Uma Abordagem para a Manutenção da Integridade de Repositórios de Gerência de Configuração

Gleiph Ghiotto Lima de Menezes, Leonardo Gresta Paulino Murta

Instituto de Computação – Universidade Federal Fluminense (UFF)
São Domingos Niterói - RJ – Brasil - CEP: 24210-240

{gmenezes, leomurta}@ic.uff.br

Resumo. *A evolução de software é um processo que deve ser acompanhado com atenção, objetivando sempre ter como resultado um software que atenda às necessidades do usuário. A Gerência de Configuração (GC) é a disciplina tradicionalmente responsável pelo controle da evolução de software. Contudo, seu ciclo de trabalho, baseado em check-out, modificação e check-in, é usualmente apoiado por verificações manuais, que são, na maioria das vezes, contraproducentes e suscetíveis a erros. Desta forma, este trabalho propõe uma abordagem que amplifica o ciclo de trabalho tradicional de GC com tarefas assíncronas, automáticas e gradativas para verificação de conflitos de ordem sintática e semântica nos artefatos, visando contribuir com a manutenção da integridade de repositórios de GC.*

Abstract. *The software evolution process should be carefully monitored, always seeking for adherence to the user needs. Configuration Management (CM) is a discipline traditionally responsible for controlling the software evolution. However, its work cycle, based on checkout, modifications, and check-in, is usually supported by manual verifications, which are, in most cases, counterproductive and error prone. Thus, this paper proposes an approach that amplifies the traditional CM work cycle with asynchronous, automatic, and incremental tasks to check for syntactic and semantic conflicts in order to contribute for maintaining the integrity of CM repositories.*

Palavras-chave. *Gerência de Configuração de Software, Integração Contínua, Sistemas de Controle de Versão e Sistemas de Gerenciamento de Construção.*

1. Caracterização do Problema

A evolução de software é um processo que deve ser acompanhado com atenção. De modo geral, o software evolui para atender às necessidades do usuário. Contudo, caso essa evolução não seja feita de forma planejada e controlada, o software tende a degenerar com o passar do tempo, tornando as evoluções futuras mais complexas. Esse cenário pode afetar negativamente a qualidade do produto sendo gerado e a produtividade da equipe de desenvolvimento.

Neste contexto se destaca a Gerência de Configuração, uma disciplina responsável pelo controle da evolução de sistemas de s*oftware* [Dart, S. 1991]. Esta disciplina faz uso de Sistemas de Controle de versões (SCV) [Murta 2006] que são caracterizados por oferecerem recursos como registrar e armazenar o histórico de

desenvolvimento de artefatos (e.g., documentos, código fonte, modelos, entre outros), além de apoiar um ciclo de trabalho composto por: *check-out*, modificações e *check-in*.

Este ciclo deixa o repositório muito exposto, permitindo que conflitos possam chegar até a linha principal (do inglês, *mainline*) do repositório. Os SCV apresentam apoio à detecção de conflitos textuais ou físicos (i.e., edições em paralelo na mesma região de um artefato). No entanto, conflitos de ordem sintática (e.g., chamada de um método inexistente) e semântica (e.g, quebra de uma regra de negócio) não são identificados por estas ferramentas e podem corromper o repositório.

Deste modo, o problema atacado neste trabalho consiste na detecção de conflitos, sintáticos e/ou semânticos, antes que afetem a integridade do repositório e causem maiores danos, como o retrabalho e inconsistências.

2. Fundamentação Teórica e Trabalhos Relacionados

Segundo Fowler [2006], Integração Contínua (IC) é uma prática de desenvolvimento de software onde membros de uma equipe integram seus trabalhos constantemente. Essa integração é verificada por um processo de construção automática. Nesta construção são executadas tarefas como: compilação e execução de testes.

Um dos pontos que se destaca em IC é o rápido *feedback*. Por isso, uma das práticas de IC defende que o *check-in* deve ser realizado pelo menos uma vez ao dia. Esta política é definida, pois com uma frequência maior de *check-ins* é esperado que a chance de ocorrer conflitos diminua. Contudo, quando conflitos são detectados, devem ser solucionados imediatamente. Caso isto não ocorra, o repositório fica com artefatos inconsistentes até que os conflitos sejam solucionados.

Ferramentas que apoiam IC na detecção de conflitos no repositório são utilizadas com o objetivo de manter o desenvolvedor ciente de sua situação. Dentre elas se destacam o *Continuum*[1], o *CruiseControl*[2] e o *Hudson*[3]. Estas ferramentas ficam em um servidor de IC esperando que um evento (e.g., *check-in*) seja disparado. Quando este evento é disparado, um processo que inclui as tarefas de compilação e execução de testes é desencadeado e, desta maneira, uma verificação é feita sobre os artefatos presentes no repositório. Quando conflitos são detectados, uma notificação (e.g., email, SMS, entre outros) descrevendo o conflito é enviada para o responsável.

É importante notar que a maneira tradicional de tratar esses eventos no repositório é via a utilização de ganchos (do inglês, *hook*). Os ganchos são mecanismos de extensão dos SCV e viabilizam a execução de programas externos antes ou depois que eventos ocorram no repositório. Um exemplo destes eventos é criação de uma nova versão ou a modificação de uma propriedade não versionada que pode acionar um gancho de envio de e-mail.

IC realiza tarefas de verificação sintática (via compilação) e semântica (via testes) dos artefatos presentes no repositório, mas o faz tardiamente. Apesar de IC ter o propósito de melhorar a qualidade do software e reduzir riscos [Duvall 2007], existem algumas características negativas que podem comprometer todo o processo (e.g., permitir que artefatos que contenham conflitos cheguem à linha principal do

[1] http://continuum.apache.org/
[2] http://cruisecontrol.sourceforge.net/overview.html
[3] http://hudson-ci.org/

repositório). Isto ocorre, pois as verificações sintáticas e semânticas só ocorrem após o *check-in*.

O *Repoguard* [Tigris.org 2010] contorna esse problema, impedindo que artefatos contendo conflitos textuais e sintáticos entrem no repositório. Essa ferramenta se comporta da seguinte maneira: quando um *check-in* é realizado, ela é executada via um gancho, realizando o pré-processamento dos artefatos que estão sendo incluídos no repositório. Este pré-processamento é composto por verificação de conflitos textuais e sintáticos e acontece antes que os artefatos entrem no repositório (i.e., antes que o *check-in* seja concluído). Apesar desta abordagem verificar as contribuições feitas ao repositório, as verificações são realizadas no momento do *check-in* e este processo pode ser demorado. Logo, o desenvolvedor fica ocioso durante a maioria do processo o que pode desmotivar a sua adoção ou mesmo desestimular a execução de verificações mais complexas, como baterias de testes de regressão.

Outra abordagem relacionada é proposta por Wloka *et al.*[2009]. Esta abordagem visa fazer uma análise de quais modificações podem ser inseridas no repositório de acordo com uma política previamente selecionada pelo desenvolvedor. Para fazer essa análise, a ferramenta precisa do conjunto de testes que cobrem o código, a versão do código fonte sem alterações e a versão modificada. O algoritmo de análise executa testes nas mudanças atômicas e então decide se elas podem ou não sofrer *check-in*. Esta abordagem apresenta alguns aspectos positivos, como a compilação e execução de casos de teste antes de permitir a entrada de artefatos no repositório. Todavia, esta abordagem pode ser demorada e desgastante para o desenvolvedor, além de exigir o fornecimento de dados específicos durante o *check-in*.

Baseado na literatura consultada, não foi encontrada uma abordagem que faça uma verificação completa (i.e., verificação textual, sintática e semântica) dos artefatos sem demandar a presença constante do desenvolvedor. Vale notar também que é fundamental a utilização de políticas configuráveis, visto que, dependendo do processo adotado, pode ser razoável efetuar *check-ins* que não passam nos testes por determinados períodos do desenvolvimento.

3. Resultados Esperados

Como objetivo central, este trabalho apresenta a definição de uma abordagem que proteja o repositório de conflitos. A abordagem proposta consiste da inclusão de duas etapas de pré-processamento: (1) no momento do *check-out* - criação de uma camada que tem como objetivo isolar a linha principal de repositório; e (2) no momento do *check-in* - execução de tarefas de verificação de acordo com a política adotada.

Um cenário para aplicação desta abordagem pode ser o seguinte: Um desenvolvedor edita alguns artefatos e realiza *check-in* o que pode inserir conflitos que não são identificados pelo SCV. Em um primeiro caso, este desenvolvedor modificou o nome de um método apenas em um artefato, sem estender esta modificação para as chamadas a esse método nos demais artefatos e, consequentemente, inseriu conflitos sintáticos. Esta ação impede que outros desenvolvedores trabalhem nos artefatos até que esses conflitos sejam removidos. Outro conflito que pode ser inserido no repositório é o de ordem semântica. Neste caso o desenvolvedor modifica o comportamento de um método invalidando regras de negócio específicas. Este tipo de conflito é identificado via execução de testes e pode ser nocivo ao desenvolvimento, podendo gerar retrabalho

e deixar o repositório inconsistente. Vale ressaltar que o objetivo desta abordagem está ligada à proteção do repositório e não a manter outros desenvolvedores cientes das alterações que foram realizadas.

Como discutido anteriormente, a abordagem proposta faz uso de políticas para resguardar o repositório sem ferir o processo de desenvolvimento adotado. As políticas devem ser definidas utilizando filtros, que permitem isolamentos progressivos do repositório. Os filtros propostos são: textual, responsável por detectar conflitos em nível textual (e.g., a modificação simultânea da mesma linha por dois usuários distintos); sintático, que é responsável por identificar conflitos relacionados à sintaxe da linguagem utilizada; e semântico, que é capaz de identificar conflitos de ordem semântica, ou seja, partes do software que não estão de acordo com os requisitos. Desta forma, foram definidas três políticas para a utilização desta abordagem: (1) permissiva - nesta política são verificados apenas aspectos textuais; (2) moderada - nesta política serão aplicados os filtros: textual e sintático; e (3) Restritiva - nesta política todos os filtros são utilizados.

Conforme foi definido, cada política tem exigências para que um *check-in* seja aceito. Entretanto, em alguns casos é desejável saber o resultado de determinadas tarefas mesmo que esse resultado não seja utilizado como fator aceitação ou recusa de um *check-in*. Por isso, a abordagem proposta também permite que tarefas sejam executadas após o *check-in*, informado ao usuário sobre possíveis conflitos que possam estar presentes no repositório.

Desta forma, tarefas de verificação podem ser vista como restritivas ou informativas em função da política adotada. Assim, a adoção de uma destas políticas deixa a entrada no repositório tão flexível quanto necessário. Por exemplo, em um momento do ciclo de vida do projeto, o responsável pode optar por aceitar *check-ins* independentemente dos resultados dos testes, mas desejar saber o resultado. Nesse caso, a política moderada seria a mais indicada, com a opção de execução de testes após o *check-in*. Nesse cenário, a tarefa de testes pode ser vista como uma tarefa meramente informativa, mas posteriormente, com a troca da política, a tarefa pode passar a ser restritiva (rejeitando *check-ins* que não sejam aprovados nos testes).

Além das políticas que foram citadas, o usuário pode estabelecer políticas adicionais. Essas políticas se caracterizam pelas intervenções feitas no momento de *check-out* e *check-in*, estabelecendo as atividades que devem ser executadas em cada momento (integração, compilação ou testes) e o nível de controle (restritivo ou informativo). Uma visão geral destas intervenções pode ser notada na Figura 1.

Tendo em vista que o *check-in* e o *check-out* obedecem à propriedade (P1) que diz que "o *check-in* é realizado na mesma linha de desenvolvimento que sofreu *check-out*", é importante isolar a linha principal para que verificações sejam executadas e conflitos sejam detectados antes que sejam de fato incorporados no repositório.

Tradicionalmente, quando se deseja isolar uma linha de desenvolvimento, o conceito de ramos (do inglês, *branches*) é utilizado. Mas como não existe uma forma pré-estabelecida de criá-los, uma estratégia foi estabelecida na abordagem proposta. No fluxo proposto foi criada uma tarefa de pré-processamento do *check-out*, que consiste na criação de um ramo cada vez que este comando é executado, de acordo com a técnica denominada *autobranch* [Murta *et al.* 2008]. O resultado final é o *check-out* feito de um

ramo e não da linha principal, local que todos os usuários têm como referência. A figura 1 ilustra o momento em que a intervenção no *check-out* é realizada.

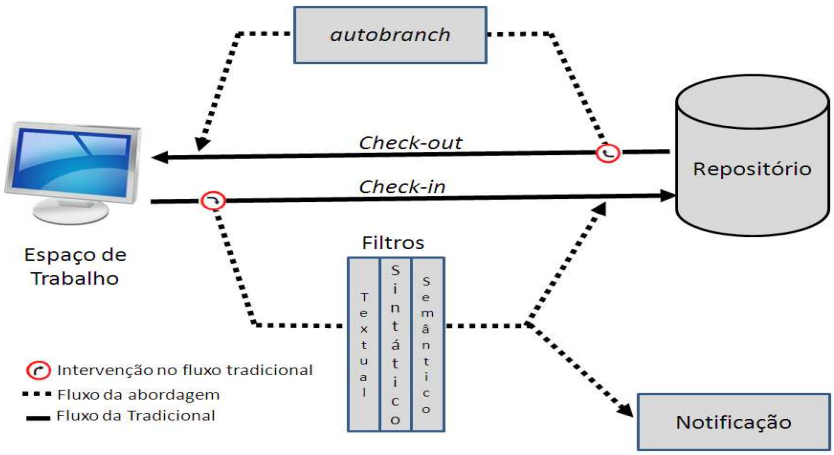

Figura 1 Comparação do Fluxo tradicional com o proposto na abordagem.

Realizado o *check-out*, o usuário faz modificações nos artefatos e realiza *check-in*, que deve enviar os artefatos livres de conflitos para a linha principal do repositório. Conforme foi descrito, os artefatos tiveram origem em um ramo, logo, pela propriedade P1, o *check-in* será feito neste ramo. Após este *check-in* ser concluído, uma tarefa de pós-processamento é disparada. Este pós-processamento é responsável pela execução dos filtros que a política escolhida define, além de integração (i.e., *merge*), execução dos filtros na configuração resultante da integração e envio ou não da configuração para o repositório, juntamente com uma notificação da situação do *check-in* ao usuário.

A verificação é realizada segundo a política adotada pelo usuário. No caso de uma política moderada, o fluxo é composto pelas seguintes tarefas: (1) verificação textual e sintática dos artefatos presentes no ramo; (2) integração do conteúdo do ramo com o da linha principal do repositório. Este passo é importante, pois outros *check-ins* podem ter ocorrido durante o tempo de desenvolvimento, logo novos conflitos podem ocorrer; e (3) verificação textual e sintática sobre a configuração resultante do passo 2. Se durante a execução das tarefas descritas for encontrado algum conflito, o fluxo é imediatamente interrompido e uma notificação é enviada ao responsável. Caso o fluxo seja executado com sucesso, o conteúdo obtido na etapa 2 é enviado para o repositório, e uma notificação é enviada ao usuário comunicando-lhe o sucesso. Para a política moderada, opcionalmente uma verificação semântica pode ser feita, mas somente a título informativo.

É importante notar que, apesar dessa descrição focar na linha principal de desenvolvimento, pode ser interessante a proteção de qualquer linha de desenvolvimento em diferentes níveis de controle [IEEE 2005]. Desta forma, diferentes políticas podem ser utilizadas para diferentes linhas de desenvolvimento. Além disso, é importante considerar que a abordagem é assíncrona sob a perspectiva do desenvolvedor, pois o *check-in* é efetuado por completo no *autobranch*, liberando o desenvolvedor para outras atividades e o informando posteriormente os resultados reais do *check-in*.

4. Metodologia e Estado atual do trabalho

Para definição da abordagem estão sendo realizadas pesquisas de ferramentas e abordagens relacionadas que contribuam para a definição de práticas que ofereçam maior segurança com relação à identificação de conflitos no repositório. Além disso, um protótipo está sendo desenvolvido em Java e apresenta apoio a projetos compatíveis com as ferramentas Subversion [Berlin 2006] e Maven [Sonatype 2008]. Vale ressaltar que as linguagens suportadas por esse protótipo estão condicionadas ao apoio à tarefa de compilação oferecido pelo Maven.

Este protótipo será alvo de uma avaliação em um caso de desenvolvimento real ainda a ser planejada e definida. Logo, trabalhos futuros são: continuação da implementação, definição do caso para avaliação e o planejamento propriamente dito da avaliação.

5. Agradecimentos

Agradecemos ao CNPQ e a FAPERJ pelo apoio financeiro.

6. Referências

Berlin, D., (2006), *Practical subversion*. 2 ed. Berkeley CA, Apress.

Dart, S., (1991), "Concepts in configuration management systems". , p. 1-18, Trondheim, Norway.

Duvall, P., (2007), *Continuous integration : improving software quality and reducing risk*. Upper Saddle River NJ, Addison-Wesley.

Fowler, M., (2006). Continuous Integration. Disponível em: <http://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 10 Abr 2010.

IEEEERRO. Std 828 - IEEE Standard for Software Configuration Management Plans.

Murta, L. G., (2006), *Gerência de Configuração no Desenvolvimento Baseado em Componentes*

Murta, L., Corrêa, C., Prudêncio, J. G., Werner, C., (2008), "Proceedings of the 2008 international workshop on Comparison and versioning of software models". , p. 25-30, Leipzig, Germany.

Sonatype, (2008), *Maven : the definitive guide*. 1 ed. Sebastopol Calif, Oreilly.

Tigris.org, (2010). Repoguard. Disponível em: <http://repoguard.tigris.org/>. Acesso em: 16 Abr 2010.

Wloka, J., Ryder, B., Tip, F., Ren, X., (2009), "Proceedings of the 2009 IEEE 31st International Conference on Software Engineering". , p. 507-517