Modelo de Integração de Especificações: Transformação de *Use Cases* em Tipos Abstratos de Dados*

Luís André Martins, Guilherme Salum Rangel, Daltro José Nunes

Universidade Federal do Rio Grande do Sul (UFRGS)
Instituto de Informática
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
{lmartins,rangel,daltro}@inf.ufrgs.br

Resumo. Durante o processo de desenvolvimento de software, o sistema é especificado em diversos formatos e linguagens. No contexto da engenharia de requisitos, o sistema é especificado através de representações meramente informais ou semi-formais, pois um dos objetivos nesta fase é a comunicação com o cliente da aplicação, por isso a necessidade de formatos de representação intuitivos e não formais. A medida que o processo avança, o ideal seria a transformação das especificação informais e/ou semi-formais em representações formais, baseadas em modelos matemáticos, com sintaxe e semântica bem definidas, o que possibilita a verificação automatizada das especificações e o projeto formal do software. Dessa forma, a tendência seria a redução de erros e inconsistências nas especificações, reduzindo o tempo e os custos do projeto. Para atingir tal meta, é necessário uma metodologia eficiente para garantir a passagem correta entre especificações de requisitos e especificações formais. Este artigo trata exatamente deste problema, propondo um modelo de integração composto por uma metodologia e um conjunto de heurísticas, que combina use cases (modelo de especificação semi-formal orientado ao cliente), com tipos abstratos de dados (modelo de especificação algébrica orientado ao sistema). Para validar a proposta, é apresentado um estudo de caso ao final do artigo.

1 Introdução

A informação utilizada para construir especificações de requisitos está imersa no contexto social de usuários, clientes, gerentes, etc. Por isso, tendem a ter representações meramente informais. Por outro lado, informações utilizadas no contexto do projeto de software tendem a ser formais, representadas por linguagens com sintaxe e semântica formalmente definidas. Mas tanto a informação formal, que é independente de contexto, quanto à informação informal, situada no contexto social, são cruciais para o sucesso de projetos de engenharia de requisitos [3].

* Este trabalho é apoiado pela FAPERGS e pelo CNPq

linguagem natural. As heurísticas apresentadas pela autora visam determinar possíveis mapeamentos entre o léxico e os cenários com a linguagem algébrica RAISE. Este modelo de cenários é mais informal do que [9], o que provavelmente o torna mais apropriado para a validação dos requisitos junto ao cliente da aplicação. Mas em contrapartida, o modelo de [9] é mais estruturado, sem deixar de ser intuitivo, o que permite criar um conjunto menor e mais formal de heurísticas. Graças a isso, também pode-se propor uma ferramenta para apoiar o processo de geração da especificação formal sem a necessidade de análise de linguagem natural.

Em [1] é apresentado um formalismo e um método para extrair um tipo abstrato de dados executável a partir de uma descrição dinâmica. O autor parte de um método de derivação que cria uma especificação algébrica a partir da análise de uma máquina de estados finita. Este método cria o lado esquerdo das operações algébricas a partir das transições de estados, enquanto que o lado direito das operações (a semântica), deve ser especificado manualmente. A estrutura da especificação algébrica resultante é bastante semelhante com a apresentada neste artigo. Todavia, na proposta deste artigo, ao contrário de [1], é tomado um modelo de requisitos orientado ao cliente como base para a especificação algébrica, sendo que a máquina de estados serve simplesmente para tornar mais claro o relacionamento entre as operações algébricas resultantes. Uma descrição na forma de máquina de estados pode não ser apropriada para o diálogo com o cliente da aplicação. Além disso, o autor não apresenta nenhuma regra ou heurística para determinar a semântica das operações.

3 *Hierarquical Use cases*

O modelo de representação de *use cases* desta proposta é o *Hieraquical Use Cases* [9], um modelo hierárquico de *use cases* com representação gráfica. Este modelo faz parte do *Usage Oriented Requirements Engineering* (UORE) [10], que por sua vez é baseado na *Use Case Driven Analysis* [6]. No UORE, *use cases* servem para aquisição e validação dos requisitos funcionais.

Os principais conceitos envolvidos no modelo são:

- **Atores:** definem tipos de usuários do sistema, ou seja, representam um grupo de usuários com características em comum. Os usuários podem tanto ser humanos ou hardware/software externo;
- ***Use Case***: modela uma situação onde um ou mais serviços do sistema são utilizados por um ou mais usuários. Cada *use case* é estruturada como uma seqüência de episódios;
- **Episódio:** define uma seqüência de eventos através de uma linha de tempo. Um episódio pode ocorrer em mais de uma *use case*;
- **Evento:** um evento pode ser de três tipos: estímulos (mensagens do usuário para o sistema), respostas (mensagens do sistema para o usuário) ou ações (eventos intrínsecos do sistema, onde não acontece comunicação com o usuário).
  A construção do modelo, é estruturada em três níveis hierárquicos.

- ***Environment Level*:** neste nível são definidos os atores, serviços e *use cases* do sistema;

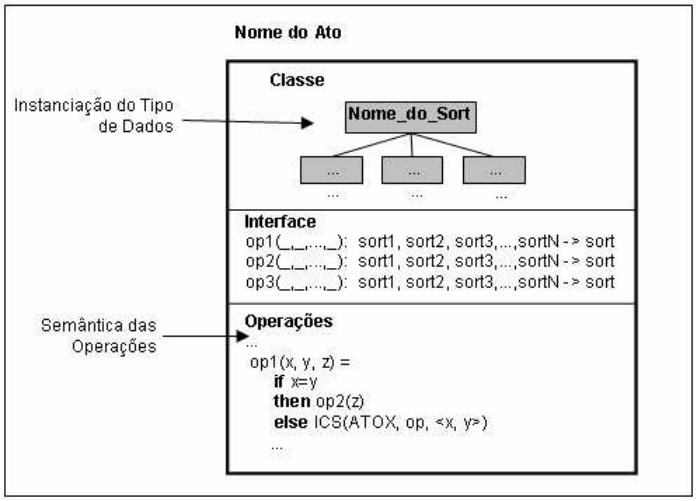


Fig. 1. Estrutura de um ATO

Todo ATO possui um nome, e na sua definição são especificados a sua **classe**, (*sort*) representada graficamente e definida através da composição de tipos de dados, a **interface** das operações e a especificação (semântica) das **operações**. Cada ATO trata apenas termos, chamados aqui de **objetos**, do *sort* por ele definido.

Os tipos de dados presentes no PROSOFT estão classificados em:

- **Primitivos:** Integer, String, Real, Char, Boolean, Date, Time;
- **Compostos:** construtores definidos no método algébrico: conjunto, lista, mapeamento, registro e união disjunta. Possuem representação gráfica;
- **Definidos pelo usuário:** são construídos a partir dos outros tipos.

Para facilitar a especificação, foi desenvolvida uma notação gráfica poderosa para a representação dos tipos primitivos e compostos (ver [8] para mais detalhes)

Cada um dos tipos primitivos e compostos traz consigo um conjunto de operações algébricas pré-definidas. Por exemplo, o tipo "Lista" possui as operações pré-definidas *concat*, *head*, *tail*, *length*, entre outras. Quando o usuário constrói um novo ATO, definindo o *sort* da especificação a partir dos tipos compostos e primitivos, as operações pré-definidas são "importadas" para a nova especificação, podendo ser aplicadas nas operações definidas pelo usuário para acessar e modificar termos do *sort*.

5 Modelo de Integração

O modelo de integração proposto irá combinar o modelo de *use cases* (semi-formal e orientado ao cliente) com o modelo algébrico do PROSOFT (formal e orientado ao sistema). Tem-se assim de um lado uma especificação orientada para a definição dos requisitos junto aos clientes da aplicação, dada a sua facilidade de entendimento, organização e estrutura, e do outro lado, uma especificação abstrata do sistema,

estado do sistema será representado através de uma instância do *sort* da especificação algébrica. Cada transição (evento) representará uma chamada a uma operação algébrica, responsável em transformar a instância do *sort.*

A máquina de estados é especificada através de operações algébricas, como mostrado na figura 2, que podem ser de dois tipos:

- **Operações de transição:** determinam as mudanças na instância do *sort*, ou seja, as mudanças no estado do sistema (na figura 2, as operações X e Y)
- **Operações de controle:** determinam a seqüência correta em que as operações de transição devem ser chamadas (na figura 2, as operações episodeB, e0, s0, entre outras);

A idéia de introduzir uma notação similar a máquinas de estados neste ponto do trabalho tem como intuito apenas tornar mais claro o entendimento da dinâmica das operações algébricas geradas. Por isso, como as operações algébricas são produzidas a partir dos *Structure Level* e do *Event Level*, as máquinas de estados apresentadas serão definidas em dois níveis correspondentes. Na máquina de estados correspondente ao *Structure Level*, as transições (arcos) e os estados (nodos) traduzem graficamente as operações de controle que determinam a correta execução dos episódios. Na máquina de estados correspondente ao *Event Level*, as transições correspondem às operações de transição, que são responsáveis em alterar o estado do sistema, enquanto que os estados correspondem às operações de controle responsáveis em determinar a correta execução das operações de transição.

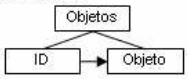
5.1 Metodologia

Definiu-se um processo apoiado por um conjunto de heurísticas para construir a especificação algébrica, tomando a especificação de *use cases* e objetos do domínio do problema como ponto de partida.

Inicialmente, parte-se de um processo de **elicitação das *use cases***, onde pode ser aplicado algum método apropriado, (como [10]). O importante aqui é ter-se ao final especificações hierárquicas de *use cases* e definições de objetos do domínio.

O passo seguinte é a **criação do *sort*** do tipo abstrato de dados, através da análise dos objetos de domínio e de um conjunto de regras. Tendo em vista que a estrutura do *sort* na notação do PROSOFT é hierárquica, pode-se analisar objetos e atributos recursivamente. Inicialmente, é criado um *sort* "raiz" do tipo registro, com o nome do sistema, como pode ser visto na figura 2. Cada um dos objetos de domínio irá gerar um novo campo do sistema. Para cada objeto, aplicam-se heurísticas para decidir de qual tipo será o *sub-sort* por ele gerado.

Tabela 1. Exemplos de regras para a determinação do *sort* da especificação

Condições do objeto	**Tipo Resultante**
Se **objeto** terá múltiplas instâncias durante a execução do sistema	Conjunto: 
Se o **objeto** terá múltiplas instâncias durante a execução do sistema **E** Se as instâncias necessitam de ordenação	Lista: 
Se o **objeto** terá múltiplas instâncias durante a execução do sistema **E** Se as instancias possuem um atributo para identificação.	Mapeamento: 

Para utilizar heurísticas na **definição das operações de transição**, parte-se antes para uma melhor classificação de cada evento, sendo que cada tipo de evento terá determinadas heurísticas para determinar a operação algébrica correspondente. A tabela 2 mostra algumas heurísticas para determinados tipos de eventos:

Tabela 2. Exemplo de regras para determinar as operações de transição

Tipo de evento	**Descrição**	**Heurísticas**
Atualização	Quando é requisitada a mudança ou inclusão de valores do sistema.	Determinar quais valores do *sort* serão atualizados; Determinar quais os parâmetros de atualização, Aplicar as operações algébricas pré-definidas para acessar e atualizar os valores do *sort*.
Verificação	Quando é requisitada a verificação de algum valor do sistema.	Construir uma operação observadora, do tipo if-then-else; Para definir o predicado do if: determinar qual parâmetro e qual valor do sort serão comparados; Aplicar as operações algébricas pré-definidas para acessar o valor do sort; then e else irão retornar TRUE ou FALSE.
Resposta	Quando o sistema envia uma mensagem para o usuário.	Inclui-se um *sub-sort* do tipo string no *sort* System, chamado de "Output". Sempre que uma resposta for enviada, uma operação correspondente irá atualizar o campo "Output". Durante a execução será possível visualizar a última mensagem enviada pelo sistema

Para **definir as operações de controle**, tem-se heurísticas para os operadores das *use cases* no *Structure Level* e para os operadores de eventos no *Event Level*, pois como foi explicado na seção anterior, a máquina de estados é definida a partir destes dois níveis. A tabela 3 relaciona as principais regras para criação das operações de controle no *Structure Level*.

6 Estudo de Caso

Será mostrado agora um estudo de caso para exemplificar as principais idéias apresentadas. Partindo da definição de um sistema simples (controle de uma biblioteca), será definido um requisito especifico dentro deste contexto (empréstimo de livros). Em seguida, serão especificados as *use cases* e objetos do domínio relativos a este problema. Ao final, será criado um tipo abstrato de dados, seguindo a metodologia e heurísticas apresentadas anteriormente.

Definição do sistema: um sistema de controle de biblioteca irá armazenar dados sobre livros, clientes e empréstimos. Irá prover serviços de empréstimos e reservas, entre outros.

Definição dos requisitos: Para efetuar um empréstimo, o usuário do sistema irá inicialmente entrar com o código do cliente, para o sistema verificar a existência do mesmo. Em seguida, o usuário irá entrar com o código do livro, para o sistema verificar se o mesmo não está reservado. Nesses dois casos, o sistema poderá enviar uma mensagem de erro ao usuário do sistema, caso o cliente não esteja cadastrado ou o livro esteja reservado. Caso contrário, o sistema irá criar um novo empréstimo, registrando o código do livro, o código do cliente e calculando a data de devolução. Ao final, o sistema enviará uma mensagem de confirmação para o usuário do sistema.

Especificação dos requisitos: A figura 3 mostra os objetos de domínio definidos para este problema. A *use case* "Efetua Emprestimo" na figura 4, define o processo de empréstimo de livros descrito anteriormente. Para simplificar a apresentação, será mostrado somente o *Structure Level* e o *Event Level*.

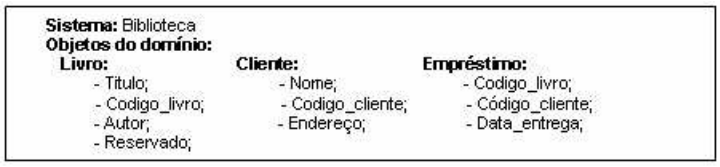


Fig. 3. Objetos do domínio biblioteca

O próximo passo é **criar as operações de controle** para definir a máquina de estados. A figura 7 mostra a máquina de estado correspondente a *use case* "Efetua Empréstimo" e aos seus três episódios. Em (a), definimos o primeiro nível da máquina de estados, correspondente ao *Structure Level*, onde a *use case* é definida como uma seqüência de três episódios. Em (b), (c) e (d) são definidos os estados e transições correspondentes a cada episódio. Dessa forma, a *use case* parte inicialmente para o estado e0, em (a), após a execução de "Validar Cliente" (b), para em seguida executar "Validar Livro" (c) e atingir o estado e1, que por sua vez irá executar "Cadastrar Emprestimo" (c) e finalmente atingir o estado final e2. A especificação formal da máquina de estados é apresentada na figura 8.

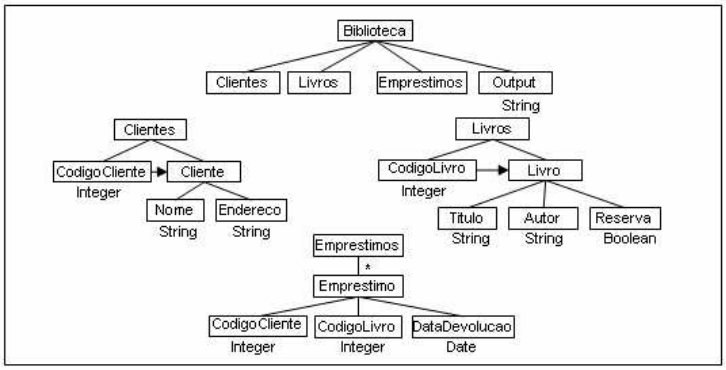


Fig. 5. Especificação do *Sort* Biblioteca

```
verifica_codigo_cliente ( biblioteca, codigo_cliente ) =
      if is_in(domain(select-Clientes(biblioteca)), codigo_cliente)
        then TRUE
        else FALSE

verifica_reserva ( biblioteca, codigo_livro ) =
     if not select-Reserva(image_of (codigo_livro, select-Livros(biblioteca)))
        then TRUE
        else FALSE

cria_emprestimo ( biblioteca, codigo_cliente, codigo_livro ) =
     (Clientes select-Clientes(biblioteca),
      Livros select-Livros(biblioteca),
      Emprestimos cons( (CodigoCliente codigo_cliente, CodigoLivro codigo_livro,
                         DataDevolução today + 7) ), select-Emprestimos(biblioteca) )
      Output select-Output(biblioteca) )

output ( (_,_,_, Output output), message ) =
                 (_, _, _, Output message)
```

Fig. 6. Especificação das operações de transição

7 Conclusões

Este artigo apresenta um modelo de integração de especificações de requisitos com especificações formais. Parte-se um modelo orientado ao cliente, baseado em *use cases* para definir os requisitos funcionais do sistema. Em seguida, é criado um tipo abstrato de dados, que graças a sua sintaxe e semântica bem-definidas, permite a execução de protótipos e ferramentas de análise formal, reduzindo tempo e custos do projeto. O modelo proposto mostra como estes dois formatos de especificação podem estar relacionados dentro do contexto do projeto formal de software.

A aplicação da metodologia e das heurísticas apresentadas visa estabelecer uma forte relação de consistência entre especificações de *use cases* e tipos abstratos de dados. Esta consistência permite trabalhar com os dois modelos de especificação, cada um dentro do seu propósito, sabendo que ambos estão direcionados ao projeto e implementação do sistema. Assim, a especificação de requisitos ganha um modelo formal de verificação, através do protótipo originado da especificação algébrica. Por outro lado, a especificação formal ganha, através da especificação dos requisitos, uma forma de representação do contexto do sistema.

Referências

1. André, P., Royer P. Building Executable Data Types from Dynamic Descriptions. Rapport de recherche, IRIN, http://www.sciences.univnantes.fr/info/perso/permanents/royer/papers/fac99.ps.gz. (1999)
2. Czarnecki, K., Zhang D., Lano, K. An Approach to Animating Model-Based Object-Oriented Formal Specifications. In: IEEE Transactions on Software Engineering, n. XX (1999)
3. Goguen J. A. Requirements Engineering as the Reconciliation of Technical and Social Issues http://citeseer.nj.nec.com/116628.html, (1994)
4. Ehrig, H.; Kreowski, H. Refinement and Implementation In E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, *Algebraic Foundations of Systems Specification*, Springer (1999) 201-242
5. ITU. Message Sequence Charts ITU-T Recomendation Z-120 (1999)
6. Jacobson, I. Object-oriented software engineering : a use case driven approach Addison-Wesley (1992)
7. Mauco M. V., George C. Using Requirements Engineering to Derive a Formal Specification Technical Report 223, UNU/IIST, Macau (2000)
8. Nunes, D. J. Estratégia Data-Driven no Desenvolvimento de Software In: Simpósio Brasileiro de Engenharia de Software (6. : 1992, nov.4-6 : Gramado). Anais. Porto Alegre: Instituto de Informática/UFRGS, Vol.1 (1992) 81-95.
9. Regnell B., Andersson M. and Bergstrand J. A Hierarquical *Use case* Model with Graphical Representation In: Proceedings IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (1996)
10. Regnell, B., Kimbler, K., Wesslen, A. Improving the Use Case Driven Approach to Requirements Engineering In: RE'95: Second IEEE International Symposium on Requirements Engineering (1995) 40-47
11. Sommerville, I. Software Engineering, Addison-Wesley (2001)
12. Watt, D. A. Programming language syntax and semantics,