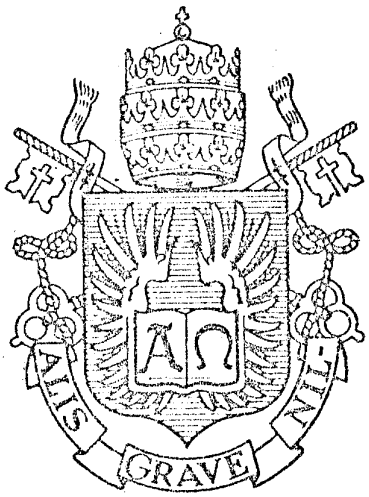

PUC

Série: Monografias em Ciência da Computação, nº 22/89

UMA ABORDAGEM LÓGICA PARA SISTEMAS EVOLUTIVOS DE SOFTWARE

Paulo Sérgio C. de Alencar

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 – CEP 22453

RIO DE JANEIRO – BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação, Nº 22/89
Editor: Paulo A. S. Veloso Outubro, 1989

UMA ABORDAGEM LÓGICA PARA SISTEMAS EVOLUTIVOS DE SOFTWARE*

Paulo Sérgio C. de Alencar**

* Trabalho parcialmente financiado pela FINEP
** Cursando o programa de doutoramento do DI.

Responsável por Publicações

Rosane Teles Lins Castilho
PUC/RJ-Depto. de Informática
Assessoria de Biblioteca, Documentação e Informação
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ

Uma Abordagem Lógica para Sistemas Evolutivos de Software

por
Paulo Sérgio C. de Alencar
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225
22453 Rio de Janeiro / RJ - BRASIL

Julho, 1989

Abstract

O processo de mudança em configurações dinâmicas de sistemas de software deve não somente ser descrito de maneira formal, mas também ser objeto de raciocínio dedutivo que informe sobre consequências de mudanças em uma dada configuração . Neste artigo apresentamos uma abordagem lógica através da qual pode-se caracterizar formalmente as noções existentes de arquitetura de um sistema, descrever as mudanças que as configurações dinâmicas sofrem e prescrever quando estas mudanças podem e têm de ocorrer. Além disso, o raciocínio dedutivo hipotético deste formalismo lógico pode ser visto como um poderoso mecanismo para uma variedade de consultas do tipo e-se que nos permite considerar os efeitos de mudanças particulares sobre uma dada configuração em relação a descrições ou prescrições de ações do sistema. O uso das caracter'isticas acima no contexto de sistemas evolutivos de software é investigado através do formalismo de uma lógica polisortida deôntica de ações (LDA).

1 Introdução

O gerenciamento de configurações de software é a parte do processo de gerenciamento de software que trata do desenvolvimento de procedimentos e padrões para a gerência de um sistema evolutivo de software. Essencialmente, esta disciplina está relacionada com mudanças: como controlar mudanças, como gerenciar sistemas de software que foram sujeitos à mudanças e como liberar versões destes sistemas alterados para usuários (ver apêndice 1).

Neste sentido, o processo de mudança em configurações dinâmicas de sistemas de software deve não somente ser descrito de maneira formal, mas também ser objeto de raciocínio dedutivo que informe sobre consequências de mudanças em uma dada configuração . Neste artigo apresentamos uma abordagem lógica através da qual pode-se caracterizar formalmente as noções existentes de arquitetura de um sistema, descrever as mudanças que as configurações dinâmicas sofrem e prescrever quando estas mudanças podem e têm de ocorrer. Além disso, o raciocínio dedutivo hipotético deste formalismo lógico pode ser visto como um poderoso mecanismo para uma variedade de consultas do tipo e-se que nos permite considerar os efeitos de mudanças particulares sobre uma dada configuração em relação a descrições ou prescrições de ações do sistema. O uso das caracter'isticas acima no contexto de sistemas evolutivos de software é investigado através do formalismo de uma lógica polisortida deôntica de ações .

Uma lógica polisortica deôntica de ações com agentes cumpre este objetivo por várias razões. Primeiramente, o componente polisortido desta lógica constitue um meio através do qual os diferentes tipos de entidades envolvidas no problema de manutenção de configurações dinâmicas podem ser explicitamente distinguidos. O componente de ações desta lógica é usado para descrever como os sistemas de software evoluem. As validações e as mudanças nas descrições de sistemas de software são, por exemplo, algumas ações descritas através deste componente lógico. O componente deôntico desta lógica pode ser usado para modelar permissões e obrigações relacionadas às entidades envolvidas no problema, constituindo-se em uma poderosa disciplina baseada em autorizações . Pode-se pensar em prescrições no uso de recursos e limitar a realização de certas ações a agentes específicos.

Deste modo vemos que é possível se construir bases de dados lógicas de descrições de sistemas através deste formalismo, controlar o comportamento dinâmico real e hipotético do sistema e raciocinar dedutivamente sobre esta descrição lógica. O comportamento dinâmico real provém de mudanças na configuração do sistema resultantes da realização de ações nesta abordagem lógica. O comportamento dinâmico hipotético provém dos efeitos de mudanças hipotéticas do sistema e leva em conta o impacto das mudanças em relação à descrição integral do sistema. O raciocínio dedutivo pode ser realizado através de um provador automático de teoremas para esta lógica deôntica de ações .

O presente artigo está estruturado como segue. Na seçao 1 descreveremos o formalismo lógico no qual o problema de manutenção de sistemas evolutivos de software será tratado. Usa-se então fato que descrições de sistemas podem ser modeladas em termos uma coleção de estados globais de informação chamados cenários e uma noção de ação que descreve o moyimento entre estes cenários. Uma descrição mais detalhada do sistema lógico apresentado aqui pode ser encontrada em [Khosla 88].

Na seção 2 discute-se as vantagens de se usar o presente formalismo lógico como um formalismo de especificação , comparando-o com outras abordagens. Entre as vantagens mais evidentes, citamos a possibilidade de se enunciar descrições de ações como pré e pós-condições e a possibilidade de se raciocinar dedutivamente sobre propriedades de ações (como o sequenciamento) sem a necessidade de se referir a meta-regras.

Na seção 3 descreveremos um método para a manutenção de sistemas evolutivos de software que enuncia uma noção rigorosa de integridade do sistema e que também fornece mecanismos para controlar a evolução de configurações ([Narayanaswamy 87a,87b]). As noções descritas nesta seção serão usadas como base para a abordagem sugerida no presente trabalho.

Finalmente, na seção 4 apresentamos uma abordagem lógica para sistemas evolutivos de software, que usa o poder expressivo e dedutivo da lógica apresentada na seção 1 no contexto deste problema. Nesta abordagem o processo de mudança em configurações dinâmicas de sistemas de software não somente é descrito de maneira formal, mas também é objeto de raciocínio dedutivo que informa sobre consequências de mudanças em uma

dada configuração . Pode-se caracterizar formalmente as noções existentes de arquitetura de um sistema, descrever as mudanças que as configurações dinâmicas sofrem e prescrever quando estas mudanças podem e têm de ocorrer. Além disso, o raciocínio dedutivo hipotético deste formalismo é visto como um poderoso mecanismo para uma variedade de consultas do tipo e-se que nos permite considerar os efeitos de alterações sobre uma dada configuração em relação a descrições ou prescrições de ações do sistema.

2 O Formalismo Lógico e seu Poder Expressivo

Nesta seçao nós descreveremos o formalismo lógico no qual o problema da manutenção de sistemas evolutivos de software será tratado. Usa-se o fato que descrições de sistemas podem ser modeladas em termos uma coleção de estados globais de informação chamados cenários e uma noção de ação que descreve o movimento entre estes cenários. Uma descrição mais detalhada do sistema lógico apresentado aqui pode ser encontrada em [Khosla 88] onde se trata a especificação de sistemas dinâmicos.

Inicialmente os aspectos estruturais do sistema são modelados através da descrição das entidades, suas propriedades e relacionamentos e das ações envolvidas. Neste sentido, a coleção de fórmulas que constitue a especificação do sistema pode ser dividida em dois tipos: aquelas a respeito de requisitos estáticos (que são invariantes a mudança) e aquelas a respeito de requisitos dinâmicos (que dizem respeito a restrições mais específicas sobre as mudanças). No componente polisortido de primeira ordem a quantificação sobre sortes específicos refere-se a domínios potenciais. Isto quer dizer que um sorte declarado deve denotar a classe de todos os objetos que potencialmente podem pertencer a ela. As ações foram introduzidas como uma nova extensão modal da lógica polisortida de primeira ordem. Para que se possa modelar o comportamento do sistema e poder lidar com os aspectos prescritivos das ações os conceitos deônticos de ação permissível e obrigatória são introduzidos. Isto nos permite separar descrições de ações de prescrições de ações , isto é, enunciar quando as ações devem e têm que acontecer em oposição a somente descrever os efeitos de tais ações .

Neste formalismo as ações são membros de um sorte especial denominado *Act*. Uma ação α é descrita pelos conectivos modais $[\_]\_$. Uma fórmula atômica modal é uma fórmula denotada por

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

onde α é um termo do universo de ações *Act* e ϕ denota uma fórmula desta lógica. Como pode-se concluir a partir das regras de formação desta lógica (ver apêndice 2) nomes de ações podem agir como argumentos de predicados e funções , e também ser o resultado de funções . Assim, os relacionamentos entre ações são expressos através dos relacionamentos entre seus

nomes. Isto também enriquece bastante o poder expressivo desta lógica, podendo a parametrização de funções por nomes de ações inclusive facilitar a geração de ações não primitivas. Isto ocorre, por exemplo, quando uma função toma como argumento dois nomes de ações e retorna a ação que corresponde à combinação sequencial das duas ações no argumento funcional. Permite-se também a parametrização de nomes de ações por nomes de ações , por ser geralmente útil formular ações que precisam mencionar os nomes de outras ações . Além disso pode-se inclusive quantificar sobre todo o universo das ações através do uso de variáveis como é usual. Assim, tomando como exemplo uma restrição global forte, uma fórmula do tipo

$$\forall \alpha \,.\, [\alpha]\, \phi$$

onde α é uma variável da categoria *Act* de ações , enuncia que a propriedade ϕ é invariante em relação a mudanças. Esta lógica ainda permite a formação de modalidades envolvendo não somente expressões de primeira ordem mas também a formação de 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 nos diz que se uma ação α for executada em un cenário s resultando em um cenário $\alpha(s)$, então será o caso que se a ação β for executada, no cenário resultante $\beta(\alpha(s))$ a propriedade ϕ será válida. Pode-se também expressar modalidades condicionais, cujo propósito é relativizar (ou condicionar) a descrição de uma ação e que têm 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 é válida.

A fórmula atômica modal expressa alguma propriedade sobre o movimento entre cenários.Isto quer dizer que a descrição de uma ação nesta lógica se baseia na caracterização do movimento hipotético entre cenários. Ao contrário de outros formalismos usados para especificação , que assumem implicitamente que as coisas permanecem como estão a menos que

uma ação explicitamente as mude, a presente abordagem contém explicitamente o requisito que qualquer propriedade que uma ação não afeta deve ser preservada através da regra/axioma do "frame". Um cenário é um estado global de informação no qual a descrição das ações se baseia. Um cenário pode ser visto como a representação da coleção de propriedades do sistema em um dado instante de observação e é representado por uma teoria consistente da lógica.

As expressões $t(s), t(s'), \ldots$ são usadas como uma meta-notação para cenários onde $s, s', \ldots$ denota as correspondentes apresentações de teoria. Uma fórmula atômica modal, por exemplo, se puder ser provada a partir de um cenário particular s enuncia que se a ação α for executada em um cenário s então o cenário alcançado após a execução de α em s, denotado por $\alpha(s)$, exibe a propriedade de primeira ordem ϕ. Uma teoria nesta lógica é introduzida através desta noção de apresentação de teoria. Uma apresentação de teoria é dada por um par $< L, A >$ onde a primeira componente denota a linguagem extralógica da teoria e a segunda componente denota uma coleção de axiomas específicos desta teoria. Uma teoria T de uma apresentação de teoria TP, denotada por $T(TP)$ é definida como o conjunto de fórmulas que podem ser provados a partir da coleção A de axiomas específicos desta teoria. Isto pode ser expresso como

$$T(TP) = \{\phi : A \vdash \phi\}$$

onde ϕ é uma fórmula expressa na linguagem L. Em adição à noção de provabilidade em relação a uma apresentação de teoria que é definida como a relação $A \vdash \phi$ (lida como ϕ pode ser provado a partir de $< L, A >$), o símbolo $\vdash_s$ denota provabilidade a partir de uma apresentação de teoria s da teoria que representa o cenário $t(s)$.

A especificação de um sistema nesta lógica corresponde a uma teoria assim como, conforme observamos anteriormente, um cenário de um sistema também é representado por uma teoria nesta lógica. A relação entre a especificação de um sistema e a coleção de cenários que esta especificação dá origem se deve à introdução da regra de necessidade

$$\frac{\vdash_{sp} \phi}{\vdash_{sp} [\phi]\,\phi}$$

onde $\vdash_{sp}$ denota provabilidade a partir de uma especificação . Esta regra tem como consequência a preservação somente da informação intensional da especificação e não necessariamente a informação específica a cenários. Com isso, duas formas de raciocínio dedutivo podem ser identificados neste formalismo: o raciocínio dedutivo sobre a propria especificação e o raciocínio dedutivo sobre um cenário da especificação . Este último em vez de simplesmente permitir inferências sobre a especificação e avaliação se esta exibe certas propriedades desejáveis, nos permite testar a especificação através do acréscimo à teoria da especificação de outros fatos específicos correspondentes a situações possíveis e verificar que consequências isto traz. Note entretanto que a regra de necessidade citada acima não é aplicável a quaisquer propriedades específicas de cenários, mas somente a propriedades da especificação . Em geral, a introdução de descrições de ações como hipóteses específicas de cenários nos permite raciocinar dedutivamente sobre como um sistema irá reagir quando certas ações (com consequências determinadas) são executadas. Esta forma de raciocínio dedutivo é especialmente aplicável à problemas que requerem quer predição quer simulação hipotética.

Como um exemplo simples considere as pilhas. Elas devem ser descritas com o uso da seguinte linguagem extralógica: $top : \longrightarrow nat$ e ações $push : nat \longrightarrow Act$ e $pop : \longrightarrow Act$. Note que top é uma constante do sorte nat. Ela age como um designador não rígido que pode ser visto como o análogo de uma variável em uma linguagem imperativa. Podemos agora introduzir os seguintes axiomas para pilhas:

$$top = n \rightarrow (([push(m)]\,[pop])\,(top = n))$$

$$[push(m)]\,(top = m)$$

O primeiro axioma enuncia que se $top = n$ é válido em um cenário particular e executarmos a ação $push$ e então a ação pop fazemos com que top permaneça com seu valor original. O segundo axioma enuncia que após executarmos a ação $push$ com o elemento m sobre a pilha obtemos que m é o novo top da pilha. Note que ambas a ações se referem implicitamente ao estado da pilha que está sendo manipulada.

Porém, além da descrição (usualmente incompleta) de todas as possíveis transformações entre cenários através da noção de ação , esta lógica de-

screve o comportamento de um sistema quando permite que se explicite quando uma determinada ação pode ou tem que ser executada. Para que se pudesse prescrever o uso de ações introduziu-se as noções deônticas de ações permissíveis e obrigatórias. Esta abordagem para o modelamento destas ações envolve a ideia de um cenário normativo. A ideia é dividir todos os cenários em dois tipos, aqueles que são normativos e aqueles que não são normativos. Um cenário é normativo se a proposição especial $\hat{n}$ é uma consequência dele e é não normativo se $\neg\hat{n}$ o for. Cenário normativos devem denotar cenários deonticamente aceitaveis enquanto cenários não normativos devem denotar aqueles cenários que são deonticamente inaceitaveis. Assim, ações permitidas são modeladas como sendo aquelas que levam a cenários normativos e ações proibidas a cenários não normativos. Incorpora-se à lógica os símbolos predicativos P, O, $Pref$ de tipo Act e o predicado OS de tipo $S(Act)$ onde $S(Act)$ é o sorte cujos elementos são sequências finitas de ações . $P(\alpha)$, $\neg P(\alpha)$ e $O(\alpha)$ significam que a ação α é permitida, não permitida e obrigatória, respectivamente. Uma ação permitida leva a um cenário normativo se já se está em um cenário alcançado após a execução de uma ação permitida. O mesmo se aplica a ações proibidas. Uma característica interessante desta caracterização do operador P é que este toma a forma de uma descrição de ação . Isto nos permite alterar a estrutura de permissões de um sistema de uma maneira particularmente elegante. Por exemplo, se a ação α leva a um cenário normativo então

$$[\alpha]\, P(\beta)$$

é equivalente a $[\alpha]\,[\beta]\,\hat{n}$ e descreve uma ação que, quando executada, resulta em um cenário no qual é permitido executar-se a ação β. Devido a esta possibilidade de descrever ações que nos permitem mudar as estruturas normativas, não é difícil ver como padrões de comportamento complexos podem ser construídos. Dois exemplos interessantes são o da expressão relativizada

$$\psi \rightarrow P(\alpha)$$

e o das sequências de obrigações

$$\psi \rightarrow [\alpha]\,(O(\beta) \wedge [\beta]\,O(\gamma)).$$

Isto é um exemplo de uma sequência de obrigações de comprimento dois.

Ela enuncia que se ψ é válida e α é executada, incorre-se na obrigação para a ação β, que quando descartada leva à introdução de outra obrigação , desta vez para a ação γ.

Ainda de acordo com esta abordagem ser permitido não executar uma ação α é equivalente ao fato de existir alguma ação diferente de α que é permitida. Para isso introduz-se o operador $Pref(\alpha)$, o qual assegura a existência de uma ação diferente de α que é permitida. Em outras palavras, enuncia que é sempre permitido executar alguma ação diferente de α. Quando a obrigação para fazer algo existe, depois que a ação obrigatória é executada a obrigação de que ela ocorra novamente é perdida. Se uma ação α é obrigatória então ela é também permitida e, além disso, nenhuma outra ação diferente de α é permitida. Isto sumariamente descreve o aspecto estático de obrigações . Para descrevermos o aspecto transicional de obrigações considere um cenário no qual um número de ações é permitido e um número de ações é proibido e se refira a isto como a p-estrutura daquele particular cenário. Suponha ainda que uma das ações permitidas seja executada e leve a um cenário no qual alguma ação é obarigatória. Isto significa que o cenário é tal que todas as ações diferentes da ação obrigatória não são permitidas. Se a obrigação incorrida é descartada a p-estrutura do cenário resultante é aquela obtida antes que a obrigação foi incorrida. Em suma, o aspecto transicional de obrigações pode ser visto como a manutenção temporária da corrente p-estrutura e sua restauração quando a obrigação for descartada. Então, embora as permissões sejam preservadas sobre a ocorrencia de ações a menos que descrições de ações forcem o contrário, as obrigações temporariamente forçam um sistema a suspender o seu comportamento corrente, executar a ação obrigatória e então reassumir o seu comportamento prévio levando em conta o efeito da ação obrigatória.

Esta abordagem também trata a formação de ações mais complexas a partir de mais simples. Isto é conseguido com a introdução de combinadores lógicos. O combinador de sequenciação , por exemplo, toma duas ações e retorna uma ação que quando executada tem o mesmo efeito de executar uma ação após a outra das duas ações mais primitivas originais.

A introdução de agentes nos permitirá associar um executor ao conceito anônimo de ações . É possível então descrever ações em relação aos sujeitos que as executam. Outro benefício de agentes é que a permissão e

a obrigação se tornam permissão e obrigação para agentes específicos realizarem determinadas ações . Com isso, uma obrigação associada a um agente específico não implica na suspensão de qualquer permissão para executar uma ação não associada ao agente em questão. Além disso, isto nos permite ser explícito sobre a interação entre os vários componentes do sistema, estando aptos a tratar, por exemplo, o caso em que o efeito de um agente executar uma ação resulte em uma obrigação para outro agente ou qualquer outra restrição de natureza prescritiva. As definição das fórmulas é generalizada para incluir $[A, \alpha]\phi$, $P(A, \alpha)$, $O(A, \alpha)$, $Pref(A, \alpha)$ e $OS(A, < \sigma, \alpha >)$.

Uma descrição mais detalhada desta lógica, incluindo categorias sintáticas, regras de formação , axiomas e regras de inferência pode ser encontrada no apêndice 2. Os axiomas da lógica polisortida de primeira ordem bem como axiomas que governam a interação entre as fórmulas de primeira orderm e as fórmulas modais podem ser encontrados ali. Pode-se notar que estes axiomas refletem o ponto de vista que cenários devem, em geral, denotar estados de informação incompletos. Isto porque a especificação de um sistema é uma teoria da lógica, e invariavelmente tais teorias são incompletas tanto devido à natureza do sistema quanto pelo entendimento do mesmo. Observamos ainda que esta lógica é um tipo de lógica modal no qual uma coleção de ações pode ser vista como uma coleção de modalidades e os mundos possíveis têm domínios constantes. Isto quer dizer que o domínio dos potenciais objetos que pertencem a um determinado sorte é comum a todos os cenários.

3 Lógica Deôntica de Ações como Meio de Especificação

A característica mais importante do formalimo apresentado na seção anterior é possibilidade de distinção explícita entre o aspecto descritivo e prescritivo em sistemas dinâmicos. A maioria dos formalismos de especificação assume que a descrição de mudanças implicitamente define o comportamento do sistema. Isto ocorre em formalismos de especificação como VDM ([Jones 86]) e Z ([Sufrin 84]) nos quais um estilo de pré e pós-condições é adotado (as ações são descritas pelas propriedades válidas antes e após a execução da ação) e onde o comportamento de um sistema nunca é explicitamente dado. Ao contrário, estes formalismos assumem que as operações podem ocorrer quando suas pré-condições são satisfeitas.

Embora o tipo de objetos que o presente formalismo permite mencionar seja ainda restrito, convém mencionarmos que a maioria dos dados estruturados encontrados em VDM ([Jones 86]), por exemplo, pode ser construido facilmente em uma lógica polisortida de primeira ordem. Dadas, por exemplo, funções de construção apropriadas, não é difícil caracterizar conjuntos e sequências de sortes particulares de objetos.

A introdução de um conectivo modal como em [Goldblatt 82] para representar descrições de ações nos fornece ainda certas vantagens sobre outras abordagens. As duas maiores vantagens são a possibilidade de enunciar descrições de ações como pré e pós-condições e a possibilidade de se raciocinar dedutivamente sobre propriedades de ações (como o sequenciamento) sem a necessidade de se referir a meta-regras (como em VDM). Como exemplo do primeiro caso temos as modalidades iteradas, isto é, um descrição de ação relativa à execução de outra ação :

$$[\alpha]\,[\beta]\,\phi.$$

Aqui, a pré-condição enuncia que α é a última ação que é executada e isto significa que podemos formular descrições de ações em relação á informação sobre que ações foram executadas previamente. O segundo importante caso do raciocínio dedutivo sobre descrição de ações , como também sua prescrição , será aplicado posteriormente ao problema de manutenção de sistemas evolutivos de software. Outras vantagens desta abordagem são a possibilidade de explicitamente se associar um executor à noção de ações (e

portanto às descrições de ações) e a possibilidade de se construir ações mais complexas a partir de ações mais primitivas através da introdução de combinadores de ações .

Embora em outros formalismos a pré-condição seja usada tanto para especificar o contexto no qual uma ação é executada e/ou uma aprovação implícita para que aquela ação seja executada, a presente abordagem considera a pré-condição somente como a identificação do contexto no qual uma ação é executada. O outro uso de pré-condições é suprido pela informação sobre a prescrição das ações .Estas descrições devem enunciar quando as ações devem e têm que ocorrer, através dos conceitos deônticos de ação permissivel e obrigatória. Isto envolve a divisão de todos os cenários em dois tipos: normativo e não normativo. Os cenários normativos são aceitáveis do ponto de vista comportamental do sistema e os não normativos são não aceitáveis e são deonticamente proibidos. Todas as ações que levam a cenários normativos são definidas como ações permissiveis e todas as ações permissiveis levam a cenários normativos.

Além disso, os estados de um sistema são considerados como situações idealizadas nas quais podemos de algum modo "congelar" o sistema e detectar todas as propriedades relevantes nesta situação . Estes estados de informação sobre o sistema são denominados cenários. A ideia de cenários em oposição a estados é significativa quando lida-se com aplicações sobre as quais não se tem um conhecimento completo. Ainda, pode-se representar informação negativa sobre ações como, por exemplo, não é o caso que a ação α produz a propriedade ϕ, sem se concluir que tal ação resulta em $\neg\phi$. Em contraste, a ideia de estado em formalismos como VDM e Z é uma estrutura explícita que afetam e são afetadas pelas execuções de ações .

4 Um Método Formal para a Manutenção de Sistemas Evolutivos de Software

Um importante aspecto da emergente disciplina de gerenciamento de configuração de software é a tarefa de manutenção de configurações de um sistema evolutivo de software. Nesta seção apresentamos um formalismo sobre a manutenção de sistemas evolutivos de software que enuncia uma noção rigorosa de integridade do sistema e mecanismos para controlar a evolução de configurações ([Narayanaswamy 87a,87b]) .

Inicialmente apresentamos como os componentes de um sistema de software e suas diferentes configurações podem ser especificados de modo que estas configurações possam ser mantidas. Descrevemos então um modêlo de configurações de um sistema de software que pode ser incorporado em uma linguagem denominada NuMIL. Pode-se também alternativamente armazenar as descrições de configurações em uma base de dados para descrição da estrutura do sistema. No nosso caso teremos uma base de dados na lógica apresentada na seção 1. Este modêlo permite fornecer definições rigorosas para vários conceitos importantes de manutenção de configurações . Apresentaremos os critérios de pertinência a uma família modular, o conceito de configurações bem formadas e a relação de compatibilidade ascendente. Os dois primeiros critérios se constituem tanto em propriedades de configurações de sistemas de software do ponto de vista estático, enquanto que o último caracteriza como o sistema de software pode ser alterado. A noção de compatibilidade ascendente entre dois membros de uma família modular nos diz quando um dos membros pode ser substituído por outro. Alterações com esta característica preservam tanto a boa formação das configurações quanto a pertinência a famílias de módulos.

Como usualmente, o sistema é visto como um grafo acíclico dirigido cujos nodos folha são famílias de módulos e os nodos internos são famílias de subsistemas([Narayanaswamy 87a]). Uma família modular é um conjunto de arquivos fonte que compartilham as mesmas propriedades de interface que caracterizam a família. Um membro particular de uma família é denominado uma versão daquela família. Uma família de subsistemas é um conjunto de configurações que satisfazem uma particular especificação de interface. Uma configuração é definida como um conjunto de módulos (ou

outras configurações) que podem ser combinados. Uma configuração particular que é um membro de uma família de subsistemas é denominada uma versão daquela família. Temos que distinguir aqui entre versões e revisões. Uma versão resulta de mudanças que requerem que um novo módulo da família seja criado enquanto uma revisão resulta de alterações sucessivas em um arquivo fonte que não levam à criação de um novo membro familiar.

Cada módulo do sistema fornece um conjunto de recursos, como tipo de dados, funções , procedimentos, variáveis, etc., para uso em outros módulos e pode requerer alguns recursos que são fornecidos por outros módulos.

A seguir apresentamos um exemplo de uma especificação na linguagem de descrições NuMIL que descreve um subsistema S que contém duas configurações . Estas configurações são formadas por versões de dois módulos M_1 e M_2, conforme segue:

```
subsistema S
        fornece a,b;
        requer c,d;
        configurações
                IBM-PC = { M1 : versão1, M2 : versão2; }
                VAX = { M2 : versão1, M1 : versão2; }
end

modulo M1
        fornece a,f;
        requer d,b;
        implementações {
                versão versão1 {
                        realização x.c;
                                fornece int a; short f;
                                requer b, d; }
                versão versão2 {
                        realização y.c;
                                fornece float a; int f;
                                requer b, d; } }
end
```

```
modulo M2
        fornece b;
        requer c,f;
        implementações {
                versão versão1 {
                        realização m.c;
                                fornece int b(s,t) char *s, *t;
                                requer c, f; }
                versão versão2 {
                        realização n.c;
                                fornece int b(m,n) char *m, *n;
                                requer c,f; } }
end
```

Consideremos agora os conceitos formais de integridade do sitema. Inicialmente descreveremos os critérios para pertinência de um módulo a uma família modular. Seja M^f uma família de módulos e seja M um módulo particular. Em geral, $p(M)$ e $p(M^f)$ denotam os conjuntos de recursos fornecidos pelo módulo M e pela família de módulos M^f, respectivamente. Analogamente, $r(M)$ denota o conjunto de recursos requeridos pelo módulo M. Podermos dizer então que um módulo M pode ser considerado um membro da família M^f se e somente se:

1. O módulo M fornece os recursos especificados na família de módulos M, i.e.,
$$p(M) \supseteq p(M^f).$$

2. O módulo M requer menos recursos que M^f, i.e.,
$$r(M) \subseteq r(M^f).$$

3. O módulo M não deve fornecer e requerer o mesmo recurso, i.e.,
$$p(M) \cap r(M) = \phi.$$

Consideremos agora o conceito de configuração bem formada. Uma configuração $C = \{C_1, C_2, \ldots, C_n\}$, onde cada C_i pode ser um módulo ou uma outra configuração , é dita bem formada se e somente se:

1. Todo recurso fornecido por C é fornecido por algum C_i, i.e.,

$$p(C) \subseteq \cup_{i=1}^{n} p(C_i).$$

2. A configuração C requer aqueles recursos requeridos por todos os $C_i s$ exceto aqueles recursos já fornecidos por algum outro componente da configuração , i.e.,

$$r(C) = \cup_{i=1}^{n} r(C_i) - \cup_{i=1}^{n} p(C_i).$$

3. A configuração C não fornece e requer os mesmos recursos,i.e.,

$$p(C) \cap r(C) = \phi.$$

4. Os recursos fornecidos e requeridos por cada componente de C são disjuntos, i.e.,

$$p(C_i) \cap r(C_i) = \phi.$$

5. Nenhum recurso é fornecido por mais de um componente, i.e.,

$$p(C_i) \cap p(C_j) = \phi,$$

para todo $C_i, C_j \in C, i \neq j$.

6. Os usos de recursos através de interfaces modulares são sintaticamente consistentes com suas definições , i.e., checagem de tipos entre módulos.

A definição acima caracteriza aquelas configurações nas quais as hipóteses feitas por um participante sobre o outro são preservadas. Isto auxilia a garantir a continua validade das configurações quando o sistema é alterado. Os módulos que participam de configurações , por exemplo, fazem hipóteses uns sobre os outros. Para que uma configuração tenha sentido

os módulos não devem fazer hipóteses erroneas sobre os recursos que eles podem requerer de outros módulos.

Até aqui consideramos propriedades de configurações de sistemas de software sob o aspecto estático. Estas propriedades nos fornecem uma maneira de checar a integridade das configurações sem levar em conta mudanças. Agora, uma vez que as descrições do sistema estão sujeitas a alterações , introduzimos um conceito que caracteriza como um sistema de software pode ser alterado. Trata-se da noção de compatibilidade ascendente. Uma versão M_1 de uma família modular M^f é ascendentemente compativel com uma versão M_2 se e somente se:

1. O módulo M_1 fornece pelo menos todos os recursos fornecidos por M_2,

$$p(M_1) \supseteq p(M_2).$$

2. O módulo M_2 não fornece recursos que não são requeridos por M_2,

$$r(M_1) \subseteq r(M_2).$$

3. As representações sintáticas de recursos comuns às duas versões (cláusulas **fornece** e **requer**) são identicas.

As definições acima capturam aquelas situações em que o uso de um particular recurso R fornecido em M_1 irá satisfazer os usos do mesmo recurso fornecido por M_2. Portanto, M_1 pode substituir M_2 em qualquer configuração bem formada, uma vez que os recursos fornecidos por M_1 sejam mutuamemte disjuntos dos recursos fornecidos por outros componentes naquela configuração . Assim, alterações ascendentemente compatíveis preservam a boa formação de configurações e preservam a pertinência à família de módulos.

5 Uma Abordagem Lógica para Sistemas Evolutivos de Software

Nesta seção mostraremos como usar o poder expressivo e dedutivo da lógica apresentada na seção 1 no contexto do problema da manutenção de sistemas evolutivos de software. Entre as vantagens obtidas com isso incluem-se aquelas citadas anteriormente quando comparamos esta lógica com outros formalismos de especificação , nos permitindo inclusive enunciar descrições de ações como pré e pós-condições e raciocinar dedutivamente sobre propriedades de ações sem a necessidade de se referir a meta-regras.

Primeiramente, como é usual, o componente polisortido desta lógica será usado como um meio através do qual os diferentes tipos de entidades envolvidas no problema podem ser explicitamente distinguidas. O componente de ações da lógica é usado para descrever como os sistemas de software evoluem. As validações e as mudanças nas descrições de sistemas de software são, por exemplo, algumas ações descritas através deste componente lógico. O componente deôntico desta lógica pode ser usado para modelar permissões e obrigações relacionadas às entidades envolvidas no problema, constituindo-se em uma poderosa disciplina baseada em autorizações . Pode-se, por exemplo, presecrever o uso de recursos e limitar a realização de certas ações a agentes específicos.

Deste modo podemos construir bases de dados lógicas de descrições de sistemas através deste formalismo, controlar as mudanças sobre estas bases de dados, o comportamento dinâmico real e hipotético do sistema e raciocinar dedutivamente sobre esta descrição lógica. O comportamento dinâmico real provém de mudanças na configuração do sistema resultantes da realização de ações nesta abordagem lógica. O comportamento dinâmico hipotético provém dos efeitos de mudanças hipotéticas do sistema e leva em conta o impacto das mudanças em relação à descrição integral do sistema. O raciocínio dedutivo é realizado através de um provador automático de teoremas para a lógica deôntica de ações . O apêndice 3 trata sobre a construção de um provador de teoremas baseado em cálculo de tableau para esta lógica.

Como já enfatizamos anteriormente a essência do problema de gerência de configurações consiste no gerenciamento das mudanças de sistemas de

software. Neste sentido torna-se tão importante quanto a descrição estrutural dos sistemas de software o controle das mudanças efetuadas sobre estas descrições . No nosso caso uma descrição de um sistema de software é uma base de dados na lógica polisortida deôntica de ações , constituindo-se em uma teoria lógica que contém todas as informações fornecidas por descrições na linguagem NuMIL conforme apresentadas anteriormente na seção 3. Assim, pode-se especificar usando-se esta lógica a evolução desejada desta base de dados através de seus aspectos descritivo e prescritivo. Neste caso serão especificadas as ações que produzem mudanças sobre as bases de dados lógicas, juntamente com informações a respeito de quando estas devem e têm de ocorrer.

Para ilustrarmos os aspectos abordados acima, consideremos a descrição de um sistema bastante simples que contém as configurações C, C_1 e C_2 e que pode ser expresso através da seguinte teoria:

1 $A(C) \rightarrow [Checa\_bf(C)]\ Bf(C)$
2 $[Incluir\_rf(p, C)\ fornecido(p, C)$.
3 $membro(C_1, C)$
4 $membro(C_2, C)$
5 $fornecido(p_1, C)$
6 $fornecido(p_2, C)$
7 $fornecido(p_1, C_1)$
8 $fornecido(p_2, C_2)$

onde

$$A(C) =_{df} \forall p\ .\ \{fornecido(p, C) \rightarrow \exists C'\ .\ membro(C', C) \wedge fornecido(p, C').$$

Note que a expressão $A(C)$ acima é a primeira das condições para a boa formação de uma configuração conforme foi definida na seção anterior. A primeira fórmula acima portanto enuncia que se estivermos em um cenário no qual esta condição for válida e executarmos a ação que checa a boa formação parcial de uma configuração ($Checa\_bf$), obteremos como resultado o fato que esta configuração é bem formada. A segunda fórmula acima

exemplifica a definição de uma mudança simples na configuração através da ação ($Incluir\_rf$) para inclusão de um recurso fornecido à uma determinada configuração . Esta pode ser considerada uma ação de edição da base de dados lógica que contém a descrição do sistema. As demais fórmulas da teoria especificam quais os membros de uma determinada configuração C e quais os recursos fornecidos por cada uma das configurações citadas.

Uma vez especificadas as maneiras através das quais uma determinada descrição de um sistema pode ser alterada, através da execução de ações , podemos raciocinar hipoteticamente a respeito de possíveis mudanças nesta descrição . Em nosso caso estamos especificamente interessados em mudanças que preservem a pertinência a uma família de módulos, a boa formação de uma configuração ou que envolvam módulos ascendentemente compatíveis. Este raciocínio hipotético é realizado através da submissão de consultas do tipo "e-se" a um provador automático de teoremas desta lógica. O tipo destas consultas pode ser essencialmente classificado naquelas que envolvem a descrição de ações , naquelas que envolvem o aspecto prescritivo das ações e naquelas que envolvem os agentes caracterizados no problema, podendo uma certa consulta ser caracterizada como de mais de um destes tipos. As consultas que envolvem a descrição de ações são usadas para se atestar se no cenário alcançado após a execução de uma dada ação uma determinada propriedade é ou não válida. No caso mais geral este tipo de consulta envolve uma combinação de ações e é relativizada por uma outra descrição de ação . Em outras palavras, deseja-se saber se, dado que uma certa descrição de ação seja válida, ao executarmos uma combinação de ações uma certa propriedade se verifica no cenário alcançado após a execução destas ações . As consultas que envolvem o aspecto prescritivo das ações tem por objetivo nos informar a respeito do estado deôntico das ações associado aos cenários, isto é, se uma determinada ação é permitida, não permitida ou obrigatória em um cenário determinado. Isto envolve a validade das pré-condições prescritivas das ações envolvidas na consulta. Finalmente, as consultas que envolvem os agentes caracterizados no problema estão relacionadas a um poderoso mecanismo de autorizações que visa restringir a execução de certas ações a agentes específicos e possibilitar que certos agentes em determinadas circunstancias tenham permissão (não permissão) ou sejam obrigados a realizar certas

ações . Além disso, as consultas podem ainda fazer referência a combinadores de ações quando estes são devidamente definidos na lógica.

Como um exemplo simples, suponhamos que se deseje verificar se a inclusão de um determinado recurso fornecido em uma dada configuração implica ou não na preservação parcial da boa formação desta configuração . Trata-se de uma consulta descrita acima como do primeiro tipo. O que se quer demonstrar é, portanto, a seguinte fórmula:

$$[Incluir\_rf(p_3, C)]\ [Checa\_bf(C)]\ Bf(C).$$

A demonstração sumarizada abaixo mostra que não se pode incluir na descrição do sistema um recurso fornecido p_3 e depois ao se checar a configuração C verificar que ela é bem formada ($Bf(C)$), uma vez que a fórmula acima não é válida. Este, portanto, é um exemplo de uma mudança na configuração do sistema que não preserva uma das condições (1) de boa formação descritas anteriormente.

1 $[Incluir\_rf(p_3, C)]\ membro(C_1, C)$ $(ax., nec)$

2 $[Incluir\_rf(p_3, C)]\ membro(C_2, C)$ $(ax., nec)$

3 $[Incluir\_rf(p_3, C)]\ fornecido(p_1, C)$ $(ax., nec)$

4 $[Incluir\_rf(p_3, C)]\ fornecido(p_2, C)$ $(ax., nec)$

5 $[Incluir\_rf(p_3, C)]\ fornecido(p_1, C_1)$ $(ax., nec)$

6 $[Incluir\_rf(p_3, C)]\ fornecido(p_2, C_2)$ $(ax., nec)$

7 $[Incluir\_rf(p_3, C)]\ A(C) \rightarrow$
$[Checa\_bf(C)]\ Bf(C)$ $(ax., nec)$

8 $[Incluir\_rf(p_3, C)]\ A(C) \rightarrow$
$[Incluir\_rf(P_3, C)]\ [Checa\_bf(C)]\ Bf(C)$ $(ax.\ log.\ 7)$

9 $[Incluir\_rf(p_3, C)]\ fornecido(p, C)$ $(ax., nec)$

10 $[Incluir\_rf(p_3, C)]\ fornecido(p_3, C)$ $(9,\ [p/p_3])$

11 $[Incluir\_rf(p_3, C)]\ \forall p\ .\ \{fornecido(p, C) \rightarrow$
$\exists C'\ .\ membro(C', C) \wedge fornecido(p, C')\}$ $(hip.\ de\ 7)$

12 $\forall p\ .\ [Incluir\_rf(p_3, C)]\ \{fornecido(p, C) \rightarrow$
$\exists C'\ .\ membro(C', C) \wedge fornecido(p, C')\}$ $(ax.\ log.\ 9)$

13 $\forall p\ .\ [Incluir\_rf(p_3, C)]\ \{fornecido(p, C) \rightarrow$
$[Incluir\_rf(p_3, C)]\ \exists C'\ .\ membro(C', C) \wedge fornecido(p, C')\}$
$(ax.\ log.\ 7)$

14 $[Incluir\_rf(p_3, C)]\ \{fornecido(p_3, C) \rightarrow$
$[Incluir\_rf(p_3, C)]\ \exists C' \,.\, membro(C', C) \wedge fornecido(p, C')\}$
$(ax.\ log.\ 4)$

15 $[Incluir\_rf(p_3, C)]\ \{fornecido(p_3, C) \rightarrow$
$[Incluir\_rf(p_3, C)]\ \exists C' \,.\, membro(C', C) \wedge fornecido(p, C')\}$
$(ax.\ log.\ 4)$

16 $[Incluir\_rf(p_3, C)]\ \exists C' \,.\, membro(C', C) \wedge fornecido(p_3, C')$
$(M.\ P.\ , 11, 7)$

17 $[Incluir\_rf(p_3, C)]$
$\neg\forall C' \,.\, \neg\{membro(C', C) \wedge fornecido(p_3, C')\} \quad (Def.\ \exists)$

18 $[Incluir\_rf(p_3, C)]$
$\neg\forall c' \,.\, \{membro(C', C) \vee fornecido(p_3, C')\} \quad (Def.\ \neg\wedge)$

19 $[Incluir\_rf(p_3, C)]$
$\neg\{membro(\bar{C}', C) \vee fornecido(p_3, \bar{C}')\} \quad (ax.\ log.\ 4)$

20 $[Incluir\_rf(p_3, C)]$
$\{membro(\bar{C}', C) \vee fornecido(p_3, \bar{C}')\} \quad (Def.\ \neg\vee)$

21 . . .

22 *Não válida*

Esta consulta nos informa, portanto, que uma dada mudança pretendida não é conveniente uma vez que não preserva parcialmente a boa formação de uma configuração . Este exemplo simples nos mostra o poder do raciocínio hipotético quando empregado para aferir as consequências da execução de uma dada ação sobre a configuração dinâmica do sistema. Nesta demonstração usamos os axiomas da teoria da especificação do sistema, denotados por *ax.*, as regras de inferência de modus ponens e da necessidade, denotadas por *M.P.* e *nec.*, respectivamente, e os axiomas lógicos descritos no apêndice 2.

Entretanto, certas alterações na configuração do sistema são convenientes por preservarem, por exemplo, determinadas condições de boa formação da configuração . Este é o caso se incluirmos na descrição do sistema o mesmo recurso fornecido p_3 porém agora à configuração C_1. A demonstração sumarizada abaixo mostra a conveniência de se incluir na descrição do sistema este recurso, uma vez que é preservada a boa formação parcial definida pela primeira condição de boa formação de uma

configuração .

1 $[Incluir\_rf(p_3, C_1)]\ membro(C_1, C)$ $(ax., nec)$

2 $[Incluir\_rf(p_3, C_1)]\ membro(C_2, C)$ $(ax., nec)$

3 $[Incluir\_rf(p_3, C_1)]\ fornecido(p_1, C)$ $(ax., nec)$

4 $[Incluir\_rf(p_3, C_1)]\ fornecido(p_2, C)$ $(ax., nec)$

5 $[Incluir\_rf(p_3, C_1)]\ fornecido(p_1, C_1)$ $(ax., nec)$

6 $[Incluir\_rf(p_3, C_1)]\ fornecido(p_2, C_2)$ $(ax., nec)$

7 $[Incluir\_rf(p_3, C_1)]\ A(C) \rightarrow$
$[Checa\_bf(C)]\ Bf(C)$ $(ax., nec)$

8 $[Incluir\_rf(p_3, C_1)]\ A(C) \rightarrow$
$[Incluir\_rf(P_3, C_1)]\ [Checa\_bf(C)]\ Bf(C)$ $(ax.\ log.\ 7)$

9 $[Incluir\_rf(p_3, C_1)]\ fornecido(p, C)$ $(ax., nec)$

10 $[Incluir\_rf(p_3, C_1)]\ fornecido(p_3, C)$ $(9,\ [p/p_3])$

11 $[Incluir\_rf(p_3, C_1)]\ \forall p\ .\ \{fornecido(p, C) \rightarrow$
$\exists C'\ .\ membro(C', C) \wedge fornecido(p, C')\}$ $(hip.\ de\ 7)$

12 $\forall p\ .\ [Incluir\_rf(p_3, C_1)]\ \{fornecido(p, C) \rightarrow$
$\exists C'\ .\ membro(C', C) \wedge fornecido(p, C')\}$ $(ax.\ log.\ 9)$

13 $\forall p\ .\ [Incluir\_rf(p_3, C_1)]\ \{fornecido(p, C) \rightarrow$
$[Incluir\_rf(p_3, C_1)]\ \exists C'\ .\ membro(C', C) \wedge fornecido(p, C')\}$
$(ax.\ log.\ 7)$

14 $[Incluir\_rf(p_3, C_1)]\ \{fornecido(p_3, C) \rightarrow$
$[Incluir\_rf(p_3, C_1)]\ \exists C'\ .\ membro(C', C) \wedge fornecido(p_3, C')\}$
$(ax.\ log.\ 4)$

15 $[Incluir\_rf(p_3, C_1)]\ \{fornecido(p_3, C) \rightarrow$
$[Incluir\_rf(p_3, C)]\ \exists C'\ .\ membro(C', C) \wedge fornecido(p_3, C')\}$
$(ax.\ log.\ 4)$

16 . . .

17 *Válida*

Esta fórmula é válida uma vez que apesar do recurso p_3 não ser fornecido pela configuração C no cenário alcançado após a execução da ação de inclusão $[Incluir_r f(p_3, C_1)]$, existe uma configuração que é membro de C , que no caso é C_1, na qual o recurso p_3 é fornecido.

Exemplifiquemos agora as consultas que envolvem o aspecto prescritivo das ações . Suponhamos o acréscimo da seguinte fórmula

$$[Incluir\_rf(p, C)]\,[Checa\_bf(C)]\,Bf(C) \rightarrow P(Incluir\_rf(p, C))$$

à especificação do sistema dada acima. Com isto, podemos inferir quando a ação de inclusão de um determinado recurso fornecido p a uma dada configuração C é permitida. Isto pode ser mostrado através de provas semelhantes àquelas apresentadas anteriormente nesta seção .

Observamos ainda que podemos generalizar a consulta acima através da associação à cada ação descrita na especificação de um determinado agente que a executa. Isto pode nos possibilitar expressarmos um poderoso mecanismo de autorizações relacionado ao sistema. Assim, a fórmula acima é generalizada ao supormos que a ação de inclusão de recursos está associada com um determinado agente a. Obtemos então

$$[a,\ Incluir\_rf(p, C)]\,[a,\ Checa\_bf(C)]\,Bf(C) \rightarrow P(a,\ Incluir\_rf(p, C)).$$

Os exemplos acima nos mostram como, dado o grande poder expressivo da lógica polisortida deôntica de ações , podemos formalizar o conceito de mudanças nas configurações de sistemas evolutivos de software em seu aspecto descritivo e prescritivo e, raciocinarmos hipoteticamente a respeito das mesmas. Este tipo de raciocínio nos permite realizarmos uma grande variedade de consultas do tipo "e-se" à base de dados lógica que contém a descrição de um determinado sistema de software.

6 Apêndice 1 : Gerenciamento de Configurações de Software

O gerenciamento de configurações de software (GCS) é definido como uma disciplina emergente para identificar a configuração de um sistema evolutivo de software com a finalidade de controlar as mudanças nesta configuração e para manter a integridade e rastreabilidade da configuração ao longo do ciclo de vida do sistema ([Babich 86], [Bersoff 80]).

As quatro funções principais do gerenciamento de configuração de software são identificação , controle, administração de estado e auditagem. Na fase de identificação a configuração de software tem que ser definida e expressa como uma sequência de itens:

$$(item_1, item_2, \ldots, item_m).$$

Na fase de controle deve-se descrever como controlar as mudanças sobre a configuração e listar os passos no processamento de mudanças que diretamente ou indiretamente afetam a configuração :

$$(passo_1, passo_2, \ldots, passo_n).$$

Na fase de administração de estado temos que descrever quais as mudanças realizadas sobre o sistema. Como resultado obtemos a configuração do sistema e as mudanças relacionadas em um dado instante:

$$(item_1, item_2, \ldots, item_m) \; +$$

$$(muda_1, muda_2, \ldots, muda_p, muda\ pendente_1, \ldots, muda\ pendente_q).$$

Finalmente, na fase de auditagem deve-se verificar se o sistema que está sendo construído satisfaz os requisitos enunciados e obter as diferenças entre o sistema até então construído e os requisitos enunciados como segue:

$$(difere_1, \ldots, difere_r).$$

Dois importantes conceitos no GCS são o de um item de configuração de software (ICS) e o de uma configuração básica. Um item de configuração de software é formalmente definido como um componente de software que é

acompanhado e controlado através do processo de desenvolvimento do sistema de software. Exemplos de itens de configuração de software são programas, funções do sistema e módulos de um sistema de software. Uma configuração básica representa um ponto de referência, isto é , um marco no desenvolvimento de um sistema, e é o produto final das atividades de uma fase de desenvolvimento e o ponto de partida da fase subsequente, sendo constituída de uma coleção de ICSs relacionados entre si.

7 Apêndice 2 : A Lógica Deôntica de Ações

Apresentamos aqui com mais detalhe uma versão da lógica polisortida de ações deôntica tratada na seção 2 do presente artigo.

LINGUAGEM

Categorias Sintáticas

- Uma coleção finita de sortes

$$S = Act \cup Agt \cup S(Act) \cup \{s_1, s_2, \ldots\}$$

onde Act denota o sorte de ações , Agt representa o sorte de agentes e $S(Act)$ denota o sorte não vazio de sequências finitas de ações .

- Símbolos constantes:

  1. Constante lógica $\hat{n}$.
  2. Constantes extra-lógicas: para cada sorte $s \in S$ existe um conjunto (possivelmente vazio) de símbolos constantes cada um dos quais é dito ser de sorte s.

- Nomes de ações : para cada $n > 0$ e cada n-upla $< s1, \ldots, s_n >$ tal que $s_1 \in S, \ldots, s_n \in Act$, existe um conjunto (possivelmente vazio) de nomes de ação de aridade n cada um dos quais é dito ser do tipo $< s_1, \ldots, s_n >$.

- Símbolos de predicados:

  1. Lógicos: P do tipo Act, O do tipo Act, $Pref$ do tipo Act e OS do tipo Act.
  2. Extra-lógicos: para cada $n > 0$ e cada n-upla $< s_1, \ldots, s_n >$ tal que $s_1 \in S, \ldots, s_n \in S$, existe um conjunto (possivelmente vazio) de simbolos de predicados de aridade n cada um dos quais é dito ser do tipo $< s_1, \ldots, s_n >$.

- Símbolos funcionais:

1. Lógicos:
   $<>$ do tipo $< Act, S(Act) >$
   , do tipo $< S(Act), Act, S(Act) >$
2. Extra-lógicos: para cada $n > 0$ e cada $n+1$-upla $< s_1, \ldots, s_{n+1} >$ tal que $s_1 \in S, \ldots, s_{n+1} \in S$ existe um conjunto (possivelmente vazio) de símbolos funcionais de aridade n cada um dos quais é dito ser do tipo $< s_1, \ldots, s_{n+1} >$.

- Símbolos de igualdade: para certos sortes $s \in S$ (possivelmente todos) existe um símbolo predicado especial $=_s$ de sorte $< s, s >$ que representa a igualdade entre objetos de sorte s.

- Variáveis: o conjunto infinito usual de variáveis distintas para cada sorte s.

- Quantificadores: para cada sorte $s \in S$ temos um quantificador universal $\forall_s$ e um quantificador existencial $\exists_s$.

- Operadores lógicos: $\neg, \leftrightarrow, \rightarrow, \wedge, \vee, [\_,\_], [[\_,\_]]$

- Pontuação : (,) e ⟩ .

Regras de Formação

- Termos:

1. para cada sorte $s \in S$ uma variável de sorte s é um termo.
2. se $t_1, \ldots, t_n$ são termos de sortes $s_1, \ldots, s_n$, respectivamente e $s_1, \ldots, s_n$ são todos de S e f é um símbolo funcional de tipo $< s_1, \ldots, s_{n+1} >$ então $f(t_1, \ldots, t_n)$ é um termo de sorte s_{n+1}.
3. se $t_1, \ldots, t_n$ são termos de sortes $s_1, \ldots, s_n$, respectivamente e $s_1, \ldots, s_n$ são todos de S e a é um simbolo de ação de tipo $< s_1, \ldots, s_n >$ então $a(t_1, \ldots, t_n)$ é um termo de sorte Act.
4. nada mais é um termo.

- Átomos:

1. se $t_1, \ldots, t_n$ são termos de sortes $s_1, \ldots, s_n$ respectivamente e $s_1, \ldots, s_n$ são todos tomados de S e p é um simbolo de predicado de tipo $< s_1, \ldots, s_n >$ então $p(t_1, \ldots, t_n)$ é um átomo.
2. para cada sorte $s \in S$, dados dois termos de s, a saber, t_1 e t_2, $t_1 =_s t_2$ é um átomo, se $=_s$ está na linguagem.
3. nada mais é um átomo.

- Fórmulas:

1. Um átomo é uma fórmula.
2. A constante lógica $\hat{n}$ é uma fórmula.
3. Se ϕ é uma fórmula então $\neg\phi$ também o é.
4. Se ϕ e ψ são fórmulas então $(\phi \vee \psi), (\phi \wedge \psi), (\phi \rightarrow \psi), (\phi \leftrightarrow \psi)$ são fórmulas também.
5. Se α é uma fórmula de sorte Act, A é um termo de sorte Agt e ϕ é uma fórmula, então $[A, \alpha]\phi$ é uma fórmula.
6. Se σ é um termo de sorte $S(Act)$, A é um termo de sorte Agt e ϕ é uma fórmula, então $[[A, \sigma]]\phi$ é uma fórmula.
7. Para cada sorte $s \in S$, se x é uma variável de sorte s e ϕ é uma fórmula então $\forall x\phi$ e $\exists x\phi$ são fórmulas.
8. nada mais é uma fórmula.

Axiomas

Se ϕ, ψ e τ são fórmulas, A é um termo de sorte Agt. α é um termo de sorte Act e σ é um termo de sorte $S(Act)$, então as expressões seguintes são axiomas:

1. $\phi \rightarrow (\psi \rightarrow \phi)$
2. $(\tau \rightarrow (\phi \rightarrow \psi)) \rightarrow ((\tau \rightarrow \phi) \rightarrow (\tau \rightarrow \psi))$
3. $(\neg\, phi \rightarrow \neg\psi) \rightarrow ((\neg\phi \rightarrow \psi) \rightarrow \phi)$
4. $\forall x\phi(x) \rightarrow \phi(t)$ onde t é livre para x em ϕ

5. $(\forall x(\phi \rightarrow \psi)) \rightarrow (\phi \rightarrow \forall x\psi)$ onde x não é livre em ϕ
6. $[A, \alpha]\top$
7. $([A, \alpha](\phi \rightarrow \psi)) \rightarrow (([A, \alpha]\phi) \rightarrow ([A, \alpha]\psi))$
8. $([A, \alpha]\neg\phi \rightarrow (\neg[A, \alpha]\phi)$
9. $\forall x([A, \alpha]\phi) \leftrightarrow ([A, \alpha]\forall x\phi)$ onde x não é livre em α ou A
10. $\exists x([A, \alpha]\phi) \rightarrow [A, \alpha]\exists x\phi)$ onde x não é livre em α ou A
11. $(([A, \alpha]\phi) \wedge ([A, \alpha]\psi)) \leftrightarrow ([A, \alpha]\phi \wedge \psi)$
12. $(([A, \alpha]\phi) \vee ([A, \alpha]\psi)) \rightarrow ([A, \alpha]\phi \vee \psi)$
13. $[[A, < \alpha >]]\phi \leftrightarrow [A, \alpha]\phi$
14. $[[A, < \sigma, \alpha >]]\phi \leftrightarrow [[A, \sigma]][A, \alpha]\phi$
15. $\hat{n} \rightarrow (P(A, \alpha) \leftrightarrow [A, \alpha]\hat{n})$
16. $\neg[A, \alpha]\hat{n} \rightarrow [A, \alpha]\neg\hat{n}$
17. $Pref(A, \alpha) \leftrightarrow \exists\beta(\neg(\beta = \alpha) \wedge P(A, \beta))$
18. $O(A, \alpha) \rightarrow P(A, \alpha)$
19. $OS(A, < \alpha >) \leftrightarrow O(A, \alpha)$
20. $OS(A, < \sigma, \alpha >) \leftrightarrow (OS(A, \sigma) \wedge [[A, \sigma]])(A, \alpha))$

Regras de Inferência

- Generalização :

$$\frac{\vdash \phi}{\vdash \forall x \,.\, \phi}$$

- Modus Ponens:

$$\frac{\vdash \phi, \vdash (\phi \rightarrow \psi)}{\vdash \psi}$$

- Necessidade:

$$\frac{\vdash_{sp} \phi}{\vdash_{sp} [\alpha]\phi}$$

8 Apêndice 3 : Um Provador de Teoremas para LDA

Nesta seção trataremos a respeito da construção de um provador automático de teoremas para a lógica deôntica de ações . Este provador será parte de um sistema de suporte para a construção de descrições de sistemas e para se raciocinar dedutivamente sobre tais descrições , devendo ser baseado em um cálculo de tableau para a lógica em questão.

Sendo um dos principais variantes do cálculo de sequentes, os sistemas de tableau semânticos surgiram a partir das provas de completude para o cálculo de sequentes sem a regra do corte descobertas independentemente nos anos cinquenta por Beth, Hintikka, Kanger, Schutte e Kripke ([Beth 59], [Hintikka 55], [Kripke 63], [Prawitz 75]). O método de tableau aplicado à lógica clássica tem seu locus classicus em [Smullyan 68]. A descrição de sistemas de tableau para as lógicas não clássicas modais e intuicionista pode ser encontrada em [Fitting 83] e [Bell,Machover 77].

Como observamos anteriormente, LDA é um tipo de lógica modal no qual uma coleção de ações pode ser vista como uma coleção de modalidades e os mundos possíveis têm domínios constantes. Isto quer dizer que o domínio dos potenciais objetos que pertencem a um determinado sorte é comum a todos os cenários. Para uma descrição detalhada sobre lógicas modais pode-se consultar [Hughes,Cresswell 68] e [Gabbay 84]. Esta última referência contém a descrição também de várias outras extensões da lógica clássica.

Os sistemas de prova baseados em cálculos de tableau constituem uma maneira bastante legível de se investigar a validade de fórmulas arbitrárias de uma determinada lógica. Além disso, como pode-se pensar em um tableau como uma exploração de um contra-modelo hipotético para uma fórmula particular, se o tableau não termina com sucesso nòs obtemos também um modêlo no qual a fórmula não é válida.

Como usualmente, adotamos uma versão da semântica de Kripke que se utiliza de fórmulas com sinal ([Fitting 83]). Uma fórmula com sinal é denotada por $+F$ ou $-F$ onde F é uma fórmula e $+$, $-$ são dois novos símbolos formais. Nos sistemas de tableau prefixado as provas são escsritas em forma de árvore, com ramificações descendentes. Em cada nodo da árvore existe uma fórmula prefixada. Uma fórmula prefixada é denotada por $\sigma\ F$ onde σ é um prefixo (sequência finita de inteiros positivos) e F é

uma fórmula com sinal. Uma tentativa de se provar F inicia-se com uma árvore de um ramo e um nodo que contém a fórmula $1 - F$. Então a árvore é aumentada através do uso de certas regras de extensão. Por exemplo, se $\sigma + (F \wedge G)$ ocorre em um ramo, então $\sigma + F$ e $\sigma + G$ devem ser adicionadas ao fim do ramo. Como outro exemplo, se $\sigma + (F \vee G)$ ocorre em um ramo, este deve se bifurcar sendo que $\sigma + F$ deve ser acrescentado ao ramo esquerdo e $\sigma + G$ deve ser acrescentado ao ramo direito. Um conjunto de regras de extensão será dado mais adiante. Um ramo é dito fechado se ele contém $\sigma + F$ e $\sigma - F$ para alguma fórmula F e algum prefixo σ. Um tableau (ou árvore) é dito fechado se cada um dos seus ramos for fechado. Um tableau fechado para $\sigma - F$ é, por definição , uma prova da fórmula F.

Apresentaremos aqui uma versão tentativa de um conjunto de regras de extensão para a componente de primeira ordem da lógica LDA incluindo apenas ações . Segue-se aqui um procedimento de prova análogo ao descrito em [Bittel,Alencar 88] para o caso da lógica intuicionista. Adotando a notação unificadora para fórmulas ([Smullyan 68]), definimos as seguintes regras de extensão:

$$(\alpha) \quad \frac{\sigma\,\alpha}{\sigma\,\alpha_1,\ \sigma\,\alpha_2}$$

$$(\beta) \quad \frac{\sigma\,\beta}{\sigma\,\beta_1 \mid \sigma\,\beta_2}$$

$$(\gamma) \quad \frac{\sigma\,\gamma}{\sigma\,\gamma(a)}$$

$$(\delta) \quad \frac{\sigma\,\delta}{\sigma\,\delta(a')}$$

$$(\nu) \quad \frac{\sigma + [\alpha]\,\phi}{\sigma'\,\phi}$$

$$(\pi) \quad \frac{\sigma - [\alpha]\,\phi}{\sigma''\,\phi}$$

Algumas observações devem ser feitas a respeito destas regras. Primeiramente, as fórmulas do tipo α são a positiva que contém o conectivo $\wedge$ e as negativas que contém os conectivos $\vee$, $\rightarrow$ e $\neg$. Suas componentes α_1 e α_2

tem sinais $(+,+)$, $(-,-)$, $(+,-)$ e $(+,+)$, respectivamente. As fórmulas do tipo β são a negativa que contém o conectivo $\wedge$ e as positivas que contém os conectivos $\vee$, $\rightarrow$ e $\neg$. Suas componentes α_1 e α_2 tem sinais $(+,+)$, $(-,-)$, $(-,+)$ e $(-,-)$, respectivamente. O tipo de fórmulas β é o único que envolve ramificação no tableau. As fórmulas do tipo γ são a positiva que contém o conectivo $\forall$ e a negativa que contém o conectivo $\exists$. Suas componentes $\gamma(a)$ tem sinais $(+)$ e $(-)$, respectivamente, onde a é qualquer constante. As fórmulas do tipo δ são a negativa que contém o conectivo $\forall$ e a positiva que contém o conectivo $\exists$. Suas componentes $\delta(a')$ tem sinais $(+)$ e $(-)$, respectivamente, onde a' é nova no ramo. Além disso, σ' é um prefixo já usado no ramo ou é uma extensão simples irrestrita de σ e σ'' é uma extensão simples irrestrita de σ.

Como mostramos na seção 4 deste artigo, certas alterações na configuração de um sistema são convenientes por preservarem, por exemplo, determinadas condições de boa formação da configuração . Este foi o caso quando incluimos na descrição do sistema o recurso fornecido p_3 à configuração C_1. A demonstração sumarizada abaixo mostra a conveniência de se incluir na descrição do sistema este recurso, uma vez que é preservada a boa formação parcial definida pela primeira condição de boa formação de uma configuração , segundo o sistema de tableau apresentado acima. Nesta prova um modo especial de numeração é usado. Um número mostra em que nodo da árvore de prova a fórmula correspondente aparece. A raiz tem nodo 1. Se um nodo com número $n.s$ tem somente um sucessor então este receberá o número $n+1.s$. No caso de dois sucessores, estes receberão os números $n+1.s.1$ e $n+2.s.2$. Com isso, n expressa a profundidade da prova e a sequência após n mostra a que ramo o nodo pertence. Esta numeração não deve ser confundida com os prefixos associados às fórmulas. Após cada fórmula (exceto as das folhas da árvore de prova) expressamos que regra será usada a seguir ou a inclusão de um axioma da teoria que descreve o sistema.

1 $\quad 1- \ [Incluir\_rf(p_3, C_1)] \ [Checa\_bf(C)] \ Bf(C) \quad (\pi)$

2 $\quad 1+ \ [Incluir\_rf(p, C)] \ fornecido(p, C) \quad (ax.)$

3 $\quad 1,1- \ [Checa\_bf(C)] \ Bf(C) \quad (de\ 1)$

4 $\quad 1,1+ A(C) \rightarrow$
$\quad\quad [Checa\_bf(C)] \ Bf(C) \quad (ax., + \rightarrow)$

5.1 $1,1 - \forall p \,.\, \{fornecido(p,C) \rightarrow$
$\exists C' \,.\, membro(C',C) \wedge fornecido(p,C')\}$ $(-\forall)$

5.2 $1,1 + [Checa\_bf(C)]\ Bf(C)$ $(closed,3)$

6.1 $1,1 + membro(C_1,C)$ $(ax.)$

7.1 $1,1 + membro(C_2,C)$ $(ax.)$

8.1 $1,1 + fornecido(p_1,C)$ $(ax.)$

9.1 $1,1 + fornecido(p_2,C)$ $(ax.)$

10.1 $1,1 + fornecido(p_1,C_1)$ $(ax.)$

11.1 $1,1 + fornecido(p_2,C_2)$ $(ax.)$

12.1 $1,1 - \{fornecido(p',C) \rightarrow$
$\exists C' \,.\, membro(C',C) \wedge fornecido(p',C')\}$ $(- \rightarrow)$

13.1 $1,1 + fornecido(p',C)$

14.1 $1,1 - \exists C' \,.\, membro(C',C) \wedge fornecido(p',C')\}$ $(-\exists)$

15.1 $1,1 - membro(C_1,C) \wedge fornecido(p',C_1)\}$ $(-\wedge)$

16.1.1 $1,1 - membro(C_1,C)$ $(closed,6.1)$

16.1.2 $1,1 - fornecido(p',C_1)$

17.1.2 $1,1 + fornecido(p',C_1)$ $(closed,2)$

Válida

Concluimos notando que este cálculo somente inclui a componente de ações da lógica LDA, devendo portanto ser extendido ainda para permitir derivações a respeito do caráter deôntico das ações , a inclusão da igualdade, o tratamento do aspecto polisortido e a inclusão de regras que tratem provas que utilizam tipos de dados primitivos de mais alto nível como conjuntos e sequências.

9 Referências

[Alencar, Lucena 88] Alencar, P.S.C., Lucena, C.J.P. *Métodos Formais para o Desenvolvimento de Programas*, Editorial Kapelusz, IV EBAI, Buenos Aires, Argentina, 1988.

[Babich 86] Babich, W., *Software Configuration Management*, Addison-Wesley, Reading, MA, 1986.

[Bershoff 80] Bershoff, E., Henderson, V., Siegel, S., *Software Configuration Management*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[Beth 59] Beth, E. W., *The foundations of Mathematics*, North-Holland, Amsterdam, 1959.

[Bittel,Alencar 88a] Bittel, O., Alencar, P.S.C., *Towards a Tableau Based Intuitionistic Theorem Prover*, RT no. 3, Departamento de Informática, PUC-RJ, 1988.

[Bittel, Alencar 88b] Bittel, O., Alencar, P. S. C., *Program Construction with an Intuitionistic Sequent Calculus*, Anais da IV Reunião de Trabalho do Projeto Estra, 1988.

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

[Enderton 72] Enderton, H. B., *A mathematical Introduction to Logic*, Academic Press, 1972.

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

[Gabbay 84] Gabbay, D., Guenthner,F. (eds.), *Handbook of Philosophical Logic*, Vols. I, II,D. Reidel Pub. Co., 1984.

[Goldblatt 82] Goldblatt, R., *Axiomatizing the Logic of Computer Programming*, Lecture Notes in Computer Science, Springer-Verlag, 1982.

[Hintikka 55] Hintikka, J., *Form and Content in Quantification Theory*, Acta Philosophica Fennica, vol. 8, p. 7-55, 1955.

[Hughes, Cresswell 68] Hughes, G. E., Cresswell, M. J., *An Introduction to Modal Logic*, Menthuen, London, 1968. Segunda edição 1972.

[Jones 80] Jones, C. B., *Software Development: a Rigorous Approach*, Prentice-Hall, International Series in Computer Science, 1980.

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

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

[Kripke 63] Kripke, S. A., *Semantical Considerations on Modal Logic*, Acta Philosophica Fennica, vol. 16, p. 83-94, 1963.

[Nakajima 80] Nakajima, R., Honda, M., Nakahura, H., *Hierarchical Program Specification and Verification - A Many-sorted Approach*, Acta Informatica, vol. 14, 1980.

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

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

[Prawitz 75] Prawitz,D., *Comments on Gentzen-style Procedures and the Classical Notion of Truth*, em Diller, J. e Muller, G. H., (eds.), Proof Theory Symposium, Lecture Notes in Mathematics 500, Springer, Berlin, p. 290-319, 1975.

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

[Sufrin 84] Sufrin, B. A., *Notes for a Z Handbook: Part I - The Mathematical Language*, Programming Research Group, University of Oxford,1984.