MAC-5701-Tópicos em Ciência da Computação
Relatório de Estudos
Revisão de Crenças para Especificação Formal em Z

Aluno : Thiago Carvalho de Sousa
Orientadora : Prof. Dr. Renata Wassermann

Sumário

1 Introdução **3**

2 Motivação **3**

3 Revisão de Crenças **4**
3.1 História 4
3.2 Um exemplo 4
3.3 Os operadores 5
3.4 Representação das Crenças 5
3.5 Postulados AGM 6
3.5.1 Revisão 6
3.5.2 Contração 7
3.6 Construções 7
3.6.1 Partial Meet Contraction 8
3.6.2 Epistemic Entrenchment 8
3.6.3 Safe Contraction (Contração Segura) 9
3.6.4 Sistema de Esferas 10
3.7 Outras abordagens 10

4 Notação Z **10**
4.1 Especificação Formal 10
4.2 Características 12
4.3 Um exemplo 13
4.4 Provas, Inconsistências e Ferramentas 16

5 ProofPower **18**
5.1 Componentes 18
5.2 Consistência das Especificações 19
5.3 Suporte a Provas em Z 19
5.3.1 Cálculo Proposicional 20
5.3.2 Cálculo de Predicado 20
5.4 Diagnóstico do ProofPower 21
5.5 Um exemplo 22

6 Trabalhos Relacionados **25**

7 Conclusões e Trabalho Futuro **25**

1 Introdução

O presente trabalho tem como objetivo apresentar as atividades de pesquisa realizadas ao longo do primeiro semestre de 2004 como parte da disciplina MAC5701 - Tópicos em Ciência da Computação. O trabalho mescla duas grandes áreas da computação : Inteligência Artificial e Engenharia de Software. Apresentaremos inicialmente na seção 2 qual a motivação desse trabalho, com uma proposta da área de IA para solucionar um antigo problema da área de ES. Na seção 3 entraremos em detalhes no campo de IA conhecido como revisão de crenças, que nos dará uma base teórica para propormos uma solução para o problema. Na seção 4 apresentaremos a notação Z, um dos mais usados métodos formais de desenvolvimento de software, que servirá de base para uma futura implementação da solução proposta. A seção 5 é dedicada a apresentar uma das ferramentas de suporte a Z mais usadas no mundo : ProofPower. Por possuir código aberto essa ferramenta nos servirá de base para a integração IA-ES. Apresentaremos sequencialmente nas seções 6 e 7 os trabalhos relacionados a essa pesquisa, bem como as conclusões inferidas e os possíveis trabalhos futuro.

2 Motivação

Durante a fase de desenvolvimento de um sistema, há a fase de análise de requisitos. Nessa fase, é freqüente a ocorrência especificações inconsistentes umas com as outras, como por exemplo, um usuário deseja que um certo icone posicione-se no lado direito da tela enquanto um outro prefere o lado esquerdo. A capacidade de corrigir e analisar o impacto das mudanças no design de um sistema durante a fase de desenvolvimento é uma das tarefas mais árduas e custosas da engenharia de software. Pesquisas indicam que, em alguns sistemas, 80% dos custos de desenvolvimento são consumidos pelas mudanças dos requisitos dos usuários. Essas mudanças ocorrem devido a alguns fatores, tais como : muitas vezes os usuários possuem visões e perspectivas diferentes do sistema; na maioria das vezes os usuários "esquecem" alguns requisitos; quase sempre os usuários desejam revisar a funcionalidade do sistema; nem sempre os engenheiros captam de maneira exata os requisitos dos usuários, etc. É preciso, portanto, prover uma ferramenta que mantenha de alguma forma a consistência desses requisitos e, ao mesmo tempo, permita revisá-los durante o processo de análise.

A nossa pesquisa tem como objetivo mostrar como a área de revisão de crenças encaixa-se perfeitamente para solucionar esse problema. Especificamente, tentaremos integrar a solução proposta por esse campo da inteligência artificial a notação de especificação formal de software Z, utilizando como base de implementação e testes a ferramenta ProofPower.

3 Revisão de Crenças

3.1 História

A necessidade de modelar o comportamento de bases de conhecimento dinâmicas que ao receberem uma certa informação se tornam inconsistentes formou a base da teoria de revisão de crenças. Essa teoria começou a se solidificar a partir do inicío da década de 80 quando Alchourrón, Gardenfors e Makinson [1] propuseram alguns postulados que descrevem as mínimas propriedades que um processo de revisão deve ter. Esses postulados ficaram conhecidos como postulados AGM em homenagem aos autores e, desde então, têm sido bastante usados na área de modelagem de sistema dinâmicos.

3.2 Um exemplo

Mostraremos o seguinte exemplo a fim de elucidarmos melhor a teoria da revisão de crenças. Suponha que se tenha uma base de dados que contenha as seguintes informações :

α : Todo brasileiro é pobre

β : João é brasileiro

γ : Quem ganha na mega-sena é rico

δ : Ricardo é jogador de futebol

Suponha também que essa base de dados possua um mecanismo lógico de inferência e que portanto seja capaz de deduzir que "João é pobre". Se essa mesma base de dados receber a informação "João ganha na mega-sena", a mesma ficará inconsistente, uma vez que João não pode ser rico e pobre simultaneamente. Como o que se deseja é manter a base de dados sempre consistente, é preciso revisá-la, o que significa abandonar algumas das informações (crenças) originais. Poderíamos simplesmente abandonar todas as crenças originais e considerar apenas a nova. Mas isso não é necessário e nem desejável, pois acarretaria perda de informações valiosas (no nosso caso, por exemplo, "Ricardo é jogador de futebol"). Um dos princípios que rege a teoria de revisão de crenças é que a mudança no meu conjunto original de crenças seja a mínima possível. Esse é o Princípio da Mudança Mínima. Em outras palavras, o que se deseja é reter o máximo de informação possível. No exemplo acima, se abandonássemos a crença α, β ou γ nossa base de dados voltaria a ficar consistente.

O problema da revisão de crenças pode tornar-se mais complicado se as crenças possuirem consequências lógicas. Nesse caso, é preciso decidir também quais as consequências a serem mantidas e quais devem ser abandonadas. Um outro questionamento possível é que a informação recebida seja descartada imediatamente, mantendo-se o conjunto original de crenças, que já era consistente. Mas a teoria da revisão de crenças sempre supõe que a nova crença é aceita, o que tem provocado críticas e gerado novas abordagens sobre o tratamento da crença recebida.

3.3 Os operadores

Como já mostramos, a revisão de crenças tem como papel "consertar" uma base de dados que era consistente e que se tornou inconsistente depois da adição de uma nova crença. Mas nem sempre é necessário "consertar" a base de dados, uma vez que a nova informação recebida seja consistente com as crenças originais. Dessa maneira, existem três tipos de operadores de mudanças no conjunto de crenças (K) quando uma nova informação α é recebida. São eles :

- Expansão : uma informação consistente α, juntamente com as suas consequências lógicas, é adicionada ao conjunto de crenças K. A representação desse operador é K + α;

- Contração : uma informação $\neg\ \alpha$ é abandonada pelo conjunto K. Como o conjunto K é fechado logicamente, talve seja necessário também abandonar outras crenças. A representação desse operador é K - $\neg\ \alpha$;

- Revisão : uma informação inconsistente α é adicionada ao conjunto K e para se manter a consistência é preciso abandonar outras crenças de K. A representação desse operador é K $*$ α.

Harper e Levi identificaram e provaram duas relações entre os operadores de revisão e contração, que depois tornaram-se conhecidas como identidade de Harper e identidade de Levi :

Identidade de Levi : K $*$ α = (K - $\neg\ \alpha$) + α
Identidade de Harper : K - α = K $\cap$ K $*$ $\neg\ \alpha$

3.4 Representação das Crenças

A teoria da revisão de crenças é focada nos estados epistêmicos (de conhecimento, crença) e nas mudanças ocorridas nesses estados. Já vimos na seção anterior quais os operadores são responsáveis pelas mudanças. No entanto, é necessário definir uma representação para esses estados. Há várias maneiras de representá-los, dentre os quais destacam-se :

* Conjunto de Crenças : é a maneira mais simples de modelar um estado epistêmico. Um conjunto de crenças caracteriza-se por ser um conjunto de fórmulas logicamente fechado, ou seja, dado um conjunto K, se K implica logicamente α, então $\alpha \in$ K. Ao usar essa representação, estamos propensos a trabalhar com conjuntos infinitos, o que não é apropriado do ponto de vista computacional.

* Base de Crenças : essa representação supõe que um conjunto de crenças possui uma base necessária para inferir todas as crenças de um conjunto. B_k é uma base para um conjunto de crenças K se e somente se B_k é um subconjunto finito de K e o

fecho lógico de B_k é igual ao conjunto K. Logicamente podemos ter várias bases para um mesmo conjunto K. Do ponto de vista computacional, por trabalhar com conjuntos finitos, essa representação tem sido bastante utilizada.

* Mundos Possíveis : essa representação baseia-se no conjunto W_k de mundos possíveis. Há uma forte relação entre conjunto de crenças e a representação por mundos possíveis. Para um conjunto W_k de mundos possíveis podemos definir um conjunto de crenças K correspondente como o conjunto das formulas que são verdadeiras em todos os mundos de W_k. Do ponto de vista computacional, essa representação é complicada, sendo, portanto, a preferida entre lógicos e filósofos.

3.5 Postulados AGM

Vamos assumir, a partir deste ponto, que a representação dos estados epistêmicos será baseada no modelo de conjunto de crenças. A motivação por trás dos postulados é que o Princípio da Mudança Mínima seja atendido quando uma mudança for realizada.

3.5.1 Revisão

Dado um conjunto de crenças K e uma sentença (crença) α, os seis postulados básicos da revisão são :

K $*$ **1** : K $*$ α é um conjunto de crenças
K $*$ **2** : $\alpha \in$ K $*$ α
K $*$ **3** : K $*$ $\alpha \subseteq$ K $+$ α
K $*$ **4** : Se $\neg \alpha \notin$ K, então K $+$ $\alpha \subseteq$ K $*$ α
K $*$ **5** : K $*$ α é consistente sse $\vdash \neg \alpha$
K $*$ **6** : Se $\vdash \alpha \leftrightarrow \beta$, então K $*$ $\beta =$ K $*$ α

O postulado 1 mostra que o resultado de uma revisão continua sendo um conjunto de crenças. O postulado 2 diz que a crença recebida deve pertencer ao conjunto de crenças revisado. O próximo postulado garante que nenhuma outra informação é adicionada ao conjunto de crenças a não ser a nova sentença e suas consequências lógicas. O postulado 4 diz que se a nova informação adicionada for consistente, a expansão está contida na revisão. E esse fato, junto com o postulado 3, mostra que a expansão é igual a revisão. O postulado 5 diz que todo conjunto de crenças revisado é consistente. O postulado 6 mostra que a revisão feita por sentenças logicamente equivalentes resultam no mesmo conjunto revisado. Há ainda dois postulados extras para revisões de conjunções :

K $*$ **7** : K $*$ $(\alpha \wedge \beta) \subseteq$ (K $*$ α) $+$ β
K $*$ **8** : Se $\neg \beta \in$ K $*$ α, então(K $*$ α) $+$ $\beta \subseteq$ K $*$ $(\alpha \wedge \beta$

O postulado 7 diz que a revisão por uma conjunção deve estar contida no conjunto resultante da revisão de uma das fórmulas pela expansão da outra. O postulado 8 afirma

que se uma das fórmulas for consistente com o conjunto revisado pela outra fórmula, então, junto com o postulado 7, o conjunto revisado pela conjunção é igual ao conjunto resultante da revisão de uma das fórmulas pela expansão da outra.

3.5.2 Contração

Dado um conjunto de crenças K e uma sentença (crença) α, os seis postulados básicos da contração são :

K - 1 : K - α é um conjunto de crenças
K - 2 : K - $\alpha \subseteq$ K
K - 3 : Se $\alpha \notin$ K, então K - α = K
K - 4 : Se não $\vdash \alpha$, então $\alpha \notin$ K - α
K - 5 : K $\subseteq$ (K - α) + α
K - 6 : Se $\vdash \alpha \leftrightarrow \beta$, então K - β = K - α

O postulado 1 diz que o resultado de uma contração continua sendo um conjunto de crenças. O postulado 2 diz que a operação de contração não acrescenta novas informações ao conjunto de crenças original. O próximo postulado garante que se a sentença contraída já não pertencia a conjunto de crenças, não haverá mudanças. O postulado 4 diz que a sentença contraída não fará parte do conjunto resultante. O postulado 5 diz que o conjunto original pode ser recuperado ao fazer a expansão pela sentença contraída. Esse é um dos mais polêmicos postulados, uma vez que é possível mostrar que nem sempre isso ocorre. O postulado 6 mostra que a contração feita por sentenças logicamente equivalentes resultam no mesmo conjunto. Há ainda dois postulado extras para contração também relacionados a conjunções :

K - 7 : K - $\alpha \cap$ K - $\beta \subseteq$ K - $(\alpha \wedge \beta)$
K - 8 : Se $\alpha \notin$ K - $(\alpha \wedge \beta)$, então K - $(\alpha \wedge \beta) \subseteq$ K - α

O postulado 7 garante que as crenças comuns a contração individual de cada uma das formulas da conjunção estão na contração pela conjunção. O postulado 8 diz que qualquer crença abandonada quando se contrai uma das formulas da conjunção é também abandonada na contração pela conjunção.

3.6 Construções

Agora que já vimos um exemplo ilustrativo, os operadores de mudanças, a representação das crenças e os postulados AGM, é hora de vermos as construções. Como uma revisão nada mais é do que uma contração seguida de uma expansão (Identidade de Levi), a solução do problemas se baseia na construção de uma função de contração que siga os postulados AGM.

3.6.1 Partial Meet Contraction

Antes de analisarmos a primeira construção [2] é preciso definir a idéia de subconjunto maximal que não implica uma dada sentença. Seja K um conjunto de crenças e α, temos que X é um subconjunto maximal de K que não implica α se :

* X $\subseteq$ K

* X não $\vdash \alpha$

* Para todo X' tal que X $\subset$ X' $\subseteq$ K, X' $\vdash \alpha$

Note que há a possibilidade de existirem vários subconjuntos maximais que se adequam as regras acima. É preciso também definir a função de seleção, onde dada uma sentença α e um conjunto K de crenças como parâmetros de entrada, é devolvido a intersecção dos subconjuntos maximais de K que não implicam α. A contração é obtida através dessa função de seleção e a essa construção damos o nome de Partial Meet Contraction.

Há dois casos que são opostos e muito conhecidos em se tratando da escolha da função de seleção : Maxichoice e Full Meet. No Maxichoice a função de seleção tem como saída um único subconjunto maximal. Essa função satisfaz o seguinte postulado :

Se $\beta \in$ K e $\beta \notin$ K - α, então $\beta \rightarrow \alpha \in$ K - α

Entretanto, esse postulado produz efeitos indesejados, pois cria subconjuntos muito grandes, com acrécimo de crenças que antes da contração eram totalmente desconhecidas. Já no Full Meet, a função de seleção tem como saída a intersecção de todos os subconjuntos maximais. Essa função satisfaz o seguinte postulado :

Para todo α e β, K - $(\alpha \wedge \beta)$ = K - $\alpha \cap$ K - β

Mas esse postulado possui também efeitos indesejáveis que são opostos ao problema do Maxichoice. Nesse caso, são criados subconjuntos pequenos que abandonam crenças, que intuitivamente deveriam ter sido preservadas. É importante ressaltar aqui que a função de seleção até aqui apresentada sem nenhuma restrição satisfaz apenas os postulados básicos. Para satisfazer também os outros dois postulados extras é preciso haver uma relação de ordem transistiva e reflexiva entre os subconjuntos maximais.

3.6.2 Epistemic Entrenchment

Essa construção [3] é baseada na idéia de que nem todas as crenças possuem a mesma importância. É feito, portanto, uma espécie de ranking de crenças baseado em algum critério. A idéia é que as crenças em posições mais baixas no ranking (ou traduzindo ao pé da letra, menos entricheiradas) sejam abandonadas no momento da contração. Dado um conjunto de crenças K, e sendo "a" $\leq$ "b" significa que "b" possui pelo menos a mesma

posição de "a" no ranking. Para que suas crenças satisfaçam a idéia do epistemic entrenchment, é preciso seguir os seguinte postulados :

EE1 : Se a $\leq$ b e b $\leq$ c então a $\leq$ c
EE2 : Se a $\vdash$ b então a $\leq$ b
EE3 : Para algum a e b, a $\leq$ a $\wedge$ b ou b $\leq$ a $\wedge$ b
EE4 : Quando K $\neq \perp$, a $\notin$ K sse a $\leq$ b, $\forall$ b
EE5 : Se a $\leq$ b $\forall$ a, então $\vdash$ b

O postulado 1 diz que a ordenação deve ser transitiva. O postulado 2 diz que se uma sentença "a" implica alguma sentença "b" é melhor abandonar "a" e reter "b". À primeira vista, esse postulado parece violar o Princípio da Mudança Mínima, porém analisando bem, se abandonarmos "b" o mesmo precisaria ser feito com "a", o que não ocorre se decidirmos abandonar apenas a sentença"a". O próximo postulado diz que se quisermos contrair "a $\wedge$ b", basta contrairmos "a" ou "b". O postulado 4 diz que qualquer sentença não pertencente ao conjunto de crenças terá o valor mínimo na ordenação. Já o postulado 5 diz que as tautologias sempre terão valor máximo na ordenação.

Essa construção foi criticada pois há a possibilidade da crença abandonada não ser a menor do ranking. Uma outra crítica a essa construção, que pode ser estendida a toda a teoria dos postulados AGM, é que não há suporte a uma função interativa, já que a ordenação é perdida logo após a primeira operação de contração. Por parecer computacionalmente uma das melhores contruções de se implementar, várias propostas de interação e ranqueamento foram providas, mas no momento não entraremos em detalhes.

3.6.3 Safe Contraction (Contração Segura)

Antes de falarmos dessa construção [4] é preciso definirmos o que é um subconjunto minimal que implica uma dada sentença. Seja K um conjunto de crenças e α uma sentença, temos que X é um subconjunto minimal de K que implica α se :

* X $\subseteq$ K

* X $\vdash \alpha$

* Para todo X' tal que X $\subset$ X' $\subseteq$ K, X' não $\vdash \alpha$

A idéia por trás dessa construção é a seguinte : Seja K um conjunto de crenças que se deseja contrair com uma sentença α e que exista uma ordem aciclica entre as sentenças de K. Um elemento β é seguro em relação a α se β não é o elemento mínimo (em relação a ordenação) de qualquer subconjunto minimal K' de K que implica α. Ou de outra forma, β é seguro se todo subconjunto minimal K' de K que implica α, ou não contém β ou contém algum $\gamma < \beta$.

Intuitivamente a idéia é que β é seguro se nunca puder ser responsável pela implicação de α. Note que, em contraste com as outras construções citadas, essa construção usa a idéia de subconjuntos minimais que implicam α ao invés de subconjuntos maximais que

não implicam α. A ordenação também pode ser vista como uma espécie de epistemic entrechment. A contração segura se dá pelo conjunto de todas as sentenças que são seguras em relação a sentença que se deseja contrair.

3.6.4 Sistema de Esferas

Até o momento, as construções apresentadas se utilizavam da representação de crenças baseada em conjunto de crenças. Essas construções podem ser facilmente estendidas para a representação das crenças através de bases. O sistema de esferas se utiliza da representação baseada em mundos possíveis.

Qualquer conjunto de crenças K pode ser representado pelo subconjunto [K] dos mundos possíveis M, que consiste em todos os conjuntos maximais onde todas as senteças de K são verdadeiras. Um sistema de esfera [5] centrado em [K] é uma coleção X dos subconjuntos dos mundos possíveis M que satisfaz as seguintes condições :

- X é totalmente ordenado
- [K] é o menor elemento dessa ordenação
- M é o maior elemento dessa ordenação
- Se qualquer esfera intersecta uma sentença α, então existe uma esfera em X que é a menor esfera que intersecta α

A contração através de uma dada sentença α nesse caso é feita juntando-se o [K] com a intersecção de [$\neg$ α]com a menor esfera de X dos mundos possíveis. Mais uma vez podemos ver essa ordenação como uma espécie de epistemic entrenchment.

Ainda há outras construções possíveis, mas ficaremos por aqui por considerarmos as quatro mais importantes. É importante ainda ressaltar que todas as construções possuem provas de equivalência e provas de que seguem os postulados AGM.

3.7 Outras abordagens

A área de revisão de crenças tem se desenvolvido com bastante força nos últimos anos, tendo-se produzido abordagens bastante interessantes, com destaque para a revisão de crenças interativa [6], kernel contraction, internal e external revision, adição de novos operadores (consolidação e semi-revisão) [7], revisão seletiva [8], screened revision [9], non-prioritized revision [10], etc. Mas vamos deixar para entrar em detalhes em uma outra oportunidade.

4 Notação Z

4.1 Especificação Formal

A especificação formal de um software nada mais é do que uma técnica de engenharia de software que usa notação matemática para descrever as funcionalidades de um sistema,

e que detalha formas de validar esta especificação e a subsequente implementação. Para falar de suas vantagens é preciso conhecermos primeiro os sete mitos de especificação formal de software [11] . São eles:

- Especificações formais podem garantir um programa perfeito

  Não é possível garantir a perfeição de nenhum programa, independentemente do método utilizado. Mesmo usando um método formal, existem várias fases do projeto onde podem surgir problemas. O ponto principal é que os benefícios de uma especificação formal independem da garantia de sucesso.

  Dito isso, especificações formais podem realmente eliminar problemas de certas classes e ajudar a evidenciar problemas na especificação das outras. O documento de especificação formal tem um mecanismo embutido que facilita a descoberta destes erros antes da implementação.

- Especificações formais só servem para provar que o programa é correto

  O uso dos especificações formais vai além da prova: do ponto de vista econômico, a parte mais importante do formalismo é na verdade a elaboração da especificação em uma notação auto-validante. Estabelecer e validar os requisitos é uma das tarefas mais importantes que um projeto de software enfrenta, e um método formal aborda de forma direta esta questão. A prova e verificação do programa associada ao método formal é a parte geralmente mais difícil do método, mas não a única que traz benefícios.

- Somente sistemas de missão crítica são beneficiados pelo uso de especificações formais

  Especificações formais são aplicáveis a qualquer projeto, e a melhora na especificação será refletida em um projeto executado de forma mais eficiente. Sistemas de missão crítica se beneficiam diretamente pela parte de verificação do método, mas todos os projetos se beneficiam de um documento de requisitos melhor elaborado.

- Especificações formais envolvem matemática complexa

  A matemática usada não é muito dificil, e de qualquer forma mais fácil do que a própria linguagem de programação. Usar qualquer ferramenta de computação requer algum aprendizado; a notação formal não deve fugir à regra. Usar treinamento e consultoria em matemática discreta e na notação formal pode suavizar a curva de aprendizado.

- Especificação formal aumenta o custo do desenvolvimento

  Na realidade, o uso de um método formal ajuda a especificar melhor o produto, e por isso, tende a encurtar o tempo total de implementação. Passa-se mais tempo detalhando a especificação e menos programando, mas na fase inicial é geralmente menos custoso consertar problemas, e por isso é importante não ter pressa neste momento. Devido ao tempo maior gasto, pode parecer na fase inicial que não se está

fazendo progresso, pois o importante é tentar entender o problema bem e registrar as tentativas que antecedem a especificação propriamente dita para se ter ideia do andamento.

- Especificação formal é incompreensível aos clientes

  A especificação formal ajuda a capturar corretamente o verdadeiro desejo do usuário, e por isso beneficia diretamente o cliente. Para que o usuário entenda o progresso que está sendo feito, no entanto, é importante que se escreva paralelamente um documento em linguagem natural.

- Ninguem usa especificaçãos formal em projetos reais

  Especificações formais estão sendo usados em diversos projetos, em várias empresas diferentes. Exemplos citados são a IBM, no sistema CICS, a Agência de Defesa Inglesa, etc.

Métodos formais são pouco compreendidos pela maior parte dos desenvolvedores, e existem alguns tabus relacionados a eles que não são exatamente confirmados. Os mais importantes, que são a dificuldade de aplicação e o custo aumentado de desenvolvimento, estão sendo contraprovados pelo uso prático em projetos importantes de várias empresas da industria de desenvolvimento de software.

É bastante provável que métodos formais de especificação se tornem uma prática padrão na engenharia de software, uma vez que é concenso que para melhorar a qualidade do software é preciso se ter um bom mecanismo de documentação nas fases de design, desenvolvimento, testes e manutenção. E os métodos formais se mostram uma ferramanta bastante útil na tarefa de uma boa documentação, ao contrário das ferramentas que se utilizam de linguagem natural, figuras e diagramas, que são imprecisas e ambíguas.

4.2 Características

Entre os mais conhecidos métodos formais de especificação de software encontra-se a notação Z, desenvolvida pela Universidade de Oxford no final dos anos 70. Z é uma notação pois não tem a intenção de ser algo executável, ou seja, não é uma linguagem de programação e sim apenas uma notação para detalhar funcionalidades de um sistema de software. A notação Z possui entre suas principais características o fato de ser tipada, ser baseada em lógica de primeira ordem e teoria dos conjuntos (ZF), ser popular na indústria, meios acadêmicos e governamentais e ser o único método formal de especificação com definição de padrão internacional (ISO). O Z possui quatro aspectos fundamentais :

- A lógica de primeira ordem juntamente com a teoria dos conjuntos proporcionam uma linguagem matemática bem expressiva e fácil de se usar.

- Permite a modularização das especificações através de esquemas, o que é importante para a organização e manipulação de grandes especificações. Um esquema representa um estado do sistema ou uma operação qua altera um determinado estado.

- Permite o uso de linguagem natural quando se nomeia as variáveis sensatamente. Além disso, permite também comentários adicionais para a perfeita compreensão humana.

- Pode-se fazer refinamentos (diminuição do nível de abstração) no modelo do sistema até se chegar a algo bem perto a ser implementado.

4.3 Um exemplo

A melhor maneira de apresentar a sintaxe e a semântica da notação Z é através de exemplos. Vamos agora mostrar o exemplo clássico do livro de aniversário [12] para elucidar as idéias por trás do Z.

Suponha que desejamos ter um sistema que armazena os aniversários das pessoas conhecidas, que dado um nome o sistema devolve ou não a data do aniversário da pessoa e que seja capaz de emitir um lembrete quando algum aniversário acontece. O primeiro passo é definir os conjuntos usados, no nosso caso, o conjunto dos nomes das pessoas e o conjunto com todas as datas possíveis. Isso é definido através de colchetes

[NOMES, DATAS]

O segundo passo é definir o espaço de estados do sistema, e fazemos isso com o seguinte esquema:

LivroAniversario $\cong$ [conhecidos : P NOMES, aniversário : NOMES $\mapsto$ DATAS | conhecidos = dom aniversário]

O que vem antes da | é a assinatura do esquema, onde se introduz as variáveis e seus respectivos tipos. É similar as declarações de variáveis nas linguagens de programação. O que vem depois da | é o predicado, que relaciona as variavéis. No exemplo, temos que conhecidos é o conjunto de todos os nomes com aniversários registrados, uma vez que P representa o powerset dos NOMES, e o aniversário é uma função que mapeia os nomes aos respectivos aniversários. O predicado nesse caso é a invariante do sistema, ou seja, o conjunto conhecidos é o mesmo que o domínio da função aniversário em qualquer estado do sistema.

Um possível estado do sistema é o seguinte :

conhecidos = {Thiago, Janine, Seiji}

aniversário = {Thiago $\mapsto$ 01/06/80, Janine $\mapsto$ 01/10/80, Seiji $\mapsto$ 9/9/79}

No exemplo não nos preocupamos em colocar um limite no número de estados possíveis no sistema, nem em definir um formato dos nomes e datas.

O passo seguinte é definir as operações do sistema. Precisamos de três operações : uma para adicionar um aniversário ao livro, outra para encontrar um aniversário através de um nome dado e outra para ser o lembrete dos aniversários do dia.

O esquema para a primeira operação é o seguinte:

AdicionaNiver $\cong$ [Δ LivroAniversario, nome? : NOMES, data? :DATAS | nome $\notin$ conhecidos, aniversário' = aniversário $\cup$ nome? $\mapsto$ data?]

A declaração Δ LivroAniversario alerta para o fato que o esquema está descrevendo uma mudança de estado. Aparecem também nessa esquema dados de entrada do operador, que por convenção terminam por um ponto de interrogação. Na parte do predicado há uma pré-condição a ser satisfeita para a operação ser bem sucedida. É preciso que o novo nome que se quer adicionar já não esteja no registros. Se a pré-condição for satisfeita a função aniversário é estendida para mapear o novo nome a nova data.

O esquema para a segunda operação é o seguinte:

EncontraNiver $\cong$ [Ξ LivroAniversario, nome? : NOMES, data! :DATAS | nome $\in$ conhecidos, data! = aniversário(nome?)]

A declaração Ξ LivroAniversario que esse esquema descreve uma operação sem mudança de estado. Como no esquema anterior, há dados de entradas. Mas há também dados de saída, que por convenção terminam com exclamação. Novamente há a pré-condição a ser satisfeita (nome $\in$ conhecidos). Caso seja satisfeita, a saída data! é o valor da função aniversário para o argumento nome?.

O esquema para a operação de lembrete é a seguinte:

LembraNiver $\cong$ [Ξ LivroAniversario, hoje? : DATAS, presentes! : P NOMES | presentes! = {n : conhecidos | aniversário(n) = hoje?}]

Essa operação devolve um conjunto presentes com uma ou mais pessoas se elas fizerem aniversário na data hoje.

Para finalizar a especificação, é preciso dizermos em qual estado o sistema começa. Para isso precisamos de um esquema que defina esse estado inicial. No nosos exemplo, temos :

InicioLivroNiver $\cong$ [LivroAniversario | conhecidos = $\oslash$]

A especificão do sistema poderia terminar por aqui. Mas note que há sérias falhas. O que acontece se tentarmos adicionar uma pessoa que já existe nos registros? E se tentarmos encontrar o aniversário de alguém desconhecido? Não há como prever o comportamento do sistema. Ele pode simplemente ignorar como também pode mostrar lixo. É preciso, portanto, ter algum tratamento para entradas incorretas.

Poderíamos modificar os nossos esquemas acima para tal tratamento. Mas isso pode tornar o esquema um escopo grande e complicado. Um alternativa, que é bem comum no uso de Z, é descrever em outros esquemas os possíveis erros e seu respectivos tratamentos. Depois podemos então combinar esses esquemas com os originais para formar esquemas mais robustos e corretos. Essa combinação entre esquemas é uma facilidade do Z conhecida como schemas calculus.

Voltando ao nosso exemplo, vamos adicionar um saída status que descreverá se a operação foi bem sucedida ou não. Assim sendo, para resolver o problema da primeira pergunta descrevemos mais um esquema Sucesso que devolve ok e mais um esquema Já-Conhecido que diz que a pessoa já está registrada.

[STATUS]

Sucesso $\cong$ [status : STATUS | status! = ok]

JáConhecido $\cong$ [Ξ LivroAniversario, nome? : NOMES, status! : STATUS | nome $\in$ conhecidos, status! = pessoa já registrada]

Usando schemas calculus podemos combinar esses dois esquemas com AdicionaNiver para formarmos uma especificação mais completa

MelhorAdicionaNiver $\doteq$ (AdicionaNiver $\wedge$ Sucesso) $\vee$ JáConhecido

Note que o novo esquema formado termina qualquer que seja a entrada. Esse esquema, como dito anteriormente, poderia ser feito diretamente, incluindo apropriamente novos predicados e declarações. Mas a idéia de modularização do Z faz com que seja mais comum separá-lo em esquemas menores.

Para resolver o problema da nossa segunda pergunta, referente ao o que acontece quando procuramos o aniversário de desconhecidos, precisamos apenas do esquema Não-Conhecido para relatar o status de pessoa desconhecida nos registros.

NãoConhecido $\cong$ [Ξ LivroAniversario, nome? : NOMES, status! : STATUS | nome $\notin$ conhecidos, status! = pessoa não registrada]

Assim temos uma melhor especificação para o EncontraNiver, que termina também com qualquer entrada.

$$\text{MelhorEncontraNiver} \doteq (\text{AdicionaNiver} \wedge \text{Sucesso}) \vee \text{NãoConhecido}$$

Vimos aqui um exemplo bem simples de como especificar um sistema usando a notação Z, particulamente como se definem os esquemas e a sua idéia de modularização. Há ainda muitos aspectos interessantes de Z a serem abordados, tais como: as variáveis livres, que são conjuntos com estrutura explicita; modularizações mais complexas, como a promoção, onde operações sobre uma única entidade são transformadas em operações sobre entidades maiores; outros tipos de dados interessantes, como sequência e bag; outros usos de schemas calculus, como extensão e manipulação de esquemas; outras operações possíveis na teoria ZF, etc. Mas para efeito ilustrativo ficaremos por aqui. Em uma outra oportunidade abordaremos melhor esses aspectos.

4.4 Provas, Inconsistências e Ferramentas

Uma das principais vantagens de se usar uma especificação formal é o fato de podermos provar algumas especificações com o intuíto de válida-las ou não. Mas a notação Z não foi criada com a mente voltada para a análise automática de provas. Há portanto alguns problemas [13] a serem encarados quando pensamos em uma ferramenta de suporte a provas em Z.

- Problemas de Sintaxe e Metodologia:

  Muitos sistemas usam uma sintaxe especial para a especificação e um outra para a prova e fazer essa tradução pode não ser muito intuitiva. Há ferramentas que usam como dados de entrada a especificação em Latex (a mais usada), outras que usam o email lexis e há ainda aquelas que usam ASCII puro. Isso gera uma imcopatibilidade de inputs. Outro problema está na metodologia de desenvolvimento utilizada, com alguns aplicando orientação a objetos, enquanto outras usam o paradigma funcional ou imperativo.

- Problemas de Semântica:

  Dividem-se em dois sub-tipos de problemas basicamente : nas funções parciais e nos esquemas. O problema da função parcial é o que fazer quando as funções são aplicadas fora do seu dominio. Isso gera uma indefinição, mas a notaçàо Z só possui duas valorações possíveis : verdadeiro ou falso. O problema dos esquemas é que eles podem representar declarações, expressões e predicados, o que faz com que não se possa substituir um esquema simplesmente pela sua definição, pois de acordo com o tipo de esquema uma variável pode ser livre ou não. Mais especificamente, as variáveis livres de um termo dependem do contexto.

Um outro fator importante a se destacar para o uso de um ferramenta de provas é que a notação Z possui quatro tipos de inconsistências, que possuem graus de complexidade diferentes em termos identificação e correção. São eles :

* Localizada : quando um predicado de um esquema é falso;

* Técnico : quando a definição de variáveis livres podem ser usadas de tal forma que seja impossível satisfazê-las em todos os conjuntos;

* Lógico : quando uma função parcial é aplicada fora do seu dominio;

* Modelagem : quando a descrição de uma operação conflita com um invariante do sistema.

Um número razoável de ferramentas de suporte a prova em Z foram criados desde o surgimento da notação. São divididas basicamente em dois tipos : aquelas que implementam um provador exclusivamente para o Z e aquelas que se utilizam de outras lógicas e de outros provadores já implementados. Dentre os últimos, há ainda aquelas ferramentas com uma implementação de encaixe semântico profundo e aquelas com uma implementação de encaixe semântico superficial. Na implementação de encaixe profundo a linguagem é representada dentro da lógica. Na implementação de encaixe superficial há uma interface que traduz a linguagem nos termos da lógica. Quanto mais profunda é a implementação, mais teoremas se pode provar, enquanto que uma implementação de encaixe superficial produz bons resultados para uma especificação em particular. Há atualmente várias ferramentas de prova que suportam type-checking e provas para a notação Z. Dentre estas, destacam-se o Z/EVES, HOL-Z, CadiZ, Alloy e ProofPower.

O Z/EVES [14] é uma implementação exclusiva de Z, sendo considerada a ferramenta mais poderosa de provas para a notação Z. É mantida pela Ora Canada e já está na versão 2.1. Possui uma interface gráfica muito boa e é implementado em Common Lisp. O grande problema, que nos fez descartar o seu uso, é ser software proprietário, o que não nos permite modificar o seu código.

O HOL-Z [15] é uma ferramenta de provas para Z que se utiliza da semelhença de Z com HOL. A ferramenta converte a representação em Latex de Z para instâncias do provador genérico Isabelle em higher-order logic (HOL). É uma implementação de encaixe semântico superficial. Utiliza um compilador para SML e está estável na atual versão 2.1.1, mas ainda possui algumas desvantagens, tais como depender de uma outra ferramenta (Zeta) como front-end e não suportar todas as construções possíveis de Z, além de ser um projeto praticamente abandonado.

Já o CadiZ [16] é uma implementação exclusiva para Z, que já vem sendo desenvolvida desde o inicio da década de 90. Primeiramente era apenas um type-checker para Z, mas atualmente já se iniciou o processo para que a ferramenta suporte provas.Possui uma excelente interface gráfica. A grande desvantagem é que não obedece o padrão internacional (ISO) e não possui código aberto.

O Alloy [17] é uma ferramenta que se ultiliza de uma linguagem baseada em lógica de primeira ordem derivada de notação Z. Ela traduz a especificação em uma fórmula booleana e repassa para um SAT solver qualquer. Depois o resultado é traduzido de volta para a linguagem da especificação. Caso algo esteja errado no modelo, é gerado um contra-exemplo. Tem vantagens em relação as demais ferramentas que são totalmente presas a notação Z : possui suporte automático de provas; compatibilidade a orientação

a objetos; independência do Latex. Está na versão estável 3.0 e é desenvolvida em Java. Seria uma boa ferramenta se não fosse precária em documentação e organização do código e ser baseada em uma linguagem pouco conhecida e sem padrão determinado.

O ProofPower [18] é uma ferramenta de especificação e provas baseado em uma implementação de um provador de teoremas HOL (Higher Order Logic), que segue o paradigma LCF em Standard ML. O ProofPower suporta especificações e provas em Z através de um encaixe semântico profundo de Z em HOL. Essa ferramenta possui uma excelente documentação e organização do código fonte. Além disso possui uma interface gráfica boa e fácil de se usar. O ProofPower começou a ser desenvolvido no final da década de 80 em um projeto envolvendo duas empresas [ICL (International Computers Ltd) e PVL (Program Verification Ltd)] e duas universidades inglesas (Cambridge e Kent). Durante 93 e 94 o suporte a notação Z foi incorporada a ferramenta. A partir de 200 a ICL cedeu os direitos do ProofPower para a Lemma 1 Ltd, que já vinha sendo responsável pelo desenvolvimento e melhoria desde 1997. Desde 2001 a Lemma 1 Ltd tornou o ProofPower (que a partir daí tornou-se OpenProofPower) uma ferramenta de código aberto, o que tem sido fundamental para o crescimento da mesma como fonte de desenvolvimento de sistemas críticos. Estima-se que atualmente cerca de 70 equipes espalhadas nos 5 continentes, entre empresas, universidades e governos, usam o ProofPower, que já está na versão 2.7.3

5 ProofPower

Depois de uma análise, a ferramenta escolhida para servir como base de implementação foi o ProofPower [19] [20] [21] pelas qualidades citadas na seção anterior. Vamos agora nos aprofundar um pouco mais nas suas características, mostrando como essa ferramenta suporta provas em Z, como trata inconsistências, quais seus componentes e como se usa.

5.1 Componentes

O coração do ProofPower é essencialmente um sistema que gerencia um conjunto de teorias e que conduz provas sobre objetos descritos por essas teorias. Essa parte do sistema segue o paradigma LCF e é implementado em SML of New Jersey, uma implementação SML desenvolvida pela Lucent Technologies. Uma vez que ML é uma linguagem interativa, ela serve não apenas como linguagem de implementação mas também como linguagem de comando. O ProofPower possui basicamente 3 pacotes : ProofPower-HOL, ProofPower-Z e XPP.

O ProofPower-HOL é a parte do sistema responsável por oferecer suporte fragmentos escritos em HOL. Foi o objetivo inicial do ProofPower, mas por não ser relevante no desenvolvimento da nossa pesquisa não entraremos em detalhes nesse pacote.

O ProofPower-Z é a parte do sistema responsável por oferecer suporte a fragmentos escritos em Z. Esse pacote providencia suporte a uma extensão de uma aproximação da notação Z. Aqui, entenda-se por aproximação da notação Z como algo entre o ISO e o *Spivey* e a extensão dessa aproximação como todo o suporte oferecido por esse pacote.

As principais diferenças entre a aproximação da notação Z e a extensão oferecida pelo ProofPower-Z são :

- A extensão é de alta ordem, o que remove a distinção sintática entre predicados e expressões em Z.
- A extensão permite misturar fragmentos de HOL com fragmentos de Z.
- A extensão possui dois operadores que são utéis durante a prova. Um deles é usado quando não se sabe o tipo de uma dada variável, enquanto o outro força o tratamento de esquemas de expressão como se fossem esquemas de declaração.

O ProofPower segue a tradição de Z cujo gerenciamento de especificações e provas é baseado na edição de documentos em Latex. O pacote que permite tal caracteristica é o XPP, uma ferramenta gráfica que providencia uma maneira de preparar, checar e executar scripts do ProofPower. Ela combina a idéia de um editor de texto com uma interface de comando ML. O XPP pode ser usado também apenas como editor de documentos ProofPower. Ela providencia várias facilidades para as construções mais comuns em Z tais como boxes de esquemas. As especificações e as provas podem ser checadas pelo ProofPower incrementalmente a medida que são desenvolvidas. Para se fazer isso basta selecionar o fragmento com o mouse e executá-lo. Isso faz com que um comando ML apropriado seja executado e, se for correto, atualiza a teoria presente no ProofPower. Se não for correto o ProofPower devolve um diagnóstico, que entraremos em detalhes mais adiante.

5.2 Consistência das Especificações

A grande vantagem de se utilizar uma ferramenta do porte do ProofPower é poder fazer provas sobre a notação Z. Provas informais nessa notação geralmente tratam a especificação como uma coleção de axiomas. Essa abordagem nos remete ao problema da trivialização, onde qualquer coisa pode ser provada a partir de um conjunto inconsistente de axiomas. É preciso, portanto, relaxar de alguma forma a lógica para que esse problema não ocorra. O mecanismo proposto é a regra da "extensão conservativa", que permite a definição de novos objetos mantendo-se aparentemente a consistência.

5.3 Suporte a Provas em Z

Apresentaremos nessa seção os métodos de prova a notação Z oferecidos pelo ProofPower. Mais especificamente, mostraremos os métodos usados para provas em cálculo proposicional e cálculo de predicado. Outros métodos de provas para Z serão mostrados em uma outra oportunidade.

5.3.1 Cálculo Proposicional

Os conetivos proposicionais de Z são mapeados diretamente aos conetivos correspondentes em HOL, e o raciocínio proposicional do ProofPower-Z comporta-se conseqüentemente de uma maneira idêntica ao raciocínio proposicional do ProofPower-HOL. Os métodos principais de prova são :

1. Foward usando regras elementares.
2. Goal orientado a prova dividida.
3. Goal orientado a prova automática.

A prova do tipo Foward usando regras elementares é raramente requerida no ProofPower-Z e funciona apenas para os conetivos proposicionais, não funcionando sobre os operadores correspondentes ao schema calculus. Algumas das regras são : *asm_rule*, que produz ou usa uma suposição; *_elim*, que é o *modus pones*; *_intro*, que prova uma implicação; *strip_∧_rule*, que quebra uma conjunção e *list_∧_intro*, que cria uma conjunção a partir de uma lisat de teoremas.

A prova do tipo Goal orientada a prova dividida é baseada no método da prova por contradição, dividindo-se o que se deseja provar em suposições. Isso é feito através do comando *a contr_tac*. Mas para uma esclarecimento maior de como a conjectura é dividida em suposições, há dois comandos : *a z_strip_tac* e *a step_strip_tac*. O primeiro divide totalmente em suposições tudo o que vem antes da implicação e, depois, divide tudo o que vem logo após a implicação, produzindo vários sub-goals. O segundo é primeiro só que passo a passo, mostrando cada divisão sequencialmente.

A prova do tipo Goal orientado a prova automática geralmente é usada para resultados que são reduzíveis para cálculo proposicional puro. Há dois comandos para a prova automática : *a prove_tac* e *asm_prove_tac*. O primeiro deve ser usado somente quando o goal não possui suposições. Se suposições forem necessárias para obter a prova, usa-se o segundo comando. É importante ressaltar que a repetição desses comandos não gera progresso algum, apesar de que mesmo falhando, pode apresentar resultados mais simples que os outros métodos.

5.3.2 Cálculo de Predicado

Vários métodos de provas podem ser usados para resultados em cálculo de predicado. São eles :

1. Prova por divisão.
2. Prova automatica.
3. Prova pelo método de duas táticas.
4. Prova usando Forward Chaining.

Os primeiros dois métodos, que são completos para o cálculo proposicional, podem falhar para cálculo de predicado puro. Para esses casos, os dois últimos métodos são suficentes para obter resultados com pouco esforço do usuário.

A prova pelo método de duas táticas é baseado em dois passos. O primeiro passo é baseado no método da prova por contradição, dividindo-se o que se deseja provar em suposições. Isso é feito através do comando *a contr_tac*, que é o mesmo usado para provas em cálculo proposicional. O segundo passo é dependente da escolha do usuário. É preciso definir as suposições universais e com quais valores serão instanciados. Isso é feito através do comando *a z_spec_asm_tac suposição valor* .

A prova usando Forward Chaining providencia uma maneira fácil de provar algo que requer instanciação de suposições universais. Quando um prova falha pela *a contr_tac*, o comando *all_asm_fc_tac* pode ser capaz de dar sequência a prova. Esse comando tenta instanciar suposições universais com valores que permitam continuar com a prova. Esse comando deve ser usado no máximo duas vezes, e se falhar, deve-se usar o método de duas táticas.

Os métodos descritos acima providenciam um bom suporte para cálculo de predicado puro, mas podem falhar quando há o uso de equações para se completar a prova. O ProofPower possui alguns comandos relacionados ao uso de equações. Esses comandos geralmente eliminam as equações, rescrevendo a conclusão do sub-goal.

5.4 Diagnóstico do ProofPower

O ProofPower reporta erros de especificação e provas via o mecanismo de exceção do ML. As mensagens de erro possuem genericamente o seguinte formato :

< Texto Explicativo Opcional >
Exception - < Tipo de Erro > ∗ < Texto > [< Função > • < Número >] ∗ raised
onde :

* Texto Explicativo Opcional : se aparecer é uma descrição estendida do problema. É bastante usado por outros sub-sistemas tais como o parser, que usa um layout customizado para suas mensagens de erro.

* Tipo de Erro : o erro pode ser do tipo Fail, indicando um erro onde uma função não pode ser usada apropriadamente no contexto, ou do tipo Error, quando não se sabe a origem do erro.

* Texto : na ausência do texto explicativo opcional, torna-se a principal descrição do erro. Essa descrição é gerada a partir de um banco de dados de erros.

* Função : nome da função ou sub-sistema que origina o erro.

* Número : é o número do erro relacionada a função com problemas. É gerado a partir de um banco de dados de erros.

5.5 Um exemplo

Para um maior esclarecimento sobre a ferramenta ProofPower nada melhor que um exemplo prático. Primeiramente é preciso termos um banco de dados que contenha o suporte a notação Z, ou seja, é necessário criamos um banco de dados cujo pai é o banco de dados pp_zed, que oferece suporte a Z. Isso é feito através do seguinte comando :

pp_make_database -p *path*/pp_zed meudb

O próximo passo é rodar o XPP com suporte a interface de comando ML através do comando seguinte :

xpp -f meudoc.doc -c pp -d meudb

É preciso agora deixar claro que onde for encontrado SML é uma referência a interface de interação da meta-linguagem com o ProofPower e onde for encontrado Output ProofPower é uma referência a saída emitida pelo sistema.

Um dos aspectos mais relevantes é a escolha da teoria. No nosso caso, precisamos de uma nova teoria que seja filha da teoria "z_library". Isso é obtido através de :

SML :

open_theory "z_library";
new_theory "minhateoria";

É importante também definir o contexto de provas. Como desejamos suportar provas em Z, o que precisamos fazer é :

SML :

set_pc "z_library";

O ProofPower armazena em uma pilha as provas dos sub-goals necessários para se alcançar o goal. Quando uma das provas falha ou é imcompleta, a mesma continua nessa pilha. Para evitar isso, o comando que limpa a pilha é :

SML :

repeat drop_main_goal;

Podemos agora evocar o comando de status para dispormos de informações sobre nosso contexto atual :

SML :

print_status();

Output ProofPower :

Current theory name : "minhateoria";
Current proof context name(s) : [z_library];
The subgoal package is not in use;
There is no current goal.
val it = () : unit

Com o contexto perfeitamente configurado podemos escrever as especificações em Z e, a partir destas fazer provas. Para atualizar a teoria com as especificação documentada, basta selecionar a especificação com o mouse e ir em Command->Execute Selection. As figuras abaixos são um screenshot do XPP com um exemplo bem simples e a prova de sua inconsistência :

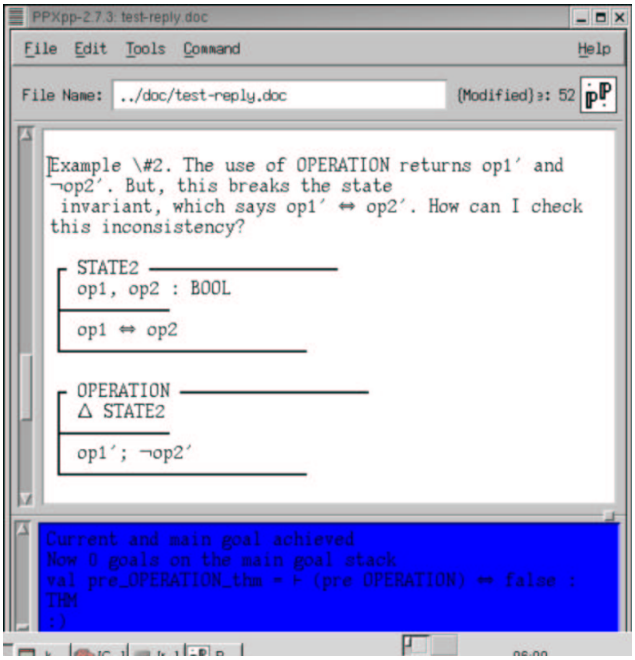

Figura 1: XPP com suporte a comando ML

A figura 1 mostra o XPP com suporte a comando ML. A parte branca é o editor de documentos. Podemos ver que há uma especificação em Z inconsistente, pois o operador OPERATION retorna op1' e ¬ op2', o que viola o invariante do sistema op1 ⇔ op2. A parte azul é saída do ProofPower.

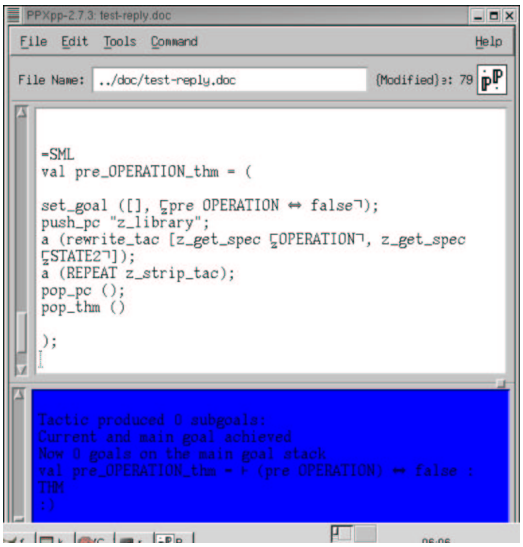

Figura 2: Código ML que exige a prova

A figura 2 mostra no editor de documentos o código ML que verifica se a pré-condição do OPERATION vale em todas as instâncias, ou seja, se nunca viola um invariante do sistema.

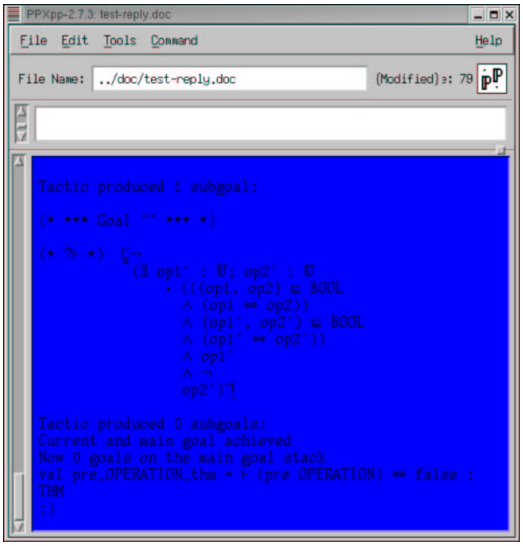

Figura 3: Saída da prova obtida pelo ProofPower

A figura 3 mostra os passos realizados pelo ProofPower para provar a inconsistência. O que se deseja é que além disso o ProofPower diga onde está o problema e proponha uma solução. No caso do exemplo seria abandonar o invariante do sistema ou op1' ou ¬ op2'.

6 Trabalhos Relacionados

A idéia de se usar revisão de crenças como uma ferramenta de manutenção de requisitos não é nova. *Zowghi et al* [22] apresentam um framework para modelar e raciocinar sobre a evolução de requisitos. Esse framework vê o modelo de requisitos como uma teoria de alguma lógica não-monotônica, cuja evolução envolve o mapeamento de uma teoria a outra. Eles implementaram operadores da revisão de crenças no sistema THEORIST (sistema de raciocínio não-monotônico) e mostram com exemplos como se faz a evolução e manutenção de requisitos.

Já *Williams et al* [23] mostram o uso de uma proposta de revisão de crenças interativa, chamada de *maxi-adjustment*, para a manutenção de requisitos. Eles usam um framework chamado "Goal Structerd Analysis", que se baseia na decomposição do goal principal em sub-goals. Eles mostram que esse framework é o ideal para se aplicar revisão de crenças baseado no epistemic entrenchment, uma vez que o goal é mais importante que os subgoals, não violando o postulado **EE2**. Eles mostram a aplicação através de um exemplo de requisitos para um sistema de freios de avião.

Rodrigues et al [24] apresentam a idéia de revisão de crenças clusterizadas para solucionar o problema. Basicamente, as crenças são agrupadas de acordo com algum critério e esses grupos ordenados parcialmente ou totalmente. A revisão é feita abandonando-se as crenças dos grupos que estão nas primeiras posições da ordenação. A idéia é derivada do epistemic entrenchment, só que com uma abstração maior. Eles usam lógica clássica proposicional (DNF) como representação dos requisitos e aplicam a idéia com um exemplo de requisitos de um sistema de controle de iluminação.

Apesar de serem propostas bem interessantes e aplicáveis, ainda estão distantes da área prática de engenharia de software, pois usam frameworks e linguaguens matemáticas que são desconhecidas até mesmo para pessoas de desenvolvimento formal de software.

7 Conclusões e Trabalho Futuro

Nesse relatório apresentamos a teoria básica de revisão de crenças e a mais usada notação de desenvolvimento formal de software. Fizemos também uma pequena introdução ao ProofPower, uma das mais usadas ferramentas de suporte a especificações e provas em Z. Desejamos estendê-la implementando um algoritmo de revisão de crenças.

Mostramos também que a notação Z possui problemas quando se deseja fazer provas automáticas. A principal caracteristica que torna complicado o uso de ferramentas de suporte para Z é o uso de nomes e a consequente complexidade da semântica dos esquemas. Isso tem um grande impacto pois significa que a substituição não pode ser feita da maneira usual. Mesmo com esses problemas, acreditamos que a revisão de crenças possa trazer grande economia de custos quando se trata de propor soluções para a inconsistência de grandes especificações escritas em Z. A pequisa não tem intenção propor que a revisão de crenças solucione todos os problemas de inconsistência em Z. O que se propõe é uma integração inicial entre essas duas áreas da computação com uma forma de efetiva aplicação da teoria da revisão de crenças.

O próximo passo é verificar como o algoritmo de provas do ProofPower detecta inconsistências e adicionar a esse provador de teoremas algum algoritmo de revisão de crenças como forma de solução para a inconsistência encontrada. Obtido isso, seria interessante tentarmos usar a idéia de prioridades para os requisitos, colocando por exemplo que um invariante de um estado é mais importante que um operador.

Referências

[1] C. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meetcontraction and revision functions. *Journal of Symbolic Logic*, 50:510–530, 1985.

[2] Peter Gärdenfors. *Knowledge in Flux: Modeling the Dynamics of Epistemic States*. MIT Press, 1988.

[3] P. Gärdenfors and D. Makinson. Revisions of knowledge systems using epistemic entrenchment. In *Proceedings of the Second Conference on Theoretical Aspects of Reasoning about Knowledge Conference*, pages 83–95. Morgan Kaufmann, 1988.

[4] C.E. Alchourron and D. Makinson. On the logic of theory change: Safe contraction. *Studia Logica*, (44):405–422, 1985.

[5] A. Grove. Two modellings for theory change. *Journal of Philosophical Logic*, (17):157–170, 1988.

[6] Adnan Darwiche and Judea Pearl. On the logic of iterated belief revision. In Ronald Fagin, editor, *Proceedings of the fifth Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 5–23. Morgan Kaufmann, Pacific Grove, CA, 1994.

[7] Sven Ove Hansson. *A Textbook of Belief Dynamics*. Kluwer Academic Publishers, 1997.

[8] E. Fermé and S. Hansson. Selective revision. *Studia Logica 63*, pages 331–342, 1999.

[9] D. Makinson. Screened revision. *Theoria 63*, pages 14–23, 1997.

[10] Meyer, Ghose, and Chopra. Non-prioritised ranked belief change. *TARK: Theoretical Aspects of Reasoning about Knowledge*, 8, 2001.

[11] Anthony Hall. Seven myths of formal methods. *IEEE Software*, pages 11–20, 1990.

[12] Mike Spivey. The z notation: a reference manual. *Prentice Hall*, 2001.

[13] Andrew Martin. Why effective proof tool support for z is hard. *Software Verification Research Centre*, pages 97–34, 1997.

[14] ORA Canada Ltd. Z/eves reference manual. available at http://www.ora.on.ca/z-eves/documentation.html.

[15] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 283–298, Turku, Finland, 1996. Springer-Verlag LNCS 1125.

[16] Ian Toyn and John A. McDermid. Cadiz: An architecture for z tools and its implementation. *Software - Practice and Experience*, 25(3):305–330, 1995.

[17] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[18] R.B. Jones. Icl proofpower. *BCS-FACS FACTS*, 1992.

[19] Lemma 1 Ltd. Proofpower description. available at http://www.lemma-one.com/proofpower/doc/doc.html.

[20] Lemma 1 Ltd. Proofpower z tutorial. available at http://www.lemma-one.com/proofpower/doc/doc.html.

[21] Lemma 1 Ltd. Xpp user guide. available at http://www.lemma-one.com/proofpower/doc/doc.html.

[22] D. Zowghi, A. Ghose, and P. Peppas. A Framework for Reasoning about Requirement Evolution. In N. Y. Foo and R. Goebel, editors, *Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence, Cairns, Australia, 1996*, pages 157–168. Springer Verlag, 1996.

[23] Williams and MacNish. From belief revision to revision design : Applying theory change to changing requirements. 1997.

[24] O. Rodrigues, A. d'Avila Garcez, and A. Russo. Reasoning about requirements evolution using clustered belief revision. In A. S. d'Avila Garcez G. Spanoudakis and A. Zisman, editors, *Proceedings of ACM ESEC/FSE International Workshop on Intelligent Technologies for Software Engineering WITSE03*, 2003.