Parametrização de Componentes de Especificação com Preservação de Semântica

Anamaria Martins Moreira[1]
Universidade Federal do Rio Grande do Norte — UFRN
Departamento de Informática e Matemática Aplicada — DIMAp
Campus Universitário de Lagoa Nova
59072-970 Natal, RN, Brasil
anamaria@dimap.ufrn.br

Resumo

Para que um componente de um produto tenha maiores chances de ser reutilizável em um contexto diferente daquele para o qual ele foi criado ele deve ser "suficientemente geral". No caso de componentes de software, essa generalidade pode ser obtida por meios de parametrização, e podemos a esse respeito identificar 2 casos de figura: componentes altamente parametrizados concebidos com vistas a sua reutilização, e componentes específicos que se quer reutilizar a posteriori. É esse segundo caso que consideramos aqui: o que fazer para aumentar o potencial de reutilização de componentes desenvolvidos para uma aplicação específica. Com esse objetivo, propomos aqui as bases teóricas de uma ferramenta para a parametrização de componentes de software a partir de sua especificacao formal. A ideia subjacente é, dado um componente e um conjunto de propriedades semânticas satisfeitas por ele e que se quer preservar, encontrar um componente parametrizado do qual o componente original é uma instanciação possível e tal que possamos garantir que toda nova instanciação satisfará as propriedades citadas. A técnica utilizada é a *generalização de provas*, através da qual determinamos os requisitos mínimos a serem exigidos de possíveis parâmetros efetivos para que as propriedades em questão sejam preservadas. Finalmente, o contexto ao qual aplicamos essas ideias é o de especificações algébricas, utilizado na especificação de componentes e cujo mecanismo de genericidade nos fornece um contexto ideal para esse trabalho.

Palavras-Chave: parametrização de componentes, reutilização de componentes, especificações formais.

Abstract

For a product component to have a high probability of reuse in some context which is not the one for which it was created, it must be "general enough". In the case of software components this generality may be obtained by parameterization and we can identify 2 situations: highly parameterized components conceived with the goal of being reused, and specific components that one tries to reuse *a posteriori*. This second situation is the one considered in this paper: what must be done in order to increase the reuse potential of components first developped for a specific application. With this goal in mind, this paper proposes the theorectical basis of a tool for the parameterization of software components based on their formal specification. The underlying idea is, given a component and a set of semantic properties that it satisfies and that we want to preserve, to find a parameterized component satisfying the following conditions: the original component is one of its possible instantiations, and we can garantee that any of its instantiations satisfy the stated properties. To reach this goal we use *proof generalization*, through which it can be determined the minimum requirements on possible efective parameters so that these stated properties are preserved. Finaly, the context to which these ideas are applied is the one of algebraic specifications, used in the specification of components and which often includes a genericity feature that provides us with an ideal framework for this work.

Keywords: component parameterization, component reuse, formal specification.

[1]Projeto parcialmente financiado pelo CNPq.

1 Introdução

A reutilização é considerada por muitos uma técnica essencial para a Engenharia de Software. No entanto, apesar de muitos trabalhos estarem sendo desenvolvidos nessa área, ainda não há uma solução definitiva para as dificuldades que impedem que melhores resultados sejam obtidos. A previsão, de qualquer maneira, é que essa solução dificilmente será única, como indicam os sucessos parciais de diversas técnicas e artigos como [6].

No trabalho que apresentamos aqui, estudamos a reutilização baseada em bibliotecas de componentes de software. Mais especificamente, queremos contribuir para a solução do problema da construção de tais bibliotecas, ou seja, a programação *para* a reutilização. Essa é nossa preocupação primeira; no entanto, a teoria que apresentamos aqui pode ser extendida para se aplicar à segunda etapa do processo de reutilização, a programação *com* reutilização.

Dentro desse contexto, uma decisão importante é a decisão entre criar componentes para reutilização futura ou criar componentes reutilizáveis a partir de uma aplicação já desenvolvida. Ambas as técnicas têm seus prós e contras. A segunda tem como principal vantagem que o software gerado já foi utilizado pelo menos uma vez, como defende M. Wirsing em [14]. Nesse caso, a principal dificuldade é transformar componentes desenvolvidos para uma aplicação específica em componentes efetivamente reutilizáveis. Isso deve ser feito através da sua generalização. Sommerville, em [13], fala desse processo de generalização, que inclui, por exemplo, uma mudança de nomes de objetos definidos no componente de maneira a abstrair da aplicação específica para o qual esse componente foi criado. A teoria que apresentamos aqui tem como principal objetivo servir de apoio nesse processo de generalização, dado que uma ferramenta construída sobre ela nos dará meios para automatizar, ao menos parcialmente, esse processo. Com uma tal ferramenta ficará mais fácil garantir a correção das transformações efetuadas com vistas a obter um componente reutilizável, evitando desde erros simples de renomagem parcial (um identificador que tem apenas algumas de suas ocorrências renomeadas) até erros complexos como geração de inconsistências semânticas no componente.

Um outro problema que se apresenta é a escolha entre reutilização de código e reutilização de especificações. Escolhemos trabalhar com especificações (especificações formais), basedos em argumentos como os seguintes:

- quanto mais cedo no processo de desenvolvimento for tomada a decisão de reutilizar um componente, maior será o ganho obtido pela reutilização;
- o maior nível de abstração de uma especificação a faz mais facilmente reutilizável do que uma implementação específica;
- especificações formais apresentam fundamentos matemáticos que permitem que ferramentas possam "entender" e manipular esses componentes em função de sua semântica.

Finalmente, dentre os formalismos de especificação existentes, utilizamos o formalismo algébrico, adequado para a especificação de componentes em geral. Em particular, as especificações algébricas são adequadas para a especificação de objetos e classes de linguagens

de programação orientadas a objetos, paradigma defendido por muitos como estimulador da reutilização. Trabalhos associando especificações algébricas e orientação objetos são comuns (p.ex. [12, 5]), e pode-se esperar fazer uso deles em uma futura aplicação da teoria que apresentamos. Adicionalmente, os mecanismos de genericidade presentes nas linguagens de especificação algébricas nos fornecem uma grande flexibilidade quanto ao nível de abstração desejado, fator que se mostrará de grande importância para esse trabalho, como veremos a seguir.

O que propomos nesse artigo são então meios para identificar o parâmetro formal que deve ser utilizado na parametrização de um componente de especificação algébrica de maneira a generalizá-lo preservando parte de sua semântica, mesmo que essa semântica dependa originalmente de propriedades dos objetos que foram substituídos por esse parâmetro formal. Para obter esse resultado, partimos das provas dessas propriedades no contexto original (não generalizado) e identificamos as condições para que essas provas sejam reprodutíveis no contexto generalizado. Essa técnica nos fornece condições suficientes para a validade das propriedades consideradas nos modelos da especificação mais geral, com a vantagem de possuir um algoritmo simples e eficiente para sua implementação.

Esse artigo está estruturado da seguinte forma: na seção 2 fornecemos uma noção superficial do que é uma linguagem de especificação algébrica e introduzimos conceitos e notações que utilizamos; na seção 3 definimos um operador de generalização por parametrização e apresentamos dois tipos de generalização — sintática e semântica; na seção 4 mostramos como generalizar de maneira preservar a validade de provas das propriedades semânticas selecionadas; para finalmente concluirmos na seção 5.

2 Especificações Algébricas

Nessa seção apresentamos alguns princípios básicos de especificações algébricas que serão utilizados ao longo do texto. O objetivo dessa apresentação é apenas o de permitir que o leitor não especialista possa acompanhar as idéias expostas nesse artigo e de introduzir o leitor especialista à notação utilizada. Por essa razão, esses princípios são apresentados informalmente sempre que possível. Uma apresentação mais detalhada e formal da área de especificações algébricas pode ser encontrada em [15] (revisão bastante técnica da área) ou nas referências correspondentes a linguagens específicas, em geral mais acessíveis ao não especialista [2, 7, 8, 3]. Para o não especialista interessado em conhecer as origens da área, a referência histórica (bastante acessível) é o artigo [9], de J. Guttag e J. J. Horning.

Atualmente, as linguagens de especificação algébricas existentes possuem muitos aspectos em comum, e alguns aspectos particulares a cada uma. Nós procuramos trabalhar sobre esse núcleo comum, de maneira que o que propomos aqui pode ser aplicado a qualquer uma delas, apesar de utilizarmos LPG [2] como referência. LPG é a linguagem de especificação que utilizamos nos exemplos e como referência nessa introdução. Ela é uma linguagem lógico-funcional de especificação (e programação) com genericidade e lógica de Horn acrescida de disequações. Nesse trabalho nos limitamos a parte funcional da linguagem, implementada através de *lógica condicional* (lógica equacional com acréscimo de condições do tipo `se-então`).

2.1 Sintaxe

As entidades sintáticas de base de uma especificação algébrica com lógica condicional são os **sortes** (S) e os **operadores** (Op), que correspondem, grosso modo, aos tipos de dados e funções (ou procedimentos) de uma linguagem de programação. Os operadores, tal como funções em linguagens de programação, possuem um nome (identificador) e um perfil, que indica os sortes aos quais se aplica e o sorte resultante de sua aplicação. O par constituído pelo conjunto de nomes de sortes e pelo conjunto de operadores de uma especificação constitui uma **assinatura** ($\Sigma = (S, Op)$). A restrição para que um par (S, Op) constitua uma assinatura é que todos os sortes do perfil dos operadores de Op pertençam a S. Essa restrição corresponde à restrição de uso de identificadores em linguagens de programação, onde todo identificador utilizado deve ser declarado. Um operador zero-ário (sem parâmetros) é também chamado de **constante**.

Exemplo 1 (Assinatura) *Seja S_1 o conjunto de sortes {nat, seq-nat} e Op_1 o conjunto de operadores {0: nat, nil:seq-nat, cons: (nat,seq-nat) → seq-nat}. O par (S_1, Op_1) constitui uma assinatura e nil é uma constante dessa assinatura.*

É importante notar no exemplo acima que, apesar dos identificadores escolhidos darem indicações da semântica desejada para essas entidades, a assinatura em si é um objeto puramente sintático e não tem nenhuma implicação semântica. Nós retornaremos a esse ponto adiante quando tratarmos da semântica das especificações.

A partir dessas entitades básicas podemos construir termos e equações. Um **Σ-termo** (termo da assinatura Σ) é definido recursivamente pela aplicação de operadores de Σ a outros Σ-termos, com os casos de base sendo variáveis e as constantes da assinatura. Essa aplicação deve respeitar a especificação do perfil do operador da mesma maneira que uma chamada de função em uma linguagem de programação tipada deve ser feita com parâmetros efetivos que correspondam aos parâmetros formais de sua declaração. Uma **Σ-equação** simples (não condicional) é um par de Σ-termos (t_1, t_2) e será representada por $t_1 == t_2$. Uma **Σ-equação condicional** corresponde a uma igualdade que só deve ser válida se as equações presentes em sua condição forem satisfeitas. Equações condicionais são representadas por $t_1 == t_2$ **se** $e_1 \wedge ... \wedge e_n$, onde e_1 a e_n são equações simples.

Exemplo 2 (Termos e Equações) *Seja $\Sigma_1 = (S_1, Op_1)$ a assinatura do exemplo 1 e V uma coleção de variáveis[2] com x1,x2:nat e s1,s2:seq-nat. Exemplos de Σ_1-termos são: nil, cons(x1,nil), cons(x2,cons(x2,s1)). Exemplos de Σ_1-equações são: nil == cons(x1,nil) e cons(x1,nil) == cons(x2,nil)* **se** *x1 == x2.*

2.2 Semântica

Assinaturas e equações têm como contrapartida semântica classes de **álgebras**. Uma Σ-álgebra é constituída por um conjunto de valores para cada sorte da assinatura Σ e uma

[2]Em geral, na teoria, essa coleção de variáveis é deixada implícita; na prática, essas variáveis também devem ser declaradas com mecanismos que variam de uma linguagem a outra. Como é de praxe na área, omitiremos as referências à assinatura Σ, às equações E ou às variáveis V quando não houver ambigüidade.

função para cada operador da assinatura, com as esperadas correspondências de perfil. A classe de todas as álgebras associadas a uma assinatura Σ é notada $Alg(\Sigma)$. Uma Σ-E-álgebra, onde E é um conjunto de Σ-equações é uma Σ-álgebra que satisfaz as equações de E. A classe de todas as álgebras associadas a uma assinatura Σ que satisfazem as equações de E é notada $Alg(\Sigma, E)$. Essa classe possui uma álgebra especial "mínima", chamada de álgebra inicial[3]. Não cabe aqui entrar em maiores detalhes sobre esse assunto, mas veremos a seguir através de um exemplo quem é essa álgebra.

Exemplo 3 (Álgebras) *1. Considere a assinatura Σ_1 dos exemplos precedentes. Então temos que a álgebra composta pelo conjunto $\mathbb{N}$ dos inteiros naturais, com o zero correspondendo a 0, e o conjunto de seqüências de naturais, com a seqüência vazia associada a nil e a concatenação de um novo elemento a uma seqüencia associada a cons, é um modelo de Σ_1 (Σ_1-álgebra). Mas nat poderia também corresponder a {verdadeiro, falso} ou qualquer outro conjunto (idem para seq-nat) e ainda assim termos uma Σ_1-álgebra.*

2. Consideremos agora a equação nil == cons(0,nil). A álgebra dos naturais e seqüencias definida acima não a satisfaz, deixando de ser um modelo válido da especificação que a contenha.

3. Finalmente, temos que a álgebra inicial correspondente a Σ_1 é a sua álgebra de termos onde o conjunto correspondente a nat é {0} e o correspondente a seq-nat é {nil, cons(0,nil), cons(0,cons(0,nil)), ...}. Se consideramos a equação acima, obtemos a álgebra inicial correspondente ao par (Σ, E), i.e., nat interpretado por {0} e seq-nat por {nil}, pois a equação iguala a nil os termos construídos a partir de nil através do operador cons.

Na prática, classes de álgebras situadas entre a classe de todas as álgebras de (Σ, E), $Alg(\Sigma, E)$, e a classe de álgebras iniciais, $Init(\Sigma, E)$, são bastante interessantes. Por exemplo, no caso da assinatura Σ_1 acima, nos interessamos pela classe de todas álgebras correspondendo a seqüências de valores de um tipo qualquer (seqüências genéricas). Para definir esse tipo de classes utilizamos componentes parametrizáveis, e são exatamente esses componentes que nos interessam nesse trabalho.

2.3 Componentes

Uma especificação no nosso contexto é simplesmente um par *(assinatura, classe de modelos)*. Componentes são os objetos concretos através dos quais construímos ou *realizamos* essas especificações.

Em LPG podemos destacar dois gêneros de componentes que se distinguem por sua semântica: **tipos** (`type`) e **propriedades** (`property`)[4]. Em ambos, o componente é composto por:

[3]Na realidade uma classe de álgebras isomorfas, mas por simplicidade, nos referenciamos a essa classe como uma só álgebra.

[4]Esses componentes se encontram, com pequenas variações, na maioria das linguagens de especificação algébrica, o que muda de uma para outra é principalmente o nome dado a eles.

- uma parte local — sortes e operadores introduzidos pelo componente e equações sobre eles;
- uma parte importada — especificações realizadas por outros componentes e utilizadas na especificação dos objetos locais. A parte importada pode ser dividida em dois tipos de importação com objetivos distintos: a importação fixa ou específica e o parâmetro formal, como esclarecemos em seguida.

Um componente pode ser representado por

$$Co = \Psi(\langle sp_I, sp_F \rangle, \Sigma_{Co}, Eq)$$

onde Ψ é T ou P se o componente for um tipo ou uma propriedade, respectivamente; sp_I, a importação específica, sp_F, o parâmetro formal, Σ_{Co}, a assinatura (local + importada), e Eq as equações acrescentadas pelo componente.

A semântica desses componentes é definida em termos de teoria de categorias e não a reproduzimos formalmente aqui. A intuição por trás dessa definição é apresentada a seguir.

Semântica de Tipos A semântica de um tipo é construtiva, no sentido que construímos os modelos da especificação implementada pelo tipo a partir dos modelos da especificação importada. Isso é feito de maneira similar à construção da álgebra de termos do exemplo 3. Se a especificação importada sp_I só possui o modelo inicial, $Init(\Sigma_I, E_I)$, ou se não houver importação, a nova especificação sp possuirá apenas o modelo inicial correspondente, $Init(\Sigma_{Co}, Eq)$. No caso de a especificação importada representar um parâmetro formal com uma maior classe de modelos, temos a especificação de um tipo genérico, como no exemplo de seqüências genéricas acima. O nome usual para esse tipo de semântica é *semântica construída livremente sobre uma classe de modelos* ou *semântica livre*. Nesse texto também é usado o termo mais intuitivo *semântica específica*.

Semântica de Propriedades As propriedades têm uma semântica mais flexível, aceitando como modelos qualquer álgebra que satisfaça as equações presentes na propriedade. Mas com propriedades há também uma maneira de obter uma classe de modelos intermediária, no caso, mais restrita, através de importação. Todo modelo de uma propriedade que importe uma especificação específica, como por exemplo os booleanos, deve preservar esse modelo específico importado. Esse é o caso quando queremos especificar uma relação de ordem, por exemplo, pois uma relação de ordem, independentemente do sorte que ordena, faz referência aos valores verdadeiro e falso dos booleanos.

2.4 Morfismos de Especificação

Morfismos de especificação são um conceito herdado da teoria de categorias e da noção de homomorfismo da álgebra. Eles servem para indicar se uma especificação pode ser "encontrada" de alguma maneira dentro de uma outra especificação. Não entraremos em detalhes aqui, mas precisaremos desse conceito no momento de definir a generalização de um componente. Exemplos de morfismos são:

- a especificação de seqüências de naturais está associada à especificação genérica de seqüências por um morfismo que indica que os naturais são um caso particular do sorte genérico (e ao mesmo tempo indica todas as associações que são conseqüência desta);
- a especificação dos naturais com a adição está associada à especificação da comutatividade de um operador binário por um morfismo que associa `nat` ao sorte da propriedade e + ao operador comutativo.

2.5 Um Exemplo em LPG

Essa seção apresenta uma especificação bastante simples que é utilizada ao longo do texto como apoio às explicações fornecidas. É uma especificação de seqüências de inteiros naturais como vimos acima, com duas operações adicionais: uma operação que calcula a soma dos elementos de uma seqüência dada e uma operação de teste que indica se uma seqüência é ou não a seqüência vazia.

```
type Seq_Nat_Ops
imports Nat, Bool
sorts seq_nat
constructors
  nil: -> seq_nat
  cons: (nat,seq_nat) -> seq_nat
operators
  sum: seq_nat -> nat
  is_empty: seq_nat -> bool
equations
  1: sum(nil) == 0
  2: sum(cons(x,s)) == x + sum(s)
  3: is_empty(nil) == true
  4: is_empty(cons(x,s)) == false
theorems
  1: sum(cons(x,nil)) == x
end Seq_Nat_Ops
```

```
property Com
sorts t
operators bin: (t,t) -> t
equations
  1: bin(x,y) == bin(y,x)
end Com

property Neutro_Direita
sorts t
operators bin: (t,t) -> t
          k: ->t
equations
  1: bin(x,k) == x
end Neutro_Direita
```

Esse componente importa as especificações dos inteiros naturais (`Nat`) e dos booleanos (`Bool`), que não reproduzimos aqui. O leitor pode assumir especificações com as operações usuais, e quando alguma em particular se fizer necessária em um exemplo, a explicitaremos. No caso da especificação acima, utilizamos a constante 0 e o operador de adição dos naturais e as constantes verdadeiro (`true`) e falso (`false`) dos booleanos. O parâmetro formal de um componente, não existente no exemplo é indicado pela palavra-chave `requires`.

Até agora não falamos de uma classe especial de operadores que ocorre nesse componente: os operadores construtores. O uso de construtores é um recurso sintático para dizer que os termos construídos apenas com esses operadores constituem o conjunto de valores associados aos sortes correspondentes. No caso do exemplo, os valores válidos de

seqüências são determinados pela aplicação repetida de `cons` a seqüências menores, até a seqüência vazia `nil`. Os outros operadores introduzidos no componente devem então ser recursivamente definidos em função dos construtores. É o que é feito para os operadores `sum` e `is_empty` acima. As equações correspondentes são também chamadas de *axiomas* da teoria definida pelo componente. São elas que serão utilizadas posteriormente na prova das propriedades semânticas satisfeitas por cada componente, como veremos na seção 4.

Nesse exemplo temos apens uma dessas propriedades semânticas explicitadas: é o teorema 1, que indica que a soma dos valores dos elementos de uma seqüência com apenas 1 elemento é o valor do próprio elemento. Como essa equação não é incluída no componente como axioma, mas sim como teorema, ela precisa ser uma conseqüência lógica da especificação, e por isso deve ser provada. É da preservação desses teoremas através da preservação de suas provas que tratamos aqui.

3 Generalização de Componentes

Essa seção define o operador de generalização de componentes por parametrização e discute em seguida os diferentes níveis de generalização que podemos obter. O operador de generalização, seção 3.1, é basicamente uma ferramenta para, uma vez definida a generalização desejada, executar as modificações correspondentes de maneira segura. A parte mais delicada do processo é no entanto a definição da generalização desejada. Para isso, propuzemos um algoritmo de generalização sintática do componente e técnicas para a sua generalização semântica em [10, 11]. Nas seções 3.2 e 3.3 apresentamos sucintamente o que entendemos por generalização sintática e semântica respectivamente.

3.1 Operador de Generalização

A operação de generalização de componentes de que tratamos aqui tem como principal efeito a substituição segura de parte da especificação importada por um parâmetro formal do qual a especificação substituida é uma especialização. Isso corresponde a aumentar a classe de modelos sobre a qual estamos construindo nossa especificação (implementada pelo componente generalizado). Como conseqüência, a classe de modelos que satisfazem a nova especificação é também uma expansão da original.

A definição formal desse operador é fornecida abaixo, seguida de uma explicação informal.

O *operador de generalização* (`generalize`) é definido por:

$Co'' =$ **generalize** Co **via** m **with** $Co' =$
se $Co = T(\langle sp_I, sp_F\rangle, \Sigma_E, Eq)$ **(1)**
$\wedge\ Co' = P(\langle sp'_I, sp_\phi\rangle, \Sigma', Eq')$ realiza sp' **(2)**
$\wedge\ m \in HomSpec(sp', sp_R)$ e é um monomorfismo **(2)**
$\wedge\ (\forall s \in \Sigma_R, \forall op \in ImpOps(Co), s \in m(\Sigma') \wedge s \in sortes(op) \Rightarrow op \in m(\Sigma'))$ **(3)**
então
se $\forall s \in \Sigma_R, s \in sortes(Op_L \cup ImpOps(Co)) \Rightarrow s \in m(\Sigma')$ **(4)**
então $T(\langle sp_\phi, sp'\rangle, \Sigma' \cup m_g(\Sigma_L), m_g^{\#}(Eq))$

senão se $\forall s, op \in \Sigma' \cap \Sigma_R,\ m(s) = s, m(op) = op$ **(5)**
então $T(\langle sp_I, sp_F \cup sp' \rangle, \Sigma_R \cup \Sigma' \cup m_g(\Sigma_L), m_g^{\#}(Eq))$
senão $\perp$

I.e.: dados um componente a generalizar Co, um morfismo de generalização m, e um componente candidato a parâmetro formal Co'; se **(1)** Co é um tipo bem definido; se **(2)** o morfismo proposto m é um morfismo de especificações injetivo de sp', realizada pela *propriedade* Co', em sp_R, a especificação importada resultante de Co; e se **(3)** esse morfismo alcança todos os operadores cujos perfis contém os sortes generalizados (os sortes que ocorrem no morfismo), de tal maneira que as equações do componente continuem bem formadas com respeito à nova assinatura após as mudanças de nomes; então:

1. se **(4)** m satisfaz a *condição de generalização*, sp' passa a ser o parâmetro formal do novo componente, substituindo a totalidade das importações do componente, com a modificação correspondente na assinatura exportada (a condição de generalização visa garantir que não deixamos de importar partes de especificação que ainda serão necessárias após a generalização);

2. se **(5)** a condição de generalização não é satisfeita, mas a inclusão de sp' como parâmetro formal através de m não gera incompatibilidade de nomes, ela é simplesmente incluída, sem que as importações originais sejam alteradas;

3. se nenhum dos casos acima se aplica, a operação de generalização não é definida.

Nos casos onde a operação é definida, os objetos da imagem de m (na parte local da assinatura exportada e nas equações locais ao componente) têm seus nomes modificados de acordo com o morfismo inverso m^{-1}.

É importante ressaltar que, apesar da aparência complexa da fórmula acima, a maioria das condições indicadas são sintáticas e facilmente verificadas através de algoritmos simples. A única complexidade potencial é a verificação da condição de morfismo, pois dependendo do caso, pode implicar na execução de provas de teoremas equacionais. Na prática, essa informação estará provavelmente registrada nos próprios componentes, sendo verificada apenas uma vez.

3.2 Generalização Sintática

O ponto mais delicado de um processo de generalização é provavelmente a escolha do nível de generalidade que se deseja. Em um caso extremo, todo componente poderia ser generalizado até a especificação vazia. Obviamente que esse tipo de resultado não nos interessa (tudo é um modelo da especificação vazia), e por isso precisamos fixar limites mais razoáveis. A convenção que adotamos, utilizada na definição do operador de generalização vista na seção 3.1, é que apenas podemos abstrair dos objetos definidos externamente ao componente considerado, definindo assim uma operação dual à instanciação.

Dentro desse contexto, ainda temos a escolha das partes da especificação importada que queremos generalizar e o quanto as queremos generalizar. O primeiro ponto, corresponde

a determinar que sortes e operadores devem ser generalizados, determinando uma assinatura. Definida essa assinatura, podemos ainda decidir sobre as propriedades semânticas da especificação original que queremos manter ou abandonar. O caso onde se generaliza uma especificação sem se ocupar de suas propriedades semânticas é chamado de *generalização sintática*. A generalização sintática é unicamente definida pela assinatura que se quer generalizar.

Exemplo 4 *Já vimos que seqüências de naturais podem ser generalizadas em seqüências genéricas de um sorte qualquer. No entanto, se quizermos definir seqüências genéricas a partir do componente* `Seq_Nat_Ops` *da seção 2.5, será necessário que esse sorte possua ao menos um operador binário para "substituir" a adição natural e uma constante para o 0. A propriedade correspondente definiria então um sorte* `t`*, uma constante* `k: -> t` *e um operador binário* `bin: (t,t) -> t`*. Qualquer sorte com uma constante e um operador binário com o perfil correspondente é então um modelo dessa propriedade e um parâmetro efetivo potencial para as seqüências correspondentes.*

Em [10, 11] propusemos um algoritmo para, dado o conjunto de sortes importados que se quer generalizar em um componente, definir a propriedade e o morfismo correspondente à máxima generalização que se pode obter. Também mostramos que isso sempre é possível dado que o conjunto das generalizações sintáticas possíveis de um componente constitui um reticulado completo.

A generalização sintática é muitas vezes demasiado, dado que perdemos todas as propriedades semânticas que dependem dos sortes generalizados. Ela apresenta no entanto duas características interessantes. Primeiramente, ela serve como ponto de partida para a generalização semântica. Segundo, é esse tipo de parametrização que encontramos em linguagens de programação com genericidade como Ada [1]. Nessas linguagens, apenas a assinatura do parâmetro é especificada na interface de um módulo genérico. Para que se possa acrescentar condições semânticas é necessário expandir a linguagem.

3.3 Generalização Semântica

O problema da generalização de um componente com preservação de propriedades semânticas representadas por teoremas equacionais pode ser enunciado como

> *generalizar um conjunto de sortes importados de um componente* Co *preservando a validade de um conjunto dado* Teo *de teoremas de maneira a poder recuperar o componente original por re-instanciação.*

Uma ampla discussão sobre esse assunto pode ser encontrada em [11]. Aqui, no entanto, nos interessamos por um problema mais restrito, que pode ser enunciado como

> *generalizar um conjunto de sortes importados de um componente* Co *preservando a validade de um conjunto dado* P *de provas de um conjunto* Teo *de teoremas de maneira a poder recuperar o componente original por re-instanciação.*

Essa segunda condição (preservação de provas) é evidentemente mais forte do que o problema original exige: a validade de uma prova implica na validade do teorema, mas o contrário nem sempre é verdade. Com essa técnica obtemos condições suficientes, mas não obrigatoriamente necessárias, para a validade dos teoremas em questão. A principal vantagem dessa técnica é sua simplicidade de implementação, pois evitamos o problema de encontrar uma prova para um teorema e o substituímos pelo problema mais simples de validar uma prova já conhecida. A seção 4 desse artigo fornece um algoritmo para a solução desse problema.

Independentemente da condição considerada, a maneira de garantir que o componente generalizado satisfará um determinado conjunto de axiomas utilizados na prova dos teoremas é incluir esses axiomas no parâmetro formal do componente. A semântica dos componentes (ver seção 2) vai então garantir que esses axiomas serão válidos na especificação realizada pelo componente generalizado, podendo ser utilizados para provar a validade dos teoremas correspondentes.

Exemplo 5 *Já vimos que* `Seq_Nat_Ops` *da seção 2.5 pode ser generalizada pela propriedade do exemplo 4. Mas se quizermos que a validade do teorema do componente seja preservada, é necessário que a constante* `k` *seja um elemento neutro da operação* `bin`. *Essa condição deve então ser acrescentada à propriedade parâmetro formal.*

4 Preservação de Provas

Como indicado na seção 3.3, queremos encontrar uma propriedade e um morfismo que substituam uma dada importação sem que um conjunto provas para um conjunto selecionado de propriedades semânticas seja perdido. Nessa seção, consideramos 3 tipos de provas:

- direta (por reescritura);
- por casos;
- por indução estrutural;

onde as provas por casos e por indução estrutural são por sua vez construídas a partir de provas por reescritura.

Na seção 4.1, apresentamos os princípios básicos de provas por reescritura e fornecemos um algoritmo para obter essa propriedade e esse morfismo a partir de uma dessas provas. Em seguida, na seção 4.2, apresentamos os princípios de provas por casos e por indução estrutural e indicamos o que muda no problema de preservação dessas provas.

4.1 Provas por Reescritura

Princípios de Provas por Reescritura Sistemas de Reescritura são frequentemente utilizados na prova de teoremas em sistemas equacionais. Uma ótima referência sobre o assunto é [4]. Aqui, apresentamos apenas superficialmente seus princípios de maneira a poder falar da generalização desse tipo de provas.

A prova por reescritura se baseia na substituição de termos por termos (semanticamente) iguais. Uma prova por reescritura do teorema `t == t'` como a abaixo

- Reescritura do termo esquerdo —

```
t     ==  t1   (eq1)
      ==  t2   (eq2)
      ==  ...
      ==  tn   (eqn)
```

- Reescritura do termo direito —

```
t'    ==  t1'  (eq1')
      ==  t2'  (eq2')
      ==  ...
      ==  tn'  (eqn')
```

significa que: (1) `t` se reescreve em `tn`, notado $\texttt{t} \rightarrow_{\mathcal{R}}^{*} \texttt{tn}$, em `n` etapas. `ti` é o resultado da reescritura na etapa `i` e `eqi` é a equação utilizada nessa etapa. Idem para `t'` e `tn'`. (2) O teorema inicial `t == t'` está provado se `tn` e `tn'` são sintaticamente idênticos. (3) Se `tn` et `tn'` não são sintaticamente idênticos, a equação `tn == tn'` se transforma em lema que deve ser provado por algum outro método.

Generalização de Provas por Reescritura Para reproduzir uma prova por reescritura no contexto generalizado é necessário que todas as equações utilizadas na prova sejam ainda válidas em todos os modelos da nova especificação. A maneira de garantir essa condição é incluí-las como axiomas dessa especificação. No caso de uma generalização, alguns axiomas correspondentes à importação generalizada podem ter sido "perdidos" nesse processo. Eles devem então ser adicionados à propriedade parâmetro formal e a semântica dos componentes apresentada na seção 2.3 garantirá a validade desses axiomas na nova especificação.

Considere então $Eq(p)$ o conjunto de todas as equações utilizadas na prova p que se quer preservar e `SINT` a propriedade de generalização sintática correspondente. $Eq(p)$ pode ser dividido em dois sub-conjuntos $Eq^{g}(p)$ e $Eq^{ng}(p)$, correspondendo às equações generalizadas (perdidas) e não generalizadas, respectivamente. Para que a prova p possa ser reproduzida devemos acrescentar a `SINT` as equações de $Eq^{g}(p)$. Em alguns casos, essas equações podem incluir sortes e operadores que não faziam parte da assinatura de `SINT`, nesses casos, a assinatura deve ser expandida para obtermos um componente bem formado.

Dado um conjunto $GS = \{s_1, .., s_n\}$ de sortes a generalizar em um componente Co com preservação de um conjunto de provas P, o morfismo de generalização pode ser calculado pelo seguinte algoritmo:

1. Utilizar o algoritmo de generalização sintática [10][5] para obter a propriedade de generalização sintática `SINT`. Seja $GS_{\texttt{SINT}} = \{s_1, .., s_m\} \supseteq GS$ o conjunto de sortes a

[5] O algoritmo de generalização sintática é bastante similar a esse, com a diferença que suas iterações têm como objetivo apenas construir uma assinatura válida.

generalizar após a definição da generalização sintática. E seja $GO_{\text{SINT}} = \{op_1, .., op_l\}$ o conjunto de operadores generalizados.

2. Identificar o conjunto $Eq(P)$ das equações utilizadas em P.

3. Identificar o sub-conjunto $Eq^g(P, GS_{\text{SINT}})$ de equações de $Eq(P)$ perdidas pela generalização (equações com referência a sortes de GS_{SINT}).

$$Eq^g(P, GS_{\text{SINT}}) = \{e \in Eq(P) - Eq_{Co} | sortes(e) \cap GS_{\text{SINT}} \neq \phi\} = \{e_1, .., e_k\}$$

4. Identificar as referências pendentes $Pend(Eq^g(P, GS_{\text{SINT}}))$ dessas equações.

$$Pend(Eq^g(P, GS_{\text{SINT}})) = sortes(Eq^g(P, GS_{\text{SINT}})) - GS_{\text{SINT}}$$

5. Se $Pend(Eq^g(P, GS_{\text{SINT}})) = \phi$ então a generalização é definida para o morfismo que associa a propriedade `SEM` abaixo à especificação importada de Co.

```
property SEM
sorts s'_1, ..., s'_m
operators op'_1, ..., op'_j
equations e'_1, ..., e'_k
```

nota: Os objetos especificados em `SEM` são os sortes e operadores generalizados com novos nomes e as equações de $Eq^g(P, GS_{\text{SINT}})$ com as renomagens correspondentes.

6. Se o conjunto $Pend(Eq^g(P, GS_{\text{SINT}})) \neq \phi$, ele deve ser adicionado ao conjunto de sortes a generalizar e o algoritmo deve ser re-executado até que se obtenha uma propriedade e um morfismo para os quais esse conjunto é vazio e a generalização, definida.

Exemplo 6 *Considere o componente* `Seq_Nat_Ops` *da seção 2.5 e seu teorema*

`sum(cons(x,nil)) == x`

Queremos abstrair da sorte `nat`. *Para que a propriedade resultante seja bem formada, ela precisa incluir um sorte formal, uma constante e um operador binário, utilizados nas equações do componente a generalizar. Considere agora a seguinte prova do teorema:*

- *Reescritura do termo esquerdo —*

```
sum(cons(x,nil)) == x + sum(nil)   (eq2.Seq_Nat_Ops)
                 == x + 0          (eq1.Seq_Nat_Ops)
                 == x              (propriedade que vem de Nat)
```

- *Reescritura do termo direito —* `x`

Como temos uma identidade `x == x`, *o teoremas está provado. Para reproduzir a prova após a generalização precisamos da equação correspondente a* `x + 0 == x` *que foi perdida. Acrescentá-la a propriedade nos dá a propriedade* `Neutro_Direita` *apresentada na seção 2.5.*

4.2 *Provas por Casos e por Indução Estrutural*

Princípios de Provas por Casos As provas por casos que consideramos aqui são um conjunto de provas por reescritura, uma para cada caso considerado. Cada caso é caracterizado por uma conjunção de equações `caso_i : caso_i.1` $\wedge$ ... $\wedge$ `caso_i.k` e, incorporando essas equações ao sistema de reescritura, tentamos provar o teorema original `t == t'` sob a suposição `caso_i`. A prova para o i-ésimo caso é uma prova por reescritura com o sistema de regras ($\mathcal{R}_i$) obtido pela adição das asserções `caso_i.1,...,caso_i.k` ao sistema de regras original. Então:

- Se essa reescritura nos leva a uma igualdade sintática, o lema `t == t' se caso_i` está provado.
- O conjunto de casos é **completo** se

$$\bigvee_{i=1}^{m} \texttt{caso\_i}$$

onde `m` é o número de casos. Se o conjunto de casos é completo e todos os lemas correspondentes foram provados, o teorema original está provado.

Princípios de Provas por Indução Estrutural O princípio de prova por indução estrutural considera que os valores do domínio (os valores de um sorte) são alcançáveis a partir de um conjunto de valores de base e de uma ou mais operações. Um exemplo típico de indução estrutural é a indução sobre os inteiros naturais, onde o valor de base é 0 e a operação é a função sucessor (ou +1). Em uma indução sobre uma variável natural `n (n: nat)` devemos provar o teorema em questão para `n` = 0 e para `n` = *sucessor*(`n'`), partindo da hipótese que o teorema é válido para o valor `n'`. Nesse texto, as provas intermediárias são provas por reescritura, o que nos dá para uma indução sobre a variável `v`:

caso de base — (`v == base`) prova por reescritura de σ_b(`th`), onde $\sigma_b = \{\texttt{v} \mapsto \texttt{base}\}$[6].

passo de indução — (`v == ind`, onde `ind` corresponde à construção do valor de `v` a partir de um valor `v'`) prova por reescritura de σ_i(`th`), onde $\sigma_i = \{\texttt{v} \mapsto \texttt{ind}\}$ sob a hipótese `hyp` : σ'(`th`), onde $\sigma' = \{\texttt{v} \mapsto \texttt{v}'\}$. O sistema de regras utilizado é o original aumentado por `hyp`, como indicado acima para as provas por casos.

Então: Os casos de base são provados normalmente por reescritura e no passo de indução provamos o lema σ_i(`th`) `se hyp`. O princípio de indução garante então a validade do teroema original na totalidade do domínio.

[6]Substituição de cada ocorrência da variável `v` no teorema original pelo valor de base.

Generalização de Provas por Casos e por Indução Estrutural Em ambos os casos temos um conjunto de provas por reescritura e as condições que fazem com que esse conjunto de provas por reescritura constituam uma prova do teorema original. A maior dificuldade que se apresenta para a generalização dessas provas é sua parte não equacional: as condições de completude dos casos considerados ou de validade do princípio de indução. Essas condições dependem em geral da semântica dos componentes correspondentes (semântica construída livremente, seção 2.3). O que se pode fazer no entanto é garantir a preservação dos lemas equacionais que levam à validade do teorema, como mostramos abaixo.

No caso de uma prova por casos com n casos teremos n lemas da forma:

$$\texttt{t == t' se caso\_i}$$

onde `caso_i` é a conjunção que caracteriza o i-ésimo caso.

E no caso de uma prova por indução com k casos de base e p passos de indução teremos $n = k + p$ lemas da forma:

$$\sigma_i(\texttt{t == t' se caso\_i})$$

onde `caso_i` é a conjunção que caracteriza cada ramo da árvore de provas por indução e σ_i é a substituição correspondente.

5 Conclusões e Trabalhos Futuros

Os resultados apresentados nesse trabalho têm como objetivo contribuir para a área de reutilização de componentes de software através de suas especificações. Em particular, esses resultados representam a base teórica a partir da qual pode ser construída uma ferramenta para a construção de bibliotecas de componentes reutilizáveis através da parametrização de componentes específicos. Essa base teórica fornece meios para garantir a validade de propriedades semânticas dos componentes reutilizáveis baseados nas propriedades semânticas dos componentes a partir dos quais eles foram gerados.

O principal resultado apresentado nesse artigo é a proposta de um algoritmo para a identificação dos parâmetros que garantem a validade dessas propriedades a partir de suas provas diretas (por reescritura) no contexto original. Mostramos também que provas mais complexas (provas por casos e provas por reescritura) também podem beneficiar desse resultado, mas apenas parcialmente. Um dos próximos passos a serem estudados agora é uma melhoria dos resultados para esses tipos de prova.

Finalmente, precisamos também desenvolver um protótipo dessa ferramenta para efetuar estudos de caso e desenvolver o aspecto metodológico de apoio à criação de bibliotecas de componentes. Paralelamente, esses estudos de caso servirão para confirmar a escalabilidade dessa técnica a sistemas reais.

Referências

[1] Alsys. *Reference Manual for the Ada programming language*, 1983.

[2] D. Bert, R. Echahed, and J.C. Reynaud. Reference manual of the LPG specification language and environment (release with disequations). Technical report, LGI-IMAG, June 1994. disponível por ftp — site ftp.imag.fr.

[3] M. Bidoit. *Pluss, un Langage pour le Développement de Spécifications Algébriques Modulaires.* PhD thesis, Université de Paris-Sud, May 1989. Thèse d'État.

[4] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theorectical Computer Science*, chapter 15. Elsevier Science Publishers B.V., 1990.

[5] R. Hennicker e C. Schmitz. Object-oriented implementation of abstract data type specifications. In M. Wirsing and M. Nivat, editors, *AMAST'96*, number 1101 in LNCS. Springer-Verlag, 1996.

[6] D. Fafchamps. Organizational factors and reuse. *IEEE Software*, 11(5):31–41, Sep. 1994.

[7] S.J. Garland and J.V. Guttag. An overview of Larch. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, number 693 in LNCS, pages 329–348. Springer-Verlag, 1993.

[8] J.A. Goguen and T.C. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 1988.

[9] J. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, (10):27–52, 1978.

[10] P. Jacquet and A. Martins. Proposals for a methodic approach to generalization of specification components. In *Proceedings of the XXI SEMISH*, pages 193–207. Sociedade Brasileira de Computação, 1994.

[11] A. Martins. *La Généralisation : un Outil pour la Réutilisation.* PhD thesis, INPG, March 1995.

[12] F. Parisi-Presicce and A. Pierantonio. Reusing object oriented design: An algebraic approach. In E. Bertino and S. Urban, editors, *Object-Oriented Methodologies and Systems*, number 858 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[13] Ian Sommerville. *Software Engineering.* Addison-Wesley, 5ª edition, 1996.

[14] M. Wirsing. Algebraic description of reusable software components. Technical Report MIP-8816, Fakultät für Mathematik und Informatik - Universität Passau, 1988.

[15] M. Wirsing. Algebraic specification. In *Handbook of Theorectical Computer Science*, chapter 13. Elsevier Science Publishers B.V., 1990.