Modularizando a Gerência de Configuração de Software

André Luiz Moura *Francisco A. C. Pinheiro*

Universidade de Brasília – UnB
Departamento de Ciência da Computação
{amoura, facp}@cic.unb.br
Brasília, DF, Brasil

Resumo

Este artigo apresenta um modelo para gerência de configuração de software em termos de módulos de projeto. Módulos de projeto determinam um escopo para as atividades do desenvolvimento, os objetos por elas criados ou transformados e as ferramentas empregadas para esse fim. O modelo é adequado para evolução do software, integrando ações relativas ao controle de versões com ações gerenciais alusivas ao controle do processo. Ele permite associar os vários componentes com as fases do ciclo de vida e atividades do desenvolvimento. Possibilita também um controle mais detalhado da configuração e contextualização das várias decisões pertinentes a ela.

Palavras-chave: componentes de software, evolução de software, gerência de configuração, modularização, processos de desenvolvimento.

Abstract

This article presents a model for software configuration management and evolution. The model is aimed at providing context to configuration management activities. We use project modules for structuring and defining the software development process. The project modules determine a scope where development activities can be carried out and objects can be created and used. The project modules may reflect, for example, the phases of a software life-cycle model. By associating the context provided by project modules to the structures used to control software evolution we promote a more tight integration between configuration management activities and the other activities of the development process.

Key-words: configuration management, modularization, software components, software evolution, software process models.

1 Introdução

Um dos aspectos relevantes da Gerência de Configuração de Software (GCS) é a determinação do contexto no qual o processo de desenvolvimento é conduzido, com a conseqüente integração entre GCS e evolução. Uma parte essencial da GCS é apoiar a evolução do software, a qual compreende todas as atividades que modificam um sistema de software, por exemplo, respostas a pedidos de mudança, melhoramentos na performance ou clareza e reparos de defeitos [Fei88, Luq89]. Para viabilizar esse apoio, é necessário entender como as diversas partes do sistema se encaixam, bem como os impactos que uma mudança pode vir a desencadear.

A multiplicidade de hardware, ambientes operacionais e base de dados compele as organizações a criarem versões alternativas de sistemas que trabalhem sobre uma variedade de plataformas. Isso aumenta a complexidade das atividades e da estrutura dos produtos criados no ciclo de vida de um sistema e reforça a importância de manter histórico da evolução do sistema. Tal histórico inclui: pessoas, atividades, ferramentas, pedidos de mudança, versões de diferentes componentes e outros artefatos [Hum90, Luq90]. O modo como esses objetos se relacionam, juntamente com informações sobre os agentes e razões de tais relacionamentos, descrevem aspectos do contexto no qual o sistema foi desenvolvido. Esse contexto deve ser compreendido pelos gerentes e demais membros da equipe de desenvolvimento, para que o trabalho seja efetivamente coordenado e a evolução do software controlada. Assim, o processo de desenvolvimento torna-se fator importante para a gerência de configuração por incorporar o contexto no qual as atividades são realizadas.

Neste artigo, apresentamos um modelo para gerência de configuração em termos de módulos de projeto. Nele são identificados e controlados ferramentas, atividades e componentes gerados durante o ciclo de vida do software. Nosso modelo é uma extensão do modelo proposto por Luqi [Luq90]. O modelo original tem como objetivo propor uma estrutura formal com a qual se possa modelar as atividades de gerência de configuração de software. Esta formalização é essencial para o desenvolvimento de ferramentas que automatizem e auxiliem essas atividades. O nosso modelo, além de preservar o objetivo do modelo original, visa a promover uma integração mais forte entre as atividades de gerência de configuração e as demais atividades do processo de desenvolvimento. Essa integração é importante, à medida que as tarefas de controle de versão e manutenção da consistência e integridade dos vários objetos produzidos são parte integrante do processo de desenvolvimento. De modo mais específico, pode-se ter as atividades relativas à gerência de configuração restritas a determinadas fases do ciclo de vida. Por exemplo, um analista, ao pesquisar no repositório de objetos sob controle de configuração a existência de especificações que possam ser reutilizadas, pode restringir sua pesquisa apenas aos objetos que foram criados na fase de análise. Adicionalmente, nosso modelo permite incluir, como itens de configuração, outros componentes que normalmente não são considerados.

Este artigo está organizado da seguinte forma: a seção 2 descreve os conceitos básicos. A seção 3 discute o uso de módulos de projeto para prover um contexto no qual os objetos são criados e as atividades realizadas. Discute também como esse contexto pode ser utilizado para a gerência de configuração. Na seção 4, relatamos alguns trabalhos relacionados e apresentamos as conclusões na seção 5.

2 Um Modelo Baseado em Grafo para Evolução de Software

O modelo descrito em Luqi [Luq90] fornece uma descrição formal integrando atividades de evolução com controle de configuração. Ele é composto de dois elementos básicos: componentes de sistema e passos de evolução. Componentes de sistema são vistos como uma coleção estruturada de componentes de software de diferentes tipos, tais como: requisitos, especificações, descrições de projeto, módulos de código-fonte, casos de teste, manuais, etc. Passos de evolução são vistos como coleções estruturadas de atividades envolvidas na produção e transformação desses componentes.

Nesse modelo, uma configuração de software é definida como uma tripla $[G, E, L]$, onde $G = [C, S, I, O]$ é um grafo bipartido direcionado e acíclico, $E \subseteq C$ é um conjunto de componentes exportáveis e L é uma função de rotulação fornecendo identificadores únicos tanto para componentes como atividades. Os nós do grafo representam componentes de software (nós C) e atividades (nós S). Essas duas classes de nós alternam-se em cada caminho do grafo. Os arcos no grafo representam relações de entrada entre componentes e passos de manufaturação ($I \subseteq C \times S$) e relações de saída entre atividades e componentes ($O \subseteq S \times C$). Intuitivamente, os arcos $<o, t>$ ligando componentes a uma atividade t indicam as entradas necessárias à execução de t. Do mesmo modo, os arcos $<t, o>$ indicam os componentes produzidos como saídas de t. O conjunto $E \subseteq C$ de componentes exportáveis corresponde a partes do sistema que podem ser utilizadas em outros desenvolvimentos, ou seja, podem fazer parte de outras configurações.

Exemplo 1. Considere uma organização que possua dois ambientes operacionais distintos: um para Windows95 e outro para UNIX. Nessa organização, alguns programas são gerados para serem executados em ambas as plataformas. Considere que nessa organização um projeto *P1* utilize a especificação *Espec-A* e, por refinamentos sucessivos, gere a especificação detalhada *Espec-A3*. Posteriormente, neste mesmo projeto, são gerados os códigos fonte, objeto e executável que implementam *Espec-A3*. Parte do grafo de configuração e evolução para esse projeto é mostrada na Figura 1(a). Os círculos representam as atividades; e os retângulos, os componentes. Por simplicidade, os vários refinamentos levando a *Espec-A3* não estão contemplados, havendo uma única atividade de Análise que gera *Espec-A3* a partir de *Espec-A*. ❑

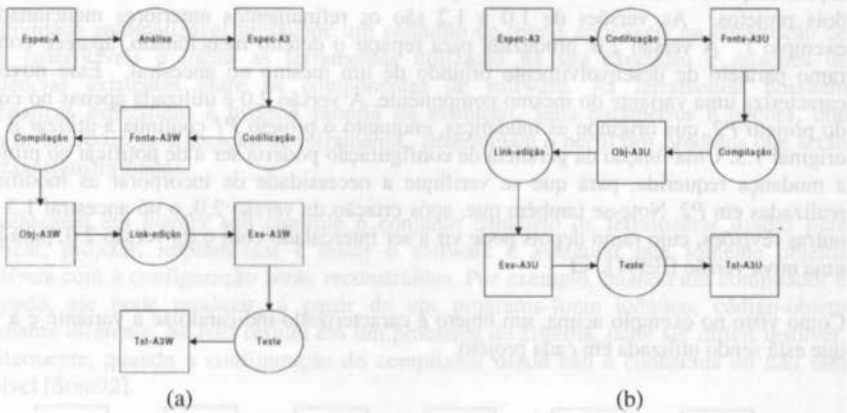

Figura 1 - Grafos de Configuração e Evolução.

Enquanto os itens de configuração (especificações, fontes, objetos, etc.) são mantidos em um repositório único, utilizando estruturas que veremos mais adiante, os grafos de configuração e evolução são associados a projetos.

Exemplo 2. Na mesma organização do exemplo anterior, suponha que um projeto *P2* utilize o mesmo componente *Espec-A3* e que, a partir dele, os respectivos códigos fonte, objeto e executável para a plataforma UNIX sejam produzidos. O grafo para o projeto *P2* é similar ao do projeto *P1* e está exibido na Figura 1(b). Observe que *Espec-A3* utilizada nos dois projetos refere-se a um mesmo componente (a função de rotulação é única para a organização). Os componentes *Fonte-A3U, Obj-A3U, Exe-A3U* e *Tst-A3U* são específicos para o projeto 2. ❑

2.1 Componentes

A evolução dos componentes é controlada através de sua história, que é expressa por meio de um grafo direcionado e acíclico. Os nós do grafo representam versões; os arcos, as mudanças e transformações de cada componente de software.

Um componente posto sob controle inicialmente recebe a versão 1.0. Quando um pedido de mudança para esse componente é autorizado, o desenvolvedor recupera a sua última versão (n) e a inclui em sua área de trabalho. Feitas e testadas as mudanças para o componente, este é armazenado de volta no repositório com um novo número de versão (n+1). Essa nova versão é representada como um novo nó no grafo de versão.

Exemplo 3. Considere que *Espec-A3* usada nos projetos *P1* e *P2*, para derivar seus respectivos códigos-fontes, seja a última de uma série de versões. Durante a fase de teste de *P2*, foi verificado que os resultados apresentados pelo código *Exe-A3U* não correspondiam aos que eram esperados. A causa do erro estava em *Espec-A3*. Esta foi revisada, dando origem a uma nova versão. A Figura 2 abaixo representa a história da evolução da especificação considerada. Assumimos que a versão 1.3 é a versão utilizada inicialmente nos dois projetos. As versões de 1.0 a 1.2 são os refinamentos anteriores mencionados no exemplo 1. A versão 2.0, produzida para reparar o defeito mencionado, aparece como um ramo paralelo de desenvolvimento oriundo de um mesmo nó ancestral. Esse novo ramo caracteriza uma variante do mesmo componente. A versão 2.0 é utilizada apenas no contexto do projeto *P2*, que originou as mudanças, enquanto o projeto *P1* continua a utilizar a versão original 1.3. Uma função da gerência de configuração poderia ser a de notificar ao projeto *P1* a mudança requerida, para que se verifique a necessidade de incorporar as modificações realizadas em *P2*. Note-se também que, após criação da versão 2.0, o nó ancestral 1.3 sofreu outras revisões, cujo ramo depois pode vir a ser intercalado com o da versão 2.0, para formar uma nova versão [Fei91]. ❑

Como visto no exemplo acima, um objeto é caracterizado indicando-se a variante e a versão que está sendo utilizada em cada projeto.

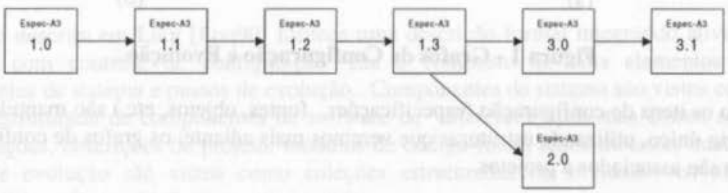

Figura 2 - História da Evolução do Componente Espec-A3.

O modelo em Luqi é definido para tratar apenas objetos que possam ser submetidos ao controle de versões. Nós estendemos esse modelo para incluir, explicitamente, objetos que estavam fora desse controle, os quais denominamos objetos não-transformáveis, e o contexto no qual as atividades são executadas e os componentes produzidos e utilizados. Os objetos não-transformáveis incluem, entre outros, as pessoas e as ferramentas. Aqui trataremos com apenas as ferramentas e atividades utilizadas no processo de desenvolvimento. A determinação do contexto é dada pela utilização de módulos de projetos, como veremos na seção 3.

Os diversos componentes de uma configuração são os itens de configuração de software (ICS), aqui particionados em dois grupos de objetos: os *transformáveis* (ICST) e os *não-transformáveis* (ICST'). Os itens transformáveis são aqueles que podem sofrer transformações em sua estrutura, por exemplo: diagramas, código-fonte, etc. Os não-transformáveis, aqueles cuja estrutura não sofre transformação, incluem: ferramentas e pessoas, entre outros. Os itens transformáveis são particionados em dois conjuntos: os *rederiváveis* (D) e os *não-rederiváveis* (D'). Um objeto rederivável é aquele que pode ser, automaticamente, reconstruído por ferramenta, a partir de outro objeto, sem intervenção humana. Objetos não-rederiváveis são os que não podem ser reconstruídos automaticamente.

2.2 Atividades e Ferramentas

A hierarquia das atividades é representada por uma estrutura em forma de árvore. Uma atividade é *composta*, se puder ser vista como uma coleção de partes relacionadas, e é *atômica* em caso contrário. Na árvore hierárquica, os nós-folha são as atividades atômicas; os nós-raiz, as atividades compostas.

Uma atividade automatizável pressupõe um conjunto de regras aplicáveis na construção de objetos rederiváveis e inclui as ferramentas utilizadas na sua execução e relações de dependências existentes entre os componentes de software. As ferramentas possuem características específicas e história própria de evolução. Seus parâmetros e opções, que variam de acordo com a geração da ferramenta, determinam o perfil dos produtos que serão criados ou transformados.

Uma vantagem em manter sob controle o conjunto completo de ferramentas usadas para especificar, projetar, implementar e testar o software é assegurar que somente objetos compatíveis com a configuração serão reconstruídos. Por exemplo, quando um compilador é modificado, ele pode produzir, a partir de um programa-fonte idêntico, código-objeto ligeiramente diferente. Se isso resulta em um problema de sistema, pode ser difícil resolver, particularmente, quando a configuração do compilador usado não é conhecida ou não está disponível [Som92].

Exemplo 4. O grafo de configuração e evolução para o projeto *P2*, apresentado no exemplo 2, contém na realidade nodos adicionais correspondendo às ferramentas utilizadas na execução de suas atividades, conforme ilustra a Figura 3. A situação já discutida, em que um erro em *Exe-A3U* foi encontrado, cuja origem foi atribuída a falhas em *Espec-A3* requer, naturalmente, que uma nova versão de *Espec-A3* seja produzida. Conseqüentemente, as atividades de codificação, compilação e link-edição terão de ser reexecutadas para produzir objetos compatíveis com a nova versão. O compilador utilizado na produção de *Obj-A3U* possui parâmetros específicos que devem ser reproduzidos na execução da atividade, uma vez que

são frutos de requisitos próprios ao projeto *P2*. Estes parâmetros são recuperados a partir da descrição do componente 'Compilador C'. ❑

O uso de ferramenta como item de configuração e a conseqüente inserção deste componentes no grafo de configuração permitem a recuperação desses objetos no estado em que se encontravam quando da última utilização.

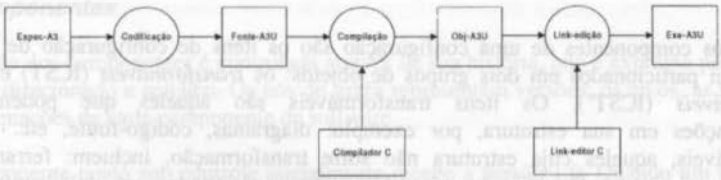

Figura 3 - O Processo de Reconstrução Consistente de Objetos.

Outros tipos de componentes não-transformáveis — e não apenas ferramentas — podem ser objeto de controle e incluídos no grafo de configuração e evolução. Este é o caso, por exemplo, de pessoas (usuários e analistas) associadas às atividades de análise.

A modelagem dessas informações, através de objetos específicos, permite maior flexibilidade do que, por exemplo, se as mesmas fossem atribuídas a arcos do grafo de configuração. As informações referentes aos objetos não-transformáveis, que incluem não apenas ferramentas, podem ser as mais variadas possíveis, sendo, desta forma, melhor modeladas através de classes específicas.

2.3 Gerenciando a Configuração

A gerência de configuração e o controle da evolução são de fato exercidos através de procedimentos que permitem a manutenção e extração de informações acerca dos objetos e atividades, incluindo o estado em que os mesmos se encontram, a consistência entre eles e os vários relacionamentos existentes. O modelo proposto por Luqi permite a definição de relações entre componentes e atividades que podem ser utilizadas para automatizar ou auxiliar a execução e a análise dos procedimentos relativos à gerência de configuração. Nossas extensões possibilitam uma maior granulação na definição dos componentes de uma configuração. Como resultado, podemos restringir o domínio e a imagem destas relações bem como definir novas relações, de acordo com a classe de componentes que queremos considerar. A seguir apresentamos alguns relacionamentos típicos da gerência de configuração.

A relação de dependência Dep^+ é definida como o fechamento transitivo da relação $Dep = (I \cup O)$ induzida pelos arcos do grafo de configuração. Essa relação formaliza a dependência entre as várias atividades do processo de desenvolvimento. Dessa maneira, a atividade t_1 depende da atividade t_2 se $(t_1, t_2) \in Dep^+$, o que equivale à existência de um caminho ligando t_1 a t_2 no grafo de configuração. Observe que, por ser uma relação induzida pelo grafo de configuração, a dependência representada por Dep^+ indica os relacionamentos que de fato existem em um dado projeto. Essa relação pode ser empregada para recuperar a

seqüência de atividades que devem ser reexecutadas no caso de modificações em um dado objeto.

A relação $Deriva \subseteq ICST \times D$ é definida entre objetos transformáveis e objetos rederiváveis. É utilizada para representar as possíveis transformações de objetos em outros. Um exemplo é a compilação de um código-fonte em código-objeto. Essa relação também é induzida pelo grafo de configuração: dados dois objetos o_1 e o_2, $(o_1, o_2) \in Deriva$, se e somente se, existe uma atividade t tendo o_1 como entrada e o_2 como saída, *i.e.*, para um grafo $G = [C, S, I, O]$, teremos que ter $(o_1, t) \in I$ e $(t, o_2) \in O$.

A relação $Usa \subseteq D' \times D'$, definida entre objetos não rederiváveis, é utilizada para representar situações de dependência entre objetos. Com, ela pode-se representar dependências estruturais entre objetos que não podem ser automaticamente gerados e, por isso mesmo, necessitam de controle rigoroso para manter a consistência entre eles. Um exemplo é a dependência representada pela declaração "include" da linguagem C.

Adicionalmente, incluímos a relação $Executa \subseteq ICST' \times T$ entre objetos não-transformáveis e atividades, indicando os objetos utilizados na execução das mesmas. Existe um arco ligando uma ferramenta f a uma atividade t sempre que $(f, t) \in Executa$. Esta é a relação utilizada, por exemplo, para associar ferramentas às atividades que delas fazem uso. Outros tipos de objetos não-transformáveis podem naturalmente ser incluídos. Este é o caso de pessoas e documentos preexistentes, como manuais e legislações.

Das relações acima, apenas Dep^+, *Deriva* e *Executa* podem ser determinadas a partir do grafo de configuração. A relação *Usa* deve ser mantida de forma independente, embora possa para alguns tipos de objetos ser automaticamente determinada, como ocorre no aplicativo Make [Fel79].

Com essas relações pode-se definir outras, necessárias à gerência da configuração. Por exemplo, para determinar todos os objetos que possivelmente possam ser afetados por modificações em um objeto não rederivável, pode-se utilizar o fechamento reflexivo e transitivo de *Usa*. Formalmente, para quaisquer dois objetos o_1 e o_2, temos que o_1 *Afeta* $o_2 \Leftrightarrow o_2$ Usa^* o_1. Da mesma forma, pode-se definir o conjunto de todas as atividades afetadas por mudanças efetuadas em um dado objeto o como $Ativ\_afetadas(o) = \{t \mid (o, t) \in Dep^+\}$. Para cada atividade, as ferramentas necessárias e o estado no qual as mesmas devem ser utilizadas são determinadas através da relação *Executa*.

O modelo também permite automatizar certos aspectos da evolução do processo, pela associação de um indicador de estado (e.g., *proposto*, *aprovado*, *determinado*, *completado* e *abandonado*) às atividades. Assim, por exemplo, quando uma atividade muda de *proposto* para *aprovado*, todas as subatividades mudam para o mesmo estado.

3 Módulos de Projeto

Os módulos de projeto definem o escopo no qual os diversos objetos, ferramentas e atividades podem ser criados, utilizados e executados. A noção de objeto em um módulo de projeto é abrangente, envolvendo não apenas os objetos de software, como diagramas, especificações e

código, mas também pessoas e representações multimídia, como transcrições audiovisuais de entrevistas, etc.

Um processo de desenvolvimento é definido utilizando-se os módulos de projeto em uma especificação do processo de desenvolvimento. A partir dessa especificação, processos reais podem ser implementados e conduzidos. Suponha que, em uma dada especificação do processo de desenvolvimento, *ProcEsp*, todas as atividades de desenho estejam definidas em um módulo denominado *Desenho*. Para que uma dessas atividades seja executada em um desenvolvimento real, segundo *ProcEsp*, um módulo deve ser criado (ou aberto, caso já exista). Esse módulo satisfará todas as propriedades e restrições definidas em *Desenho*, e.g., as pré e pós-condições válidas para cada atividade.

Para um dado processo de desenvolvimento, existem os módulos formais, definidos na especificação do processo, e os módulos reais, criados quando da implementação e condução do processo. Chamamos ambos os tipos de módulos de projeto. O contexto será suficiente para indicar quando falamos da especificação do processo ou dos módulos reais que provêem um contexto no qual as atividades podem ser, de fato, executadas.

Os módulos de projeto podem, na especificação do processo de desenvolvimento, ser parametrizados. Dessa forma, vários processos podem ser instanciados com base em uma única definição do processo visando a atender às circunstâncias especiais do desenvolvimento.

Considere o módulo *Des* abaixo, cuja definição inicia com a palavra reservada *pmod* e termina em *endp*. Nesse módulo, o parâmetro *X* é definido através da teoria *DiagrTh* que estabelece a estrutura e as propriedades que os argumentos devem satisfazer. No exemplo em questão, *DiagrTh* define apenas as classes *DiagrO*, *DiagrORev* e *DiagrF*, sendo *DiagrORev* subclasse de *DiagrO*. O módulo *Des* também importa o módulo *Analise*, onde as classes *Analista* e *Espec* são definidas.

```
pth DiagrTh is
   classes DiagrO DiagrORev DiagrF .
   subclass DiagrORev < DiagrO .
endth

pmod Des [ X :: DiagrTh ] is
   importa Analise .
   atv desenho : Analista DiagrF Espec -> DiagrO .
   atv revisao : DiagrF DiagrO -> DiagrORev .
endp
```

Desse modo, *Des* especifica que a atividade *desenho* é realizada por um analista (objeto da classe *Analista*) com base em uma especificação (objeto da classe *Espec*), utilizando uma ferramenta de desenho (objeto da classe *DiagrF*), e produz um modelo da especificação em forma de diagrama (objeto da classe *DiagrO*). Também é especificado que a atividade *revisao* é realizada, através de uma ferramenta de desenho, e produz um diagrama revisado (objeto da classe *DiagrORev*).

Em um ambiente onde se utilize o método Booch para desenho orientado a objeto, o módulo *DesBooch* abaixo pode ser definido para especificar de modo preciso as classes de componentes e as ferramentas permitidas.

```
pmod DesBooch is
   classes DiagrBooch DiagrBoochRev OODesigner .
   subclass DiagrBoochRev < DiagrBooch .
endp
```

Para um processo de desenvolvimento específico, o módulo *Desenho* pode ser definido através da instanciação de *Des* por *DesBooch*.

```
pmod Desenho is
   Des [ DesBooch ] .
endp
```

O resultado é o equivalente à definição de *Desenho*, especificando-se as classes de objetos e ferramentas particulares ao método *Booch*.

```
pmod Desenho is
   importa Analise .
   classes DiagrRev < DiagrBooch .
   atv desenho : Analista OODesigner Espec -> DiagrBooch .
   atv revisao : OODesigner DiagrBooch -> DiagrRev .
endp
```

Os módulos criados no decorrer de um processo de desenvolvimento formam uma hierarquia determinada por um grafo, onde os arcos representam as relações de importação entre eles. No exemplo acima, os módulos *Analise* e *Desenho* são relacionados, visto que um é importado pelo outro.

As diversas atividades só podem ocorrer no contexto de um módulo, i.e., o responsável por uma atividade deve abrir um módulo existente (ou criar um novo módulo) que contenha a atividade a ser realizada. Os objetos de entrada, bem como aqueles produzidos como saída, e as ferramentas utilizadas devem satisfazer ao que está especificado para o módulo em questão. Dessa forma, cada objeto é criado e utilizado no contexto de um módulo específico, havendo um relacionamento entre módulos de projeto, componentes e atividades de uma configuração.

Os detalhes da sintaxe e semântica dos módulos de projeto fogem do escopo deste artigo. A especificação do processo de desenvolvimento é, na verdade, uma especificação algébrica orientada a objeto utilizando uma extensão da linguagem FOOPS [Rap92]. Módulos de projeto são baseados nos módulos FOOPS. Uma aplicação desses módulos em um contexto semelhante ao aqui descrito é fornecida em [Pin96] e uma discussão detalhada dos módulos de projeto para o problema de rastreamento de requisitos é apresentada em [Pin97]. Para a gerência de configuração é suficiente esclarecer que, dado um módulo de projeto, pode-se obter o conjunto dos objetos que nele foram criados e utilizados, bem como as atividades nele definidas, como se verá adiante.

3.1 Gerência da Configuração Utilizando Módulos de Projeto

O contexto determinado pelos módulos de projeto pode ser 'transferido' para o grafo de configuração, no sentido de que qualquer objeto do grafo é criado ou utilizado a partir de um módulo. O mesmo ocorre com as atividades. Formalizamos essa noção a partir do conceito de grafo de configuração induzido por um módulo.

Definição. Dados um módulo M e um grafo de configuração $G = [C, A, E, S]$, denominamos grafo de configuração induzido por M o subgrafo $G|_M = [C_M, A_M, E_M, S_M]$, onde $C_M \subseteq C$ é o conjunto de objetos criados ou utilizados em M, $A_M \subseteq A$ é o conjunto das atividades executadas em M e $E_M \subseteq E$ e $S_M \subseteq S$ são as relações de entrada e saída entre componentes e atividades restritas aos componentes e atividades em C_M e A_M respectivamente. ❑

O subgrafo correspondente a um conjunto de módulos pode, de forma semelhante, ser facilmente determinado. Dessa forma, a maioria das relações determinadas a partir do grafo de configuração pode ser restrita a um módulo ou conjunto de módulos do processo de desenvolvimento. Denotamos por $rel|_M$ a restrição da relação *rel* ao (conjunto de) módulo M.

Exemplo 5. Restringindo *Usa* ao Módulo M, teremos:

$Usa|_M = \{(c_1, c_2) \mid c_1, c_2 \in C_M \wedge c_1\ Usa\ c_2\}$

A partir desse ponto, outras relações podem ser definida da forma usual

$c_1\ Afeta|_M\ c_2 \Leftrightarrow c_2\ Usa|_M^*\ c_1$ ❑

Também temos que, para um módulo M:

- *U-comp(M)* é o conjunto de objetos utilizados em M.
- *C-comp(M)* é o conjunto de objetos criados em M.
- *E-ativ(M)* é o conjunto de atividades que podem ser executadas em M.

A partir desses conjuntos, pode-se determinar os módulos afetados por uma mudança na configuração do projeto. Assim, suponha que um determinado objeto tenha sido identificado no grafo de configuração como necessitando ser revisto. A partir desse objeto, pode-se percorrer a hierarquia de módulos e verificar aqueles em que o objeto foi utilizado. Essa integração entre gerência de configuração e do processo de desenvolvimento permite que problemas na configuração sirvam de entrada para análise dos processos de desenvolvimento e vice-versa.

Prover um contexto para a gerência de configuração é um grande auxílio para os projetos de desenvolvimento de elevada complexidade, onde existem inúmeros objetos e atividades. Nesses casos, é conveniente ter-se mecanismos para analisar os impactos de mudanças circunscritos a determinadas fases do desenvolvimento (ou módulos de projeto). Outros dois benefícios do uso de módulos de projeto na gerência da configuração são a integração entre processos de desenvolvimento de um modo geral e a manutenção da consistência entre os diversos objetos de uma configuração em particular. A análise do processo de desenvolvimento só é possível a partir de processos bem definidos, de preferência, de modo formal. Os módulos de projeto utilizados para a especificação de processos de desenvolvimento permitem essa análise. Em função de modificações realizadas no processo

de desenvolvimento, atividades podem ser reexecutadas. Por exemplo, suponha que em algum momento seja verificado que as atividades de inspeção utilizadas para testar objetos de software precisem ser reformuladas. Suponha ainda que essas atividades sejam definidas em um módulo *Inspec* utilizado (importado) por diversos módulos como *InspecDes* (para inspeção de desenho) e *InspecUnid* (para inspeção de unidades de software). Após a determinação das mudanças nas atividades definidas nos respectivos módulos, pode-se recorrer ao grafo de configuração restrito a esses módulos para verificar se algumas dessas atividades foram efetivamente realizadas e executá-las novamente, se for o caso.

Os vários relacionamentos entre as diversas estruturas do nosso modelo estão indicados esquematicamente na Figura 4 a seguir.

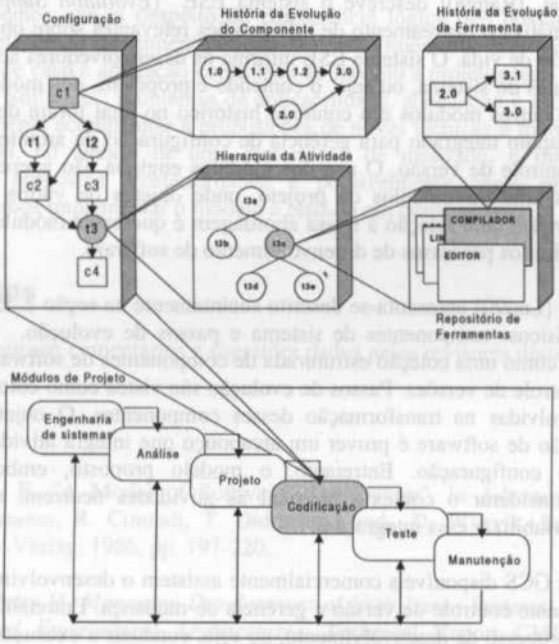

Figura 4 - Diferentes Perspectivas de Módulos de Projeto.

4 Trabalhos Relacionados

Lin e Reiss [Lin95] apresentam um modelo no qual módulos de software são utilizados como unidades para o controle da configuração. É um avanço em termos dos modelos tradicionais que utilizam arquivos. É proposta uma estrutura para ambientes de programação que permite a gerência de configuração diretamente dos módulos de código-fonte. Para tanto, funções e classes de programas são representadas por objetos chamados *unidades de software*, que contêm conjuntos de *operações*, *atributos de dados* e *ligações*. Entre as operações, destacam-se: editar, compilar, link-editar, revisar, congelar. Quanto às ligações, há duas

classes: ligações *usa* e *parte-de*. Cada unidade de software tem um conjunto de ligações *usa* apontando para as unidades de software que representam seus submódulos e uma ligação opcional *parte-de* apontando para a unidade que representa seu módulo-pai. Uma unidade de software encapsula todo o código-fonte, objetos derivados, documentações e constrói um *script* da correspondente função ou classe e provê um conjunto de operações para programadores. Os programadores gerenciam o software invocando as operações das unidades de software, em vez de usar ferramentas que estão separadas do repositório de dados. Assim, a gerência ocorre em paralelo com a modularização do código-fonte. Embora permitindo o controle de algumas atividades, estas são encapsuladas como operações nas unidades de software. Também os módulos utilizados referem-se apenas a módulos de programas. O processo de desenvolvimento não é integrado à gerência de configuração.

Ramamoorthy et al. [Ram90] descreve o sistema ESE (*Evolution Support Environment System*), que automatiza o rastreamento de informações relevantes sobre objetos de software durante todo o ciclo de vida. O sistema ESE informa os desenvolvedores acerca das relações entre os componentes do sistema, ou seja, o conteúdo e propósito dos módulos no sistema e suas relações com outros módulos e o contexto histórico no qual foram desenvolvidos. Seu objetivo é prover apoio integrado para gerência de configuração da arquitetura do software, ciclo de vida e controle de versão. O uso dos módulos engloba não apenas os módulos de programação, mas também módulos de projeto, onde objetos de vários tipos podem ser agrupados. A diferença com relação à nossa abordagem é que esses módulos não podem ser utilizados para definir os processos de desenvolvimento de software.

O modelo de Luqi [Luq90] apresenta-se descrito sucintamente na seção 2. Ele é composto de dois elementos básicos: componentes de sistema e passos de evolução. Componentes de sistema são vistos como uma coleção estruturada de componentes de software que podem ser submetidos ao controle de versões. Passos de evolução são vistos como coleções estruturadas de atividades envolvidas na transformação desses componentes. O objetivo principal do modelo de evolução de software é prover um arcabouço que integra atividades de evolução com controle de configuração. Entretanto, o modelo proposto, embora reconheça a necessidade de considerar o contexto no qual as atividades ocorrem, não apresenta os mecanismos para viabilizar essa integração.

As ferramentas de GCS disponíveis comercialmente assistem o desenvolvimento de software provendo basicamente controle de versão e gerência de mudança. Entretanto, são deficientes em disciplinar o processo de desenvolvimento, ou seja, conduzir a execução de atividades, a criação e uso de objetos de software em conformidade com as exigências de cada fase do ciclo de vida. Além do mais, as ferramentas estão limitadas a relatar o estado de componentes individuais em isolado, não levando em conta o contexto em que a evolução ocorre, o qual deve incluir, entre outras, as relações técnicas e sociais, sobretudo as pessoas participantes e as razões para as transformações sofridas.

5 Conclusões

O modelo original de Luqi [Luq90] trata apenas questões relativas a objetos que podem ser submetidos ao controle de versões. Estendemos seu modelo com novas classes de objetos e incorporamos o conceito de módulos de projeto. Essa extensão possibilita determinar o contexto no qual o processo de desenvolvimento é conduzido, envolvendo: pessoas,

ferramentas empregadas e objetos criados e utilizados nas atividades do desenvolvimento. Uma vantagem do modelo por nós proposto é considerar a arquitetura do processo de desenvolvimento na identificação, auditoria, controle e relato da situação dos itens de software. Isso permite aos gerentes por exemplo, conhecer como os esforços do desenvolvimento estão distribuídos nas diversas fases do ciclo de vida do software. Além do quê, restringir a criação de componentes à fase onde previamente foram declarados.

O modelo aqui apresentado descreve uma estrutura formal que facilitará a implementação de ferramentas de apoio às atividades relativas à gerência de configuração e evolução. Em sua elaboração, tomamos a decisão de somar esforços a pesquisas já existentes [Luq90, Nar87, Bor86, Hei88, Mos89]. Como benefício, tivemos a facilidade na identificação dos aspectos críticos relativos à gerência de configuração e aos processos de evolução de software.

Entretanto, entendemos que, a despeito da consistência lógica que possamos assegurar, a validação do modelo como instrumento útil ao desenvolvimento de software se dará apenas através de estudos de caso e aplicação controlada do modelo a projetos reais. Tal é a natureza da "Engenharia de Software": problemas de aplicabilidade, adequação e escala, entre outros, devem necessariamente acompanhar a proposições de novas técnicas e modelos. Atualmente estamos detalhando, no âmbito de uma dissertação de mestrado, o funcionamento dos mecanismos aqui propostos. Posteriormente e possivelmente como novo trabalho de mestrado, pretendemos implementar o modelo e testá-lo em casos práticos.

Agradecimentos

Agradecemos os valiosos comentários e sugestões dados pelos revisores deste artigo.

Referências

[Bor86] Borison, E. "A Model of Software Manufacture", in *Advanced Programming Environments*, R. Conradi, T. Didriksen and D. Wanvik, Eds. New York: Springer-Verlag, 1986, pp. 197-220.

[Fei88] Feiler, Peter H. *Managing Development of Very Large Systems: Implications for Integrated Environment Architectures*, Technical Report CMU/SEI-88-TR-11, Software Engineering Institute, Carnegie Mellon University, May 1988.

[Fei91] Feiler, Peter H. *Configuration Management Models in Commercial Environments*, Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, March 1991.

[Fel79] Feldman, Stuart I. "MAKE – a Program for Maintaining Computer Programs", *Software Practice and Experience*, Vol. 9, nº 3, pp. 255-265, March 1979.

[Hei88] Heimbigner D. and Krane S. "A Graph Transform Model for Configuration Management Environments", in *Proc. ACM Software Eng. Notes/SIGPLAN Notices Software Engineering Symp. Practical-Software Development Environments*, 1988, pp. 216-225.

[Hum90] Humphrey, Watts S. *Managing the Software Process*, Software Engineering Institute, Addison-Wesley Publishing Company, 1990.

[Lin95] Lin, Yi-Jing and Reiss, Steven P. *Configuration Management in terms of Modules*, Technical Report CS-94-95, Department of Computer Science, Brown University, Providence, RI 02912, USA, May 1995.

[Luq89] Luqi. "Software Evolution through Rapid Prototyping", *IEEE Computer*, Vol. 22, nº 5, pp. 13-25, May 1989.

[Luq90] Luqi. "A Graph Model for Software Evolution", *IEEE Transactions on Software Engineering*, Vol. 16, nº 8, pp. 917-927, August 1990.

[Mos89] Mostov, I., Luqi and Hefner, K. *A Graph Model of Sofware Maintenance*, Dep. Comput. Sci., Naval Postgraduate School, Tech. Rep. NP552-90-014, August 1989.

[Nar87] Narayanaswamy K. and Scacchi W. "Maintaining Configurations of Evolving Software Systems", *IEEE Transactions on Software Engineering*, Vol. SE-13, nº 3, pp. 324-334, March 1987.

[Pin96] Pinheiro, Francisco A. C. and Goguen, Joseph A. "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, Vol. 13, nº 2, pp 52-64, March 1996.

[Pin97] Pinheiro, Francisco A. C. *Design of a Hyper-Environment for Tracing Object-Oriented Requirements*, Ph.D. Thesis, Oxford University Computing Laboratory, Oxford University, Oxford, 1997.

[Ram90] Ramamoorthy, C. V, Usuda, Yutaka, Prakash, Atul e Tsai, W. T. "The Evolution Support Environment System", *IEEE Transactions on Software Engineering*, Vol. 16, nº 11, pp. 1225-1234, November 1990.

[Rap92] Rapanotti, Lucia and Socorro, Adolfo. *Introducing FOOPS*, Technical Report PRG-TR-28-92, Programming Research Group, Oxford University Computing Laboratory, Oxford, 1992.

[Som92] Sommerville, Ian. *Software Engineering*, 4ª edition, Addison-Wesley Publishing Company, 1992.