desses componentes de forma integrada ao ambiente de desenvolvimento. Entretanto, o apoio à reutilização desses componentes é limitado. Nenhuma dessas abordagens considera a separação entre as interfaces e os componentes propriamente ditos. Esse tipo de postura privilegia a substituição de componentes em detrimento à reutilização. Além disso, o JBuilder define a política pessimista como obrigatória para o versionamento do modelo de componentes, e o XDoclet agrupa todas as informações relacionadas a um componente dentro de um único artefato.

De forma análoga ao JBuilder e ao XDoclet, o Ménage não possibilita o controle de modificações integrado ao ambiente de desenvolvimento. Porém, o suporte à integração do controle de versões de linhas de produtos é atendido a contento.

3.7 Conclusão

Apesar da existência de uma ampla infra-estrutura de GCS para o desenvolvimento convencional, a integração da GCS com paradigmas específicos do desenvolvimento de software ainda é muito carente (ESTUBLIER et al., 2002). O DBC é um exemplo de paradigma específico do desenvolvimento de software que necessita um maior apoio na evolução controlada de seus artefatos.

O DBC tem um alto grau de complexidade inerente às suas atividades, e já foi detectado que uma das maiores causas de falha em soluções inovadoras e consideradas promissoras da GCS está relacionada com a complexidade adicional que essas soluções introduzem nas atividades já existentes do desenvolvimento de software (ESTUBLIER et al., 2002). Por esse motivo, a aplicação de GCS no contexto de DBC se torna um grande desafio, que só será sobreposto com um maior investimento em pesquisa nessa área.

Neste capítulo, foram apresentadas abordagens que visam diminuir essa carência de GCS no DBC. Todavia, essas abordagens levantam novos problemas em cada um dos cinco tópicos discutidos. No próximo capítulo, são apresentados alguns caminhos para minimizar o efeito desses problemas, possibilitando que outros trabalhos mais aprofundados possam ser desenvolvidos.

Capítulo 4 - Odyssey-SCM

4.1 Introdução

Com o intuito de prover controle ao DBC através de GCS, é proposta a abordagem Odyssey-SCM (SCM: *Software Configuration Management*) (MURTA et al., 2004). Esta abordagem foi concebida no contexto do Projeto Odyssey (ODYSSEY, 2004), que tem como objetivo fornecer infra-estrutura para reutilização de software através de técnicas de Engenharia de Domínio, Linha de Produtos e DBC. A abordagem Odyssey-SCM é composta por cinco sub-abordagens com enfoque nos elementos da taxonomia definida no Capítulo 2, que são:

- Odyssey-SCMP (SCMP: *Software Configuration Management Process*): Processos, normas, procedimentos, políticas e padrões de GCS para o contexto de DBC;
- Odyssey-CCS (CCS: *Change Control System*): Sistema de controle de modificações configurável e extensível;
- Odyssey-VCS (VCS: *Version Control System*): Sistema de controle de versões baseado em políticas com suporte aos diversos níveis de abstração dos artefatos que descrevem componentes, interfaces e conectores;
- Odyssey-BRCS (BRCS: *Build and Release Control System*): Sistema de controle de construções e liberações orientado a arquitetura de componentes;
- Odyssey-WI (WI: *Workspace Integration*): Integração dos espaços de trabalho de GCS e DBC.

Cada uma dessas sub-abordagens agrega funcionalidades importantes para que o Odyssey-SCM atenda à hipótese descrita no Capítulo 1, viabilizando a aplicação de GCS no contexto de DBC com o intuito de prover um maior controle na evolução de artefatos em altos níveis de abstração. Todavia, um requisito fundamental da hipótese é a necessidade da abordagem não sobrecarregar demasiadamente as atividades de DBC com as novas atividades relacionadas a GCS.

A Figura 4 apresenta um panorama do relacionamento entre a abordagem Odyssey-SCM, suas sub-abordagens e as equipes produtoras e consumidoras de componentes. Segundo a abordagem proposta, toda e qualquer atividade de reutilização deverá ser passível de controle pelo Odyssey-SCM, possibilitando que o conhecimento referente a GCS seja extraído, processado e armazenado para posterior consulta.

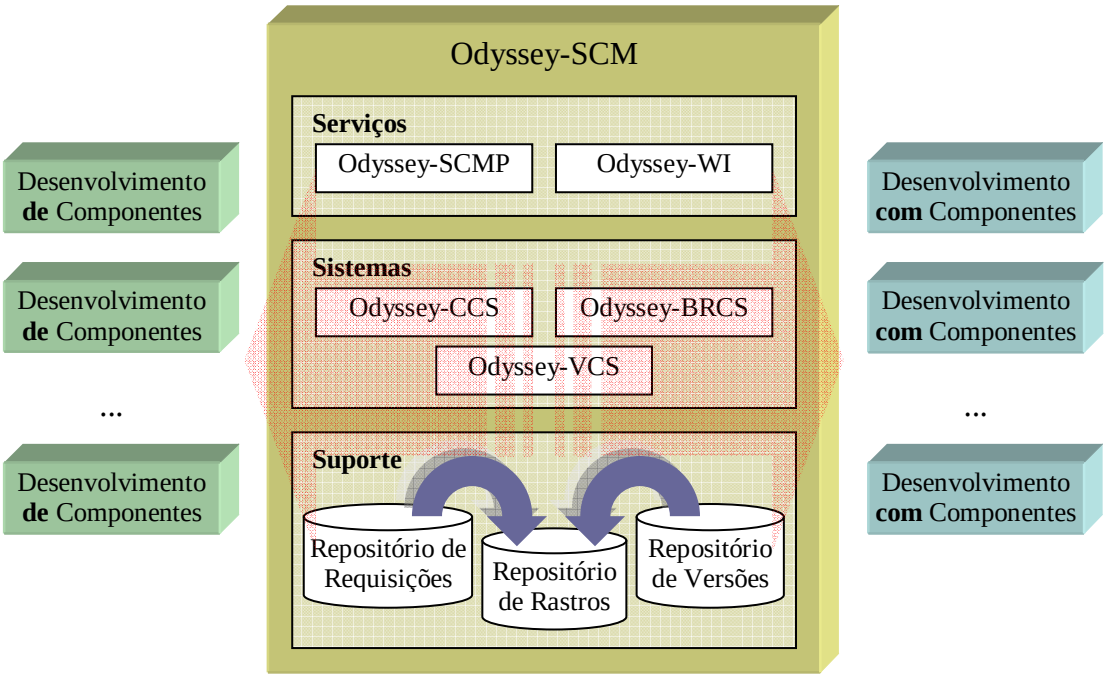

Figura 4: Panorama da solução proposta

Cada uma dessas abordagens é detalhada nas demais seções desse capítulo. Nesse detalhamento, inicialmente, cada abordagem é contextualizada e os problemas específicos são identificados. Posteriormente, é apresentada a proposta de solução referente à abordagem, e, finalmente, é relatado o andamento dos trabalhos de pesquisa referentes à abordagem em questão.

4.2 Odyssey-SCMP

Existem diversas normas para a aplicação de GCS no desenvolvimento convencional de software. Dentre essas normas, as que mais se destacam são a ISO 10007, a IEEE Std 828 e a IEEE Std 1042. Além disso, os modelos CMM e CMMI consideram a GCS como área chave de processo e determinam objetivos e atividades para atender às suas necessidades.

Apesar dessa grande variedade de normas para a GCS, nenhuma delas considera questões específicas do DBC. Essas normas tratam das quatro funções de GCS, que são

(1) identificação, (2) controle, (3) acompanhamento e (4) auditoria. Contudo, somente uma equipe de desenvolvimento de software é considerada nessas funções. No cenário de DBC, o número de equipes de desenvolvimento de software aumenta, assim como o relacionamento entre elas.

4.2.1 Abordagem proposta

Um processo genérico de DBC engloba, ao menos, duas etapas: desenvolvimento **de** componentes e desenvolvimento **com** componentes. Essas etapas são executadas respectivamente pela equipe produtora de componentes e pela equipe consumidora de componentes. Contudo, também podem existir equipes híbridas, que fazem uso de componentes para desenvolver outros componentes. Desta forma, podem ser identificados, ao menos, quatro participantes no processo de DBC, no que tange a GCS: equipes produtoras, equipes híbridas, equipes consumidoras e usuários finais.

Esses participantes devem interagir sempre que uma modificação for necessária. Diferentemente do processo convencional de GCS, uma requisição de modificações pode ser propagada para outras equipes caso o componente afetado tenha sido reutilizado, como exibido na Figura 5. Essa propagação deve levar em conta questões contratuais e análises de custo-benefício em relação ao desenvolvimento local da funcionalidade.

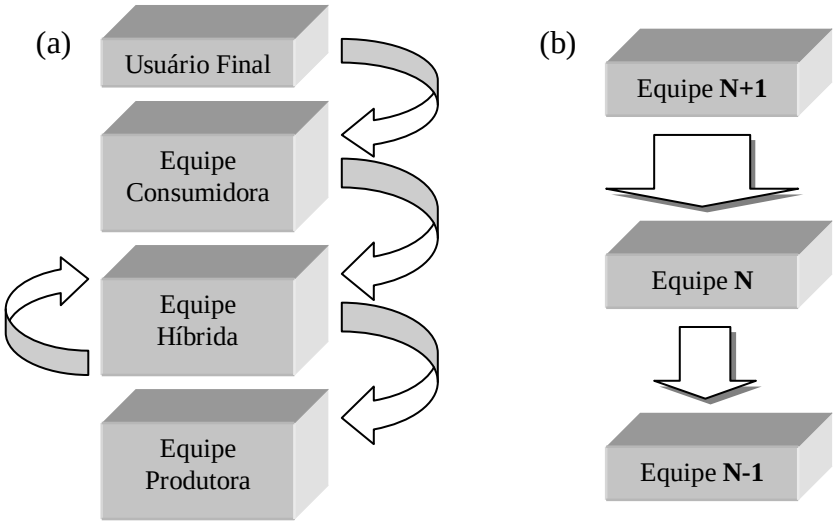

Figura 5: Propagação de requisições de modificação do processo de GCS no DBC

Neste cenário, como exibido na Figura 5.a, uma requisição de modificação é efetuada pelo usuário final. A equipe consumidora responsável pelo software alvo dessa requisição de modificação detecta, utilizando ferramental de suporte apropriado, que

parte da requisição diz respeito a um componente reutilizado. Devido a condições contratuais favoráveis, essa parte da requisição é propagada para a equipe que produziu o componente, recursivamente. Desta forma, o processo de GCS passa a se comportar como um processo multi-nível, onde processos de determinadas equipes interagem com processos de outras equipes, delegando parte das requisições sempre que necessário, como exibido na Figura 5.b.

O Odyssey-SCMP tem por objetivo definir um processo de GCS customizado para o DBC. A solução proposta consiste no estudo das normas ISO 10007 (ISO, 1995a), IEEE Std 1042 (IEEE, 1987), IEEE Std 828 (IEEE, 1998) e dos modelos CMM e CMMI (CHRISSIS et al., 2003) com o intuito de adaptá-los, levando em conta os requisitos peculiares do DBC. Além disso, o processo de referência gerado deverá estar disponível na infra-estrutura via o sistema de controle de requisições Odyssey-CCS para a sua customização e instanciação em projetos específicos.

4.2.1.1 Esforço de Reutilização

A reutilização de software no contexto de DBC pode ocorrer em diferentes etapas do ciclo de vida do componente. A reutilização de serviços de um componente em execução, ou a reutilização de componentes executáveis, porém não implantados, são as formas ideais de reutilização devido ao baixo custo de desenvolvimento adicional envolvido. Entretanto, pode não ser possível obter o componente desejado nesse estágio de desenvolvimento, ou até mesmo não ser viável adaptar o componente para as necessidades devido a incompatibilidades de plataforma.

Nesses casos, a reutilização de artefatos de mais alto nível de abstração pode ser uma boa alternativa. Quanto mais alto for o nível de abstração do artefato, menor será a dependência desse artefato com tecnologias específicas. Por esta razão, a probabilidade de reutilizar o artefato sem um excesso de adaptações aumentará. Contudo, artefatos muito abstratos tendem a ser demasiadamente genéricos, perdendo utilidade para situações específicas.

Desta forma, como exibido na Figura 6, um componente pode ser reutilizado em qualquer etapa do seu desenvolvimento, contudo, quanto mais precoce for essa reutilização, maior será o esforço de continuação de desenvolvimento despendido pela equipe consumidora. Por outro lado, componentes muito específicos podem ser difíceis de reutilizar, ou podem requerer um alto grau de adaptação.

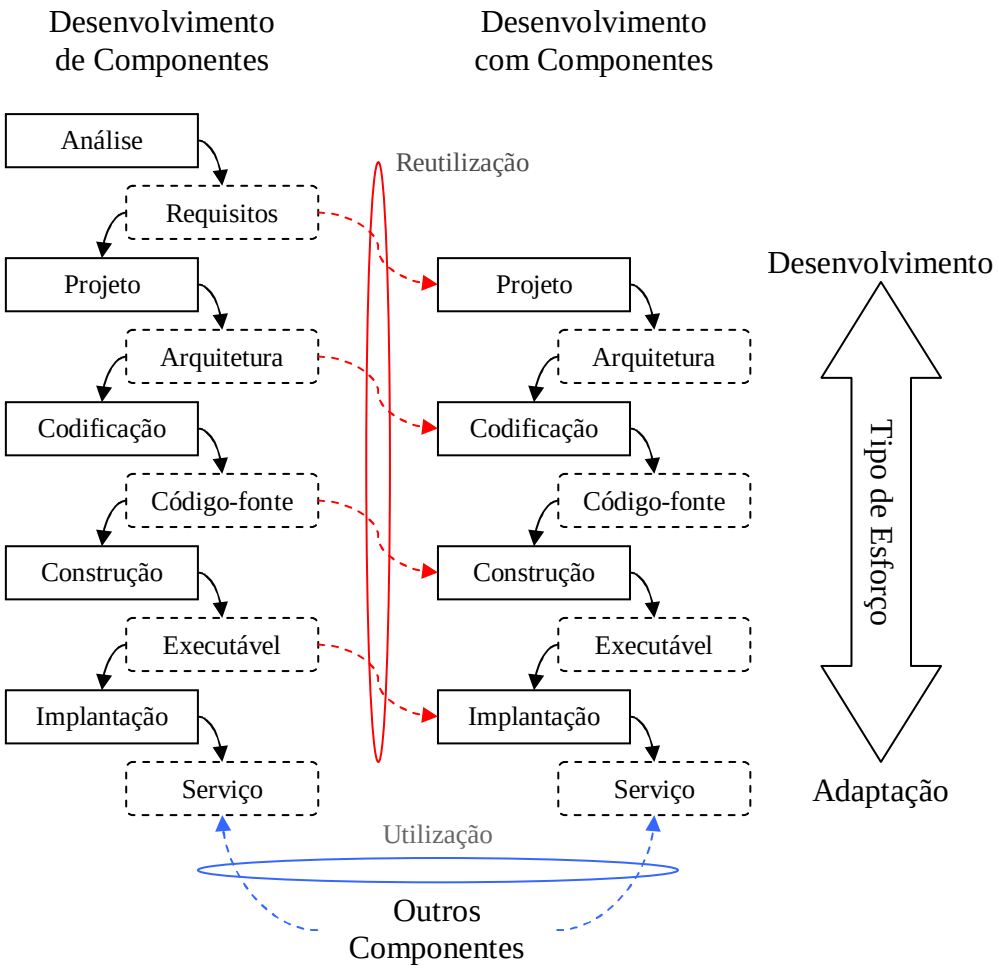

Figura 6: Esforço envolvido na reutilização de componentes em diferentes níveis de abstração

O esforço total (E_t) de produção e reutilização de componentes pode ser representado pela equação $E_t = E_d + \sum_{i=1}^{n} E_r^i$, onde E_d representa o esforço de desenvolvimento despendido pela equipe de produção de componentes e E_r representa o esforço de reutilização despendido por cada uma das n equipes de desenvolvimento com componentes. Desta forma, sempre que for possível transferir esforço das diversas equipes consumidoras para a equipe produtora, menor será o esforço total, com redução na taxa de $n:(1+E_a)$, onde E_a é o esforço adicional de adaptação do componente introduzido devido ao aumento de especificidade do mesmo. Essa relação de esforço também será válida durante a manutenção do componente.

4.2.1.2 Delegação de responsabilidades

De forma análoga ao desenvolvimento, a responsabilidade sobre a manutenção de um componente está intimamente relacionada com o ponto em que o componente foi

reutilizado. Quando um componente é reutilizado, todas as informações relacionadas ao fornecedor desse componente e às condições contratuais devem ser mantidas pelo sistema de GCS. Essas informações, juntamente com o conhecimento sobre quais partes do componente foram reutilizadas e quais partes do componente foram desenvolvidas ou adaptadas pela equipe consumidora, serão úteis para apoiar a decisão de propagar ou não uma requisição de modificação, como exibido na Figura 7.

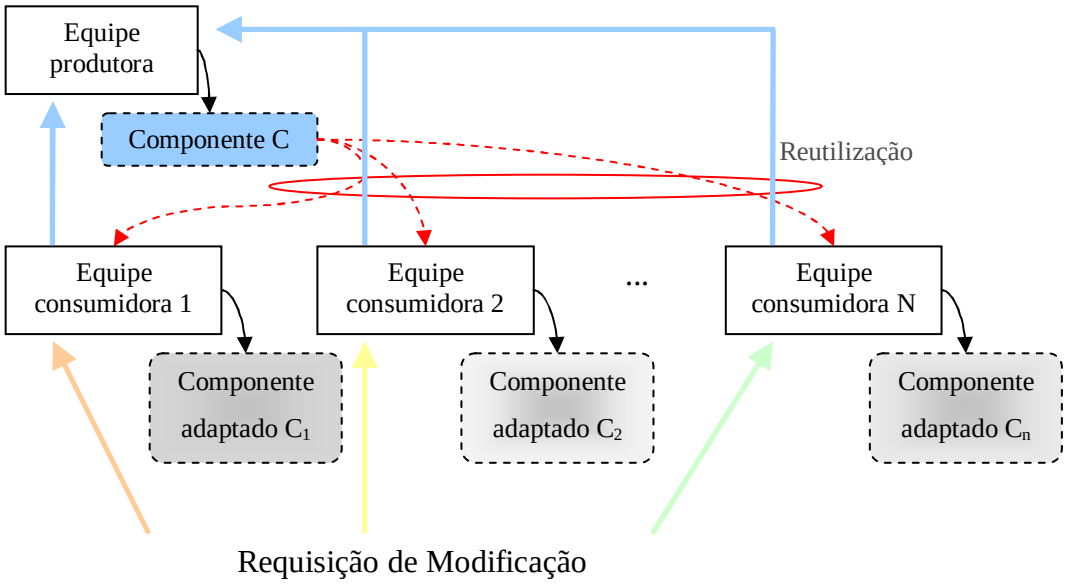

Figura 7: Propagação de requisições de modificação

A detecção correta de responsabilidades de manutenção é fundamental para evitar retrabalho e possibilitar que os componentes atinjam um elevado grau de qualidade. Quando um componente é reutilizado por diferentes equipes consumidoras, cada equipe consumidora efetua desenvolvimento adicional e adaptações sobre o componente, como já discutido anteriormente. Apesar da ocorrência dessas modificações, parte do componente mantém concordância com o componente reutilizável original. Por esta razão, modificações no componente devem ser precedidas de uma análise apropriada, para detectar quem é o verdadeiro responsável pela modificação.

Caso uma modificação de responsabilidade da equipe produtora seja implementada pela equipe consumidora, as demais equipes consumidoras estarão utilizando um componente de menor qualidade ou funcionalidade, ou então serão obrigadas a repetir o trabalho efetuado pela primeira equipe consumidora. Contudo, caso a modificação seja especificamente relacionada com adaptações ou necessidades

de uma determinada equipe consumidora, o esforço para implementar essa modificação no contexto da equipe produtora pode não compensar, devido a um pequeno interesse das demais equipes consumidoras.

4.2.2 Detalhamento

Uma primeira versão desse processo, ainda muito incipiente, foi elaborada (DANTAS et al., 2003). Nessa versão, são tratadas algumas questões específicas de DBC nas funções de identificação e controle da configuração. A abordagem utilizada foi, a partir da análise dos processos propostos pela IEEE e ISO, identificar pontos de adaptação que possibilitariam acrescentar suporte ao DBC.

4.2.2.1 Função de identificação

Na função de identificação são executadas tarefas referentes à seleção e documentação dos ICs, que podem estar em diferentes níveis de abstração (análise, projeto, codificação, teste, etc).

Inicialmente, em função da demanda das equipes consumidoras, pode ser priorizada uma ordem de construção de novos componentes. Neste contexto, os ICs podem ser classificados como internos, quando são desenvolvidos pela própria equipe consumidora, ou externos, quando são vindos da equipe produtora, seja ela pertencente à própria organização ou a terceiros.

A documentação dos ICs pode variar em relação a esta classificação. Para cada participante da equipe, pode ser definido um nível de acesso que determine a visibilidade da informação. O que, no entanto, deve ser ressaltado, é que toda a documentação que estiver disponível para o IC, inclusive a descrição de algoritmos e código-fonte, deve ser adicionada ao próprio IC, para que o mesmo possa ser projetado e testado de forma independente.

A granularidade do IC também deve ser levada em consideração. Heurísticas podem ser utilizadas para estipular o que considerar como ICs, visto que este pode englobar qualquer artefato produzido durante o desenvolvimento de software, ou um conjunto de artefatos. Uma possível heurística seria considerar elementos coesos e fracamente acoplados.

Deve-se ainda considerar a possibilidade de definição de interface como item de configuração em DBC. Apesar de se assumir a estabilidade dos contratos de interface como premissa para a compatibilidade entre componentes, em certas circunstâncias as

próprias interfaces evoluem, e todos os clientes de componentes que requerem ou provêem essas interfaces devem ser notificados para que possam se adequar, de forma não traumática, a essas modificações contratuais.

4.2.2.2 Função de controle

A função de controle é responsável pela autorização, implementação e verificação das modificações sobre os ICs. As tarefas que compõem a função de controle são: (1) requisição de modificação, (2) classificação, (3) análise, (4) avaliação, (5) implementação e (6) verificação. Segundo as normas de GCS, após a execução completa de um ciclo de controle, deve ser definida uma nova configuração de referência do sistema em desenvolvimento ou manutenção. Posteriormente ao estabelecimento de uma configuração de referência, os ICs só podem ser alterados utilizando esse processo formal de controle da mudança.

O controle da configuração seria adaptado para contemplar necessidades específicas de DBC, tanto na equipe produtora de componentes quanto nas equipes consumidoras de componentes.

4.2.2.2.1 Adaptação da função de controle na equipe produtora

No contexto de desenvolvimento de componentes, a requisição da modificação partiria das equipes consumidoras de componentes. Durante a classificação, seria necessário levar em consideração o impacto da modificação dentre as diversas equipes consumidoras para definir prioridades. Alem disso, quanto à análise de impacto, se a modificação implicasse na criação de novos componentes, seria necessário avaliar, além de custo e tempo de desenvolvimento, o interesse das demais equipes consumidoras.

Após a execução de um ciclo da função de controle, a função de acompanhamento deve relatar as modificações aos interessados. Contudo, para possibilitar que a função de acompanhamento seja capaz de notificar às equipes consumidoras sobre a modificação, a equipe produtora deve manter o rastro de reutilização em relação a todas as equipes consumidoras que adquiriram o componente, assim como um controle das versões utilizadas por cada uma dessas equipes.

4.2.2.2.2 Adaptação da função de controle nas equipes consumidoras

No contexto de desenvolvimento com componentes, a análise de impacto da equipe consumidora deve interagir com o processo de GCS da equipe produtora para

estimar o impacto da modificação, caso essa modificação atinja alguma parte reutilizada de componentes. Essa interação entre os processos de análise de impacto visa avaliar se é realmente viável modificar o componente no contexto da equipe produtora. Para isso, é fundamental o acesso aos dados da equipe produtora e às condições contratuais de reutilização. Caso a decisão de modificar o componente junto à equipe produtora não seja viável, ainda existem as opções de reutilizar um componente equivalente de outro fornecedor, desde que esse novo componente obedeça ao contrato de interfaces do componente atual, ou ainda, modificar o componente internamente à equipe consumidora, desde que esse componente seja distribuído como caixa-branca.

Se o componente for um COTS, usualmente distribuído como caixa-preta, a modificação deve ocorrer necessariamente junto ao fornecedor. Todavia, mesmo nesse cenário ainda existe a possibilidade da definição de adaptadores sobre o componente (GAMMA et al., 1995).

Na etapa de avaliação da modificação, a opção de aceitar a modificação passa a ser dividida entre aceitar optando pelo desenvolvimento, ou optando pela reutilização de componentes existentes. Neste último caso, é criada uma nova opção de escolha além das existentes no processo de GCS. Desta forma, uma requisição de modificação no processo de GCS da equipe consumidora pode motivar o início de um processo de reutilização de componentes.

Caso a implementação da modificação seja feita na equipe produtora, a equipe consumidora deve, ao receber a nova versão do componente, empacotá-lo em um IC dentro de seu processo de GCS e documentá-lo para que fique identificado o produtor do componente, as restrições e os direitos contratuais envolvidos na aquisição.

A equipe consumidora também deve verificar o componente realizando testes de integração e regressão. Esses testes não podem ser abandonados mesmo depois da equipe produtora assegurar que o componente, em conformidade com as suas interfaces, atende à especificação anterior ou especificação acrescida de uma nova funcionalidade. Os testes de regressão e integração também são importantes como forma de averiguar se os requisitos não funcionais continuam sendo atendidos no contexto de uso do componente.

4.2.3 Andamento

A partir de um trabalho inicial de customização do processo de GCS para as necessidades do DBC (DANTAS et al., 2003), novas demandas foram detectadas e

devem ser elaboradas levando em consideração, além das normas ISO e IEEE, os modelos CMM e CMMI e o processo de manutenção do método KobrA. Mais ainda, as funções de acompanhamento e auditoria também devem ser consideradas, juntamente com as atuais funções de identificação e controle.

Além disso, um maior estudo se mostra necessário para determinar o grau de suporte fornecido pelos sistemas de GCS nas atividades do processo. Esse suporte é fundamental para viabilizar a execução do processo, visto a complexidade existente devido ao escopo multi-nível envolvendo o controle de informações de rastro entre as equipes produtoras, híbridas e consumidoras de componentes.

4.3 Odyssey-CCS

Como apresentado no capítulo 3, referente à revisão da literatura de GCS aplicada no DBC, os processos de GCS relacionados com o DBC ainda estão imaturos e tendem a sofrer modificações até que uma maior estabilidade possa ser alcançada. Ainda não existe um consenso em relação a quais informações devem ser coletadas em cada atividade do processo. Também não existem normas que estabeleçam quais atividades são obrigatórias e quais atividades são opcionais na adoção de GCS no DBC.

Por estas razões, é essencial possibilitar a adaptação do sistema de controle de modificações a novos processos e permitir a configuração da coleta de informações em função das necessidades dos outros sistemas. Mais ainda, essa adaptação e configuração devem ocorrer não somente para os novos processos, mas também nas instâncias dos processos em execução, sem que ocorra perda de informações anteriores. Esse requisito é fundamental para que a abordagem seja viável em cenários reais de desenvolvimento.

4.3.1 Abordagem proposta

A solução proposta consiste na elaboração de uma abordagem configurável para o controle de modificações que permita o gerenciamento do grafo, que descreve o processo, e que faça uso de padrões de documentação para a coleta de informações durante as atividades do processo. Essa infra-estrutura pode ser utilizada pelos participantes de ambientes de DBC de duas formas: o usuário final pode solicitar modificações à equipe de desenvolvimento com componentes, e a equipe de desenvolvimento com componentes pode repassar parte das solicitações para a equipe de desenvolvimento de componentes, dependendo de como os artefatos afetados foram adaptados durante o processo de reutilização.

Para possibilitar a configuração do processo de GCS para DBC, a abordagem prevê a existência de processos primitivos e compostos. Os processos primitivos, que representam as atividades de GCS, permitem, entre outras características, a definição dos papéis autorizados a executar a atividade em questão e dos formulários para a documentação dessa atividade. Já os processos compostos fazem uso de um diagrama de atividades da UML para descrever os seus subprocessos. A Figura 8 exibe um modelo simplificado com os conceitos envolvidos no sistema de controle de modificações proposto.

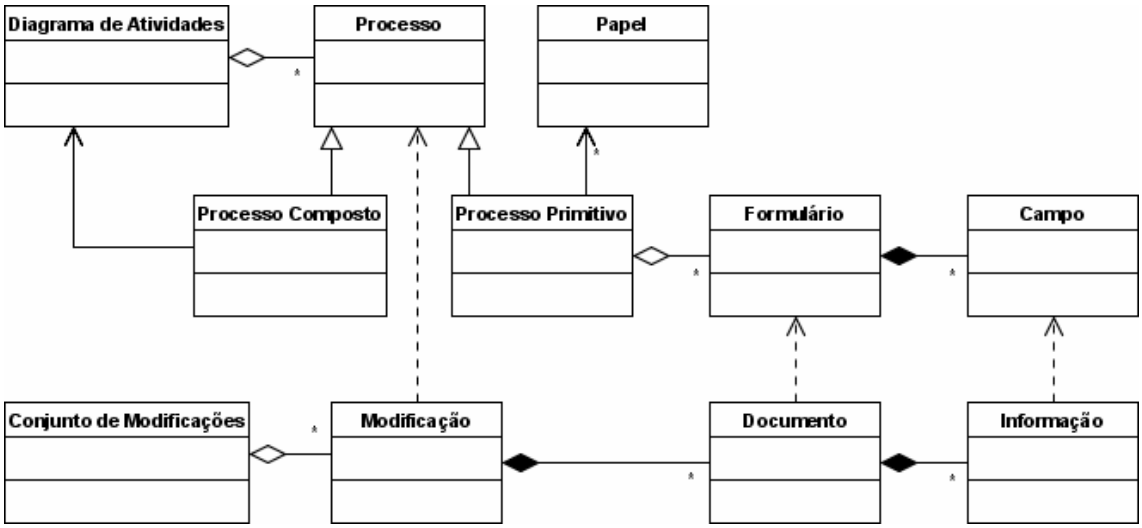

Figura 8: Modelo simplificado de controle de modificações

Na parte superior do modelo, são exibidos os elementos que representam o nível meta da abordagem. Para descrever um processo completo de controle de modificações pode ser necessário utilizar diagramas de atividades em diferentes níveis de abstração, possibilitando a reutilização de subprocessos. Esse recurso é fornecido através dos processos compostos.

De forma mais concreta, os processos primitivos possibilitam a interação entre os usuários do sistema e o sistema propriamente dito. Para cada processo primitivo, podem ser atribuídos os formulários necessários para a geração da documentação requerida pela atividade. Esses formulários são configuráveis em função das necessidades do processo e facilmente adaptáveis caso o processo precise evoluir.

Na parte inferior do modelo, são exibidos os elementos que instanciam o nível meta. Para cada execução do processo completo de controle de modificações, é criado um objeto que representa a modificação propriamente dita. Esse objeto concatena todos os documentos produzidos através do preenchimento dos formulários durante a

execução das atividades de GCS. Além disso, é permitida a criação de conjuntos de modificações que agreguem valor às equipes participantes, facilitando a troca de informações entre as equipes produtoras e consumidoras de componentes.

4.3.1.1 Máquina de processos

Com o intuito de viabilizar a modelagem e execução dos processos de controle de modificações é proposto o uso da máquina de processos Charon (MURTA, 2002; MURTA et al., 2002a; MURTA et al., 2002b), adaptada para atender aos requisitos específicos do problema em questão. A Charon possibilita a modelagem de processo através de uma notação estendida do diagrama de atividades da UML. Após a modelagem, o processo é instanciado constituindo uma base de conhecimento Prolog. Sobre essa base de conhecimento, agentes específicos para a simulação, a execução e o acompanhamento do processo são acionados. Além disso, é possível monitorar graficamente a execução do processo e, caso necessário, modificar o processo sem que as instâncias em execução sejam prejudicadas. A Figura 9 exibe o fluxo dessas atividades no contexto da máquina de processos Charon.

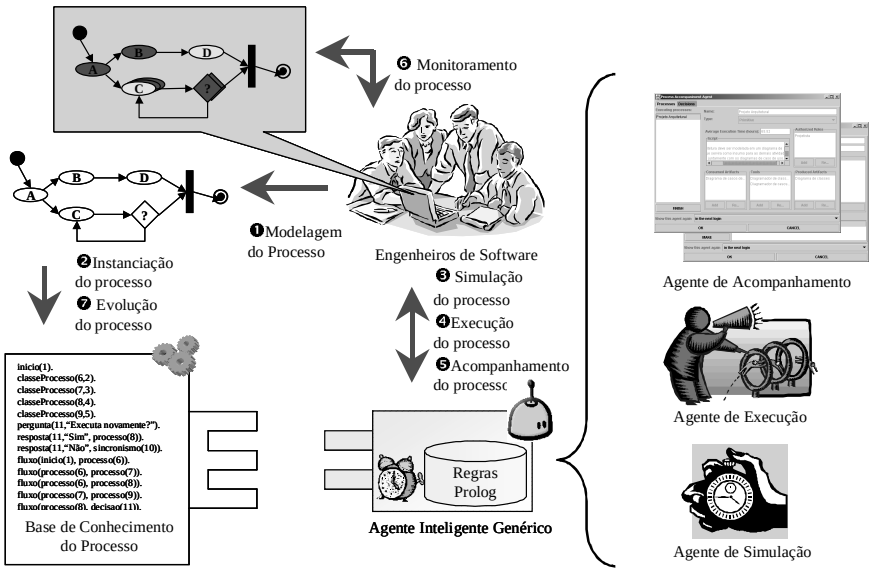

Figura 9: Atividades contempladas pela máquina de processos Charon.

Além de prover os agentes de simulação, execução e acompanhamento, a máquina de processos Charon define um *framework* para a criação de novos agentes. Esses novos agentes são descritos através de regras de inferência Prolog e conectados automaticamente às bases de conhecimento sempre que necessário. Essas regras de inferência Prolog fazem uso de uma meta-modelo genericamente definido para todas as

bases de conhecimento da Charon, com o intuito de manipular o conhecimento existente em função de um objetivo específico.

Desta forma, a máquina de processos Charon atende aos requisitos do Odyssey-CCS referentes ao suporte à modelagem de processos de GCS e adaptação desses processos durante as suas execuções. Mais ainda, devido ao fácil acesso ao conhecimento contido nas bases através do uso de agentes, é possível criar novos agentes que contemplem necessidades específicas de GCS, como, por exemplo, a análise das informações geradas pelo processo para a obtenção de métricas.

4.3.1.2 Documentação das atividades

Cada atividade do processo agrega novos conhecimentos ao ciclo de vida das modificações. Na atividade de requisição, por exemplo, informações referentes ao tipo, razão e gravidade da modificação, entre outras, são necessárias para possibilitar a execução das demais atividades. Entretanto, na atividade de análise ou de avaliação, o conjunto de informações necessárias é completamente distinto. Para possibilitar a configuração de formulários de documentação e posterior adaptação desses formulários, é proposto o uso do *framework* de documentação FrameDoc (MURTA, 1999; MURTA et al., 2001).

Na sua concepção original, o FrameDoc se destinava à documentação de artefatos reutilizáveis. Todavia, suas características são propensas para o cenário em questão, pois é possível criar formulários customizados de documentação e associar esses formulários às atividades de GCS modeladas através da Charon, sem a necessidade de adaptações profundas.

Esses formulários podem ser utilizados nas diversas atividades do processo de GCS com intuitos diferentes. Do ponto de vista de análise de impacto, eles podem, por exemplo, apoiar à análise de causa através de campos específicos para manutenções corretivas. Já do ponto de vista de identificação de componentes, eles podem, por exemplo, coletar informações referentes aos fornecedores de componentes, viabilizando o suporte ao desenvolvimento de software através de diversas camadas contratuais de produção e consumo de componentes.

4.3.2 Detalhamento

A utilização do Odyssey-CCS consiste, principalmente, nas etapas referentes à preparação do ambiente e execução dos processos. A etapa de preparação do ambiente é

executada pelo gerente de configuração responsável por implantar no Odyssey-CCS a norma adotada pela organização. Esse participante deve criar os formulários necessários e, para cada formulário criado, criar os campos que compõem esse formulário. Posteriormente, o processo deverá ser modelado e associado aos respectivos formulários. Além disso, devem ser determinados os participantes que exercerão os papéis associados às atividades. Após essa etapa, a preparação do sistema está finalizada e o sistema pode entrar em execução.

Para cada vez em que um processo for instanciado, um objeto que representa a modificação que motivou a instanciação desse processo será criado. Esse objeto atua como repositório para todos os documentos gerados pelos participantes durante as atividades do processo. A geração desses documentos ocorre através do preenchimento dos formulários associados às atividades. Ao término do ciclo de vida de uma modificação, o objeto que representa a modificação contém todo o conhecimento referente ao processo, organizado hierarquicamente através das estruturas de atividades, formulários e campos.

Esse conhecimento, acumulado através das diversas instanciações de processos de controle de modificações, será útil para possibilitar a execução das funções de acompanhamento da configuração e de auditoria da configuração. Além disso, a análise sobre essas informações pode permitir a detecção de deficiências no próprio processo de controle de modificações.

4.3.3 Andamento

Em trabalhos anteriores, foram definidos mecanismos para a configuração de padrões de documentação e para a modelagem, instanciação, simulação, execução, monitoramento e evolução de processos de software. Esses trabalhos servirão de base para possibilitar, respectivamente, a configuração das informações a serem coletadas em cada atividade do processo de GCS e a modelagem e execução desse processo.

Dentre as atividades de pesquisa necessárias para possibilitar uma especificação mais detalhada da abordagem, estão: (1) a definição de mecanismos de rastro entre as modificações cadastradas no Odyssey-CCS e as versões de componentes existentes no Odyssey-VCS; (2) a representação do conhecimento referente à colaboração entre as equipes produtoras e consumidoras de componentes e os usuários finais; e (3) o estudo dos tipos de campo necessários para a construção de formulários de GCS.

4.4 Odyssey-VCS

Usualmente, um componente é descrito e construído usando artefatos pertencentes a diversos níveis de abstração, estruturados através de diferentes modelos de dados. Atualmente, existe suporte satisfatório para o controle de versões sobre artefatos de baixo nível de abstração, como, por exemplo, código-fonte. Contudo, existe uma grande carência no suporte aos artefatos de alto nível de abstração, como, por exemplo, modelos de análise e projeto.

Entretanto, esses artefatos de alto nível de abstração seguem modelos de dados específicos. A implementação de sistemas de controle de versões dependentes desses modelos de dados seria muito custosa, especialmente no que se refere ao acompanhamento da evolução e à integração desses modelos. Uma abordagem possível para lidar com esses artefatos é fazer uso de modelos de dados padrões.

Alem disso, questões referentes à utilização de metáfora homogênea para os diversos tipos de ICs, a possibilidade de desenvolvimento concorrente desses ICs e a distribuição através da Internet são requisitos desejáveis para qualquer sistema de controle de versões.

4.4.1 Abordagem proposta

Mesmo restringindo o contexto para artefatos de alto nível de abstração, ainda existem diferentes tipos de ICs que são sujeitos ao controle de versões. No caso específico de artefatos UML, esses ICs podem ser classes, casos de uso, atributos, métodos, modelos, etc. Como pode ser percebido, apesar de todos esses artefatos pertencerem à UML, eles estão em níveis diferentes de uma mesma hierarquia de composição. Por exemplo, métodos e atributos fazem parte de classes, e classes fazem parte de modelos.

Para possibilitar a discussão sobre o relacionamento entre esses diversos elementos, foram definidos os conceitos de grão de comparação e grão de versionamento. O conceito de grão de comparação estabelece os artefatos na hierarquia de composição que devem ser utilizados na comparação de modificações. De forma análoga, o conceito de grão de versionamento estabelece os artefatos na hierarquia de composição que necessitam conter informações sobre versão.

A solução proposta consiste na definição de uma abordagem de controle de versões que atue sobre modelos descritos por meta-modelos MOF, fazendo uso de

políticas configuráveis para os grãos de comparação e versionamento. O MOF é um padrão para a descrição de meta-modelos e modelos independentemente de plataforma, definido pela OMG. A UML, o CWM (*Common Warehouse Metamodel*) (OMG, 2003a) e o próprio MOF são exemplos de meta-modelos MOF que poderiam ser beneficiados por essa abordagem. Além desses meta-modelos, é possível definir meta-modelos customizados usando diversas ferramentas disponíveis livremente para a comunidade.

4.4.1.1 Grão de comparação convencional

Quando abordagens convencionais, como o CVS ou ClearCase, são adotadas para o controle de versões, o grão de comparação utilizado é linha de arquivo do sistema operacional. No caso de ICs contendo documentos texto, esse grão de comparação é perfeitamente adequado, pois uma linha é delimitada pelos caracteres especiais CR e LF, o que estabelece equivalência para parágrafos nesses documentos. Parágrafos têm grau de coesão suficiente para serem utilizados como grãos de comparação. A Figura 10.a exibe o grão de comparação em documentos texto utilizado pelas abordagens convencionais de controle de versões.

(a) Documento texto (b) Código-fonte Java

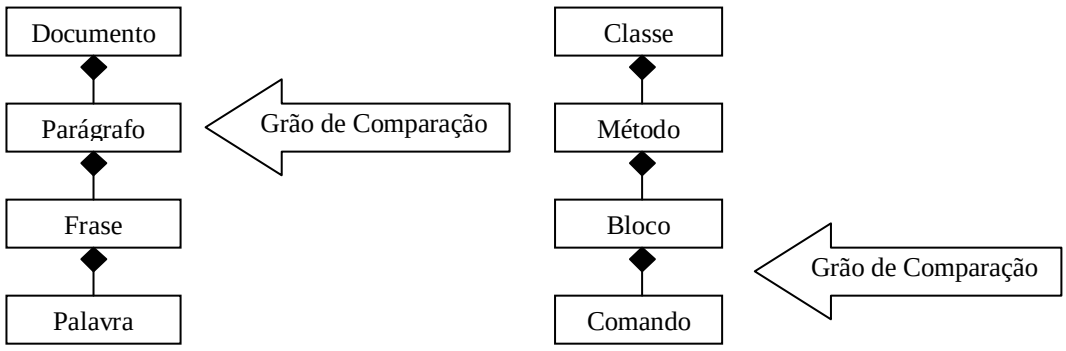

Figura 10: Grão de comparação em documento texto e código-fonte Java

Porém, nem sempre é compatível o casamento entre o conceito de linha de arquivo do sistema operacional, utilizado pelas abordagens convencionais como grão de comparação, e os conceitos utilizados logicamente pelas estruturas que serão versionadas. No caso de código-fonte Java, por exemplo, o grão de comparação será mapeado para algo maior ou igual a um comando e menor ou igual a um bloco de comandos, como exibido na Figura 10.b. Isso ocorre porque em Java uma mesma linha

pode conter mais de um comando, e um bloco de comandos, circunscrito por chaves, pode ocupar mais de uma linha.

Fazendo análise semelhante ao caso de parágrafos em documentos texto, pode ser constatada a falta de coesão no grão de comparação utilizado pelas abordagens convencionais para código-fonte Java. Essa falta de coesão irá gerar dificuldades na detecção de conflitos, pois o grão de comparação é fino demais. Para essa situação, um grão de comparação mais adequado seria o método como um todo.

Desta forma, o uso de grão de comparação demasiadamente fino pode impossibilitar a detecção de conflitos existentes ou possíveis colaborações entre os participantes do desenvolvimento de software. Por outro lado, o uso de grão de comparação demasiadamente grosso pode inviabilizar a diferenciação entre as partes isoladas do sistema, culminando na detecção incorreta de um maior número de conflitos. O conceito de coesão pode ser útil para a definição de um grão de comparação correto. Contudo, vale lembrar que a coesão é dependente do modelo de dados específico do IC. Por exemplo, em documento texto, o parágrafo tem um grau adequado de coesão. No caso de código-fonte Java, o método pode ser visto como a unidade mínima de coesão.

4.4.1.2 Grão de versionamento convencional

A adoção das abordagens convencionais também introduz restrições referentes ao grão de versionamento. O grão de versionamento utilizado por essas abordagens é o próprio arquivo do sistema operacional. Desta forma, esse grão de versionamento é perfeitamente adequado para o caso de documentos texto, como exibido na Figura 11.a, onde existe um mapeamento para o documento como um todo. Contudo, no caso de código-fonte Java, o mapeamento não é satisfatório, como exibido na Figura 11.b, pois um arquivo pode conter uma ou mais classes. Seria mais adequada a utilização de classes como grão de versionamento nessa situação.

A escolha de um grão de versionamento demasiadamente fino introduz complexidade desnecessária ao sistema de controle de versões, tornando inviável a identificação das versões devido ao excesso delas. Por outro lado, a escolha de um grão de versionamento demasiadamente grosso impossibilita o controle da evolução dos elementos que compõem o sistema e aumenta o número de bloqueios, caso essa política esteja sendo utilizada.

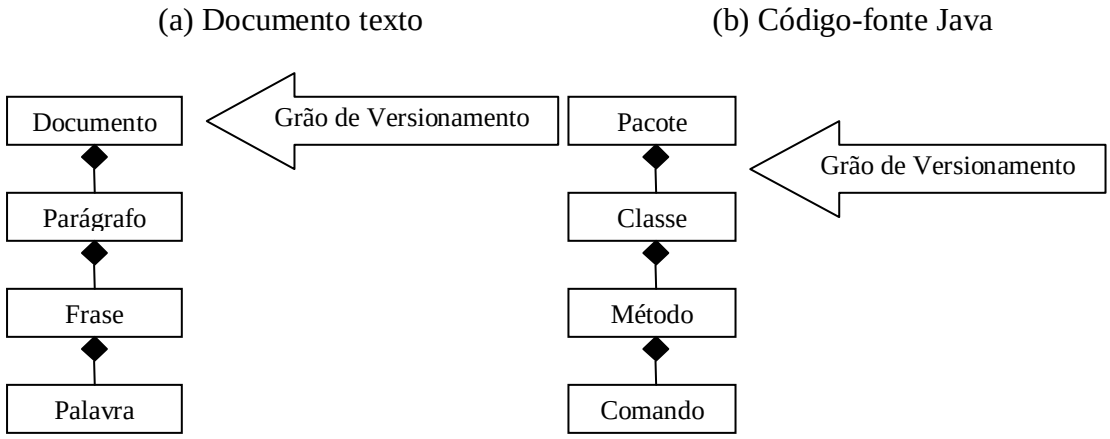

Figura 11: Grão de versionamento em documento texto e código-fonte Java

4.4.1.3 Grãos de comparação e versionamento convencionais utilizados em modelos

Quando os ICs são modelos, a situação é ainda mais catastrófica caso as abordagens convencionais venham a ser adotadas. Os modelos que seguem meta-modelos MOF, como, por exemplo, a UML e o CWM, podem ser externados através de XMI. Esses modelos XMI, que são usualmente armazenados em arquivos e colocados sob versionamento, têm um grão de comparação extremamente fino, conforme exibido na Figura 12. Por exemplo, um atributo de uma classe modelada em UML pode ocupar diversas linhas do arquivo XMI gerado. Desta forma, cada uma dessas linhas tem, se analisada isoladamente, uma baixa coesão e um grande acoplamento com as demais linhas. Por essa razão, grande parte dos conflitos reais será ignorada.

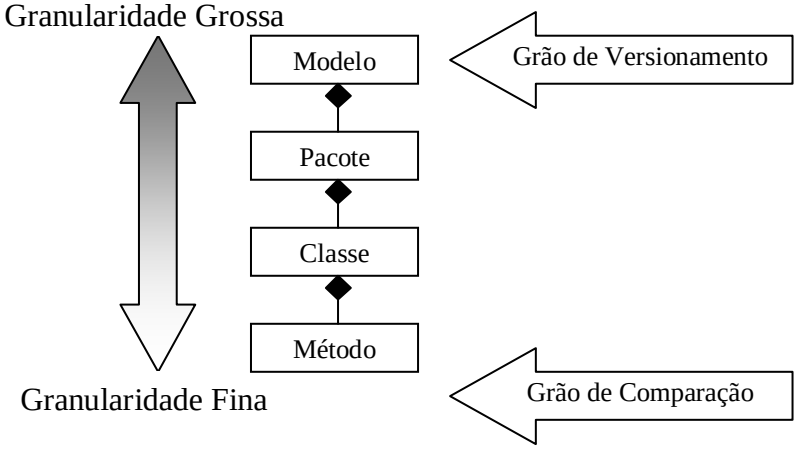

Figura 12: Grãos de comparação e de versionamento em modelos XMI

Nesta mesma situação, o grão de versionamento será extremamente grosso, pois o modelo completo do sistema é armazenado em um único arquivo XMI. Essa postura

inviabiliza o controle individual das versões de elementos do modelo, como, por exemplo, casos de uso e classes. A Figura 12 exibe esse cenário, onde um sistema completo, que pode ser composto por milhares de classes, tem grão de versionamento equivalente ao sistema como um todo. Nesse cenário, onde os ICs são modelos, seria mais indicado configurar o grão de comparação para elementos do tipo característica (*Feature*) do meta-modelo da UML, como, por exemplo, métodos e atributos, e configurar o grão de versionamento para elementos do tipo classificador (*Classifier*) do meta-modelo da UML, como, por exemplo, classes e casos de uso.

4.4.2 Detalhamento

Diferentemente das abordagens convencionais, que tem grãos de comparação e versionamento fixos, a abordagem proposta permite que esses elementos sejam configuráveis em função do próprio meta-modelo dos ICs. Para que seja possível controlar as versões de um determinado meta-modelo MOF, é necessário estabelecer a hierarquia de composição dos seus elementos e determinar quais elementos são grãos de comparação e de versionamento. Utilizando essa informação, o Odyssey-VCS é capaz de lidar com os modelos criados a partir desse meta-modelo.

A abordagem proposta faz uso do repositório NetBeans MDR (MATULA, 2003), para persistir as versões de modelos criados a partir de meta-modelos MOF. Esse repositório se baseia na especificação JMI (DIRCKZE, 2002), que possibilita a geração de APIs a partir de meta-modelos MOF e utilização dessas APIs para acessar os elementos persistidos no repositório MOF. A Figura 13 exibe um exemplo de trecho de código que descreve grãos de comparação e de versionamento para elementos do meta-modelo da UML.

```
...
<elemento tipo=”org.omg.uml.foundation.core.UmlClass” graoVersionamento=”true”>
  <elemento tipo=”org.omg.uml.foundation.core.Attribute” graoComparacao=”true” />
  <elemento tipo=”org.omg.uml.foundation.core.Operation” graoComparacao=”true” />
</elemento>
...
```

Figura 13: Exemplo de configuração dos grãos de comparação e versionamento para modelos UML

Nesse descritor, elementos do tipo classe são considerados grãos de versionamento e elementos dos tipos atributos e métodos são considerados grãos de comparação. Além disso, é definida a hierarquia de composição onde atributo e método são partes de classe.

Caso seja detectado que a configuração inicial para um determinado meta-modelo MOF não é adequada, basta alterar o descritor de configuração para que novos grãos de comparação ou versionamento sejam utilizados. Para cada interação do cliente com o repositório, esse descritor é consultado e as políticas corretas são utilizadas. Caso o elemento não tenha política definida, uma política global estipulada no nível do meta-meta-modelo MOF é adotada.

O Odyssey-VCS pode ser utilizado por qualquer ambiente de desenvolvimento de software que tenha funcionalidade de importação e exportação de seus modelos no padrão XMI. Esses modelos são transmitidos para o Odyssey-VCS como exibido na Figura 14.

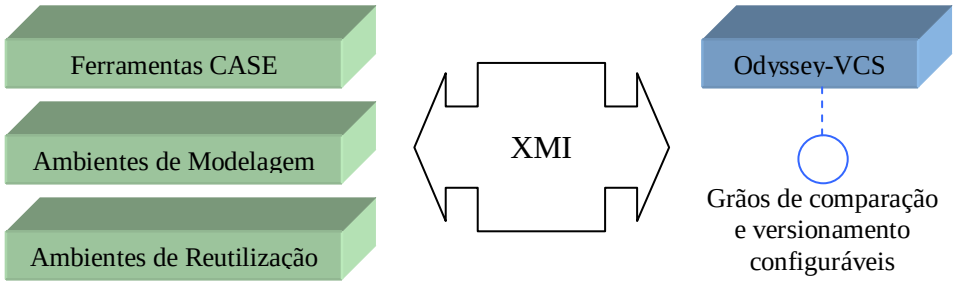

Figura 14: Cenário de utilização do Odyssey-VCS

4.4.3 Andamento

Em experiências anteriores foi construída a ferramenta Token (MURTA et al., 2000), que possibilita o controle de versões sobre módulos de sistemas através de políticas de bloqueio. Essa abordagem foi aprimorada para atender ao controle de versões de elementos de modelo do ambiente Odyssey, desencadeando na criação da ferramenta LockED (TEIXEIRA et al., 2001a; TEIXEIRA et al., 2001b). A ferramenta LockED, apesar de apoiar o controle de versões sobre elementos de modelo, está presa ao meta-modelo do ambiente Odyssey e herda a política de bloqueio da ferramenta Token. O objetivo do Odyssey-VCS é contornar essas limitações através da utilização do meta-modelo MOF e de grãos de comparação e versionamento configuráveis.

Atualmente, uma versão inicial do Odyssey-VCS está sendo implementada por um aluno de mestrado. Essa versão irá fornecer a infra-estrutura para a conexão de comportamentos através de novas classes Java. Essa infra-estrutura possibilitará que, para cada novo elemento definido em um meta-modelo MOF, seja construída uma classe “tratadora” para a comparação e o versionamento desse elemento. Além disso, será construída a política genérica, situada no nível do meta-meta-modelo MOF,

contendo definições para comparação e versionamento de elementos que não têm classes “tratadoras” definidos.

A partir dessa versão inicial, serão implementadas extensões para possibilitar o uso de descritores de grãos de comparação e versionamento. A infra-estrutura de conexão de comportamentos através de classes “tratadores” continuará disponível para que a abordagem possibilite, para situações complexas, um ajuste mais fino da sua execução.

4.5 Odyssey-BRCS

Atualmente, existem diversas soluções para a construção de executáveis a partir de código-fonte. Dentre elas estão o make (FELDMAN, 1979) e o Ant (HATCHER et al., 2004). Contudo, o conceito de construção não está relacionado somente com geração de executáveis, mas sim no processo de obtenção de ICs derivados a partir de um conjunto de ICs fonte.

No caso de DBC, é necessário fornecer a metáfora de GCS embutida nos próprios conceitos de componente, conectores e interfaces. Para que isso seja possível, é proposta a utilização de uma camada de mapeamento entre os diversos sistemas de GCS e o ambiente de DBC. Esta camada, denominada Odyssey-BRCS, lida com o problema da estruturação dos componentes em relação aos seus artefatos internos e em relação aos demais componentes relacionados.

4.5.1 Abordagem proposta

A abordagem proposta consiste no tratamento de um componente como configuração de referência dos seus artefatos internos, e utilização das interfaces e conectores como contratos entre componentes, fazendo uso de linguagens de descrição de arquitetura (ADL – *Architecture Description Language*) para representar os modelos de construção de sistemas (*system models*).

O controle de versões sobre componentes tem sido feito, até então, pelas mesmas ferramentas usadas para controlar versões no desenvolvimento convencional de software. Por essa razão, apesar de ser possível controlar individualmente as versões das partes de um componente, não existe um controle de versões automatizado para o componente como um todo. Esse problema introduz as seguintes questões:

- Como seria possível controlar a evolução dos artefatos que descrevem componentes de uma forma consistente, usando os próprios conceitos do domínio de DBC?
- Como o relacionamento entre componentes pode ser expresso através das suas partes armazenadas em repositórios de GCS?
- Como as partes de componentes podem ser correlacionadas? É possível manter de forma automática ou semi-automática a consistência entre elas?
- Qual é o relacionamento entre o versionamento dos artefatos que descrevem os componentes e o versionamento dos componentes propriamente ditos?

Para responder essas questões, que fazem parte do sistema de controle de construções e liberações, está sendo definida uma infra-estrutura que alinhe os conceitos de DBC e GCS. Essa infra-estrutura pode ser vista como uma camada de mapeamento entre o idioma usado pelos usuários de DBC e o idioma usado pelas soluções existentes de GCS.

4.5.1.1 Integração contínua

Para apoiar o DBC, é proposta a utilização de técnicas específicas para a integração contínua dos componentes. A integração dos componentes pode ser decomposta em três níveis de atuação: (1) detecção de conflitos, (2) detecção de quebras e (3) detecção de quebras lógicas. As ferramentas convencionais de controle de versão atuam no nível de detecção de conflitos. Contudo, devido a um grão de comparação inadequado, vários conflitos passam despercebidos e só são detectados quando o sistema é compilado.

A compilação do sistema é capaz de encontrar conflitos como, por exemplo, quando dois desenvolvedores editam linhas diferentes que manipulam um mesmo elemento. Nessas situações o sistema pode parar de compilar, e será possível detectar os desenvolvedores envolvidos nas modificações suspeitas da quebra.

Em um cenário ainda mais complexo, o conflito passa despercebido inclusive pela compilação, contudo, quando a bateria de testes é executada, é possível detectar que algo indesejável ocorreu nas últimas modificações. Para automatizar o processo de

integração contínua dos componentes escritos em Java, são utilizadas as ferramentas Ant, javac e junit.

Com o objetivo de atender aos requisitos de integração contínua (FOWLER et al., 2004), os componentes deverão conter, além dos artefatos de modelo e código-fonte, rotinas de teste automatizadas. Devido à estruturação de componentes em hierarquias de composição, o teste de unidade de um componente pode ser visto como teste de integração dos seus sub-componentes. Na Figura 15, é exibido este cenário onde uma bateria de testes de unidade é criada para o componente A. Esses testes de unidade do componente A manipulam os componentes B, C e D. Sob a perspectiva desses componentes, os testes de unidade de A passam a ser considerados testes de integração. Desta forma, a quantidade de testes de integração de um determinado componente pode variar em função das ocorrências de reutilização desse componente.

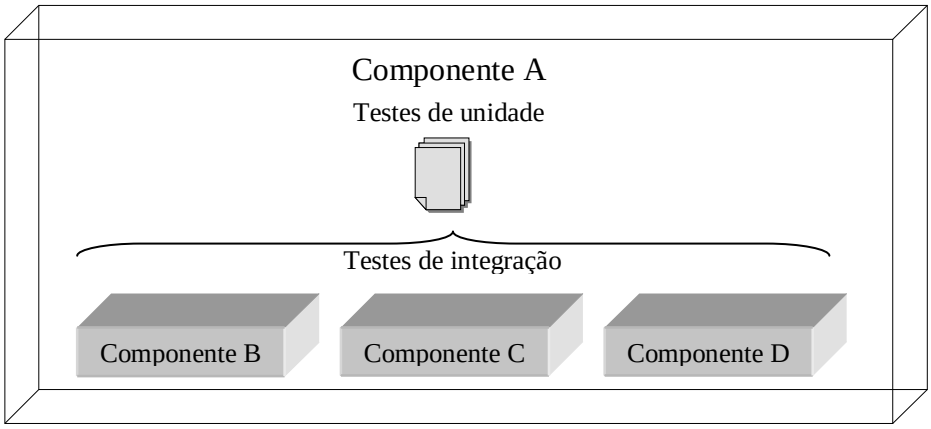

Figura 15: Perspectivas de teste no contexto de DBC

4.5.2 Detalhamento

A meta do Odyssey-BRCS é construir uma infra-estrutura de GCS customizada para DBC. A Figura 16 mostra as várias camadas que compõem a solução proposta. A camada mais alta representa a arquitetura de sistemas. Essa arquitetura pode ser considerada pela GCS como um modelo de construção de sistemas, que mostra como as partes principais de um sistema se relacionam, fornecendo suporte para as atividades de construção e liberação de GCS. A camada seguinte, de cima para baixo, correlaciona as partes dos componentes. Cada componente pode abranger especificações, modelos, códigos-fonte, executáveis, testes e serviços em execução. O controle do relacionamento e a evolução desses artefatos devem ocorrer de forma consistente. Para isso, um novo artefato passa a ser necessário: o descritor de integração. A camada

seguinte, de mediação, mapeia cada tipo de artefato com o correto repositório de GCS. Já a camada de controle de versões fornece acesso aos repositórios de GCS, como, por exemplo, CVS (CEDERQVIST, 2003), MCCM (LINGEN et al., 2004) ou o próprio Odyssey-VCS.

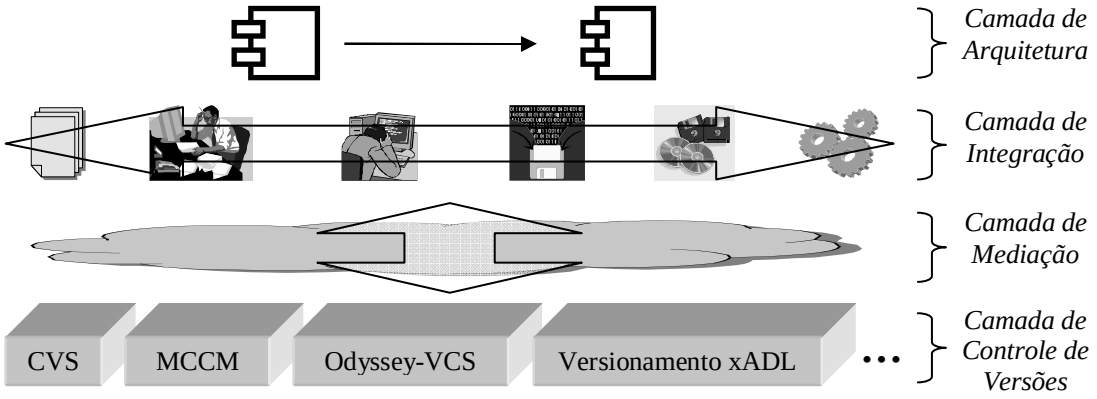

Figura 16: Gerência de Configuração Baseada em Arquitetura.

A meta apresentada pode ser dividida em objetivos específicos de cada camada descrita na Figura 16. Os objetivos específicos da **Camada de Arquitetura** são:

- Usar a arquitetura de sistema como um modelo de construção de sistemas (um documento que descreve as partes dos sistemas). A arquitetura de sistemas está para o DBC assim como um descritor de construção do Ant está para códigos-fonte Java;
- Tratar um componente como a configuração de referência das suas partes (sub-componentes) e tratar um componente como membro das configurações de referência descritas por outros componentes que ele faz parte;
- Tratar interfaces como contratos entre componentes, permitindo a associação automática de novas versões de componentes às configurações de referência como opções no espaço de versões se, e somente se, suas interfaces não tiverem sido modificadas; e
- Usar conectores como descritores do relacionamento entre membros de contratos no contexto de configurações de referência específicas. Essa técnica pode ser usada para marcar componentes inconsistentes sempre que uma nova versão de interface for adotada por outros componentes relacionados.

Os objetivos específicos da **Camada de Integração** são:

- Definir cada artefato possível, levando em conta a necessidade específica do DBC, que possa descrever um componente (ex.: modelos UML, códigos-fonte Java/EJB, descritores de implantação, casos de teste, endereços de serviços para a *web*, etc.), uma interface (ex.: CORBA-IDL, códigos-fonte Java/EJB, especificações de contrato, etc.) e um conector (ex.: padrões de adaptação, descritores de conexão, etc.);
- Representar componentes usando esses artefatos e os seus sub-componentes;
- Descrever o relacionamento entre esses artefatos usando o conceito de componentes, interfaces e conectores como indexadores; e
- Controlar a integridade e a consistência dos componentes durante a evolução das suas partes.

Os objetivos específicos da **Camada de Mediação** são:

- Mapear cada tipo de artefato para o sistema de controle de versões correto. Alguns sistemas de controle de versões são mais bem aplicados para determinados tipos de artefatos. Por esse motivo, os artefatos podem precisar residir em diferentes repositórios (ex.: código-fonte no CVS e modelos UML no Odyssey-VCS);
- Especificar, para casos especiais, como artefatos podem ser divididos e armazenados em diferentes repositórios, ou agrupados e armazenados no mesmo repositório; e
- Definir uma forma de descrever o relacionamento entre os artefatos e seus repositórios.

Os objetivos específicos da **Camada de Versionamento** são:

- Selecionar os sistemas de controle de versão mais apropriados para cada tipo de artefato.

4.5.3 Andamento

Atualmente, estão sendo experimentadas técnicas de integração automatizadas utilizando Ant e javac para a construção noturna (*nightly build*) do ambiente Odyssey. Essas técnicas devem ser aprimoradas para contemplar a execução de testes junit e notificação dos resultados através de páginas HTML e e-mail. Além disso, estão sendo definidos procedimentos de comunicação com o repositório de versões para obtenção dos nomes dos suspeitos de quebra.

Os aspectos referentes a integração das partes de componentes do Odyssey-BRCS deverão ser tratados durante o estágio de doutorado na Universidade da Califórnia, em Irvine, junto à equipe que definiu a ADL a ser utilizada: a xADL (DASHOFY et al., 2001).

4.6 Odyssey-WI

O sucesso da adoção de novos métodos e ferramentas no desenvolvimento de software está intimamente relacionado com a relação entre os benefícios providos e o esforço adicional necessário para atingir esses benefícios. O caso da GCS aplicada ao DBC não é diferente. O DBC por si só é composto por atividades de alta complexidade, e o aumento da burocracia na execução dessas atividades pode inviabilizar a adoção da GCS.

Por esta razão, uma atenção especial deve ser despendida na integração entre o ferramental e os processos de GCS com o ferramental e os processos existentes no DBC. Para que essa interação seja a mais suave possível, as funcionalidades de GCS devem ficar encapsuladas na metáfora atual de DBC, que engloba componentes, interfaces e conectores, conforme proposto pelo Odyssey-BRCS.

4.6.1 Abordagem proposta

A integração entre GCS e DBC pode ser dividida em dois níveis de complexidade: (1) integração simples e (2) integração avançada. A integração simples abrange os aspectos referentes à integração entre as próprias ferramentas de GCS, entre as ferramentas de GCS e de DBC, o acesso aos recursos de GCS na interface de usuário dos ambientes de DBC e o mecanismo de carga de componentes dentro do ambiente de DBC. Esses aspectos de integração são usualmente tratados dentro do próprio escopo dos demais sistemas propostos para o Odyssey-SCM.

A integração avançada busca a propagação de conhecimento entre os ambientes de GCS e de DBC. A função de acompanhamento da configuração da GCS é responsável por adquirir e armazenar informações durante a execução do desenvolvimento e manutenção de software. Essas informações, existentes no contexto dos sistemas de controle de modificações e de controle de versões, podem ser utilizadas para apoiar a execução das atividades do DBC.

Sempre que uma requisição de modificação é aprovada, e a implementação é iniciada, modelos de análise e projeto podem ser alterados para atender às novas necessidades. Quando elementos de modelo estão sendo alterados, é importante detectar quais outros elementos podem também necessitar de alterações. Para responder a esse requisito, é necessário encontrar rastros entre os elementos que indiquem dependências de manutenção, e que possam ser utilizados para indicar os elementos que precisam ser modificados em conjunto.

O uso de técnicas de mineração de dados pode apoiar na detecção dessas dependências de manutenção, encontrando regras do tipo: “desenvolvedores que modificam esses elementos também modificam esses outros elementos”. Esse tipo de conhecimento pode ser útil para apoiar diversas atividades do DBC e da GCS, como, por exemplo: sugerir modificações futuras, prevenir erros oriundos de modificações incompletas, detectar efeitos colaterais de modificações, apoiar na análise de impacto de modificações e detectar falhas de projeto devido a alto acoplamento entre artefatos não correlatos.

Desta forma, o Odyssey-WI trata questões na fronteira entre os sistemas de GCS e os ambientes de DBC. A solução proposta consiste em interferir o mínimo necessário no trabalho de DBC. Dentro do possível, as questões referentes a GCS deverão ser tratadas de forma automática, ou ao menos encapsuladas dentro de uma metáfora de DBC, fazendo uso dos mesmos termos e ferramentas que o desenvolvedor está habituado. Mais ainda, a infra-estrutura de GCS deverá fornecer novas funcionalidades para o ambiente de DBC, como, por exemplo, a detecção automática de dependências entre artefatos através da mineração de dados no repositório de versões, para compensar uma possível perda de produtividade relacionada com a mudança na forma usual de trabalho.

4.6.1.1 Rastros de modificações

Dentre os rastros de pós-especificação, ou pós-rastros (GOTEL et al., 1994), podem ser detectados ao menos dois tipos diferentes de rastros (KOWALCZYKIEWICZ et al., 2002): (1) rastros intermodelos, que são rastos entre os diversos níveis de abstração de um mesmo elemento conceitual e (2) rastros intramodelo, que são rastros entre diferentes elementos conceituais em um mesmo nível de abstração. Independentemente do tipo do pós-rastro, a sua detecção é considerada um problema em aberto na engenharia de software, devido à complexidade envolvida no descobrimento dos reais motivos para o surgimento desses rastros.

Com o intuito de prover algum apoio na detecção desses rastros, é proposta uma abordagem baseada em GCS. Sempre que uma modificação é implementada, é possível correlacionar as informações existentes no sistema de controle de modificações com os artefatos modificados no sistema de controle de versões. Essa correlação pode apoiar no surgimento gradativo tanto de rastros intermodelos quanto de rastros intramodelo, como exibido na Figura 17.

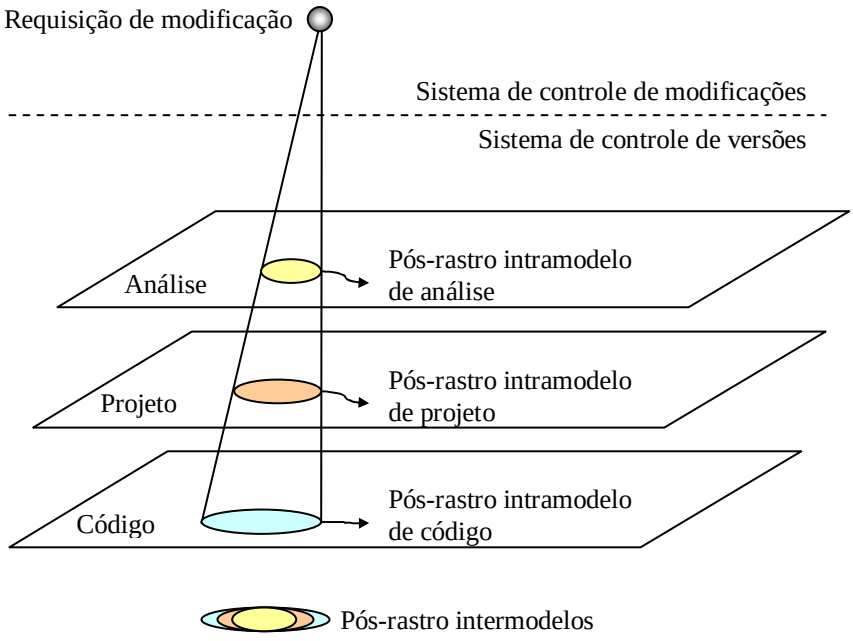

Figura 17: Rastros de modificação

Para cada requisição de modificação, é possível detectar informações referentes a **quem** implementou a modificação, **quando** a modificação foi implementada, **como** foi implementada a modificação, **onde** foi necessário alterar para implementar a modificação, **o quê** foi realmente feito e **por quê** a modificação é necessária. Essas informações são fundamentais para apoiar na construção dos rastros entre os artefatos.

As atividades do processo de controle de modificações, juntamente com informações existentes no sistema de controle de versões, apóiam na coleta dessas informações da seguinte forma:

- **Quem?** Obtido nos registros de *check-in* do sistema de controle de versões e na atividade de implementação do sistema de controle de modificações;
- **Quando?** Obtido nos registros de *check-in* do sistema de controle de versões e na atividade de implementação do sistema de controle de modificações;
- **Como?** Obtido no laudo de análise de impacto gerado a partir da atividade de análise, no sistema de controle de modificações;
- **Onde?** Obtido nos registros de *check-in* no sistema de controle de versões, que relatam quais partes de quais artefatos foram modificadas;
- **O quê?** Obtido no laudo de verificação gerado a partir da atividade de verificação da modificação, que relata o que realmente foi feito, e é necessário para aprovar ou não a integração da modificação na configuração de referência; e
- **Por quê?** Obtido no documento de requisição da modificação, que informa a razão da modificação e serve como base para o prosseguimento do processo de controle de modificação.

4.6.2 Detalhamento

Dentre os aspectos de integração simples, o mecanismo de carga de componentes, que já está operacional no ambiente Odyssey, faz uso da técnica de grafos de dependência (LARSSON, 2000) para possibilitar a detecção recursiva de quais componentes devem ser instalados ou removidos sempre que uma nova funcionalidade for necessária no ambiente, como exibido na Figura 18.

A partir da lista de componentes necessários, o sistema de integração faz a transferência física dos componentes e carrega dinamicamente esses componentes no ambiente. Para possibilitar a carga dinâmica dos componentes, são utilizados mecanismos de reflexão existentes na linguagem Java e dispositivos especiais para a carga das classes (*Class Loader*).

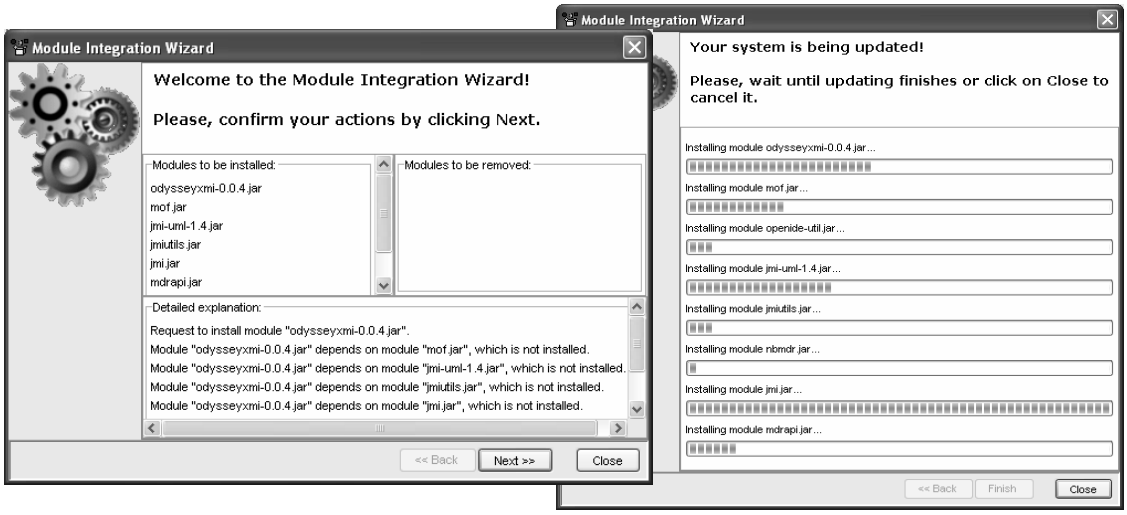

Figura 18: Carga dinâmica de componentes no ambiente Odyssey

Dentre os aspectos de integração avançada, o Odyssey-WI se propõe a aplicar técnicas de mineração de dados sobre o repositório MOF de versões Odyssey-VCS e sobre o repositório de requisições de modificações Odyssey-CCS para gerar pós-rastros intramodelo e intermodelos entre elementos UML. Esses rastros serão utilizados para detectar conjuntos de modificações e guiar o desenvolvedor na implementação desses conjuntos de modificações, pois sempre que uma modificação é implementada, a própria implementação da modificação pode motivar o surgimento de novas modificações, como exibido na Figura 19.

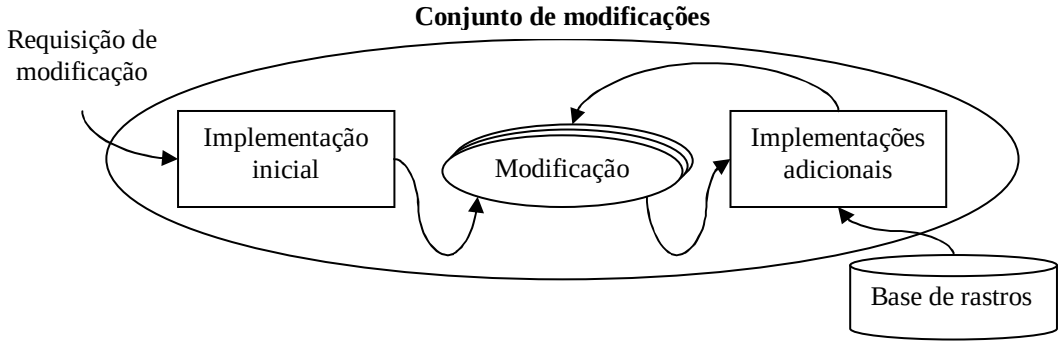

Figura 19: Detecção de conjuntos de modificações

4.6.3 Andamento

Inicialmente, foi definido e implementado um mecanismo de carga dinâmica para o ambiente Odyssey, para possibilitar que a versão atual do ambiente, com suporte ao DBC, possa ser estendida para receber funcionalidade de GCS de forma dinâmica e incremental. Esse mecanismo, que já está operacional, serve como alicerce para que seja possível definir questões específicas de integração para cada sistema do Odyssey-SCM.

Atualmente, está sendo definida, por uma aluna de mestrado, a técnica de mineração de dados apropriada para o problema em questão. Após essa definição, será implementado um protótipo que faça uso dessas técnicas para a obtenção de rastros de modificação e utilização desses rastros no apoio ao DBC.

4.7 Conclusão

As contribuições previstas do Odyssey-SCM podem ser agrupadas em função do seu campo principal de atuação. As principais contribuições, referentes ao Odyssey-SCMP, são:

- Customização do processo de GCS para o DBC;
- Colaboração entre participantes do processo multi-nível de produção, consumo e utilização de componentes;
- Propagação de partes de requisições em função da responsabilidade contratual de manutenção;
- Embasamento nas normas ISO 10007, IEEE Std 282, IEEE Std 1042, CMM, CMMI e KobrA; e
- Implantação do processo no sistema Odyssey-CCS;

As principais contribuições, referentes ao Odyssey-CCS, são:

- Modelagem gráfica, simulação e execução do processo de GCS;
- Configuração da coleta de informações necessárias para as atividades do processo;
- Adaptação do processo sem prejudicar as instâncias em execução;
- Metáfora centrada na modificação, permitindo o acesso a todos os documentos produzidos durante o ciclo de vida da modificação; e
- Coleta de métricas para a análise da execução do processo via construção de agentes inteligentes.

As principais contribuições, referentes ao Odyssey-VCS, são:

- Modelo de dados padrão OMG (MOF);
- Políticas configuráveis para grão de comparação e versionamento;
- Política genérica predefinida no nível do meta-meta-modelo MOF; e
- Metáfora homogênea de GCS para os diversos elementos de modelo.

As principais contribuições, referentes ao Odyssey-BRCS, são:

- Mapeamento entre os conceitos de DBC e GCS, permitindo a manipulação dos diversos artefatos que compõem um componente via o próprio componente, através de acessos distribuídos a repositórios heterogêneos de versão;
- Uso de ADL como modelo de construção de componentes;
- Integração contínua de componentes; e
- Emprego de contratos de interfaces na automatização da evolução de componentes.

As principais contribuições, referentes ao Odyssey-WI, são:

- Integração dos sistemas de GCS no ambiente de DBC Odyssey via mecanismo de carga dinâmica de componentes;
- Utilização de mineração de dados para propagação de conhecimentos de GCS para o ambiente de DBC;
- Construção de pós-rastros intramodelo e intermodelos para apoiar a detecção de modificações incompletas, análises de impacto, e reengenharia dos componentes; e
- Emprego de conjuntos de modificações para propagação de modificações entre os ambientes produtores e consumidores de componentes;

Apesar de oferecer contribuições pontuais a subsistemas específicos de GCS, a coleção dessas contribuições pontuais confere ao Odyssey-SCM uma abordagem ampla de GCS no contexto de DBC, com características inovadoras tanto individualmente, para cada um dos cinco subsistemas, quanto como solução integrada para o problema de GCS no DBC como um todo. Além disso, com esse suporte mínimo, necessário aos cinco tópicos discutidos, será possível investir em uma contribuição mais profunda no tópico de controle de construções e liberações, referente ao Odyssey-BRCS, que tem se mostrado essencial na automação das atividades susceptíveis a erro do DBC.

Capítulo 5 - Conclusão

5.1 Resultados esperados

Como apresentado no Capítulo 3, o suporte de GCS no DBC ainda deixa a desejar em relação a diversas funcionalidades necessárias. Este trabalho visa aumentar o apoio existente na área, fornecendo soluções com aspectos inovadores nos cinco objetivos definidos no Capítulo 1, que são:

- Fornecer um processo de GCS, customizado para os problemas do DBC, levando em consideração as principais normas de GCS e as características específicas do DBC;
- Possibilitar a modelagem, execução e adaptação do processo de GCS proposto, sem perda de informações. Além disso, permitir a configuração das informações necessária a cada atividade do processo de GCS;
- Fornecer suporte à configuração de políticas que determinem os grãos de comparação e de versionamento para o controle de versões de modelos;
- Mapear a metáfora de GCS dentro dos elementos de DBC, permitindo a atuação do desenvolvedor no nível arquitetural e a propagação das ações para os sistemas de base, que controlam artefatos de níveis mais baixos de abstração;
- Permitir a propagação do conhecimento obtido pelos sistemas de GCS para o ambiente de DBC, aumentando a capacidade do desenvolvedor de perceber o relacionamento existente entre os diferentes artefatos de um mesmo ambiente de desenvolvimento e entre as diferentes instâncias de um mesmo artefato, manipuladas por equipes distintas.

Apesar da solução proposta contemplar os cinco objetivos definidos no Capítulo 1, uma maior ênfase será dada nos aspectos relacionados ao mapeamento entre as metáforas de GCS e de DBC, que fazem parte da abordagem proposta para o Odyssey-BRCS. Este tópico foi escolhido para um maior esforço de pesquisa, em detrimento aos demais, devido a sua importância na automação de atividades que atualmente são executadas manualmente, implicando na baixa produtividade das equipes de DBC e possíveis comprometimentos na qualidade final dos componentes.

No restante deste capítulo, serão apresentados: na seção 5.2, os resultados preliminares relacionados aos trabalhos em andamento; na seção 5.3, a metodologia a ser utilizada no decorrer das pesquisas; e, na seção 5.4, a definição das atividades e do cronograma para continuação dos trabalhos.

5.2 Resultados preliminares

Para alcançar os resultados esperados, diversas frentes de trabalho foram iniciadas em conjunto com dois alunos de mestrado. No decorrer desse trabalho, algumas decisões foram tomadas e resultados preliminares foram obtidos. Esses resultados preliminares possibilitaram que medidas corretivas fossem adotadas com o intuito de nortear o foco da abordagem. Dentre essas decisões e resultados, podemos citar, no que se refere ao Odyssey-SCMP:

- Uma versão inicial do processo de GCS customizado para o DBC, que serve como ponto de partida para o Odyssey-SCMP, foi apresentada no WDBC 2003 (DANTAS et al., 2003).

No que se refere ao Odyssey-CCS:

- O *framework* de documentação FrameDoc, que foi apresentado no ISKM/DM 2001 (MURTA et al., 2001), será utilizado como base na construção do Odyssey-CCS; e
- A máquina de processos Charon, que foi apresentada no IDEAS 2002 (MURTA et al., 2002b) e premiada em 2º lugar na sessão de ferramentas do SBES 2002 (MURTA et al., 2002a), também será utilizada como base na construção do Odyssey-CCS.

No que se refere ao Odyssey-VCS:

- Predecessor do Odyssey-VCS, o sistema de controle de versões LockED, foi apresentado no IDEAS 2001 (TEIXEIRA et al., 2001a) e premiado em 1º lugar na sessão de ferramentas do SBES 2001 (TEIXEIRA et al., 2001b); e
- Predecessor do LockED, o sistema de controle de versões Token, foi premiado em 1º lugar na sessão de ferramentas do SBES 2000 (MURTA et al., 2000).

No que se refere ao Odyssey-BRCS:

- Foi solicitada uma bolsa PDEE/CAPES de estágio de doutorado, com duração de 4 meses, para estágio junto à equipe do professor André van der Hoek, na Universidade da Califórnia, em Irvine. O objetivo desta interação é definir mecanismos de GCS no nível de abstração de arquiteturas de componentes.

No que se refere ao Odyssey-WI:

- Foi implementada uma ferramenta de carga dinâmica de componentes, que será submetida para a sessão de ferramentas do SBES 2004.

5.3 Metodologia

No momento, a especificação detalhada da solução está sendo elaborada, e será implementada, posteriormente, no ambiente Odyssey (WERNER et al., 2003). Parte desse trabalho será feita durante o estágio de doutorado, junto à equipe que desenvolveu a linguagem de descrição de arquiteturas que será utilizada como modelo de construção de sistemas. Subseqüentemente, ocorrerão estudos experimentais e análise dos resultados em paralelo à escrita da tese. Os estudos experimentais deverão servir como um instrumento para a avaliação de viabilidade da abordagem proposta juntamente com seus processos e sistemas, segundo a hipótese apresentada no Capítulo 1.

5.4 Cronograma

De acordo com a metodologia, as atividades pendentes até a defesa da tese englobam: a especificação e implementação da solução, os estudos experimentais e a redação da tese. Essas atividades podem ser expandidas nos seguintes itens:

1 - Especificação do processo;
2 - Especificação do controle de requisições;
3 - Especificação do controle de versões;
4 - Especificação do controle de construções e liberações;
5 - Especificação da integração;
6 - Implementação do processo na máquina de controle de requisições;
7 - Implementação do controle de requisições;
8 - Implementação do controle de versões;

9 - Implementação do controle de construções e liberações;

10 - Implementação da integração em ambientes de DBC;

11 - Redação de artigos;

12 - Planejamento dos estudos experimentais;

13 - Execução dos estudos experimentais;

14 - Análise dos estudos experimentais;

15 - Redação da tese; e

16 - Defesa da tese.

O cronograma previsto para essas atividades é exibido na Tabela 1.

Tabela 1: Cronograma previsto até a defesa da tese.

Atividades	2004												2005												2006		
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
1																											
2																											
3																											
4																											
5																											
6																											
7																											
8																											
9																											
10																											
11																											
12																											
13																											
14																											
15																											
16																											

Durante esse processo serão redigidos artigos, sempre que alguma contribuição bem definida for detectada. Esses artigos serão direcionados preferencialmente para os seguintes fóruns:

- Workshop de Desenvolvimento Baseado em Componentes (WDBC);
- Simpósio Brasileiro de Qualidade de Software (SBQS);
- Simpósio Brasileiro de Engenharia de Software (SBES);
- International Workshop on Software Configuration Management (SCM);
- International Conference on Software Maintenance (ICSM);

- International Conference on Software Reuse (ICSR);
- International Symposium on Component-based Software Engineering (CBSE);
- International Conference on Software Engineering (ICSE); e
- Revistas especializadas.

O período de julho a outubro de 2004, que está destacado na Tabela 1, representa a época em que o estágio de doutorado ocorrerá. As atividades desse período, expandidas para um maior detalhamento, constituem:

1 - Especificação do mapeamento entre os conceitos de GCS e DBC;
2 - Especificação da integração com Ménage;
3 - Especificação da integração com sistemas de controle de versões;
4 - Implementação da camada de tradução dos idiomas de GCS e DBC;
5 - Implementação da integração com Ménage;
6 - Execução de um exemplo usando a solução proposta;
7 - Redação de um artigo descrevendo a solução proposta;

O cronograma das atividades a serem executadas durante o estágio de doutorado é exibido na Tabela 2.

Tabela 2: Cronograma das atividades a serem executadas durante o estágio de doutorado.

Atividades	2004			
	7	8	9	10
1				
2				
3				
4				
5				
6				
7				

Referências Bibliográficas

ARANGO, G., 1988, *Domain Engineering for Software Reuse*, Ph.D., University of California at Irvine, Irvine, CA.

ARANGO, G., 1994, "Software Reusability - Domain analysis methods". *Ellis HorWood*.

ASKLUND, U., BENDIX, L., 2002, "A Study of Configuration Management in Open Source Software". In: *IEE Proceedings - Software*, v. 149, pp. 40-46, February.

ATKINSON, C., BAYER, J., BUNSE, C., et al., 2001, *Component-Based Product Line Engineering with UML*. 1st. ed., Addison-Wesley.

ATKINSON, C., BAYER, J., LAITENBERGER, O., et al., 2000, "Component-Based Software Engineering: The KobrA Approach". In: *3rd International Workshop on Component-based Software Engineering*, Limerick, Ireland, June.

BALL, T., KIM, J., PORTER, A. A., et al., 1997, "If your version control system could talk". In: *ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, Boston, MA, May.

BARNSON, M.P., STEENHAGEN J., 2003, *The Bugzilla Guide - 2.17.5 Development Release*. The Bugzilla Team.

BECK, K., 1999, *Extreme Programming Explained: Embrace Change*. 1 ed., Boston, MA, Addison-Wesley.

BLOIS, A. P. T. B., 2004, *Uma Proposta de Projeto Arquitetural Baseado em Componentes no Contexto de Engenharia de Domínio*, Exame de Qualificação, COPPE/UFRJ, Rio de Janeiro, RJ.

BOLINGER, D., BRONSON, T., 1995, *Applying RCS and SCCS*. 1st. ed., O'Reilly.

BORLAND, 2004a, "Borland InterBase Software Cross Platform Embedded Database". In: http://www.borland.com/interbase/, Accessed in 04/02/2004.

BORLAND, 2004b, "Java Development Solution for Java Applications: Borland JBuilder X". In: http://www.borland.com/jbuilder/, Accessed in 03/02/2004.

BRAGA, R. M. M., 2000, *Busca e Recuperação de Componentes em um Ambiente de Reuso*, Tese de Doutorado, COPPE, UFRJ, Rio de Janeiro, RJ.

BRIAND, L. C., LABICHE, Y., O'SULLIVAN, L., 2003, "Impact Analysis and Change Management of UML Models". In: *International Conference on Software Maintenance (ICSM) 2003*, pp. 256-265, Amsterdam, Netherlands, September.

BROWN, A. W., 2000, *Large Scale Component Based Development*. 1 ed., Prentice Hall PTR.

CEDERQVIST, P., 2003, *Version Management with CVS*. Free Software Foundation.

CHEESMAN, J., DANIELS, J., 2000, *UML Components: A Simple Process for Specifying Component-Based Software*. 1st. ed., Addison-Wesley.

CHEN, P., CRITCHLOW, M., GARG, A., et al., 2003, "Differencing and Merging within an Evolving Product Line Architecture". In: *Fifth International Workshop on Product Family Engineering*, Siena, Italy, November.

CHIEN, S., TSOTRAS, V. J., ZANIOLO, C., 2001, "XML Document Versioning", *ACM SIGMOD Record*, v. 30, n. 3, pp. 46-53.

CHRISSIS, M. B., KONRAD, M., SHRUM, S., 2003, *CMMI: Guidelines for Process Integration and Product Improvement*, Boston, MA, Addison-Wesley.

CHRISTENSEN, H. B., 1999a, "The Ragnarok Architectural Software Configuration Management Model". In: *32nd Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January.

CHRISTENSEN, H. B., 1999b, "The Ragnarok Software Development Environment", *Nordic Journal of Computing*, v. 6, n. 1, pp. 4-21.

CHRISTENSEN, M. J., THAYER, R. H., 2002, *The Project Manager's Guide to Software Engineering's Best Practices*. 1 ed., IEEE Computer Society Press and John Wiley & Sons.

CLEMENTS, P., NORTHROP, L. M., 2001, *Software Product Lines: Practices and Patterns*. 1 ed., Boston, MA, Addison-Wesley.

CMTODAY, 2004a, "Commercial Issue Tracking / Workflow Systems". In: http://www.cmtoday.com/yp/tracking.html, Accessed in 08/01/2004.

CMTODAY, 2004b, "Public Domain or Free Configuration Management Systems". In: http://www.cmtoday.com/yp/free.html, Accessed in 08/01/2004.

COBENA, G., ABITEBOUL, S., MARIAN, A., 2002, "Detecting Changes in XML Documents". In: *18th International Conference on Data Engineering*, pp. 41-52, San Jose, CA, February.

COLLINS-SUSSMAN B., FITZPATRICK B.W., PILATO C.M., 2004, *Version Control with Subversion - Draft Revision 8263*. TBA.

D'SOUZA, D., WILLS, A., 1998, *Objects, components, and frameworks with UML: The catalysis approach*, Addison Wesley.

DANTAS, C. R., OLIVEIRA, H. L. R., MURTA, L. G. P., et al., 2003, "Um Estudo sobre Gerência de Configuração de Software aplicada ao Desenvolvimento Baseado em Componentes". In: *Terceiro Workshop de Desenvolvimento Baseado em Componentes*, São Carlos, SP, September.

DASHOFY, E., HOEK, A., TAYLOR, R. N., 2001, "A Highly-Extensible, XML-Based Architecture Description Language". In: *Working IEEE/IFIP Conference on*

Software Architectures (WICSA 2001), pp. 103-112, Amsterdam, Netherlands, August.

DASHOFY, E., HOEK, A., TAYLOR, R. N., 2002, "An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages". In: *International Conference on Software Engineering (ICSE 2002)*, pp. 266-276, Orlando, Fl, May.

DIRCKZE, R., 2002, *Java™ Metadata Interface (JMI) Specification - Version 1.0*. Unisys Corporation and Sun Microsystems.

DOD, 1962, *AFSCM 375-1 - CM During the Development & Acquisition Phases*.

DOD, 1971, *MIL Std 483 - CM Practices for Systems, Equipment, Munitions, & Computer Programs*.

DOD, 1985, *DOD Std 2167A - Defense System Software Development*.

DRAHEIM, D., PEKACKI, L., 2003, *Analytical Processing of Version Control Data: Towards a Process-Centric Viewpoint*. Free University Berlin.

ECLIPSE FOUNDATION, 2004, "Eclipse.org Main Page". In: http://www.eclipse.org, Accessed in 03/02/2004.

EL-JAICK, D., 2004, *MIMIX: Sistema de Apoio à Modelagem Cooperativa de Software Utilizando Ferramentas CASE Heterogêneas*, Dissertação de mestrado (em fase final de elaboração), COPPE, UFRJ, Rio de Janeiro, RJ.

ESTRADA, A. F., 2003, *Un modelo de referencia para la Gestión de Configuración en la PYME de Software*, D.Sc., Instituto Superior Politécnico "José Antonio Echeverria", Havana, Cuba.

ESTUBLIER, J., 2000, "Software Configuration Management: a Roadmap". In: *Proceedings of 22nd International Conference on Software Engineering, The Future of Software Engineering*, pp. 279-289, Limerick, Ireland, June.

ESTUBLIER, J., 2001, "Objects Control for Software Configuration Management". In: *CAiSE 2001*, pp. 359-373, Interlaken, Suiça, June.

ESTUBLIER, J., LEBLANG, D., CLEMM G., et al., 2002, "Impact of the Research Community On the Field of Software Configuration Management". In: *Software Engineering Notes (ACM)*, v. 27, pp. 31-39, Orlando, Florida, September.

FELDMAN, S. I., 1979, "Make-A Program for Maintaining Computer Programs", *Software - Practice and Experience*, v. 9, n. 4, pp. 255-265.

FIGUEIREDO, S. M., 2004, *Gerência de Configuração em Ambientes de Desenvolvimento de Software Orientados à Organização*, B.Sc., DCC - IM / UFRJ, Rio de Janeiro, RJ.

FOGEL, K., BAR, M., 2001, *Open Source Development with CVS*. 2 ed., Scottsdale, Arizona, The Coriolis Group.

FOWLER, M., FOEMMEL, M., 2004, "Continuous Integration". In: http://martinfowler.com/articles/continuousIntegration.html, Accessed in 21/01/2004.

FSF, 2002, *Comparing and Merging Files*. Free Software Foundation.

GAMMA, E., HELM, R., JOHNSON, R., et al., 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*. primeira ed., Addison Wesley.

GARG, A., CRITCHLOW, M., CHEN, P., et al., 2003, "An Environment for Managing Evolving Product Line Architectures". In: *International Conference on Software Maintenance*, pp. 358-367, Amsterdam, Netherlands, September.

GOLDBERG, E., 2002, *Bug Writing Guidelines*. The Mozilla Organization.

GOTEL, O., FINKELSTEIN, A., 1994, "An Analysis of the Requirements Traceability Problem". In: *International Conference on Requirements Engineering*, pp. 94-101, Colorado, USA.

HASS, A. M. J., 2003, *Configuration Management Principles and Practices*, Boston, MA, Pearson Education, Inc.

HATCHER, E., 2001, *Automating the build and test process: Ant and JUnit bring you one step closer to XP nirvana*. IBM.

HATCHER, E., LOUGHRAN, S., 2004, *Java Development with Ant*, Greenwich, CT, Manning Publications Company.

HEINEMAN, G. T., COUNCILL, W. T., 2001, *Component Based Software Engineering: Putting the Pieces Together*. 1 ed., Addison-Wesley Pub Co.

HERZUM, P., SIMS, O., 1999, *Business Component Factory : A Comprehensive Overview of Component-Based Development for the Enterprise*. 1 ed., John Wiley & Sons.

HNETYNKA, P., PLASIL, F., 2004, "Distributed Versioning Model for MOF". In: *WISICT*, pp. 489-494, Cancun, México, January.

HOEK, A., 2004, "Design-Time Product Line Architectures for Any-Time Variability", *Science of Computer Programming*, n. special issue on Software Variability Management.

HOEK, A., CARZANIGA, A., HEIMBIGNER, D., et al., 2002, "A Testbed for Configuration Management Policy Programming", v. 28, n. 1, pp. 79-99.

IBM RATIONAL, 2004, "Rational Rose XDE Modeler - Product Overview - IBM Software". In: http://www-306.ibm.com/software/awdtools/developer/modeler/, Accessed in 05/02/2004.

IEEE, 1987, *Std 1042 - IEEE Guide to Software Configuration Management*.

IEEE, 1990, *Std 610.12 - IEEE Standard Glossary of Software Engineering Terminology*.

IEEE, 1998, *Std 828 - IEEE Standard for Software Configuration Management Plans*.

ISO, 1995a, *ISO 10007, Quality Management - Guidelines for Configuration Management*.

ISO, 1995b, *ISO/IEC 12207 - Information technology - Software life cycle processes*.

JALOTE, P., 1999, *CMM in Practice: Processes for Executing Software Projects at Infosys*. 1 ed., Boston, MA, Addison-Wesley.

KNETHEN, A., GRUND, M., 2003, "QuaTrace: A Tool Environment for (Semi-) Automated Impact Analysis Based on Traces". In: *International Conference on Software Maintenance (ICSM) 2003*, pp. 246-255, Amsterdam, Netherlands, September.

KOWALCZYKIEWICZ, K., WEISS, D., 2002, "Traceability: Taming uncontrolled change in software development". In: *IV National Software Engineering Conference*, Tarnowo Podgorne, Poland.

KRUCHTEN P., 2001, *The Rational Unified Process: An Introduction*. 2nd. ed., Addison-Wesley.

KWON, O., SHIN, G., BOLDYREFF, C., et al., 1999, "Maintenance with Reuse: An Integrated Approach Based on Software Configuration Management". In: *Sixth Asia Pacific Software Engineering Conference*, pp. 507-515, Takamatsu, Japan, December.

LARSSON, M., 2000, *Applying Configuration Management Techniques to Component-Based Systems*, Licentiate Thesis, Deparment of Information Technology, Uppsala University, Sweden.

LARSSON, M., CRNKOVIC, I., 2000, "Component Configuration Management". In: *5th Workshop on Component Oriented Programming*, Sophia Antipolis, France, June.

LEBSACK, C. S., MROCZEK, A. J., MUELLER, C. J., 2001, "Controlling Configuration Items in Component Based Software Development". In: *Tenth International Workshop on Software Configuration Management (SCM-10)*, Toronto, Canada, May.

LEHMAN, M. M., PERRY, D. E., RAMIL, J. F., et al., 1997, "Metrics and Laws of Software Evolution: The Nineties View". In: *4th International Symposium on Software Metrics (Metrics 97)*, pp. 20-32, Albuquerque, NM, November.

LEON, A., 2000, *A Guide to Software Configuration Management*, Norwood, MA, Artech House Publishers.

LINGEN, R., HOEK, A., 2004, "An Experimental, Pluggable Infrastructure for Modular Configuration Management Policy Composition". In: *International Conference on Software Engineering (ICSE) 2004*, Edinburgh, Scotland, May.

MATULA, M., 2003, *NetBeans Metadata Repository*. NetBeans Community.

MEI, H., ZHANG, L., YANG, F., 2001, "A software configuration management model for supporting component-based software development", *ACM SIGSOFT Software Engineering Notes*, v. 26, n. 2, pp. 53-58.

MEYER, B., 1992, "Applying design by contract", *IEEE Computer*, v. 25, n. 10, pp. 40-51.

MICROSOFT, 2004a, "Microsoft COM Technologies - Information and Resources for the Component Object Model-based technologies". In: http://www.microsoft.com/com/, Accessed in 17/02/2004.

MICROSOFT, 2004b, "Microsoft Office". In: http://office.microsoft.com, Accessed in 05/02/2004.

MICROSOFT, 2004c, "Windows XP Home Page". In: http://www.microsoft.com/windowsxp, Accessed in 08/02/2004c.

MILER, N., 2000, *A Engenharia de Aplicações no Contexto da Reutilização baseada em Modelos de Domínio*, M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil.

MORO, M. M., SAGGIORATO, S. M., EDELWEISS, N., et al., 2001, "A Temporal Versions Model for Time-Evolving Systems Specification". In: *SEKE - 13th International Conference on Software Engineering and Knowledge Engineering*, pp. 252-259, Buenos Aires, Argentina, June.

MORO, M. M., ZAUPA, A. P., EDELWEISS, N., et al., 2002, "TVQL - Temporal Versioned Query Language". In: *DEXA - 13th International Conference on Database and Expert Systems Applications*, pp. 618-627, Aix en Provence, France, September.

MOZILLA, 2004a, "bonsai". In: http://www.mozilla.org/bonsai.html, Accessed in 03/02/2004.

MOZILLA, 2004b, "tinderbox". In: http://www.mozilla.org/tinderbox.html, Accessed in 03/02/2004.

MURTA, L. G. P., 1999, *FrameDoc: Um Framework para a Documentação de Componentes Reutilizáveis*, Projeto Final de Curso, DCC/IM, UFRJ, Rio de Janeiro, Brasil.

MURTA, L. G. P., 2002, *Charon: Uma Máquina de Processos Extensível Baseada em Agentes Inteligentes*, M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

MURTA, L. G. P., BARROS, M., WERNER, C. M. L., 2000, "Token: Uma Ferramenta para o Controle de Alterações em Projetos de Software em Desenvolvimento".

XIV Simpósio Brasileiro de Engenharia de Software, Sessão de Ferramentas, João Pessoa.

MURTA, L. G. P., BARROS, M., WERNER, C. M. L., 2002a, "Charon: Uma Ferramenta para a Modelagem, Simulação, Execução e Acompanhamento de Processos de Software". In: *XVI Simpósio Brasileiro de Engenharia de Software*, pp. 366-371, Gramado, RS, Brazil, October.

MURTA, L. G. P., BARROS, M., WERNER, C. M. L., 2002b, "Charon: Uma máquina de processos extensível baseada em agentes inteligentes". In: *V Workshop Ibero-americano de Engenharia de Requisitos e Ambientes de Software (IDEAS'02)*, Havana, Cuba, April.

MURTA, L. G. P., BARROS, M. O., WERNER, C. M. L., 2001, "FrameDoc: Um Framework para a Documentação de Componentes Reutilizáveis". In: *IV International Symposium on Knowledge Management/Document Management (ISKM/DM'2001)*, pp. 241-259, Curitiba, Brasil.

MURTA, L. G. P., DANTAS, C. R., OLIVEIRA, H. L. R., et al., 2004, "Odyssey-SCM". In: http://www.cos.ufrj.br/~odyssey/scm, Accessed in 02/03/2004.

MYERS, E. W., 1986, "An O(ND) Difference Algorithm and its Variations", *Algorithmica*, v. 1, n. 2, pp. 251-266.

NEIGHBORS, J., 1980, *Software Construction Using Components*, Department of Information and Computer Science, University of California.

NETBEANS COMMUNITY, 2004, "Welcome to Netbeans". In: http://www.netbeans.org, Accessed in 03/02/2004.

NETSCAPE, 2004, "Netscape 7.1". In: http://channels.netscape.com/ns/browsers/download.jsp, Accessed in 08/02/2004.

ODYSSEY, 2004, "Projeto Odyssey". In: http://www.cos.ufrj.br/~odyssey, Accessed in 02/03/2004.

OLIVEIRA, A. A. A. C. P., PRIMO, F. F., CRUZ, J. L., et al., 2001, *Gerência de Configuração de Software - Evolução de Software sob Controle*. Instituto Nacional de Tecnologia da Informação (ITI), Ministério da Ciência e Tecnologia.

OMG, 2002a, *Meta Object Facility (MOF) Specification, version 1.4*. Object Management Group.

OMG, 2002b, *UML 2.0 Superstructure: Final Adopted Specification*. Object Management Group.

OMG, 2003a, *Common Warehouse Metamodel (CWM) Specification, version 1.1*. Object Management Group.

OMG, 2003b, *Unified Modeling Language (UML) Specification, version 1.5*. Object Management Group.

OMG, 2003c, *XML Metadata Interchange (XMI) Specification, Version 2.0*. Object Management Group.

PAGE-JONES, M., 1999, *Fundamentals of Object-Oriented Design in UML*. 1st. ed., Addison-Wesley.

PRESSMAN, R. S., 1997, *Software Engineering: A Practitioner's Approach*. 4 ed., McGraw-Hill.

PRIETO-DIAZ, R., 1990, "Domain Analysis: An Introduction", *Software Engineering Notes*, v. 15, n. 2, pp. 47-54.

ROCHA, A. R. C., AGUIAR, T. C., SOUZA, J. M., 1990, "TABA: A Heuristic Workstation for Software Development". In: *COMPEURO 90*, Tel Aviv, Israel, May.

ROCHE, T., WHIPPLE, L. C., 2001, *Essential SourceSafe*, Hentzenwerke Corporation.

ROCHKIND, M. J., 1975, "The Source Code Control System", *IEEE Transactions on Software Engineering*, v. 1, n. 4, pp. 364-370.

SCHNEIDER, R. L., 2001, *Um Sistema de Gerência Cooperativa de Configuração de Software*, M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ.

SCOTT, J. A., NISSE, D., 2001, "Software Configuration Management", *Guide to Software Engineering Body of Knowledge*, chapter 7, IEEE Computer Society Press.

SEI, 2002a, *Capability Maturity Model Integration (CMMI) Version 1.1 - Continuous Representation*. Carnegie Mellon University.

SEI, 2002b, *Capability Maturity Model Integration (CMMI) Version 1.1 - Staged Representation*. Carnegie Mellon University.

SHAW, M., GARLAN, D., 1996, *Software Architecture: Perspectives on an Emerging Discipline*. 1 ed., Prentice Hall.

SILVA, F. A., COSTA, R. V. C., EDELWEISS, N., et al., 2003, "Using the Temporal Versions Model in a Software Configuration Management Environment". In: *Simpósio Brasileiro de Engenharia de Software*, Manaus, Amazonas, October.

SMDS, 1994, *Aide de Camp Product Overview*. Software Maintenance & Development Systems.

SUN, 2004a, "Enterprise JavaBeans Technology". In: http://java.sun.com/products/ejb, Accessed in 17/02/2004.

SUN, 2004b, "Javadoc Tool Home Page". In: http://java.sun.com/j2se/javadoc, Accessed in 17/02/2004.

SVAHNBERG, M., GURP, J., BOSCH, J., 2002, *A Taxonomy of Variability Realization Techniques*. Blekinge Institute of Technology, Sweden.

SZYPERSKI, C., 2002, *Component Software: Beyond object-oriented programming*. 2nd. ed., Addison-Wesley.

TECHTARGET, 2004, "WhatIs.com". In: http://www.whatis.com, Accessed in 06/02/2004.

TEIXEIRA, H. V., 2003, *Geração de Componentes de Negócio a Partir de Modelos de Análise*, M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ.

TEIXEIRA, H. V., MURTA, L. G. P., WERNER, C. M. L., 2001a, "LockED: Uma Abordagem para o Controle de Alterações de Artefatos de Software". In: *IV Workshop Ibero-americano de Engenharia de Requisitos e Ambientes de Software (IDEAS'01)*, pp. 348-359, San José, Costa Rica, April.

TEIXEIRA, H. V., MURTA, L. G. P., WERNER, C. M. L., 2001b, "LockED: Uma Ferramenta para o Controle de Alterações no Desenvolvimento Distribuído de Artefatos de Software". In: *XV Simpósio Brasileiro de Engenharia de Software, Caderno de Ferramentas*, pp. 380-385, Rio de Janeiro, RJ, October.

TICHY, W., 1985, "RCS: a system for version control", *Software - Practice and Experience*, v. 15, n. 7, pp. 637-654.

TIGRIS, 2004, "Scarab". In: http://scarab.tigris.org/, Accessed in 08/01/2004.

TORVALDS, L., 2004, "The Linux Home Page at Linux Online". In: http://www.linux.org/, Accessed in 04/02/2004.

TRAVASSOS, G. H., 1994, *O Modelo de Integração de Ferramentas da Estação TABA*, D.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil.

VENUGOPALAN, V., 2002, *CVS Best Practices - Revision 0.6*. Free Software Foundation.

W3C, 1999, *HTML 4.01 Specification*. World Wide Web Consortium.

W3C, 2001a, *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium.

W3C, 2001b, *XML Schema 1.0*. World Wide Web Consortium.

W3C, 2004, *Extensible Markup Language (XML) 1.0 (Third Edition)*. World Wide Web Consortium.

WALLNAU, K., HISSAM, S., SEACORD, R., 2001, *Building Systems from Commercial Components*. 1 ed., Addison-Wesley Pub Co.

WANG, Y., DE WITT, D. J., CAI, J., 2003, "X-Diff: An Effective Change Detection Algorithm for XML Documents". In: *19th International Conference on Data Engineering*, pp. 519-530, Bangalore, India, March.

WERNER, C. M. L., MANGAN, M. A. S., MURTA, L. G. P., et al., 2003, "OdysseyShare: an Environment for Collaborative Component-Based Development". In: *IEEE Conference on Information Reuse and Integration (IRI)*, Las Vegas, Nevada, October.

WESTHUIZEN, C., HOEK, A., 2002, "Understanding and Propagating Architectural Changes". In: *Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA 3)*, pp. 95-109, Montreal, Canada, August.

WHITE, B. A., 2000, *Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction*, Addison-Wesley.

XDOCLET TEAM, 2004, "XDoclet: Attribute-Oriented Programming - Welcome". In: http://xdoclet.sourceforge.net, Accessed in 17/02/2004.

ZHANG, L., MEI, H., ZHU, H., 2001, "A Configuration Management System Supporting Component-Based Software Development". In: *25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, pp. 25-30, Chicago, Illinois, October.