3.
Estado da Arte

Algumas das ferramentas de controle de versão comerciais mais conhecidas atualmente são: *Concurrent Version System* (CVS) [CEDERQVIST, 1993], *Microsoft Visual SourceSafe* (MVSS) [MICROSOFT, 2000], *Subversion* (SVN) [PILATO, 2004], *Team Foundation Version Control* (TFVC) [MICROSOFT, 2007], *GIT* [SWICEGOOD, 2008], *Mercurial* (HG) [MERCURIAL, 2006], *Bazaar* [POOL, 2007], *Rational Team Concert* [IBM, 2008], etc. Ferramentas como CVS e MVSS definem versões no nível de arquivos, porém as ferramentas mais recentes definem versões no nível de *changesets*, que é o conjunto total de modificações marcados com um único nome, o seu identificador. Dentre as ferramentas mais recentes encontramos os controles de versão distribuídos, como GIT [SWICEGOOD, 2008], Mercurial [MERCURIAL, 2006] e Bazaar [POOL, 2007], que não seguem o modelo tradicional em que o repositório é armazenado em um servidor e utiliza-se um cliente para manipulá-lo. Nestes controles de versão distribuídos cada *checkout* cria um novo repositório no ambiente de trabalho do usuário, contendo as mesmas informações do original. Uma das principais vantagens de se manter um repositório completo na estação de trabalho do usuário é permitir que todas as operações realizadas sobre o repositório sejam armazenadas localmente, e somente sejam exportadas quando for desejado. Desta forma é possível realizar um conjunto de *commits* locais, que somente se tornarão efetivos quando o repositório for explicitamente sincronizado. Resumidamente, este modelo mantém em cada cópia um repositório completo capaz de atuar como cliente ou como servidor. Observe que isso não impede a existência de uma cópia atuando apenas como servidor, semelhante ao modelo tradicional, porém, permite também que equipes de desenvolvimento sincronizem cada uma com seu repositório-servidor e estes sejam sincronizados entre si somente quando desejado. Na prática, estas ferramentas fornecem ao usuário controles complexos, que por um lado são bastante poderosos, porém por outro dificultam sua utilização por usuários menos experientes. Estes problemas têm sido tratados pelo *Bazaar*,

que se sobrepõe aos outros por apresentar uma IHC mais clara, tendendo a reduzir o número de possíveis operações equivocadas realizadas pelo usuário.

Estudando trabalhos na literatura que buscam ferramentas de comparação, unificação e sistemas de controles de verão mais precisos e mais informativos, encontramos tendências que serão discutidas a seguir.

3.1 Baseado em operações

Existem propostas voltadas para um mecanismo orientado a operações [LIPPE e OOSTEROM, 1992] [SHEN e SUN, 2001] [IGNAT e NORRIE, 2005] [DIG, *et al.*, 2006] [DIG, NGUYEN e JOHNSON, 2006] [DIG, *et al.*, 2006] [FREESE, 2006]. Estes trabalhos defendem que, conhecendo as operações que levaram a uma modificação, é possível determinar sua semântica, utilizar esta informação para melhorar a qualidade do resultado, e, consequentemente, instruir a ferramenta de unificação como proceder ao encontrar um conflito.

Este mecanismo consiste em registrar em uma sequência de passos as operações de edição realizadas sobre o documento. Durante um incremento de versão esta sequência de passos é utilizada em conjunto com os estados inicial e final do documento para inferir as diferenças. Indo além, dependendo do formato de armazenamento de incrementos do controle de versão, a própria sequência de operações pode ser utilizada para esta finalidade. Neste caso, para navegar entre versões é necessário aplicar as operações sobre o documento na ordem crescente ou decrescente, dependendo da direção que se está navegando.

Como dito, esta abordagem é capaz de identificar diferenças com precisão e auxiliar a ferramenta de unificação, porém seu principal ponto negativo é um obstáculo considerável para aplicá-la em ambientes de desenvolvimento reais: é dependente de ferramentas especiais capazes de gravar a sequência de passos sobre o documento. Ou seja, é uma solução que impões restrições sobre as ferramentas que serão utilizadas, tornando-se inadequada para a maioria dos ambientes de desenvolvimento que adotam ferramentas otimizadas para o tipo de aplicação que está sendo desenvolvida.

Completando, alguns trabalhos como [DIG, *et al.*, 2006] [DIG, NGUYEN e JOHNSON, 2006] [DIG, *et al.*, 2006] [FREESE, 2006] armazenam na sequência de operações as operações de refatoramento executadas pela IDE e fazem uso

deste conhecimento para desconsiderá-las na comparação. Esta é uma iniciativa importante no contexto de desenvolvimento atual, em que as IDEs oferecem recursos que sintaticamente realizam uma operação sobre um determinado elemento, que na prática afeta diferentes trechos de código. Logo, seria interessante que tais operações não fossem consideradas, ou consideradas com menor peso, na comparação dos documentos. Esta tendência será comentada na seção seguinte, com abordagens diferentes das citadas, baseadas na sequência de operações.

3.2 Baseado em refatoração

Como já foi comentado, existem trabalhos que buscam melhorar a qualidade do resultado de uma comparação a partir do reconhecimento de informações de refatoramento, seja esta uma melhoria para visualizar as diferenças efetivas ou para aumentar o grau de automatização da unificação. Operações de refatoramento são operações disparadas por comandos em ambientes de desenvolvimento que executam tarefas de forma automática ou semi-automatica sobre o código fonte modificando sua semântica, porém mantendo a consistência do software. Podemos citar como exemplo: troca do nome de um método, classe, atributo, variável; extração de método; reordenação de parâmetros; migração de declarações em sua hierarquia, etc. Estas informações de refatoramento podem ser identificadas na sequência de operações realizadas [DIG, *et al.*, 2006] [DIG, NGUYEN e JOHNSON, 2006] [DIG, *et al.*, 2006] [FREESE, 2006], como já foi citado, ou no resultado da comparação dos estados inicial e final de uma evolução de um documento [DIG, *et al.*, 2006] [DIG, *et al.*, 2005] [DIG, JOHNSON e MARINOV, 2006].

De forma geral, a identificação das operações de refatoramento em uma comparação baseada em estados consiste em buscar padrões nas diferenças encontradas ao comparar duas versões consecutivas de um documento. Como exemplo, uma operação de renomeação de um método é caracterizada por uma declaração cujo nome foi modificado de *m1* para *m2*, e, todos os pontos do código que invocavam o método *m1* a chamada foi trocada por uma chamada ao método *m2*. Entretanto, este exemplo mostra um requisito adicional para buscar por este

padrão: uma AST (*Abstract Syntax Tree*) do código que está sendo comparado para avaliar as chamadas de método.

Comparativamente, este mecanismo de identificação das operações de refatoramento em uma comparação baseada em estados, mesmo que menos preciso que o mecanismo baseado na sequência de operações, é capaz de atingir uma precisão alta, como avaliado por Dig [DIG, *et al.*, 2006] em uma implementação no ambiente de desenvolvimento Eclipse, para a linguagem de programação Java, onde foi alcançado 85% de precisão quando testado em aplicações reais. Além disso, apesar deste mecanismo baseado em comparação supostamente alcançar uma precisão menor, independe das ferramentas de edição utilizadas, desta forma, não compartilhando do ponto negativo das abordagens que restringem as ferramentas de desenvolvimento pela imposição de um ambiente específico em que todas as ferramentas são instrumentadas com mecanismos para armazenar em um histórico comum as operações realizadas.

Completando, como o trabalho de Dig [DIG, *et al.*, 2006] [DIG, *et al.*, 2005] [DIG, JOHNSON e MARINOV, 2006] tem como objetivo a detecção de operações de refatoramento, a análise sintática do código é feita até o escopo de métodos e atributos apenas.

3.3 Baseado em sintaxe

Seguindo a mesma motivação desta dissertação, encontramos na literatura trabalhos que buscam melhorar a qualidade de ferramentas de comparação [YANG, 1991] [GODFREY, 2005] [KIM e NOTKIN, 2006] e unificação [ZÜNDORF, WADSACK e ROCKEL, 2001] [KLIER, 2005] [MUNSON e DEWAN, 1994] [YANG, HORWITZ e REPS, 1992] [WANG, PIERCE e MCFARLING, 2000] [MENS, 2002] [IGNAT e NORRIE, 2005] a partir da comparação da estrutura sintática de documentos. Dentre estes trabalhos é unânime a opinião que documentos devem ser comparados segundo a sua estrutura sintática, respeitando os limites de cada elemento, a sua estrutura hierárquica e possivelmente levando em consideração as informações semânticas quando presentes.

Este tipo de abordagem para o problema exige um interpretador para o tipo de conteúdo específico do documento, o que limita a ferramenta para os formatos

conhecidos. Assim como o presente trabalho, os demais citados dirigem seus esforços para produzir uma ferramenta genérica cujas definições sintáticas de cada tipo de documento possam ser acopladas. Isso permite que, além do autor da ferramenta, demais grupos interessados sejam capazes de criar componentes que implementem a definição sintática de outros tipos de documentos a fim de suportá-los na ferramenta.

3.4 Orientado a modelos

Considerando o contexto de comparação sintática, existe um caso especial que trata do versionamento de modelos [ZÜNDORF, WADSACK e ROCKEL, 2001] [EL-KHOURY, 2005] [NGUYEN, 2009] [BARTELT, 2008] [MURTA, *et al.*, 2007]. Ferramentas comerciais comuns são direcionadas para documentos contendo informações textuais, desconsiderando outros tipos de documentos. Mesmo que estes outros tipos de documentos persistam suas informações em um formato textual (como XML [QUIN, 2003]), este não é adequado para ser comparado e unificado nas ferramentas que tratam de texto, pois a partir da leitura de seu conteúdo é difícil extrair as informações de mais alto nível contidas neles, neste caso, informações sobre os modelos.

Logo, concluímos que a comparação sintática neste caso é mais relevante, pois não apenas melhora o resultado, mas viabiliza que tais documentos possam ao menos ser comparados e unificados. E, considerando os avanços obtidos no desenvolvimento de ambientes que apoiem MDA [KLEPPE, WARMER e BAST, 2003], é eminente a necessidade de sistemas de controle de versão e ferramentas capazes de comparar e unificar modelos com clareza.

3.5 SCMs com granularidade variável

Também podemos encontrar trabalhos na literatura que vão além das ferramentas de comparação e unificação e propõem controles de versão inteiros trabalhando sobre elementos com granularidade configurável [WESTFECHTEL, *et al.*, 2001] [NGUYEN, *et al.*, 2004] [JUNQUEIRA, BITTAR e FORTES, 2008] [PIETROBON, 1995].

Assim como os demais, estes trabalhos defendem que a representação textual sobre todos os tipos de documentos impõe limitações e dificuldades na

utilização dos controles de versão quando são persistidos documentos não-textuais. Logo, para versionar documentos de acordo com a sua sintaxe, cada um propõe à sua maneira uma forma de especificar o tipo de dado que compõe os documentos, para que o controle de versão armazene um incremento estrutural em vez de um incremento textual. Desta forma, as ferramentas de comparação e unificação associadas a estes controles de versão tornam-se capazes de trabalhar diretamente com a estrutura sintática dos documentos, sem necessariamente conhecer o tipo de documento específico.

Em comparação com as ferramentas apresentadas nas seções anteriores, a diferença desta abordagem é que as ferramentas recebem os documentos em sua forma estrutural em vez de passar por um pré-processamento que transforma os documentos para uma forma canônica (estrutural) a ser trabalhada.

Controles de versão deste tipo associados a ambientes de desenvolvimento, como [PIETROBON, 1995], são capazes de manter identificadores por elemento sintático durante toda a história armazenada, a partir da aplicação do mecanismo de gravar a sequência de operações. Certamente esta abordagem produz resultados de comparação e unificação extremamente precisos, pois cada elemento existente no programa é rastreado durante toda a sua evolução.

3.6 Comparação híbrida

Outro trabalho relacionado é um artigo que descreve uma pesquisa sobre as técnicas existentes para comparação de código-fonte de software [KIM e NOTKIN, 2006]. Esta pesquisa apresenta as dificuldades conhecidas, cada técnica de comparação, e a avaliação de um experimento em que cada técnica foi aplicada em dois cenários de evolução de código. Um dos resultados mais interessantes deste experimento é que a utilização de uma única técnica não é o ideal para abordar todos os tipos de cenário existentes, porém um comparador híbrido, composto por diferentes técnicas, é capaz de obter um resultado melhor na identificação das diferenças. Este resultado influenciou os primeiros passos dessa dissertação, onde se optou por um mecanismo de comparação em que a técnica de comparação de cada elemento é determinada pelo seu tipo, como será mostrado nos capítulos seguintes.