Um Modelo Unificado de Controle de Consistência para Ambientes de Desenvolvimento de Software

Carlos Alberto Marques Pietrobon
Arndt Von Staa

RESUMO: Modelo para controle da consistência dos dados de projeto de um Ambiente de Desenvolvimento de Software Distribuído baseado na unificação do conceito de transação com controle de versão e configuração e, considerando-se todos os objetos de projetos criados como inalteráveis.

PALAVRAS-CHAVE: Ambiente de Desenvolvimento, Controle de Versão, Controle de Configuração, Imutabilidade, Engenharia de Software, Consistência de Dados.

INTRODUÇÃO

Para aumentar a produtividade (software de qualidade assegurada produzido por unidade de tempo), tem sido propostos Ambientes de Desenvolvimento de Software Distribuído (ADSD), que são sistemas automatizados e que fazem uso de banco de dados para armazenar as informações relativas aos sistemas desenvolvidos, tais como, código fonte, especificacões, documentação, etc.

Em adição, ADSDs tem sido propostos para apoiar o desenvolvimento de grandes projetos de software, realizados por equipes, as quais podem estar geográficamente distribuidas, podendo seus membros acessar concorrentemente os mesmos dados. Isto dificulta a manutenção da consistência, dos dados do projeto e dos dados de controle do ambiente de desenvolvimento.

Como em um ADSD as transações podem ser longas (podem durar dias), não são atômicas, podem ser aninhadas, e podem acessar concorrentemente a mesma informação, percebe-se que as técnicas de manutenção da consistência adotadas em banco de dados tradicionais (comerciais), baseadas exclusivamente em transações, não se aplicam em um ADSD, exigindo que novas técnicas sejam estudadas. Este trabalho descreve uma técnica que:

- unifica os conceitos de transação, controle de versão e controle de configuração, de forma a controlar o acesso ao banco de dados e, conseqüentemente, a consistência dos dados do projeto;

- considera os objetos de projeto (módulos de programa, especificações, etc) como imutáveis. A imutabilidade de um objeto significa que, uma vez criados, eles não podem mais ser alterados. Qualquer alteração em um objeto imutável implicará na criação de novas versões deste objeto. Na seção 2 serão discutidos as características de um ADSD, no que diz respeito à consistência dos dados. Na seção 3 o conceito de objetos imutáveis será abordado. Na seção 4 será avaliada esta nova técnica, mostrando sua adequação à ADSDs.

CARACTERÍSTICAS DE UM ADSD

A manipulação consistente e confiável dos dados, é uma característica importante para que um ADSD possa efetivamente apoiar o desenvolvimento de software. Até recentemente, os esforços de pesquisa foram concentrados em SGBD para área comercial, onde transação é considerada a unidade de consistência, restauração e sincronização. As idéias oriundas destas pesquisas são igualmente aplicáveis em ADSD, sendo que o modelo baseado únicamente em transação precisa ser modificado para poder cooperar com as exigências próprias de um ADSD, uma vez que as técnicas válidas para ambientes convencionais carecem de flexibilidade e eficiência requerida por um ADSD [PRIC89, WALP87]. A seguir são relacionadas as exigências de um ADSD para controlar as mudanças no seu banco de dados, estando elas grupadas segundo cada uma das funções de controle que permitam controlar a consistência:

TRANSAÇÕES

Nos bancos de dados convencionais, o objeto típico de manipulação é um registro, com tamanho e tipo bem determinado. Em um banco de dados que apoie um ADSD, este objeto tem uma granularidade bem maior, podendo conter grandes quantidades de dados, não necessariamente estruturados, tais como texto ou código fonte. Além disso, transações em um ADSD incluem operações tais como edição de diagramas, que podem ser demoradas, podendo levar horas ou dias. A combinação destas duas características faz com que a transação seja longa [WALP88].

As transações podem ser aninhadas, ou seja, compostas de sub-transações que por sua vez podem também ser aninhadas. Este tipo de transação permite que a estrutura e interdependências de operações dentro de transações muito grandes seja representada. Isto é particularmente útil dentro de ADSD para atividades como construção de sistemas. Transações aninhadas podem também ser usadas para aumentar o desempenho de transações muito grandes, permitindo que sub-transações executem em paralelo.

Dois ou mais usuários que desenvolvam um objeto (parte de um sistema), em conjunto, possuem o direito de atualizá-lo. Portanto, duas ou mais transações devem poder acessar o mesmo objeto, possivelmente de modo concorrente. Este acesso concorrente, junto com os relacionamentos complexos entre objetos, dificulta a manutenção da consistência do banco de dados.

A transação vai operar em ambiente distribuído. Portanto, ela tem que levar em conta três grandes problemas [WALP87]: a possibilidade de falha parcial do sistema, o alto nível de atividades paralelas e a falta de uma informação acurada do estado global do sistema.

As técnicas atuais de transação não são adequadas para dar suporte as características expostas acima [WALP88]. Por exemplo, transações longas dificultam ou mesmo impossibilitam as técnicas de sincronização de acesso existentes, baseadas na noção de serialização. Ou seja, como nestes casos a transação retém os recursos que ela usa durante a sua existência, outro usuário que eventualmente necessitasse destes recursos, teria que esperar o término da transação para que pudesse acessá-los. Em ambientes de desenvolvimento isto seria altamente ineficiente, por não permitir paralelismo das atividades. Além disso, caso haja uma falha e a transação tenha que ser interrompida, grandes quantidades de informação podem ser perdidas. Assim, é necessário investigar novas técnicas que suportem transações longas e transações não serializáveis [WALP88]. Transações aninhadas também apresentam problemas para os mecanismos usuais. Por exemplo, o alto nível de multiprogramação gerado pode exarcebar o problema de "deadlock". Se considerarmos que a transação é longa e o ambiente distribuído, o controle de "deadlock" torna-se extremamente difícil. Portanto, é benéfico para um ADSD, se dispor de mecanismos de transação que estejam livres de "deadlock" [WALP88].

CONTROLE DE VERSÃO

Um ADSD deve permitir que a evolução de um objeto de projeto ao longo do tempo seja registrada. Isto é conseguido armazenando-se vários de seus valores ou versões, tendo-se assim, um histórico de versões. Esta característica difere do banco de dados que apoia ambientes comerciais, onde, em geral, apenas o valor mais recente é mantido.

Por outro lado, existem diversas maneiras de derivação de uma versão. Assim, é interessante que este histórico de versões consiga expressar os diversos tipos de relacionamento de derivação ou sucessão. Segundo [BECK89 , VICT89], as versões podem ser organizadas como uma sequência linear, como uma árvore ou como um grafo. A sequência linear ocorre quando a nova versão é derivada sempre a partir da versão mais recentemente criada. Quando mais de uma versão de objeto de projeto está sendo trabalhada ao mesmo tempo e mais de uma versão pode ser derivada da mesma versão antecessora, é necessário um histórico em forma de árvore para organizar estes conceitos. Por fim, um histórico de versões na forma de grafo permite representar a derivação

de uma versão a partir de duas ou mais versões antecessoras. O tratamento de ve
rsões, além de permitir que o usuário visualise a evolução do objeto, também dev
e permitir alterações experimentais, acesso a novas e velhas versões uniformemen
te e [DITT88], permitir ou facilitar o controle da concorrência, a restauração,
o aumento a performance em siste
mas distribuidos e facilitar o desenvolvimento organizado de software.

CONTROLE DE CONFIGURAÇÃO

Objetos de projeto podem ser tanto compostos quanto primi
tivos, conforme possam ou não ser decompostos em outros objetos de projeto. Obje
tos compostos são contruidos a partir de referências a outros, por sua vez compo
stos ou primitivos, que podem se relacionar entre si. Portanto, uma alteração fe
ita em um objeto pode impactar outros objetos com os quais o objeto alterado se
relaciona. O controle de configu
ração se preocupa com as mudanças realizadas em cada grupo de objetos que se rel
acionam, de forma a manter a consistência entre eles.

O controle de configura
ção se realiza por meio de atividades a nível de sistema e a nivel técnico-admin
istrativo. A nivel de sistema temos a parte automatizada do processo de controle

Para isto deve ser fornecido pelo ambiente de desenvolvimento todo um suporte
que permita automatizar aspectos de controle não criativos que se manifestam ao
longo do ciclo de vida do softwa
re. Por exemplo, o ADSD alertar ao usuário sempre que ele tente modificar um obj
eto de projeto que já esteja sendo alterado por outro usuário. A nivel tecnico-a
dministrativo, temos a coordenação da equipe de desenvolvimento (gerência de des
envolvimento), de forma a se obter o produto desejado, com um nivel de qualidade
aceitável, dentro do prazo e com os recursos disponiveis. Foge ao escopo deste
trabalho um estudo detalhado desta função técnico-administrativa. Para o nosso p
ropósito, basta saber que ela va
i ser a responsável por solucionar problemas que possam ocasionar perda de consi
stência e que não podem ser resolvidos a nivel de sistemas (por exemplo, dois us
uários que querem alterar a mesma porção de código fonte).

OBJETOS DE PROJETO IMUTÁVEIS

1) IMUTABILIDADE

Imutabilidade de um obje
to de projeto significa que, os objetos uma vez criados, não podem ser alterados
("updating"). A noção de atualização (como existe em banco de dados comerciais)
, de um objeto de projeto, é sub
stituida, aqui, pela noção de transformação. Uma transformação consiste de uma s
equência de operações as quais produzem um novo objeto. O objeto mais antigo per
manece inalterado. Esta propriedade é importante para a manutenção da consistênc
ia, como será visto mais a frente.

Uma outra propriedade importante das trans
formações, é que elas são atômicas. A atomicidade das transformações é conseguid
a pela criação de uma cópia do objeto antigo. As operações são então realizadas
sobre esta cópia, que só será vi
sível quando a transformação terminar ("transformation commit"). Desta maneira,
os estados intermediários do novo objeto permanecem invisiveis até o término da
transformação.

Um sistema com objetos imutáveis é consideràvelmente diferente
de um convencional, baseado na atualização do objeto mais recente disponivel. N
este último, tem-se apenas um valor do objeto, que é o valor mais atual, enquant
o que nos sistemas imutáveis, objetos existem como uma série de versões represen
tando sua evolução ao longo do tempo.

2) RELACIONAMENTO ENTRE OBJETOS

Assumindo que um objeto de projeto
imutável possa ser transformado ao longo do tempo para produzir novos estados i
mutáveis (ou versões), um objeto de projeto conceitual é essencialmente um grafo
da história das versões. A figura 1.a ilustra um grafo da história de um objeto
conceitual: a partir de uma única versão original de um objeto, sucessivas tran
sformações fazem o grafo se desenvolver ao longo do tempo. Este grafo efetivamen

te representa relacionamentos que expressam as derivações das versões.

A outra forma de relacionamento entre objetos, é aquela que existe entre objetos conceituais, sendo definida como configuração. Em um sistema imutável, a configuração pode ser vista como uma coleção de versões e seu inter-relacionamento (uma versão para cada objeto conceitual relacionado).

3) CONSISTÊNCIA ENTRE OBJETOS IMUTÁVEIS

Em sistemas mutáveis, a consistência está interessada nos relacionamentos entre os únicos valores de múltiplos objetos. Objetos estão relacionados pelas restrições de consistência de um sistema, que definem os relacionamentos que devem ser mantidos para que o sistema permaneça consistente. Em sistemas imutáveis, múltiplas versões de objetos conceituais são mantidas. Consequentemente, restrições de consistência podem existir entre algumas versões de objetos conceituais relacionados, mas não entre outras. Por exemplo, a versão 1 do objeto conceitual A pode ser consistente com a versão 1 do objeto conceitual B, e pode não ser, entretanto, consistente com a versão 10, mais recente de B.

Assim, para se manter consistência em sistemas imutáveis, tem-se que indicar quais objetos dentro do sistema constituem um estado consistente particular, pois existem vários conjuntos de objetos consistentes, o que difere dos sistemas mutáveis onde existe apenas um conjunto consistente. Esta indicação é feita nomeando-se cada configuração consistente. A nomeação de configurações consistentes substituem a noção de retrições de consistência em ambientes mutáveis pois, se algum relacionamento inicialmente for válido entre dois objetos, o relacionamento precisa sempre ser válido, devido a imutabilidade das versões relacionadas. A seguir é descrita uma técnica de nomeação.

4 - NOMEAÇÃO POR ENDEREÇAMENTO RELATIVO AO DOMÍNIO (ERD)

Esta técnica divide o espaço de objetos do sistema em domínios consistentes separados (configurações), onde cada

domínio contém uma versão de cada objeto conceitual que se relaciona consistentemente. Cada domínio é então nomeado univocamente.

4.1 - MECANISMOS PARA SUPORTAR ERD

Para suportar ERD dentro de um ADSD, o problema de como nomear objetos de projeto e domínios consistentes precisa ser solucionado. A solução proposta em [NIC089] envolve a inclusão de três categorias especificas de objetos: objeto de história, objeto de configuração e objeto de transação.

Objeto de configuração é um conjunto desordenado de versões de objetos conceituais relacionados. O objeto de história, é um grafo direcionado de objetos de configuração (fig. 1.b). Desta maneira, a evolução do sistema é registrada pela criação de novas configurações.

Usando este esquema, objetos são nomeados primeiro acessando um objeto de história, para obter um objeto de configuração. Isto designa um domínio consistente em particular. O objeto desejado é então nomeado relativamente a este domínio (possivelmente através de um 'pathname' que concatena o nome do objeto de configuração e o nome da versão). Os outros objetos com os quais ele é consistente, estão contidos no mesmo domínio consistente e são também nomeados considerando-se o mesmo objeto de configuração.

Objeto de transação é um objeto de configuração de uso particular e cujos dados podem ser armazenados indefinidamente no banco de dados. Ele é criado pela transação, que também é responsável pela sua nomeação (bem como das versões de objeto de projeto componentes), ficando privado ao usuário até ao fim da transação ("commit"). Este objeto pode ser aberto e fechado. O fechamento de um objeto de transação implica em que os resultados parciais da transação associada sejam salvos em uma nova versão do objeto de transação, assim criando um ponto de salvamento. A abertura do objeto de transação faz com que a transação seja reiniciada a partir de um especifico ponto de salvamento.

Quando a transação termina, o objeto de transação torna-se um objeto de configuração e é incluido na estrutura de nomeação (objeto de história) do sistema. Objetos de transação podem ser considerados como objetos de configuração em estado transiente.

5 - TRANSAÇÃO EM AMBIENTE IMUTÁVEL COM ERD

Segundo [KOHL81, BERN81], algoritmos de transação podem ser considerados como consistindo de duas partes separadas: sincronização e restauração. A seguir será discutida a interpretação dada a estes dois conceitos em ambientes imutáveis.

5.1 - SINCRONIZAÇÃO

Em um ambiente mutável, aplica-se sincronização para se obter consistência, o que é conseguido através da exclusão mútua. Assim, no máximo uma transação pode estar alterando o estado de um objeto, o que reduz o nível de concorrência e, consequentemente, o desempenho do sistema. Esta redução pode ser quase imperceptível quando o tempo requerido pelas transações for pequeno.

Em um ambiente imutável, a sincronização é opcional, podendo ou não ser requerida, em função da semântica de sincronização do objeto de configuração. Três semânticas de sincronização foram identificadas [WALP87]: ramificação livre, ramificação controlada e ramificação linear. Quando nenhuma sincronização é requerida, temos um objeto com ramificação livre (figura 2.a). Transações concorrentes sobre a mesma versão nunca são evitadas. Apesar disso, a consistência de versões individuais é garantida: cada transação (T1, T2, T3), resulta na criação de uma nova versão de um objeto de configuração, mantendo todos os objetos de projeto da versão de configuração original inalterados.

Pretende-se que esta classe de objetos seja usada por configurações que não sejam compartilhadas, assim como um grupo de objetos sendo desenvolvido no espaço de trabalho local do usuário.

Em um projeto cooperativo, é frequentemente requerido o acesso compartilhado a certos objetos conceituais [VICT89]. Assim, é definida um outra classe de semântica de sincronização (ramificação controlada), que prevê este conflito (fig. 2.b). Caso uma transação (T3), tente transformar um objeto de configuração que pertença a esta classe e que já está sendo transformado, um alerta é gerado (devido a um "advisory lock" que é associado a todo o objeto de configuração desta classe, que esteja sendo transformado). Deve-se, então, tomar uma decisão autorizando ou não a continuação da segunda transformação. O processo de autorização leva em consideração aspectos gerenciais, como a adequabilidade do momento para se fazer tal transformação, a segurança (direito de transformação), etc. Caso a segunda transformação seja autorizada a prosseguir, um novo ramo é criado no grafo de histórias do objeto de configuação.

Observa-se que o alerta de conflito pode ser desrespeitado, podendo a transação criar um novo ramo na história de versões da configuração, sendo um atitude intencional, na qual o usuário está consciente dos possíveis conflitos que venham a surgir.

Finalmente, tem-se a classe dos objetos de configuração lineares (fig. 2.c). A semântica de sincronização é que uma versão no máximo está disponivel para o acesso a qualquer momento, isto é, apenas a versão mais recente está disponivel, estando as versões anteriores inacessíveis. Esta classe essencialmente simula o comportamento dos objetos sofrendo atualizações em sistemas mutáveis. Para conseguir esta sincronização, um "lock" de leitura e escrita é associado a versão, quando uma transação é autorizada a modificá-la. A partir daí, qualquer requisição de acesso a este objeto é cancelada, enquanto a primeira transação não terminar.

5.2 - RESTAURAÇÃO

Caso ocorra uma falha do sistema ou erro de operação, é necessário garantir que o sistema retorne a um estado consistente. Em ambientes mutáveis, o processo de restauração pode tornar-se difícil porque a última informação consistente disponivel pode ter sido destruida. Já em um ambiente imutável, estados consistentes anteriores não são nunca destruidos e, para se obter um estado consistente, basta acessar uma versão anterior.

Desde que, objetos criados como resultados parciais de uma transação somente são feitos visíveis após o término da transação, não é estritamente necessário, em termos de consistência, remove- lo

s após as falhas, pois eles aind
a não fazem parte da história de configurações. Assim, esta parte da restauração ficou reduzida a uma ação de "garbage collection".

A outra parte da restaura
ção implica na remoção de "locks" aplicados, o que depende da classe semântica d o objeto de configuração. Caso seja um objeto de ramificação livre, nenhuma rest auração é necessária pois nenhum "lock" foi aplicado. Para o caso dos objetos de ramificação controlada e linear, a retirada de locks aplicados pela transação é optativa no primeiro caso (pode-se desrespeitar o aviso de advertência gerado pelo sistema) e obrigatório no segundo caso.

AVALIAÇÃO DA TÉCNICA

Nesta seção será fe
ita uma avaliação mostrando como o conceito de imutabilidade em conjunto com a u nificação das técnicas de controle de alteração de banco de dados permitem apoia r um ADSD. A avaliação é feita por característica desejada.

- Suporte para transações longas

Transações longas são suportadas naturalmente pelo uso de ob
jetos de transação e pontos de s
alvamento. Por ser o objeto de transação de uso particular e um domínio (configu ração) consistente, o usuário pode trabalhar sobre esta configuração por longo p eríodo com alta autonomia do resto do sistema sendo desenvolvido. Por outro lado , o uso de pontos de salvamento permite que resultados parciais sejam salvos, em bora ao custo da criação de uma nova versão da configuração. Finalmente, como ve rsões podem ser replicadas, mais segurança e eficiência é obtida.

- Suporte para transações aninhadas

O mo
delo de transação pode ser extendido para permitir que objetos de configuração s ejam aninhados, formando uma hierarquia de objetos de configuração. Assim, cada sub-transação dentro de uma transação mais externa, cria seu próprio objeto de c onfiguração quando ele começa. Ao fim da sub-transação, o conteúdo de seu objeto de configuração é incluído no objeto de configuração da transação pai. O objeto

de configuração pai só termina quando todos os seus filhos tiverem terminado. A transação mais externa, termina
fazendo incluir seu objeto de configuração na estrutura de nomeação do sistema. Assim, os resultados de uma transação aninhada tornam- se visíveis em um único passo atômico.

- Sincronização sem serialização das transações

Como cada t
ransação trabalha em cima de seu próprio objeto de transação, as modificações oc orrerão em cima de versões diferentes do mesmo objeto conceitual, não havendo, p ortanto, necessidade de serialização das transações para sincronizar o acesso, a excessão a das transações cuja
semântica de sincronização é determinada por configurações lineares, que simulam o efeito da serialização. A nomeação por ERD também ajuda a evitar serialização por não impor qualquer ordenação global dos objetos do sistema para fins de nom eação, como é feito em [GIFF88].

- Transações livres de "deadlock"

Como cada transação tem sua própria área de trabalho (objeto de transação), duas transações não acessam os mesmos recursos, e portanto, não há o perigo de "deadlock". Esta restrição pode ser relaxada para permitir que duas ou mais configurações sejam juntadas, gerando um único domínio resultante, que contenha o resultado das alterações feitas nos objetos das duas configurações.

- Desenvolvimento de software em ambiente distribuído

Objetos de transação e configuração permitem que existam versões privadas ao usuário (versões em desenvolvimento), bem como versões públicas (visíveis por to da a comunidade), o que é desejável neste tipo de ambiente [CHOU86, DITT88, PRIC 89, VICT89].

Como réplicas s
ão permitidas, a eficiência e a segurança são incrementadas, pela diminuição do número de acessos remotos e pela duplicação da informação.

Finalmente, como
objetos são imutáveis, não há perigo deles serem alterados por outros usuários.

- Controle de versões de objetos de projeto

O conceito de imutabilidade juntamente com a técnica de nomeação por endereçamento relativo ao domínio, permite que várias versões sejam criadas para um objeto conceitual, bem como fornece um mecanismo de nomeação que permite uma identificação única e um acesso uniforme a qualquer versão de objeto de projeto.

A história das versões de um objeto conceitual está implícita na h
istória de versões da configuração, sendo função da integração do controle de ve
rsão e política de sincronização de transação em um ambiente baseado em ERD. Ela
será uma sequência linear para o caso de objeto com semântica de sincronização
linear, podendo apresentar-se com o formato de árvore ou grafo, para as outras d
uas semânticas de sincronização.

- Controle de configurações de objetos de projeto

O modelo unificado fornece uma base ideal para o controle de configuração por satisfazer várias condições, como será exposto a seguir.

Através do conceito de objetos de configuraçã
o se consegue nomear configurações e, graças ao conceito de imutabilidade garant
e-se que uma vez definido os relacionamentos dentro de uma configuração, eles nã
o podem mais serem invalidados.

Devido a integração entre controle de versão e controle de configuração, tor
na-se possível que as configurações sejam controladas por versão, permitindo que
múltiplas versões de configuração sejam suportadas. Por outro lado, a integraçã
o entre transação e controle de configuração permite que sistemas de software se
jam construídos atômicamente.

As mudanças sobre objetos constituintes da con
figuração podem ser controlados de uma maneira que uma visão consistente de uma
entidade de mais alto nível seja fornecida. Este controle é feito parcialmente p
elo sistema que implementa o mod
elo, através da criação de cópias e sinalização de perigo de conflito e, parcial
mente a um nível técnico-administrativo, onde conflitos são resolvidos via negoc
iação entre as partes interessadas, estabelecendo-se e controlando-se compromiss

os de modificação.

- Concorrência e Restauração

A definição de três tipos
de acesso, de uma política para controlar este acesso em cada caso e, o uso de o
bjetos imutáveis, permite controlar o acesso concorrente. Já a restauração, é fa
cilitada pela existência de versões anteriores dos objetos.

CONCLUSÕES

Este trabalho m
ostrou que, as técnicas para controle de consistência usadas em banco de dados c
onvencionais, não se aplicam a ADSD, sendo proposta uma nova técnica que unifica
as três atividades que controlam a consistência em ADSD, baseada no conceito da
imutabilidade dos objetos de projeto. A imutabilidade simplifica o problema de
sincronização, facilitando o mecanismo de transação, por transferir a complexida
de de se manusear objetos concor
rentemente para o controle de versão e configuração. Mais precisamente, a consis
tência entre transações conflitantes (acesso distribuído, "programming in the ma
ny"), é assegurado pelo controle de configuração. A consistência entre objetos (
"programming in the large") é realizada pelo controle de versão e configuração,
enquanto que o controle de objetos em ponto pequeno ("programming in small"), é
assegurada pelas transações de atualização. Finalmente, o conceito de imutabili
dade reduz o controle de consist
ência a um problema de nomeação. O preço a pagar é um consumo maior de espaço de
armazenamento e a necessidade de se manter "scripts" de configuração.

AGRADECIMENTOS

A John Nicol [NIC089] pelas dúvidas sanadas e material fornecido.