Formalização de Conceitos em Gerência de Configurações

por

Paulo Sérgio C. de Alencar
Departamento de Ciência da Computação
Universidade de Brasília
Campus Universitário - Asa Norte
Brasília - DF - 70090 - Brasil

e

Carlos José Pereira de Lucena
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225
Rio de Janeiro - RJ - 22453 - Brasil

Resumo

Neste artigo apresentamos uma abordagem em lógica do processo de evolução de configurações de sistemas de software que pode ser visto como um processo de transformações em "ponto grande" aplicado à descrições arquiteturais de sistemas de software. Neste sentido, uma descrição formal generalizada de arquiteturas de sistemas de software do ponto de vista de sistemas evolutivos de software é apresentada. A semântica do processo de mudança de estados de configuração de software (tomados como grafos de estrutura de software) é definida através de um formalismo lógico que envolve teorias representando estes estados e suas mudanças quando afetados por ações . O formalismo lógico adotado nos permite descrever e raciocinar dedutivamente sobre mudanças na estrutura, na interface e na funcionalidade dos componentes da descrição arquitetural do sistema de software. O raciocínio dedutivo sobre aspectos funcionais destas descrições é conseguido através da interação de um provador para o formalismo metalógico proposto e um provador de teoremas para raciocínio dedutivo sobre programas sequenciais.

1 Introdução

O processo de configuração de software, que é crítico para a manutenção de grandes sistemas de software, pode ser definido como o processo de gerenciamento de mudanças destes sistemas. Versões de software são criadas através da realização de alterações em software existente e talvez o mais crítico papel da equipe de gerenciamento de configurações é o controle de mudanças. Mudanças descontroladas rapidamente levam ao caos e o gerenciamento

de configurações trata do impacto destas mudanças, o seu custo e da decisão sobre a viabilidade e a oportunidade de serem aplicadas. Além disso, o desenvolvimento de sistemas de software de larga-escala irá mudar os padrões de trabalho de organizações de desenvolvimento de sistemas. Esta mudança esperada é em parte devida à aplicação de métodos formais e baseados em conhecimento para a solução de problemas de engenharia de software e para a construção de ferramentas avançadas de software que suportem subfases específicas do ciclo de vida de sistemas de software. Isto está em concordância com as características esperadas dos sistemas para a chamada quarta geração tecnológica do gerenciamento de configurações e controle de versões ([Ambriola, Bendix, Ciancarini 90]).

Para tratar o problema da evolução de configurações de sistemas de software, torna-se importante fornecer uma base formal para as noções existentes de arquitetura de sistemas, noções formais de restrições sobre suas configurações e mecanismos para controlar a evolução de suas descrições estruturais e funcionais. A solução proposta neste artigo adota um formalismo lógico que nos permite descrever e raciocinar sobre mudanças na estrutura, sobre as interdependências modulares vistas como as dependências entre as interfaces dos módulos comunicantes (mudanças nas interfaces) e sobre os aspectos funcionais de uma descrição arquitetural de um sistema de software. Uma extensão modal (com ações) da lógica polisortida de primeira ordem é usada para capturar os aspectos de mudanças destas descrições de alto nível de abstração tratadas como teorias na abordagem lógica proposta. O raciocínio dedutivo sobre aspectos funcionais destas descrições é obtido através da interação de um provador de teoremas para a metalógica proposta e um provador de teoremas para deduções sobre programas sequenciais. Na nossa abordagem as alterações das descrições de software podem ser vistas como transformações em ponto grande que são enunciadas por meio das características deônticas das ações .

Este artigo está estruturado como segue. Primeiramente, na seção 2 apresentamos uma descrição formal generalizada de arquiteturas de sistemas de software do ponto de vista de sistemas evolutivos de software. Estados de configuração de software são definidos através da noção de grafos de estrutura de software e suas sequências de desenvolvimento. A noção de estados de configuração válidos é motivada. Na seção 3 apresentamos a abordagem lógica adotada e mostramos como os estados de configuração de software são representados neste formalismo. A semântica do processo de mudança dos estados de configuração é descrita através desta base lógica. Indicamos também um método de raciocínio dedutivo através do qual os sistemas evolutivos de software são tratados. É um método de prova geral baseado em cálculo de tableau para lógica de ações de primeira ordem com domínio constante com ações arbitrárias (constantes e não constantes) e algumas características deônticas. Este método de prova pode ser visto como uma realização da semântica do processo evolutivo de configurações de sistemas de software. Na seção 4 continuamos a descrição da abordagem lógica mostrando como mudanças estruturais e de interconexão ocorrem na base lógica apresentada e descrevendo como se especificar e raciocinar dedutivamente sobre estes tipos de mudança de configurações de sistemas de software. Extendemos também a abordagem lógica para investigar o efeito de mudanças de natureza funcional nos estados de configuração . Finalmente, na seção 5 apresentamos nossas conclusões.

2 Os Grafos de Estrutura de Software

O processo de mudança de descrições (arquiteturais) de sistemas de software pode ser visto simplificadamente como segue. Primeiramente, começamos com uma descrição inicial da arquitetura do sistema de software SS_0 que evolui através da realização de uma mudança particular r_0 para uma descrição SS_1. Esta descrição de software intermediária então evolui para descrições SS_i, $(i = 2,3,\ldots)$ através da realização de sucessivas mudanças r_i, $(i = 2,3,\ldots)$, até que uma descrição corrente do sistema de software seja alcançada.

Cada **descrição arquitetural** SS_k $(k = 1,\ldots)$ pode ser vista como um estado da configuração do sistema de software. Estes **estados de configuração** serão definidos posteriormente como grafos de estrutura de software, sendo representados por grafos acíclicos dirigidos cujos nodos folha são famílias modulares e os nodos internos são famílias de subsistemas ([Narayanaswamy 87a,87b], [Tichy 82]). Com isso, essencialmente, buscamos descrever como os componentes de um sistema de software e suas diferentes configurações podem ser especificados com o objetivo de serem mantidos.

Uma **família modular** é um conjunto de arquivos fonte que partilham as mesmas propriedades de interface que caracterizam a família. Um membro (módulo) particular da família modular é denominado uma versão daquela família. Cada um dos módulos do sistema de software fornece um conjunto de recursos, como tipos de dados, funções , procedimentos, variáveis, etc., para uso em outros módulos e pode requerer alguns recursos que são fornecidos por outros módulos. Uma **família de subsistemas** é um conjunto de configurações que satisfazem uma particular interface de especificação . Uma configuração é definida como um conjunto de módulos (ou outras configurações) que podem ser combinados. Uma particular configuração que é um membro de uma família de subsistemas é chamada uma versão daquela família. Distinguimos aqui entre versões e revisões. Uma versão resulta de mudanças que requerem que um novo módulo de uma família seja criado, enquanto uma revisão resulta de alterações sucessivas de um arquivo fonte que não leva à criação de um novo membro da família modular.

A descrição do tipo de informação que captura as interdependências entre as interfaces dos componentes dos sistemas de software bem como a arquitetura dos sistemas de software enunciados segundo uma formalização generalizada são dados a seguir. Basicamente, esta descrição tem que conter a especificação das famílias modulares e das famílias de subsistemas. Cada uma das famílias modulares tem versões e cada versão é realizada por um arquivo fonte e todas as suas revisões. Além disso, cada família de subsistemas é um conjunto de configurações onde cada configuração consiste de uma lista de componentes que podem ser módulos ou outras configurações .

Definição
Um **grafo de estrutura de software** é denotado pela tupla

$$SS = < SG, SR, SV, SC >,$$

onde
a) SG é um grafo de estrutura.
Este componente da descrição arquitetural do sistema de software contém informações sobre

os nodos e as arestas do grafo e os nomes de cada um dos nodos.
b) SR é um grafo de estrutura extendido aos recursos.
Este componente da descrição caracteriza os recursos e os tipos dos recursos fornecidos e requeridos por cada um dos nodos do grafo.
c) SV é um grafo de estrutura extendido às versões.
Este componente da descrição arquitetural caracteriza primeiramente se o nodo é uma família de subsistemas ou uma família modular. Caso o nodo seja uma família modular, estão descritas aqui as propriedades funcionais apresentadas de modo não procedural e independente da implementação em uma especificação de interface abstrata de cada recurso que é uma operação (e.g. procedimentos), as várias versões pertencentes à família modular e suas implementações associadas, os recursos fornecidos e requeridos por cada uma das versões e uma descrição das propriedades funcionais apresentadas em uma especificação de interface concreta de cada recurso que é uma operação . Técnicas de modelo abstratas, e.g. VDM, e asserções de pré- e pós-condições em um estilo do tipo Hoare podem ser usadas na especificação de interface abstrata e concreta dos recursos funcionais, respectivamente.
d) SC é um grafo de estrutura extendido às configurações .
Este último grafo componente da descrição basicamente caracteriza para os nodos que são famílias de subsistemas um conjunto de configurações , onde cada uma destas configurações contém recursos fornecidos e requeridos de outros componentes da descrição arquitetural.

Consideremos, por exemplo, o seguinte diagrama hierárquico que caracteriza um grafo de estrutura simplificado de software contendo duas famílias de subsistemas ($fs.i$; $i = 1,2$) e quatro famílias de módulos ($fm.j$; $j = 1,2,3,4$) conforme a figura abaixo:

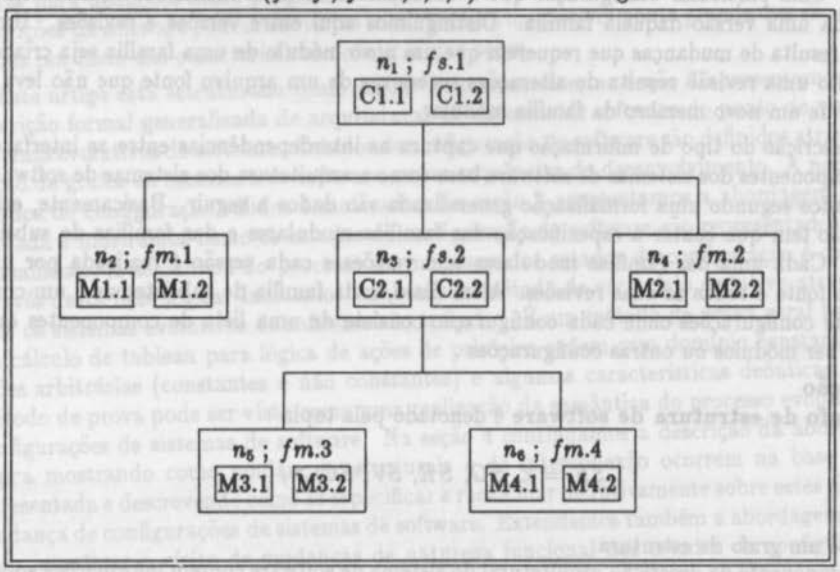

Fig. 1: Grafo de Estrutura de Software

Neste grafo de estrutura de software que contém os nodos $N = \{n_1, \ldots, n_6\}$ e seus nomes, os componentes (membros ou versões) das famílias de subsistemas são denotados por $Ci.j$, onde C indica que este membro familiar é uma configuração , i é um inteiro relacionado à família $fs.i$ que contém $Ci.j$ e j é o número da configuração dentro da família considerada. Do mesmo modo, os membros de famílias modulares $fm.i$ são denotados por $Mi.j$, onde M indica que este membro familiar é uma versão modular, i é um inteiro relacionado à família modular $fm.i$ que contém $Mi.j$ e j é o número da versão modular dentro da família considerada.

De acordo com a descrição formal para sistemas de software adotada acima, os nodos de um grafo de estrutura podem ser uma família de subsistemas ou uma família de módulos. Fornecemos no que segue as descrições destes dois tipos de nodos.

A figura abaixo ilustra o detalhamento da informação associada aos nodos referentes às famílias modulares $fm.3$ e $fm.4$ da figura 1. Note que a família $fm.3$ possue as duas versões modulares $M3.1$ e $M3.2$ com implementações $I3.1$ e $I3.2$, respectivamente.

$S_f(n_5) = fm$; $S_n(n_5) = fm.3$	
$Pr(n_5)$; $Rq(n_5)$	
$AIS(n_5)$	
$M_V(n_5) = \{M3.1, M3.2\}$	
$M3.1$	$M3.2$
$Prv(M3.1)$ $Rqv(M3.1)$	$Prv(M3.2)$ $Rqv(M3.2)$
$CIS(M3.1)$	$CIS(M3.2)$
$M_I(M3.1) = I3.1$	$M_I(M3.2) = I3.2$

$S_f(n_6) = fm$; $S_n(n_6) = fm.4$	
$Pr(n_6)$; $Rq(n_6)$	
$AIS(n_6)$	
$M_V(n_6) = \{M4.1, M4.2\}$	
$M4.1$	$M4.2$
$Prv(M4.1)$ $Rqv(M4.1)$	$Prv(M4.2)$ $Rqv(M4.2)$
$CIS(M4.1)$	$CIS(M4.2)$
$M_I(M4.1) = I4.1$	$M_I(M4.2) = I4.2$

Fig. 2: Famílias de Módulos

Se o componente da estrutura de software é uma família modular, a seguinte informação é associada com o seu nodo m correspondente:

a) $S_f(m) = fm$ (família modular);

b) $S_n(m)$: nome da familia modular;

c) $Pr(m)$ e $Rq(m)$ (recursos fornecidos e requeridos);

d) $AIS(m)$ denota a especificação abstrata de interface. Esta especificação é usada para se garantir que a especificação concreta de cada uma das versões de uma família modular satisfaz a sua especificação abstrata;

e) $M_V(m) = \{v_1, v_2, \ldots, v_n\}$ tal que os v_i's são membros da família modular ou versões;
f) para cada versão modular v_k existe uma realização desta versão tal que $M_I(v_k) = i_k$;
g) para cada versão modular v_k existe uma especificação concreta de interface que nos dá os conjuntos de recursos fornecidos e requeridos pela versão modular, denotados respectivamente por $Prv(v_k)$ e $Rqv(v_k)$;
h) para cada versão modular v_k existe um conjunto de propriedades funcionais ou uma especificação concreta de interface denotada por $CIS(v_k)$. Usa-se esta especificação concreta quando se tem que mostrar que a implementação de cada versão está correta em relação à (ou satisfaz) sua especificação concreta.

Se o componente da estrutura de software for uma família de subsistemas, a seguinte informação é associada com o seu nodo s correspondente:
a) $S_f(s) = fs$ (família de subsistemas);
b) $S_n(s)$: nome da família de subsistemas;
c) o conjunto de recursos fornecidos e requeridos pela família de subsistemas: $Pr(s)$ e $Rq(s)$;
d) $C_S = \{ c_1, c_2, \ldots, c_n \}$ denota o conjunto de configurações da família de subsistemas;
e) para cada configuração c_k existe um conjunto $M_C(c_k)$ de componentes desta configuração , i.e., um conjunto de tuplas $< S_n(k'), fm >$ onde o primeiro elemento é o nome de um módulo ou uma família de subsistemas e o segundo elemento mf da tupla denota um membro familiar (este membro familiar pode ser uma versão modular v se $S_f(k') = fm$ ou pode ser uma configuração c se $S_f(k') = fs$);
f) para cada configuração c_k existem dois conjuntos de recursos fornecidos e requeridos por esta configuração e denotados por $Prc(s)$ e $Rqc(s)$, respectivamente.

Ilustramos através da figura abaixo o detalhamento da informação associada aos nodos referentes às famílias de subsistemas $fs.1$ e $fs.2$ da figura 1. Note que a família $fs.1$ possue as duas configurações $C1.1$ e $C1.2$. Os componentes destas configurações $C1.1$ e $C1.2$ são definidos respectivamente por:

$$M_C(C1.1) = \{ < fm.1, M1.1 >, < fs.2, C2.1 >, < fm.2, M2.1 > \},$$

$$M_C(C1.2) = \{ < fm.1, M1.2 >, < fs.2, C2.2 >, < fm.2, M2.2 > \}.$$

Analogamente, a família $fs.2$ possue as duas configurações $C2.1$ e $C2.2$. Os componentes destas configurações são definidos respectivamente por:

$$M_C(C2.1) = \{ < fm.3, M3.1 >, < fm.4, M4.1 > \},$$

$$M_C(C2.2) = \{ < fm.3, M3.2 >, < fm.4, M4.2 > \}.$$

$S_f(n_1) = fs\ ;\ S_n(n_1) = fs.1$	
$Pr(n_1)\ ;\ Rq(n_1)$	
$C_S(n_1) = \{C1.1, C1.2\}$	
$C1.1$	$C1.2$
$M_C(C1.1)$	$M_C(C1.2)$
$Prc(C1.1)$ $Rqc(C1.1)$	$Prc(C1.2)$ $Rqc(C1.2)$

$S_f(n_3) = fs\ ;\ S_n(n_3) = fs.2$	
$Pr(n_3)\ ;\ Rq(n_3)$	
$C_S(n_3) = \{C2.1, C2.2\}$	
$C2.1$	$C2.2$
$M_C(C2.1)$	$M_C(C2.2)$
$Prc(C2.1)$ $Rqc(C2.1)$	$Prc(C2.2)$ $Rqc(C2.2)$

Fig. 3: Famílias de Subsistemas

Observe aqui que as descrições de software caracterizadas acima podem ser usadas para descrever a arquitetura de configurações de sistemas de software cuja estrutura modular foi construida por qualquer técnica de *design* particular. Maiores detalhes sobre esta caracterização formal generalizada estão descritos em [Alencar, Lucena 91a,91b].

Para introduzirmos a idéia de uma sequência de desenvolvimento de grafos de estrutura de software, definimos a noção de sistema de grafos de estrutura de software.

Definição
Um **sistema de grafos de estrutura de software** é definido como a tupla:
$S = < L, L', SS_0, R >$
onde
L e L' são linguagens definidas anteriormente para a especificação concreta e abstrata, respectivamente, dos recursos funcionais, SS_0 é um grafo de estrutura de software inicial, e R é um conjunto de regras de mudança (incluindo restrições às suas aplicações) que controlam a alteração de descrições de configuração de software. As restrições à aplicação de uma determinada regra de mudança podem ser vistas como as condições de aplicabilidade de regras de transformação para sistemas de software ("ponto grande").

Definição
Se SS e SS' são dois grafos de estrutura de software, dizemos que SS' é uma **extensão direta** de SS no sistema de grafos de estrutura de software S se existe uma e somente uma regra de alteração tal que $r(SS) = SS'$. Extensões, em geral, de um particular grafo de estrutura de software podem ser obtidas pela aplicação de um conjunto de regras de

mudança.

Dizemos que SS' é um desenvolvimento de SS no sistema S se existe uma sequência de grafos de estrutura de software $SS_0, \ldots, SS_n$ tal que $SS_0 = SS$, $SS_n = SS'$ e para todo $i = 1, \ldots, n$, SS_i é uma extensão direta de SS_{i-1} no sistema S. Isto nos leva à seguinte definição .

Definição
Seja $\{ SS_i \}_{i \in Nat}$ uma sequência de grafos de estrutura de software, onde Nat é um conjunto de números naturais (i.e. $Nat \in \{0, 1, \ldots\}$), tal que para cada $i \in Nat$ então SS_i é uma extensão direta de SS_{i-1} no sistema de grafos de estrutura de software S; dizemos que $\{ SS_i \}_{i \in Nat}$ é uma **sequência de desenvolvimento** de configurações de software. Tomamos cada grafo de estrutura de software SS_i desta sequência como um estado da configuração de um sistema evolutivo de software.

Pelas definições anteriores notamos que as mudanças realizadas através das regras de alteração do conjunto de regras R podem no caso geral basicamente afetar os nodos N (mudança dos nodos do grafo de estrutura), seus vértices denotados por E, os nomes dos nodos dados por S_n, que é a função que distingue se os nodos são famílias modulares ou de subsistemas (S_f), os recursos fornecidos ou requeridos, e as especificações abstrata e concreta de interface.

Portanto, as mudanças associadas ao conjunto de regras R podem ser classificadas essencialmente nos seguintes grupos:
a) Mudanças estruturais: mudanças que afetam basicamente o grafo de estrutura;
b) Mudanças dirigidas aos recursos: mudanças que alteram o número e o tipo de recursos fornecidos e requeridos por cada família ou membro familiar.
c) Mudanças dirigidas aos membros familiares: mudanças que afetam as versões modulares e os componentes de configuração de uma particular família de módulos ou subsistemas, respectivamente.
d) Mudanças dirigidas à funcionalidade: mudanças que dependem das especificações concreta e abstrata de interface, i.e. os aspectos funcionais dos recursos que são operações e suas descrições abstratas.

Estas mudanças realizadas sobre os grafos de estrutura de software estão sujeitas à restrições que podem ser vistas como as condições de aplicabilidade para transformações "em ponto grande" de sistemas de software. Tais restrições denotam de um modo geral propriedades estáticas e dinâmicas das configurações do sistema de software. Como exemplos de tais restrições citamos a que define uma configuração de sistema de software como bem formada e a que define alterações de ascendentemente compatíveis de sistemas de software ([Habermann 81], [Tichy 82], [Narayanaswamy 87a,87b]).

3 A Abordagem Lógica e as Descrições de Software

Para que se possa modelar do ponto de vista lógico o processo evolutivo das configurações de sistemas de software, teremos que representar neste formalismo lógico cada uma das descrições de sistemas de software (grafos de estrutura de software) por uma apresentação de teoria na lógica escolhida. Esta apresentação de teoria é denotada por um par

$< EL, \Delta >$ onde EL denota os símbolos extra-lógicos escolhidos para a representação dos estados de configuração e Δ é o conjunto de axiomas usados para definir as propriedades da representação .

No processo de criação de um apresentação de teoria $< EL, \Delta >$ pode-se encontrar três escolhas importantes e inter-relacionadas: a escolha da lógica, a escolha dos símbolos extra-lógicos EL e a escolha dos axiomas Δ para as propriedades da representação . A escolha da linguagem na qual descreveremos as descrições de alto nível de sistemas de software é crítica porque afeta especialmente o que se pode dizer, como se pode dizer e o que se torna plausível. A escolha da lógica também é difícil. Esta escolha influencia particularmente o que podemos dizer, a dificuldade em expressar isto, aquilo que o que dizemos comunica, como podemos raciocinar a partir do que dizemos e como podemos introduzir mudanças. A terceira escolha para a criação de um sistema de software em uma abordagem lógica é a escolha de uma apresentação de teoria e envolve a escolha de um conjunto de axiomas para definir as propriedades da representação . No que segue descrevemos com maiores detalhes nossas decisões em relação à estas escolhas.

Trabalharemos com o aparato do formalismo de uma lógica polisortida deôntica de ações . Esta abordagem lógica nos permite modelar a descrição do processo de mudança de configurações de sistemas de software fornecendo uma maneira através da qual podemos descrever formalmente a configuração de um sistema de software como também o modo como este evolui, i.e., como as configurações de software mudam e que ações produzem aquela mudança. Esta abordagem nos permitirá portanto expressar as noções existentes da arquitetura de um sistema de software, descrever as mudanças que podem ser realizadas em configurações particulares de software e prescrever quando estas mudanças devem ou tem que ocorrer. Além disso, seremos capazes de raciocinar dedutivamente sobre as descrições de sistemas de software para investigar os efeitos de determinadas mudanças em uma dada configuração em relação à descrições e prescrições de ações . O formalismo lógico adotado é uma versão de um subsistema do sistema lógico descrito em [Khosla 88] para a especificação de sistemas dinâmicos baseados em computador com múltiplos componentes bem como bases de dados distribuídas e redes de comunicação .

Neste formalismo as ações são membros de um sorte especial denotado por *Act*. Uma ação α é descrita através do conectivo modal $[\_]\_$. Uma fórmula atômica é uma fórmula denotada por

$$[\alpha]\,\phi$$

onde α é um termo do universo de ações *Act* e ϕ denota uma fórmula qualquer desta lógica. Podemos observar pelas regras de formação da lógica que os nomes de ações podem atuar como argumentos de predicados e funções e também ser o resultado de funções . Portanto, os relacionamentos de ações podem ser expressos através dos relacionamentos entre os seus nomes. A parametrização de uma função por nomes de ações pode facilitar a geração de ações não primitivas. Isto aumenta enormemente o poder expressivo do formalismo lógico. Isto ocorre, por exemplo, quando uma função toma como argumento dois nomes de ações e retorna uma ação que corresponde à combinação sequencial das duas ações no argumento funcional. Também se permite a parametrização de nomes de ações por nomes de ações , porque é geralmente útil formular ações que precisam mencionar os nomes de outras ações .

Além disso, a lógica também nos permite formar modalidades envolvendo outras modalidades, como por exemplo,

$$\vdash_s [\alpha]([\beta]\phi)$$

onde ϕ pode ser uma fórmula modal ou de primeira ordem. Esta expressão enuncia que se uma ação α é executada em um cenário s resultando em um cenário $\alpha(s)$, então se a ação β for executada, no cenário resultante final $\beta(\alpha(s))$ a propriedade ϕ será válida. Podemos também expressar modalidades condicionais, que são fórmulas que apresentam descrições relativizadas de ações tendo a forma geral

$$\psi \rightarrow [\alpha]\phi.$$

Esta expressão enuncia que se ψ é válida em um cenário, então se executarmos α neste cenário alcançaremos um cenário no qual ϕ será válida.

Uma fórmula atômica modal expressa alguma propriedade sobre o movimento entre cenários. Isto significa que a descrição de uma ação nesta lógica é baseada na caracterização do movimento hipotético entre cenários. Um cenário é um estado global de informação no qual as descrições de ações estáo baseadas. Pode ser visto como uma representação de uma coleção de propriedades do sistema de configuração de software em um dado instante de observação e é representado por uma teoria lógica consistente. Em adição à esta descrição (usualmente incompleta) todas as possíveis transformações entre cenários através da noção de ação , esta lógica descreve o comportamento de descrições de sistema quando permite a explicitação de quando uma particular ação deve ou tem que ocorrer. Para prescrevermos o uso de ações , as noções deônticas de ações permitidas e obrigatórias foram introdusidas. Os seguintes símbolos são incorporados à lógica: os predicados *per*, *obl*, *pref* do tipo *Act* e o predicado *obls* do tipo $T(Act)$ onde $T(Act)$ é o sorte cujos elementos são sequências finitas de ações . Os símbolos $per(\alpha)$, $\neg per(\alpha)$ e $obl(\alpha)$ denotam que uma ação α é permitida, não permitida e obrigada, respectivamente. Uma descrição mais detalhada desta lógica incluindo as categorias sintáticas, as regras de formação , os axiomas e as regras de inferência é dada, por exemplo, em [Alencar 89] e [Alencar, Buchsbaum 90a,90b].

A realização da semântica para o processo de mudança das descrições de configuração de software é dada por meio de um método de raciocínio dedutivo através do qual as descrições evolutivas dos sistemas de software são tratadas. Este é um método de prova geral baseado em cálculo de tableau para a lógica de ações de primeira ordem com domínio constante permitindo ações arbitrárias (constantes e não constantes). Este método leva por fim à um provador automático de teoremas baseado em tableau através do qual mudanças estruturais, de dependências através das interconexões modulares e funcionais podem ser realizadas. O método é geral no sentido que várias modalidades de ação que caracterizam sistemas lógicos diferentes podem ser tratados considerando-se regras de inferência iguais para estas várias modalidades de ações e um algorítmo de unificação que pode ser especializado mudando-se algumas de suas condições com a finalidade de descrever as condições correspondentes sobre a relação de acessibilidade na semântica de Kripke. Este método de prova geral pode ser considerado um extensão dos métodos de prova particulares para a lógica deôntica de ações e para a lógica modal de primeira ordem descritos em [Alencar, Buchsbaum 90a,90b] and [Jackson 88]. Nestes artigos anteriores também comparamos a abordagem dedutiva resultante com aquela de [Costa 88]. O propósito aqui é

a elaboração de métodos de prova flexíveis o bastante para permitir experimentação com diferentes lógicas de ação no contexto do processo de mudança na estrutura e funcionalidade de sistemas de software. Sistemas extendidos de ações com componentes deônticos fornecem uma estrutura de apoio apropriada para o estudo formal de sistemas evolutivos de software. Descrições detalhadas de sistemas de tableaux para várias lógicas podem ser encontradas em [Smullyan 68], [Fitting 83] e [Bell,Machover 77].

De acordo com a descrição formal e generalizada de um estado de configuração de um sistema de software dado como um grafo de estrutura de software de uma sequência particular de desenvolvimento, as descrições de sistemas de software podem ser representadas essencialmente pelos seguintes predicados:

1. *familia_modular*(*nome_objeto*, *projetista*)
2. *familia_subsistema*(*nome_objeto*, *projetista*)
3. *objeto_fornece*(*nome_objeto*, *nome_recurso*)
4. *objeto_requer*(*nome_objeto*, *nome_recurso*)
5. *versao_modular*(*nome_modulo*, *nome_versao*, *nome_arquivo*)
6. *configuracoes_subsistema*(*nome_subsistema*, *nome_configuracao*)
7. *componentes_configuracao*(*nome_subsistema*, *nome_configuracao*, *nome_componente*, *seletor*)
8. *versao_fornece*(*nome_modulo*, *nome_versao*, *nome_recurso*, *nome_tipo*)
9. *versao_requer*(*nome_modulo*, *nome_versao*, *nome_recurso*, *nome_tipo*)

Os nomes destes predicados são auto-explicativos. Quanto ao elemento *seletor* do predicado *componentes_configuracao*, este é usado para indicar qual versão particular de uma família modular ou de subsistemas tem que ser selecionada para participar de uma determinada configuração . O componente que não tiver uma tal versão receberá o valor *null* como conteúdo de *seletor*.

4 Mudanças na Estrutura, Interconexão e Funcionalidade

Nesta seção continuamos a descrever a semântica do processo de mudança de estados de configuração de software através da abordagem lógica, mostrando como mudanças estruturais, de interconexão e funcionais ocorrem nesta base lógica apresentada. Este propósito envolve a representação no formalismo lógico das condições para alterações válidas de configurações de software que foram mencionadas na seção 2. Neste sentido temos que garantir que as configurações permanecem válidas depois que o sistema de software é alterado. Os módulos de uma particular configuração , por exemplo, fazem algumas hipóteses um sobre o outro e uma particular configuração faz sentido se seus módulos componentes não fazem hipóteses errôneas a respeito dos recursos que eles requerem de outros módulos. Além disso, mostramos como uma ação de mudança de configuração de software pode ser descrita e usada de um modo dedutivo para demontrar se uma mudança desejada à um particular estado de configuração de software é conveniente ou não. Descrevemos o papel dos aspectos prescritivos de ações e mostramos que na abordagem logica adotada as alterações de

descrições de software de alto nível podem ser vistas como transformações de ponto grande que têm tanto aspectos descritivos quanto prescritivos. Finalmente, mostramos como as mudanças de natureza funcional são tratadas na abordagem lógica apresentada.

Podemos representar no formalismo lógico quaisquer das condições para alterações válidas de configurações de software, entre elas a condição de boa formação para configurações . No contexto do formalismo lógico as condições acima podem ser vistas como condições associadas com transformações em programação em ponto grande das descrições de alto nível de sistemas de software. Estas transformações são enunciadas por meio de descrições de ações e das características deônticas da lógica. Para ilustrar estes aspectos consideremos a descrição simples do um sistema de software representado na figura 1.

Este exemplo envolve a checagem de uma das condições para boa formação das configurações depois que algum recurso é acrescentado à interface de um dos componentes do sistema de software. As fórmulas que descrevem esta mudança e a investigação sobre a validade da descrição resultante são dadas por:

1 $A(cfg) \rightarrow [Checa\_bf(cfg)]\ Bf(cfg)$
2 $[Incluir\_rp(rp, cfg)]\ objeto\_fornece(rp, cfg)$

onde

$$A(cfg) =_{def} \forall sn\ configuracoes\_subsistema(sn, cfg) \rightarrow \forall rn\ .\ \{objeto\_fornece(sn, rn) \rightarrow \exists cpn, sv\ .\ componentes\_configuracao(sn, cfg, cpn, sv) \wedge objeto\_fornece(cpn, rn)\}.$$

Note que a expressão $A(cfg)$ acima na primeira fórmula é uma das condições de boa formação de uma configuração cfg e enuncia que qualquer recurso fornecido pela configuração cfg é também fornecido por algum componente cpn desta configuração . Esta fórmula enuncia que se estamos em um cenário no qual esta condição é válida e executarmos a ação que checa se a configuração cfg é bem formada ($Checa\_bf$), obteremos o fato resultante que ela é realmente bem formada. A segunda fórmula exemplifica a definição de uma mudança simples na descrição de software através da ação ($Incluir\_rp$) que é usada para inclusão de um recurso fornecido à uma certa configuração .

Uma vez que temos especificado como uma descrição particular de software pode ser alterada através da execução de ações , podemos raciocinar dedutivamente sobre as possíveis mudanças nesta descrição . No exemplo, estamos interessados em mudanças que preservam uma condição de boa formação de configurações de software. Em geral, qualquer propriedade (incluindo aquelas definidas aqui) que queremos preservar através de execuções de ações pode ser enunciada. O processo de raciocínio dedutivo é realizado por um provador de teoremas para esta lógica, ao qual podemos submeter consultas da forma "e se?". Estas consultas podem envolver tanto as descrições de ações e seus aspectos prescritivos como os agentes definidos na descrição particular.

Suponha que queremos verificar se a inclusão de um certo recurso em uma dada configuração implica na preservação de uma das condições de boa formação . Este é um exemplo de uma consulta envolvendo descrições de ações . O que queremos provar é a validade da seguinte formula:

$$[Incluir\_rp(g, M3.1)]\ [Checa\_bf(C2.1)]\ Bf(C2.1).$$

Uma prova desta fórmula mostra que uma vez que a formula é válida, podemos incluir na dada descrição de sistema de software um recurso fornecido g e ainda assim preservar a boa formação ($Bf(C2.1)$) da configuração $C2.1$. Os detalhes estão apresentados em [Alencar, Lucena 91a,91b].

Esta consulta demonstra que a mudança desejada na descrição de software é conveniente uma vez que ela preserva a condição para que uma configuração seja bem formada. Este exemplo simples mostra o uso do processo de raciocínio dedutivo para inferir consequências da execução de uma dada ação de mudança na configuração dinâmica do sistema. Entretanto, se tentarmos incluir na presente descrição de sistema de software o mesmo recurso fornecido g ao componente $C2.1$, no cenário resultante $C2.1$ não irá obedecer a mesma condição prévia de boa formação . Em outras palavras, a fórmula seguinte

$$[Incluir\_rp(g, C2.1)]\ [Checa\_bf(C2.1)]\ Bf(C2.1).$$

não é uma formula válida, se assumimos que embora o recurso g seja fornecido pela configuração $C2.1$, no cenário alcançado depois da execução da inclusão $[Incluide\_rp(g, C2.1)]$, não existe um membro desta configuração $C2.1$ no qual o recurso g seja fornecido. Isto significa que a mundança proposta não é conveniente.

Exemplificamos consultas envolvendo o aspecto prescritivo de ações , acrescentando a seguinte formula

$$[Incluir\_rp(p, C)]\ [Checa\_bf(C)]\ Bf(C) \rightarrow per(Incluir\_rp(p, C))$$

à descrição de software acima. Em consequência desta adição , podemos inferir se a ação de inclusão de um recurso fornecido particular p à uma dada configuração C é permitida. Em geral, isto é feito considerando-se a validade da permissão para executar a ação e a prova correspondente é obtida do mesmo modo como a prova anterior descrita acima nesta seção . Note que as alterações de descrições de alto nível de software podem ser vistas como transformações de programação em ponto grande e que estas transformações tem tanto um aspecto descritivo quanto um aspecto prescritivo.

Em relação às mudanças de natureza funcional dos estados de configuração de sistemas de software, duas dimensões podem ser consideradas. Estas duas dimensões correspondem à duas noções gerais de satisfatibilidade. A primeira noção de satisfatibilidade é concernente à prova de correção de uma descrição de um recurso funcional de uma maneira dependente da linguagem (um programa ou um segmento de programa) em relação à sua especificação de interface concreta (CIS). Isto é comumente denominado correção parcial ou total de programas e pode ser mostrado por exemplo através de sistemas dedutivos similares aos de [Hoare 69] e [Gries 81]. Neste caso queremos provar, por exemplo, que um procedimento particular (recurso funcional) escrito na linguagem de implementação obedece

a sua caracterização de interface concreta. Como foi descrito na seção 2 esta noção de satisfatibilidade é usada para garantir que a implementação de uma particular versão modular satisfaz a sua especificação de interface concreta. A segunda noção de satisfatibilidade refere-se à prova da correção da especificação concreta de interface dada de um modo dependente da linguagem de implementação em relação à uma especificação abstrata de interface dos recursos. Esta segunda noção de satisfatibilidade pode ser usada para assegurar que uma versão de família modular com uma especificação concreta de interface satisfaz a especificação abstrata associada de sua família modular. Neste caso podemos usar, por exemplo, uma lógica similar á lógica subjacente ao método VDM ([Jones 80,86], [Barringer 84]).

Deste modo, se queremos descrever o processo de mudança nos dois níveis de satisfatibilidade descritos acima precisamos provar, usando a lógica deôntica de ações como metalinguagem, quando fórmulas particulares associadas à lógica subjacente de uma técnica abstrata de especificação (como o método VDM) ou à uma lógica de programação , por exemplo, são válidas. Consequentemente, em geral, a interação entre um provador de teoremas para a metalógica deôntica de ações e as duas outras lógicas que expressam os dois níveis de satisfatibilidade descritos acima tem que ser considerada. Entretanto, no que segue somente consideraremos a interação entre os provadorres para a linguagem metalógica adotada e para a lógica do tipo Hoare para a verificação de um conjunto de programas determinísticos chamados programas do tipo *while*.

Consideremos, por exemplo, a pertinência à de uma versão modular *vn* à uma família modular *on*. Denotamos esta condição de pertinência sumarisadamente através do predicado $pertinencia\_familia\_cond(on, vn)$. Quando introdusimos mudanças de descrições de sistemas de software, temos que considerar se o arquivo fonte da versão *vn* (denotado por *prog*) satisfaz a interface concreta da versão modular. Em outras palavras, temos que investigar se as propriedades funcionais dos recursos especificados na especificação concreta de interface e expressos através de asserções de pre- e pós-condições no estilo de Hoare são satisfeitas pelo código fonte correspondente para estes recursos. Assumimos que as operações que aparecem na interface concreta de uma versão modular e suas pre- e pos-condições são dadas pelos predicados $modulo\_oper(vn, op)$, $modulo\_pre(op, pre)$ e $modulo\_pos(op, post)$ nos quais *pre* e *post* são termos da lógica deôntica que representam fórmulas da lógica de programação de Hoare.

Notamos que esta ação de verificação de pertinência é permitida somente se para cada operação de interface concreta, ambas as suas pré- e pós-condições são dadas. Isto é expresso pela fórmula:

$$
\begin{array}{l}
versao\_modulo(on, vn, fn) \rightarrow \\
\forall\, op,\ pre,\ post\ (modulo\_oper(on, vn, fn) \rightarrow \\
(modulo\_pre(on, pre) \wedge modulo\_pos(on, post))) \\
\rightarrow per(teste\_pertinencia(vn, on))
\end{array}
$$

A correspondente descrição da ação para $teste\_pertinencia(vn, on)$ é dada por:

$pertinencia\_familia\_cond(vn, on) \wedge$
$codigo\_versao(vn, prog) \wedge modulo\_pre(vn, pre) \wedge$
$modulo\_pos(vn, pos) \wedge prove_{PL}(\{ pre \}\ prog\ \{ pos \})$
$\rightarrow [teste\_pertinencia(vn, on)]\ membro\_familia(vn, on)$

Na expressão acima o predicado $prove_{PL}$ é usado para codificar a relação de derivabilidade da lógica de programação de Hoare na lógica deôntica de ações . Para este propósito pode-se usar um provador de teoremas para a lógica de programação que trata as pré- e pós-condições como as especificações concretas de entrada e saída, respectivamente.

Ilustremos o processo dedutivo sobre os aspectos funcionais da pertinência à uma família modular consultando se uma particular versão modular *vn* pertence à família modular *on*. Como podemos ver das duas últimas expressões, a ação para testar esta pertinência é permitida se assumimos que todas as operações da versão tem pré- e pós-condições . Neste caso podemos investigar a validade da seguinte fórmula construindo sua prova na abordagem lógica. De fato, queremos provar usando a última expressão apresentada que:

$$[teste\_pertinencia(V, M)]\ membro\_familia(V, M)$$

Quando chegamos no ponto de provar a asserção do predicado $prove_{PL}$, os provadores de teoremas interagem da seguinte maneira. Primeiro, o provador da metalinguagem deôntica de ações chama o provador para a lógica de programação com a asserção no predicado $prove_{PL}$. Então, se o provador da lógica de programação obtém sucesso em provar a asserção , o ramo associado também se fecha. Isto é o mesmo que tomar $prove_{PL}$ como um axioma não lógico. Senão, o ramo não pode ser fechado e a fórmula não é válida. Os detalhes são dados em [Alencar, Lucena 91a,91b].

Deste modo, podemos investigar como alterações estruturais e funcionais gerais de uma descrição de software podem afetar as noções de integridade para descrições arquiteturais de sistemas de software. Para isso basta considerarmos o efeito de uma sequência qualquer de ações individualmente propostas ($[\![\omega]\!]$ na lógica apresentada) e que caracterizam alterações estruturais e/ou funcionais nas descrições de sistemas de software.

5 Conclusão

Apresentamos neste artigo a formalização de conceitos relevantes na área de gerência de configurações . Uma descrição formal generalizada de arquiteturas de sistemas de software do ponto de vista de sistemas evolutivos de software foi inicialmente caracterizada. Esta descrição usa a noção de grafos de estrutura de software como os estados de configuração de software e a noção de sequências de desenvolvimento de estados. Estados válidos de configuração de software foram usados para controlar a evolução dos grafos de estrutura de software dos pontos de vista estrutural e funcional.

A semântica do processo de mudança dos estados de configuração de software foi dada através de uma abordagem lógica que envolve teorias representando configurações de software e suas mudanças quando afetadas por ações num formalismo que também inclui as características deônticas das ações . A descrição da abordagem lógica também mostra como

mudanças estruturais, de interconexão e funcionais ocorrem na base lógica apresentada. Isto envolve a representação no formalismo lógico das condições para alterações válidas de configurações de software que foram formalmente apresentadas. Neste sentido, temos que garantir que as configurações continuem válidas depois que o sistema de software é alterado. Os módulos de uma particular configuraçãao, por exemplo, fazem algumas hipóteses um sobre o outro e uma particular configuração faz sentido essencialmente se os seus módulos componentes não fazem hipóteses errôneas sobre os recursos que eles requerem de outros módulos. Além disso, mostramos como ações de mudança de configurações de software podem ser descritas e usadas de um modo dedutivo para demonstrar se uma mudança desejada em um particular estado de configuração de software é conveniente ou não. O método apresentado de raciocínio dedutivo pode ser visto como a realização da semântica para o processo de mudanças de configurações de software. Descrevemos o papel dos aspectos prescritivos de ações e mostramos que na abordagem lógica adotada as alterações de descrições em um alto nível de abstração de sistemas de software podem ser vistas como transformações de programação em ponto grande que tem tanto um aspecto descritivo quanto um aspecto prescritivo. O aspecto funcional das mudanças de configuração foi tratado através da interação entre um provador de teoremas construído para a metalógica deôntica de ações e um provador dado para uma lógica de programação .

Consultas sobre a conveniência de ações de mudança particulares à estados de configuração de software do ponto de vista de aplicações práticas podem ter a sua eficiência aumentada se considerarmos, por exemplo, a possibilidade de particionar a teoria de configuração global para um sistema de software em segmentos modulares cada um dos quais descrevendo um particular aspecto (e.g. estrutural ou dirigido aos recursos) e de submeter cada consulta juntamente com seu associado segmento modular ao provador. De fato, entretando, estamos mais interessados em modelar o processo de mudanças e, assim, nossa abordagem lógica pode ser vista essencialmente como uma especificação formal de uma aplicação prática em configurações de software.

6 Referências

[Alencar, Lucena 91a] Alencar, P.S.C., Lucena, C.J.P. *A Logical Viewpoint of Software Configuration Maintainance*, submetido para publicação .

[Alencar, Lucena 91b] Alencar, P.S.C., Lucena, C.J.P. *A Logical Framework for Evolving Software Systems*, submetido para publicação .

[Alencar 89] Alencar, P.S.C. *Uma Abordagem Lógica para Sistemas Evolutivos de Software*, Relatório Técnico, Departamento de Informática, 1989.

[Alencar, Buchsbaum 90a] Alencar, P.S.C. *An Automated Reasoning Method for a Many-Sorted Deontic Action Logic*, Relatório Técnico, Departamento de Informática, 1990.

[Alencar, Buchsbaum 90b] Alencar, P.S.C. *A General Reasoning Method for Many-Sorted First-Order Action Logic*, a ser submetido para publicação .

[Ambriola, Bendix, Ciancarini 90] Ambriola, V., Bendix, L. Ciancarini, P., *The Evolution of Configuration Management and Version Control* , Software Engineering Journal, November 1990.

[Barringer 84] Barringer, H., Cheng, J. H., Jones, C. B., *A Logic Covering Undefinedness in Program Proofs*, Acta Informatica, vol. 21, pp. 251-259, 1984.

[Bell, Machover 77] Bell, J., Machover, M., *A Course in Mathematical Logic*, North-Holland Pub. Co., 1977.

[Costa, Cunningham 88] Costa, M., Cunningham, R. J., *Mechanized Deduction and Modal Action Logic*, Forest Report 5, Imperial College, University of London.

[Fitting 83] Fitting, M., *Proof Methods for Modal and Intuitionistic Logic*, D. Reidel Pub. Co., Dordrecht, 1983.

[Gries 81] Gries, D., *The Science of Programming*, Springer-Verlag, NY, 1981.

[Habermann et al. 81] Habermann, A. N. et al., *System Composition and Version Control for Ada*, Software Engineering Environments, H. Hunke (editor), North-Holland Pub. Company, 1981.

[Hoare 69] Hoare, C. A. R., *An Axiomatic Basis for Computer Programming*, Communications ACM 12, pp. 576-583, 1969.

[Jackson, Reichgelt 88] Jackson, P., Reichgelt, H., *A General Proof Method for First-Order Modal Logic*, Proceedings of the IJCAI, pp. 942-944, 1988.

[Jones 80] Jones, C. B., *Software Development: A Rigorous Approach*, Prentice-Hall International, 1980.

[Jones 86] Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall International, 1986.

[Khosla 88] Khosla, S., *System Specification: A Deontic Approach*, PhD Thesis, Department of Computing, Imperial College of Science and Technology, 1988.

[Narayanaswamy 87a] Narayanaswamy, K., Scacchi, W., *Maintaining Configurations of Evolving Software Systems*, IEEE Trans. Soft. Eng., vol. SE-13, no. 3, pp. 324-334, 1987.

[Narayanaswamy 87b] Narayanaswamy, K., Scacchi, W., *A Database Foundation to Support Software System Evolution*, The Journal of Systems and Software 7, pp. 37-49, 1987.

[Smullyan 68] Smullyan, R. M., *First Order Logic*, Springer-Verlag, Berlin, 1968

[Tichy 82] Tichy, W. F., *A Data Model for Programming Support Environments and its Application*, in Automated Tools for Information System Design and Development, Schneider, H., Wasserman, A., Eds., Amsterdam, The Netherlands: North-Holland, pp. 31-48, 1982.