Ouriço: Uma Abordagem para Manutenção da Consistência em Repositórios de Gerência de Configuração

Gleiph Ghiotto Lima de Menezes, Leonardo Gresta Paulino Murta

Instituto de Computação – Universidade Federal Fluminense (UFF)
Niterói – RJ

gmenezes@ic.uff.br, leomurta@ic.uff.br

Abstract. *Configuration Management is a discipline traditionally responsible for controlling the software evolution. However, its work cycle, based on check-out, changes, and check-in, is usually supported by manual verifications, which are, in most cases, counterproductive and error prone. Thus, this work proposes an approach that amplifies the traditional work cycle with asynchronous, automatic, and incremental tasks for maintaining the consistency of the repository. Our preliminary results show that our approach was able to identify broken artifacts without a significant delay in the flow of check-ins.*

Resumo. *Gerência de Configuração é uma disciplina tradicionalmente responsável por controlar a evolução de software. Entretanto, seu ciclo de trabalho baseado em check-out, modificações e check-in pode se tornar contraproducente e propenso a erros, ainda que apoiada por abordagens manuais de verificação. Deste modo, este trabalho propõe uma abordagem que amplifica o ciclo de trabalho tradicional através de tarefas assíncronas, automáticas e incrementais para manter a consistência do repositório. Avaliações preliminares mostraram que esta abordagem foi capaz de encontrar artefatos quebrados sem que o fluxo de trabalho do desenvolvedor fosse significantemente afetado.*

1. Introdução

Gerência de Configuração (GC) é uma disciplina tradicionalmente utilizada para auxiliar na evolução de software (Dart 1991). Esta disciplina faz uso de Sistemas de Controle de Versão (SCV) para gerenciar modificações em artefatos de software (e.g., código fonte, documentação, etc.), provendo um acesso controlado ao repositório. Além disso, os SCV permitem que os desenvolvedores obtenham versões dos artefatos de software referentes a diferentes momentos do ciclo de desenvolvimento (Duvall et al. 2007).

O ciclo de trabalho dos SCV é basicamente composto por três passos, que são executados pelo desenvolvedor. O primeiro passo é o comando de *check-out*, que é responsável por obter artefatos de software de um repositório e enviá-los para um espaço de trabalho na máquina do desenvolvedor. O segundo passo é a realização de modificações, que podem ser adição, remoção ou edição dos artefatos de software previamente obtidos. Finalmente, o terceiro passo é a execução do comando de *check-in*, que é utilizado para enviar as modificações realizadas no espaço de trabalho do desenvolvedor para o repositório, tornando-as visíveis para todos os demais desenvolvedores. Alternativo ao ciclo padrão, o comando de *update* pode ser executado

para atualizar o espaço de trabalho do desenvolvedor com modificações presentes no repositório. Durante o comando de check-in, o SCV é capaz de identificar conflitos físicos (i.e., regiões dos artefatos editadas em paralelo por diferentes desenvolvedores). Entretanto, outros problemas não são verificados, permitindo que artefatos que não compilam (i.e., sintaticamente quebrados) ou que não passam nos testes (i.e., semanticamente quebrados) cheguem ao repositório e sejam acessados por outros desenvolvedores.

A quebra sintática ocorre quando alguma alteração resulta em uma revisão que não está aderente com a gramática da linguagem de programação. A Figura 1 mostra um exemplo onde dois desenvolvedores realizam *check-out* de um repositório e realizam as seguintes ações: João altera o nome de um método, de *celsiusParaKelvin* para *transformacao*, e realiza *check-in* (Figura 1.a); Paralelamente, Maria adiciona uma nova classe que utiliza o método removido por João e, em seguida, realiza check-in (Figura 1.b). Após essas modificações terem sido enviadas para o repositório, os desenvolvedores que realizarem *check-out* obterão uma configuração que não compila.

A quebra semântica ocorre quando alguma regra de negócio não é respeitada. A Figura 2 mostra um exemplo onde dois desenvolvedores realizam *check-out* de um repositório e executam as seguintes ações: Arnaldo altera o nome de um método, de *celciusParaKelvin* para *transformacao*, corrigindo o *check-in* de Maria, e realiza *check-in* dessa configuração (Figura 2.a); Paralelamente, Willian altera o comportamento do método *transformacao* (Figura 2.b), que passa a transformar a temperatura de Celsius para Fahrenheit, e realiza *check-in*, enviando sua configuração para o repositório. Após essas alterações os desenvolvedores que realizarem *check-out* obterão uma configuração que não passa nos testes, devido a uma quebra da regra de negócio, dado que o método volume espera a temperatura em Kelvin e não em Fahrenheit.

```
1 public class Transformacoes {
2
3    public static final int K = 273;
4
5     public static double celsiusParaKelvin(double tCelsius){
6     public static double transformacao(double tCelsius){
7         return tCelsius + K;
8     }
9 }
```

(a)

```
1 public class LeiDoGasIdeal {
2
3   public final double R = 8.314472;
4
5   public double volume(double p, double n, double tCelsius){
6        double tKelvin = Transformacoes.celsiusParaKelvin(tCelsius);
7        return n * R * tKelvin / p;
8    }
9 }
```

(b)

Figura 1. Cenário de quebra sintática onde o desenvolvedor renomeia um método (a) enquanto outro desenvolvedor faz referência a esse método(b).

```
1  public class LeiDoGasIdeal {
2
3    public final double R = 8.314472;
4
5    public double volume(double p, double n, double tCelsius){
6        double tKelvin = Transformacoes.celciusParaKelvin(tCelsius);
7        double tKelvin = Transformacoes.transformacao(tCelsius);
8        return n * R * tKelvin / p;
9    }
10 }
```

(a)

```
1 public class Transformacoes {
2
3    private static final int K = 273; //Celsius
4
5     public static double transformacao(double tCelsius){
6         return tcelsius + K;
7         return 5 * (tCelsius - 32) / 9;
8    }
9 }
```

(b)

Figura 2. Cenário de quebra semântica onde um desenvolvedor realiza uma correção (a) e outro altera o comportamento de um método(b).

Devido ao suporte limitado para prevenir a entrada de artefatos quebrados no repositório (Duvall et al. 2007) e à natureza evolutiva do software, é proposta a abordagem Ouriço. Tal abordagem faz uso de tarefas assíncronas, automáticas e incrementais para detectar quebras sintáticas e semânticas antes que estas sejam propagadas aos demais desenvolvedores, mantendo assim a consistência do repositório. O trabalho foi avaliado em um experimento com o objetivo de responder às seguintes questões de pesquisa: (1) "O Ouriço é capaz de identificar artefatos inconsistentes antes que estes alcancem o repositório?" e (2) "A utilização de tal abordagem resulta em algum atraso no ciclo de desenvolvimento de software?".

O presente artigo está organizado em quatro seções, além dessa de introdução. Na Seção 2 é apresentada uma breve discussão de trabalhos relacionados à consistência de repositórios de gerência de configuração. Na Seção 3 é descrita a abordagem proposta e sua implementação. Na Seção 4 é apresentada a avaliação experimental sobre quatro projetos *open source* e na Seção 5 é apresentada a conclusão do presente trabalho.

2. Manutenção da Consistência em Repositórios de Gerência de Configuração

A identificação de artefatos quebrados é uma prática importante para a manutenção da consistência de repositórios. Durante a execução do presente trabalho foram realizadas buscas por abordagens que identifiquem tais artefatos e foi possível identificar duas vertentes principais: integração contínua e percepção (do inglês, *awareness*).

A Integração Contínua (IC) é uma prática de desenvolvimento de software na qual os membros de uma equipe de desenvolvimento integram o seu trabalho frequentemente (*e.g.,* uma vez ao dia). A integração é composta por um processo de

construção que é composto por: compilação, execução de testes, inspeção, entre outros. A IC pode ser realizada de duas formas distintas: (1) automática e (2) manual. A forma automática não evita que artefatos quebrados cheguem até o repositório. Deste modo, desenvolvedores que realizam *check-in* em repositórios que utilizam apenas a IC automática não têm garantias que artefatos quebrados não entrarão no repositório. Usuários de IC acreditam que o rápido *feedback* é responsável pelo poder da IC, que é dado pelos servidores de IC. Alguns servidores de IC populares são: Continuum (Apache Foundation 2010), Cruise Control (CruiseControl development team 2010) e Hudson (Sun Microsystems 2011). Por outro lado, na IC manual o processo de integração é realizado individualmente, possibilitando que apenas um desenvolvedor realize *check-in* no repositório durante o intervalo de integração. Na integração manual são realizadas tarefas como: compilação, execução de testes, verificações estáticas e demais tarefas que sejam de interesse da equipe de desenvolvimento. Este meio de integração é bastante efetivo para evitar construções quebradas. Entretanto, pode se tornar inviável para grandes equipes de desenvolvimento (Duvall et al. 2007), devido a serialização do *check-in*.

A IC pode ser classificada como uma abordagem reativa, pois identifica artefatos quebrados apenas quando já estão no repositório. Por outro lado, existem abordagens como FASTDash (Biehl et al. 2007), Palantír (Sarma et al. 2003, 2008) e Safe-Commit (Wloka et al. 2009) que objetivam manter os desenvolvedores cientes que possíveis problemas podem ocorrer durante o *check-in*. Contudo, a maioria delas foca apenas em conflitos físicos ou sintáticos e trabalha no espaço de trabalho do desenvolvedor não sendo capaz de garantir que não haverá *check-ins* quebrados. No geral, abordagens baseadas em percepção visam manter o desenvolvedor ciente de possíveis problemas que podem ocorrer durante a integração, mas não realizam verificação sobre a revisão que está sendo enviada para o repositório. Deste modo, estas abordagens podem ser utilizadas em conjunto com o Ouriço, para evitar que artefatos quebrados entrem no repositório e manter o repositório confiável e seguro, quando desenvolvedores "conscientes" realizam equivocadamente *check-in* com artefatos quebrados.

3. Abordagem Ouriço

A abordagem Ouriço (Menezes 2011) estende o ciclo básico de GC para realizar tarefas de verificação sintática e/ou semântica e, deste modo, garantir confiança sobre os artefatos presentes no repositório de GC. Para evitar que um ou mais artefatos quebrados alcancem o repositório, a abordagem proposta utiliza um mecanismo, chamado de *autobranch*, que viabiliza a verificação e, quando necessário, a rejeição desse tipo de artefato. Para viabilizar a rejeição de artefatos quebrados durante a execução do *check-in*, tarefas de verificação, de primeiro e segundo nível, são executadas para garantir a consistência dos artefatos antes que sejam integrados à linha principal do repositório. Ou seja, se uma configuração apresenta algum artefato quebrado, ela é rejeitada; caso contrário, a configuração é enviada para a linha principal do repositório, que permanecerá consistente. Este esquema é ilustrado na Figura 3.

No ciclo da abordagem Ouriço os artefatos passam por duas verificações, sendo uma de primeiro nível e outra de segundo nível. A verificação de primeiro nível ocorre na configuração que é denominada configuração do desenvolvedor. Esse nome é dado justamente por se tratar da configuração que o desenvolvedor envia para o repositório. Já a verificação de segundo nível atua sobre a configuração candidata que recebe esse

nome, por ser o resultado da junção da configuração do desenvolvedor com a configuração corrente do repositório.

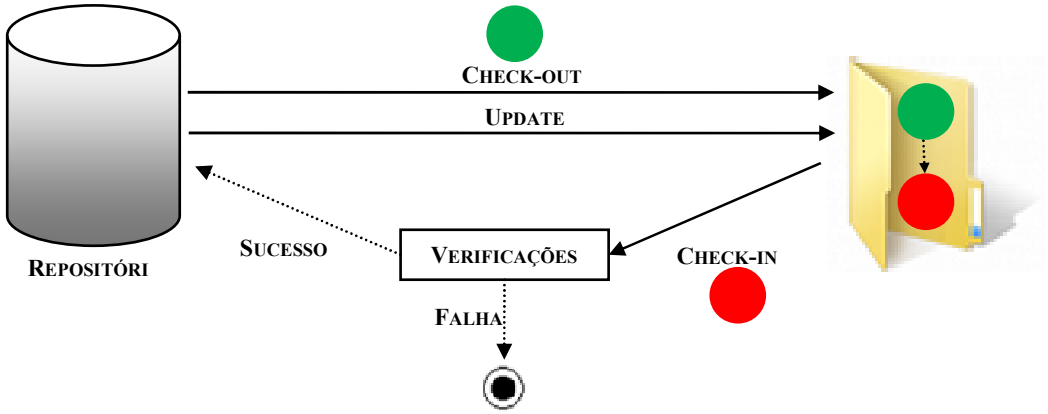

Figura 3. Ciclo executado pela abordagem Ouriço.

As verificações realizadas sobre os artefatos de software podem ser configuráveis para que linhas de desenvolvimento distintas possam receber tratamentos diferenciados (IEEE 2005). O Ouriço é composto de quatro políticas: a permissiva (política menos severa), a moderada, a restritiva (política mais severa) e a dinâmica (calibrada conforme o estado do projeto no momento do *check-out*). Cada linha de desenvolvimento pode receber um tratamento diferenciado, através da escolha de políticas diferentes, por motivos como: restrição de prazo (i.e., tempo máximo que uma tarefa pode ser executada); existência ou não de testes, entre outros. Um cenário de utilização destas políticas é o caso em que os ramos recebem um tratamento menos severo, enquanto a linha principal recebe um tratamento mais severo, por ser o local onde a maioria dos desenvolvedores realiza contribuições e, portanto, necessita maior grau de confiança nos artefatos.

As políticas utilizadas pelo Ouriço são compostas por filtros que possibilitam a detecção de problemas específicos nos artefatos de software. Ouriço faz uso de três filtros: físico, sintático e semântico. Estes filtros são utilizados quando a execução de alguma dessas verificações é necessária para a política em questão. Os filtros que compõem cada política são descritos com maiores detalhes na Seção 3.1.

A abordagem Ouriço tem o objetivo de ser não intrusiva, portanto os desenvolvedores continuam tendo a mesma percepção do ciclo de GC, que também é composto por *check-out*, modificações e *check-in*. Além disso, o Ouriço trata o comando de *update*, que é utilizado em situações onde é desejável obter novidades do repositório. Durante a execução do *check-out* o Ouriço realiza tarefas como criação de *autobranches* (i.e., camada que tem o objetivo de proteger o repositório e viabilizar verificações) e verificações iniciais. Essas verificações do *check-out* são realizadas após a entrega da configuração ao desenvolvedor (i.e., em paralelo com as suas edições) com o objetivo de aumentar o desempenho das verificações que são realizadas durante o *check-in* e calibrar a política dinâmica, discutida na Seção 3.2. Durante a execução do *check-in* o Ouriço realiza novamente verificações sobre a configuração candidata, que será então integrada ao repositório em caso de sucesso.

No restante desta seção, os conceitos são descritos com maiores detalhes visando dar um melhor entendimento da diferença do ciclo tradicional de GC para o ciclo de

trabalho do Ouriço. Esse ciclo incorpora verificações e outras tarefas inexistentes no ciclo tradicional de GC.

3.1. Filtros

A abordagem Ouriço executa verificações para identificar problemas específicos em artefatos de software. Conforme foi discutido na Seção 1, os dois problemas principais que podem ocorrer em repositórios de GC são as quebras sintática e semântica. Além disso, conflitos físicos podem também ocorrer. Estes problemas podem ser identificados por meio dos filtros físico, sintático ou semântico, descritos nesta seção.

O filtro físico identifica conflitos que ocorrem entre a configuração do desenvolvedor, que está sofrendo *check-in*, e a configuração corrente do repositório de GC. Estes conflitos ocorrem quando mais de um desenvolvedor edita a mesma região de um artefato de software e tenta enviá-la ao repositório. O Ouriço utiliza as ferramentas dos SCV, como a de junção, para identificá-los e, posteriormente, reportá-los ao desenvolvedor. O filtro sintático é utilizado para identificar quebras sintáticas. Tal filtro é auxiliado por Sistemas de Gerenciamento de Construção (SGC) (*i.e.*, sistemas capazes de construir projetos executando tarefas como compilação, testes, entre outros). O filtro semântico é utilizado para identificar quebras semânticas (*i.e.,* situações onde às regras de negócio não são respeitadas, resultando em casos de testes quebrados). Para identificar quebras em regras de negócio, o filtro semântico executa testes sobre a configuração que está sendo verificada, através de um SGC.

O objetivo desses filtros é identificar problemas (i.e., conflitos físicos e quebras sintáticas ou semânticas) que posteriormente serão enviados aos desenvolvedores, por meio de uma notificação. Esta notificação é composta pela natureza da quebra (*i.e.,* física, sintática ou semântica) e sua descrição (*e.g.*, *class not found*).

Por fim, filtros podem ser adotados em dois modos distintos de operação, bloqueante ou informativo. Os filtros bloqueantes param o ciclo de trabalho quando algum problema é encontrado e, deste modo, evitam que o repositório fique em estado inconsistente. Por outro lado, os filtros informativos não param o ciclo de trabalho quando algum problema é identificado, mas notificam o desenvolvedor sobre a existência de artefatos quebrados no repositório.

3.2. Políticas

O Ouriço tem como um dos objetivos realizar verificações em diferentes níveis de controle para diferentes compartimentos lógicos do repositório (IEEE 2005). Para realizar essas verificações, o uso de políticas foi adotado. Estas políticas foram classificadas como: permissiva, moderada, restritiva e dinâmica. Essas políticas são compostas por filtros bloqueantes e, com o objetivo de prover maior informação sobre os artefatos de software presentes no repositório, podem ser estendidas por filtros informativos. Foram definidas quatro políticas para o Ouriço que são descritas nesta seção.

A política permissiva possui um nível de controle menos severo. Tal política aplica um ciclo de verificação muito similar ao aplicado pelos SCV tradicionais. Por *default*, a política permissiva identifica apenas conflitos físicos, via execução do filtro físico bloqueante discutido anteriormente. Portando, somente artefatos que não possuem

conflitos físicos são aceitos na linha de desenvolvimento do repositório controlado por esta política.

A política moderada apresenta um nível de controle superior à política permissiva. Esta política aplica um ciclo de verificação que estende as verificações realizadas pelos SCV tradicionais. Isto é, por padrão, a política moderada pode identificar conflitos físicos e quebras sintáticas através dos filtros físico e sintático em modo bloqueante. Deste modo, somente artefatos que não resultem em conflitos físicos e que possam ser compilados serão aceitos na linha de desenvolvimento do repositório protegida por essa política.

A política restritiva apresenta o nível de controle mais severo desta abordagem. Essa política aplica uma verificação que estende as verificações realizadas pelos SCV tradicionais. Por padrão, a política restritiva é capaz de identificar conflitos físicos, sintáticos e semânticos através da adoção dos respectivos filtros em modo bloqueante. Assim, somente artefatos que não resultem em conflitos físicos, sejam compiláveis e passem pelos testes serão aceitos na linha de desenvolvimento do repositório que é protegida por essa política.

A política dinâmica é configurada de acordo com o estado da configuração obtida do repositório. Durante o processo de *check-out*, a configuração que está sendo obtida do repositório é verificada sintaticamente e semanticamente para que esta política seja calibrada conforme uma das políticas anteriores. A principal diferença desta política para as anteriores é que o desenvolvedor deverá devolver os artefatos de software da maneira que recebeu ou ainda mais consistentes.

É válido ressaltar que nas políticas permissiva e moderada o desenvolvedor pode optar por adicionar filtros informativos que darão maior poder de detecção de artefatos quebrados. Por exemplo, a política permissiva pode ter filtros sintáticos ou semânticos informativos, e a política moderada pode ter filtros semânticos informativos. Desta forma, quando alguma configuração apresentar algum problema que vai além do originalmente tratado pela política, o desenvolvedor será alertado sobre a sua causa, mas a configuração não será rejeitada.

3.3. Ciclo de Trabalho

O Ouriço apresenta um ciclo de trabalho baseado no ciclo tradicional de GC. O ciclo de trabalho do Ouriço implementa os comandos de *check-out*, *check-in* e *update*. Como discutido anteriormente, para viabilizar a verificação de artefatos foi incluído o conceito de *autobranch*, que são ramos protegidos criados de maneira automática quando um *check-out* é realizado e permitem que alterações sejam realizadas sem refletir nos artefatos originais. No restante dessa seção são descritos os comandos do Ouriço.

3.3.1. *Check-out*

A execução do *check-out* é feita em duas etapas (Figura 4): (1) criação de um *autobranch* baseado na configuração que o desenvolvedor requisitar e (2) *check-out* do *autobranch* criado para o espaço de trabalho do desenvolvedor. Além disso, o *check-out* também realiza verificações prévias (3) quando a políticas moderada, restritiva ou dinâmica está sendo utilizada.

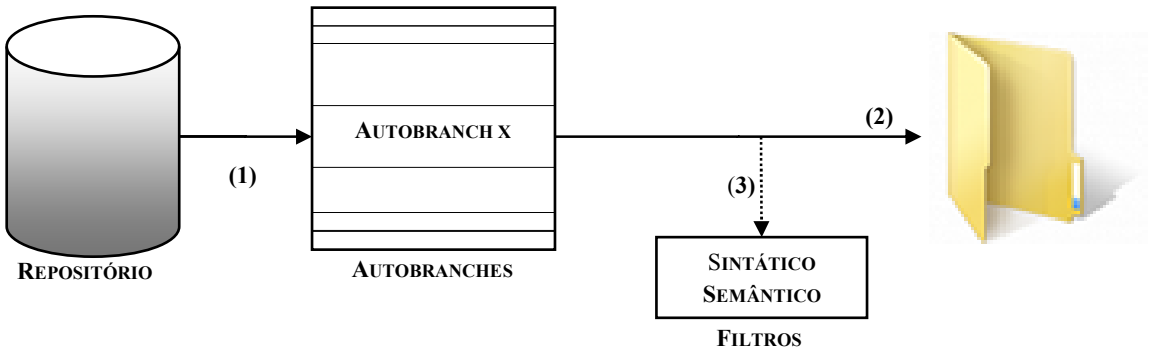

Figura 4. Comando de check-out do Ouriço.

A execução do passo 3 da Figura 4 permite que no momento do *check-in* as verificações sejam realizadas em menos tempo, devido às habilidades do SGC de só recompilar o que foi modificado. Contudo, ele se torna ainda mais importante quando a política dinâmica é escolhida. Conforme discutido anteriormente, a política dinâmica não possui filtros pré-definidos, devido ao seu objetivo de garantir que o desenvolvedor possa enviar artefatos para o repositório no nível de qualidade em que foram obtidos. Tal política é calibrada durante o processamento do comando de *check-out* da seguinte forma: são aplicados os filtros sintático e semântico sobre a configuração alvo; e se a configuração não passar por nenhum filtro a política é calibrada conforme a política permissiva; se a configuração passar apenas pelo filtro sintático, a política é calibrada conforme a política moderada; e se passar pelos filtros sintático e semântico a política é calibrada conforme a política restritiva.

3.3.2. *Check-in*

O comando de *check-in* do Ouriço possui o objetivo de evitar que artefatos quebrados entrem no repositório. Para tanto, tal comando foi estendido e é composto por 5 passos que realizam verificações, junções e integração de artefatos ao repositório. Esses passos são os seguintes (Figura 5): (1) c*heck-in* tradicional, (2) verificação de primeiro nível, (3) obtenção da configuração candidata, (4) verificação de segundo nível e (5) integração da contribuição do desenvolvedor.

O comando de *check-in* pode ser interrompido se qualquer irregularidade for identificada. Conforme discutido anteriormente, esta abordagem visa encontrar irregularidades em artefatos de software e impedi-los de chegar ao repositório de GC. Portanto, quando um defeito é encontrado, o comando de *check-in* é imediatamente interrompido, em qualquer passo, e uma notificação, explicando o problema identificado e o tipo, é enviada ao desenvolvedor. Caso contrário, a configuração do desenvolvedor é integrada (passo 5), pois está apta a entrar no repositório sem deixá-lo inconsistente.

Além de se preocupar com a consistência do repositório, a presente abordagem também se preocupa com o tempo que o desenvolvedor pode ficar ocioso esperando por verificações que, em alguns casos, podem ser demoradas e executadas de maneira automática. Por isso, as verificações realizadas por esta abordagem serão executadas de maneira assíncrona em relação ao desenvolvedor. Assim, o desenvolvedor fica livre para se dedicar a outras tarefas, enquanto o Ouriço executa as tarefas de verificação, que em alguns casos podem ser longas e desgastantes. Um exemplo de verificação

demorada é a compilação do OpenOffice Ximian, que leva aproximadamente 11 horas (Linux Reviews 2011).

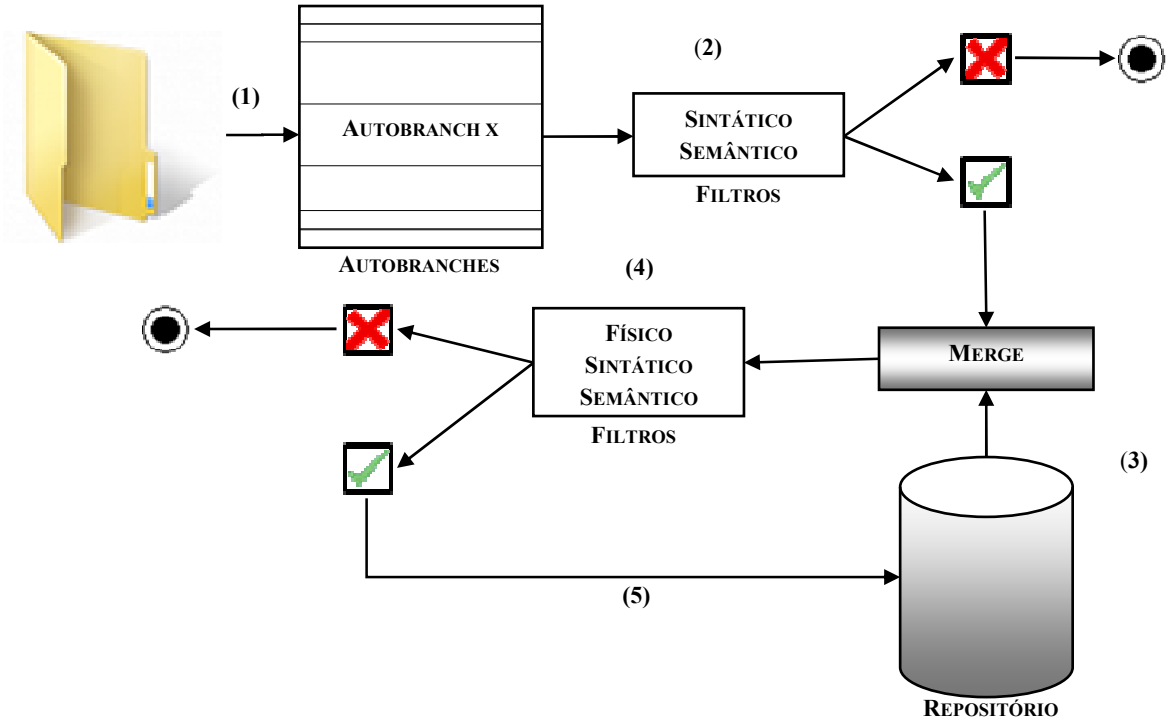

Figura 5. Comando de *check-in* do Ouriço.

3.3.2. *Update*

O comando de *update* é utilizado tradicionalmente para trazer contribuições que estão no repositório, mas que ainda não foram incorporadas no espaço de trabalho do desenvolvedor. Este comando é tradicionalmente executado quando um desenvolvedor quer integrar as contribuições feitas por outros desenvolvedores no seu espaço de trabalho.

O comando de *update* tradicional seguiria os seguintes passos se aplicado no Ouriço: (1) análise do *autobranch* para detectar as modificações presentes no repositório, mas que ainda não foram incorporadas ao espaço de trabalho, e (2) integração das modificações no espaço de trabalho do desenvolvedor. Este comando resultaria na incorporação de todas as modificações realizadas no *autobranch* para o espaço de trabalho do desenvolvedor. No entanto, este não seria o resultado desejado.

Por isso, o comando de *update* do Ouriço foi reestruturado para obter as informações do repositório original, deixando o *autobranch* somente para receber as alterações executadas no espaço de trabalho. O comando de *update* do Ouriço é composto pelos seguintes passos: (1) análise da linha de desenvolvimento que originou o *autobranch* para identificar as modificações e (2) reflexão destas modificações no espaço de trabalho do desenvolvedor.

Durante o comando de *check-out*, o Ouriço guarda uma relação entre a linha de desenvolvimento que originou *autobranch* e o próprio *autobranch*. Deste modo, é possível identificar qual linha de desenvolvimento originou os artefatos de software, que o desenvolvedor possui em seu espaço de trabalho. Assim sendo, no momento que o

Ouriço for obter as modificações, que serão incorporadas no espaço de trabalho, a linha de desenvolvimento que deu origem aos artefatos do espaço de trabalho do desenvolvedor será utilizada como referência.

3.4. Aspectos de implementação

O Ouriço possui os comandos de *check-out*, *check-in* e *update* implementados tanto via interface gráfica quanto linha de comando. Além disso, o Ouriço possui uma interface web onde é possível acompanhar o andamento das tarefas (Figura 6), uma vez que estas verificações são realizadas de forma assíncrona ao desenvolvedor (*i.e.*, sem interação direta com o desenvolvedor). Desta forma, essa interface web funciona como um painel de controle do projeto. Na Figura 6 é possível identificar dados como o *autobranch*, a revisão que ele é baseado, a url do projeto e o estado atual do *autobranch*, que varia desde *checked-out* (quando foi apenas obtido) até *integration successfully performed* (quando o *autobranch* é integrado ao repositório).

Autobranch	Revision	User	Repository URL	Current State
1	4	marapao	https://10.0.0.100/svn/trunk	Checked-out
2	5	marapao	https://10.0.0.100/svn/trunk	Checked-out
3	6	marapao	https://10.0.0.100/svn/trunk	Checked-out
4	9	marapao	https://10.0.0.100/svn/trunk	Checked-out
5	11	marapao	https://10.0.0.100/svn/trunk	Integration successfully performed
6	20	marapao	https://10.0.0.100/svn/trunk	Integration successfully performed
7	24	marapao	https://10.0.0.100/svn/trunk	Integration successfully performed
8	28	marapao	https://10.0.0.100/svn/trunk	Integration successfully performed
9	33	marapao	https://10.0.0.100/svn/trunk	First level syntactic verification failed.
10	34	marapao	https://10.0.0.100/svn/trunk	First level semantic verification failed.
11	4	marapao	http://localhost/svn/trunk	Integration successfully performed
12	9	marapao	http://localhost/svn/trunk	Integration successfully performed
13	16	marapao	http://localhost/svn/trunk	Checked-out
14	18	marapao	http://localhost/svn/trunk	Integration successfully performed
15	23	marapao	http://192.168.0.101/svn/trunk	Integration successfully performed

Figura 6. Painel de acompanhamento de *autobranches*.

Para acompanhar o projeto com mais detalhes é possível interagir com esse painel e identificar características relacionadas a cada *autobranch*, como é possível visualizar na Figura 7. O *autobranch* detalhado nesse exemplo é o 12 e nele é possível identificar que foram realizadas quatro tentativas de *check-in*, onde as três primeiras falharam e somente na última foi possível integrar o conteúdo do *autobranch* com sucesso.

Start Time	End Time	Description
2011-06-11 21:48:33.997	2011-06-11 21:48:46.101	Checked-out
2011-06-11 21:49:44.1	2011-06-11 21:49:44.593	First level syntactic verification failed. (Detail)
2011-06-11 21:50:36.1	2011-06-11 21:50:36.612	First level syntactic verification failed. (Detail)
2011-06-11 21:51:39.1	2011-06-11 21:51:39.766	First level syntactic verification failed. (Detail)
2011-06-11 21:52:34.1	2011-06-11 21:52:34.699	First level syntactic verification successfully performed.
2011-06-11 21:52:34.704	2011-06-11 21:52:35.383	First level semantic verification successfully performed.
2011-06-11 21:52:35.386	2011-06-11 21:52:36.112	Second level physic verification successfully performed
2011-06-11 21:52:36.115	2011-06-11 21:52:36.333	Second level syntactic verification successfully performed
2011-06-11 21:52:36.337	2011-06-11 21:52:37.008	Second level semantic verification successfully performed.
2011-06-11 21:52:37.011	2011-06-11 21:52:41.105	Integration successfully performed

Figura 7. Painel de detalhamento de um *autobranch*.

O Ouriço foi implementado em Java e atualmente é capaz de apoiar projetos que fazem uso do Maven (Sonatype 2008) e que são armazenados em repositórios Subversion (Berlin, Rooney 2006). É válido ressaltar que o Maven é capaz de gerenciar dependências e executar compilações e testes automáticos de acordo com a configuração do projeto, viabilizando a execução dos filtros sintáticos e semânticos. Para implementar a interface web foram utilizadas tecnologias como JSF, Facelets e Rich Faces. A camada de banco de dados foi implementada utilizando JPA, Hibernate e PostgreSQL. Além disso.

4. Avaliação da Abordagem

O Ouriço foi avaliado utilizando quatro projetos com as seguintes características: armazenados em repositórios Subversion, compatíveis com Maven e que possuem testes automatizados. Devido a essas restrições foram selecionados os seguintes projetos: BCEL (Apache Software Foundation 2011), SQL Maven Plugin (Codehaus 2012), Native Maven Plugin (Codehaus 2010) e Checkstyle (Checkstyle 2011). O BCEL visa dar aos usuários a possibilidade de criar e manipular arquivos binários da linguagem Java. Este projeto possui aproximadamente 90.000 linhas de código. O SQL Maven Plugin é responsável por executar instruções SQL e possui aproximadamente 3.400 linhas de código. O Native Maven Plugin é responsável por realizar construção de projetos que utilizam as linguagens C e C++ via Maven. Este projeto possui cerca de 16.200 linhas de código. Finalmente, o Checkstyle é uma ferramenta utilizada para realizar verificações estáticas em artefatos de software. Este projeto contém 213.620 linhas de código.

O Ouriço objetiva manter o repositório consistente sem que atrasos perceptíveis sejam inseridos no ciclo de trabalho do desenvolvedor. Por isso, foram executadas avaliações sobre duas perspectivas, que são: (1) “por quanto tempo o repositório que

contém o projeto permanece inconsistente?"; e (2) "existe algum atraso relacionado à adoção do Ouriço para verificação dos artefatos?". É importante ressaltar que a primeira perspectiva avaliar ao quanto o Ouriço seria útil para os projetos em questão, e que a segunda perspectiva avalia se a adoção do Ouriço inseriria atrasos indesejáveis para o ciclo de trabalho do desenvolvedor. Estas duas perspectivas são discutidas nessa seção.

Para responder a questão "por quanto tempo o repositório que contém o projeto permanece inconsistente?", foram realizadas verificações sobre todas as revisões presentes na linha principal dos repositórios dos projetos, simulando a adoção do Ouriço. Essas verificações são compostas pela execução do ciclo de trabalho do Ouriço (Figura 8) sobre todas as revisões do repositório. Cada ciclo é composto pelos seguintes passos: (1) *check-out* do repositório de estudo (inicialmente vazio, mas que é protegido pelo Ouriço) para o espaço de trabalho de estudo; (2) exportação de uma revisão do repositório original para o espaço de trabalho temporário; (3) junção do conteúdo do espaço de trabalho temporário com o conteúdo do espaço de trabalho de estudo; e (4) *check-in* do espaço de trabalho de estudo no repositório de estudo.

É importante ressaltar que este processo de avaliação foi escolhido para permitir a simulação dos *check-ins* reais. O resultado é uma análise retrospectiva que mostra como o Ouriço poderia agir se estivesse sendo utilizado durante o desenvolvimento do projeto. Este mecanismo apresenta algumas vantagens como: ele pode ser executado em dias, enquanto a adoção do Ouriço em projetos reais poderia demandar meses para gerar resultados; e não gera efeitos negativos ao projeto no caso de falha do Ouriço. Porém, não apresenta resultados qualitativos que um experimento real poderia gerar.

Durante a avaliação do Ouriço foi utilizada a politica restritiva. Esta decisão foi tomada, pois é possível classificar o repositório como inconsistente se algum artefato quebrado está presente no repositório, independentemente do tipo de quebra (física, sintática ou semântica).

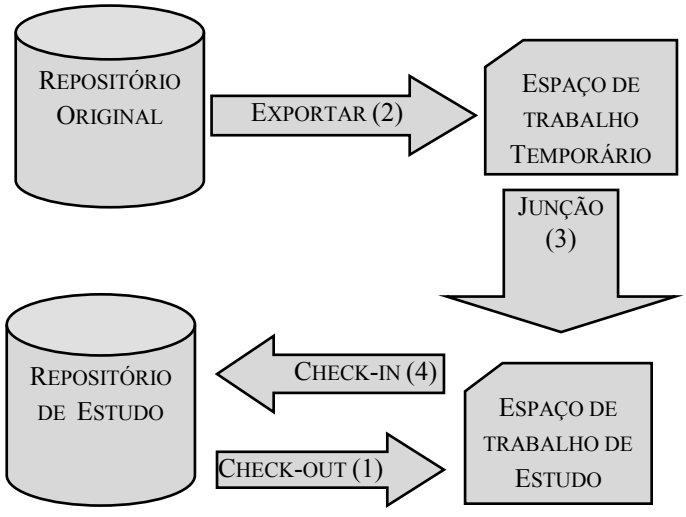

Figura 8. Ciclo de experimentação do Ouriço.

A Tabela 1 mostra os resultados relacionados à verificação dos projetos. É possível observar que é comum ter inconsistências no repositório, ao menos para os projetos utilizados neste experimento. Por exemplo, o repositório do Native Maven Plugin está quebrado por aproximadamente 35 mil horas (72,33% do período de

desenvolvimento) e em 229 *check-ins* (76,85% do total dos *check-ins*). Tal prática se torna contraproducente, pois os desenvolvedores possuem uma pequena chance (aproximadamente 28% do período de desenvolvimento e 23% das revisões) de realizar check-out e obter uma revisão que compila e passa nos testes. É possível notar que este projeto possui uma alta taxa de inconsistências, mesmo utilizando IC. Porém, de acordo com análises posteriores foi possível identificar que a partir da revisão 223 a taxa de inconsistência cai de 72,33% para apenas 8% de revisões quebradas. Este número é um indício de que a integração contínua foi empregada por volta desta revisão. Porém, mesmo após essa queda, o número ainda é grande, podendo afetar negativamente na produtividade da equipe de desenvolvimento.

Tabela 1. Resultados da verificação

Projeto	Período de desenvolvimento (horas)	Período inconsistente (horas)	Período consistente (horas)	Revisões estudadas	*Check-ins* inconsistentes	Check-ins Consistentes
BCEL	6.951,96	664,25	6.287,70	30	3	27
SQL Maven Plugin	44.386,16	4937,87	39.448,28	135	28	107
Native Maven Plugin	48.656,44	35.193,83	13.462,61	298	229	69
Checkstyle	33.026,03	312,76	32.713,28	309	43	266

Por outro lado, existem projetos que mantiveram o repositório consistente na grande maioria do tempo. Um exemplo é o Checkstyle, que mantém o repositório consistente durante 32.713,28 horas (99,05% do período de desenvolvimento analisado). Entretanto, ainda nesse caso o Ouriço pode ser utilizado para aumentar a confiança no repositório, pois 43 *check-ins* tornam o repositório inconsistente (13,92% do total de *check-ins*). Com o repositório inconsistente, a chance de obter artefatos de software quebrados aumenta.

Conforme discutido anteriormente, foi analisada outra perspectiva para responder a seguinte pergunta: “existe algum atraso relacionado à adoção do Ouriço para verificação dos artefatos?”. Nesse experimento foi analisada a quantidade de *check-ins* que seriam atrasados devido às verificações do Ouriço. Os resultados são apresentados na Tabela 2.

Tabela 2. Resultados de performance

Projeto	Tempo médio de Verificação (s)	Intervalo entre *check-ins* (s)	*Check-ins* atrasados
BCEL	25,82	834.235,22	2
SQL Maven Plugin	9,03	1.183.631,00	0
Native Maven Plugin	6,01	587.795,94	7
Checkstyle	11,95	384.769,32	0

O Ouriço teve bons resultados quando comparado com o intervalo entre *check-ins*. De acordo com a Tabela 2, o tempo gasto em verificação é, na média, bem inferior ao intervalo entre *check-ins* consecutivos. No SQL Maven plugin, o pior casos ocorre quando o projeto está correto e precisa passar por todas as etapas de verificação, gastando 9,85 segundos. Contudo, o intervalo médio entre *check-ins* é de 1.183.631 segundos em média (aproximadamente 13 dias). Por esse motivo, apenas 9 *check-ins* (2 do BCEL e 7 do Native Maven Plugin), de um total de 772, sofreram atraso devido à

interferência do Ouriço, representando somente 1,17% dos *check-ins*. Esse bom desempenho pode ser explicado pela utilização de SGC que possuem a capacidade de construir apenas as partes do projeto que foram alteradas.

Os resultados apresentados levantam indícios de que o Ouriço apresenta um bom potencial para detecção de artefatos quebrados e que não é contraproducente, pois o ciclo de trabalho de GC não é atrasado de maneira significante (atraso máximo de 97 segundos). Porém, algumas ameaças à validade podem ser levantadas: os experimentos foram executados em quatro projetos que, em alguns casos, possuem poucas revisões; não existe conhecimento da maturidade das equipes que desenvolveram os projetos, o que limita nossas analises em termos qualitativos; o número pequeno de projetos não é representativo o bastante para generalizar conclusões; e, finalmente, foi realizado um estudo retrospectivo, portanto os resultados voltados para o efeito da inclusão do Ouriço em um ambiente real ainda não foi obtido.

5. Conclusão

O Ouriço demonstrou ser uma abordagem eficiente para detectar artefatos quebrados. De acordo com os estudos realizados Ouriço foi capaz de identificar *check-ins* inconsistentes (de 14% a 77% dos *check-ins*) e evitar que eles alcançassem o repositório. Além disso, foi possível perceber que é usual o envio de artefatos quebrados para o repositório, permitindo a existência de artefatos sintaticamente ou semanticamente quebrados. Outro resultado observado é que o Ouriço foi pouco intrusivo no ciclo de trabalho, dado que o tempo gasto para verificação é menor, na maioria dos casos, que o intervalo entre *check-ins*.

Os seguintes trabalhos futuros puderam ser vislumbrados: a execução de experimentos *in-vivo* (*i.e.*, desenvolvedores utilizando o Ouriço durante o ciclo de desenvolvimento de software); a extensão para outros tipos de projetos, implementando conectores para outros SCV e SGC; a integração do Ouriço com abordagens baseadas em percepção, para atenuar aspectos negativos do isolamento entre desenvolvedores; a realização de estudos sobre a dificuldade de junção de ramos; e, finalmente, a introdução de técnicas de testes seletivos (Hsia et al. 1997) para acelerar a análise realizada pelos filtros semânticos em projetos grandes, com muitos casos de teste.

Agradecimento

Agradecemos à CAPES, ao CNPq (305283/2011-1), e à FAPERJ (E-26/100.196/2010, E-26/111.281/2011 e E-26/103.253/2011) pelo apoio financeiro.

Referências

Apache Foundation, (2010), Continuum. Disponível em: <"http://continuum.apache.org/">. Acesso em: 16 Apr 2010.

Apache Software Foundation, (2011), Apache Commons BCEL™ -. Disponível em: <"http://commons.apache.org/bcel/">. Acesso em: 11 Mar 2012.

Berlin, D. and Rooney, G., (2006), *Practical Subversion*. 2nd ed. ed. Apress.

Biehl, J. T. and Czerwinski, M. and Smith, G. and Robertson, G. G., (2007), "FASTDash: a visual dashboard for fostering awareness in software teams". In: *Conference on Human Factors in Computing Systems (CHI)*, p. 1313–1322

Checkstyle, (2011), Checkstyle. Disponível em: <"http://checkstyle.sourceforge.net/writingchecks.html">. Acesso em: 8 Apr 2011.

Codehaus, (2010), Native Maven Plugin. Disponível em: <"http://mojo.codehaus.org/maven-native/native-maven-plugin/">. Acesso em: 8 Apr 2011.

Codehaus, (2012), *SQL Maven Plugin*. Disponível em: <"http://mojo.codehaus.org/sql-maven-plugin/">. Acesso em: 7 May 2010.

CruiseControl development team, (2010), CruiseControl. Disponível em: <"http://cruisecontrol.sourceforge.net/">. Acesso em: 9 Apr 2011.

Dart, S., (1991), "Concepts in configuration management systems". , p. 1–18, New York, NY, USA.

Duvall, P. M. and Matyas, S. and Glover, A., (2007), *Continuous Integration: Improving Software Quality and Reducing Risk*. 1 ed. ed. Addison-Wesley.

Hsia, P. and Li, X. and Chenho Kung, D. and Hsu, C. and Li, L. and Toyoshima, Y. and Chen, C., (1997), "A technique for the selective revalidation of OO software", *Journal of Software Maintenance: Research and Practice*, v. 9, n. 4 (Jul.), p. 217-233.

IEEE, (2005), *IEEE Std 828 - standard for software configuration management plans*. New York, N.Y. :, Institute of Electrical and Electronics Engineers,.

Linux Reviews, (2011), Compile time stats. Disponível em: <"http://linuxreviews.org/gentoo/compiletimes/">. Acesso em: 31 Jan 2011.

Menezes, G., (2011), *Ouriço: Uma Abordagem para Manutenção da Consistência em Repositórios de Gerência de Configuração*. Dissertação de Mestrado, Universidade Federal Fluminense - UFF

Sarma, A. and Noroozi, Z. and van der Hoek, A., (2003), "Palantír: Raising Awareness among Configuration Management Workspaces". In: *International Conference on Software Engineering (ICSE)*, p. 444-454, Portland, Oregon.

Sarma, A. and Redmiles, D. and van der Hoek, A., (2008), "Empirical evidence of the benefits of workspace awareness in software configuration management", *Symposium on Foundations of Software Engineering (FSE)*, p. 113–123.

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

Sun Microsystems, (2011), Hudson Continuous Integration. Disponível em: <"http://hudson-ci.org/">. Acesso em: 9 Apr 2011.

Wloka, J. and Ryder, B. and Tip, F. and Xiaoxia Ren, (2009), "Safe-commit analysis to facilitate team software development". *International Conference on Software Engineering (ICSE)*, p. 507-517