Universidade Federal de Pernambuco
Graduação em Ciências da Computação
Centro de Informática
2009.1


# Geração de Modelos de Teste em CSP a Partir de Casos de Uso

Trabalho de Graduação

Gleibson Rodrigo Silva de Oliveira

Orientador: Augusto Cezar AlvesSampaio

Co-orientador: Sidney Carvalho Nogueira

Universidade Federal de Pernambuco
Graduação em Ciências da Computação
Centro de Informática
2009.1


# Geração de Modelos de Teste em CSP a Partir de Casos de Uso

Trabalho de Graduação

Gleibson Rodrigo Silva de Oliveira

Projeto de Graduação apresentado no Centro de Informática da Universidade Federal de Pernambuco por Gleibson Rodrigo Silva de Oliveira, orientado pelo Prof. Augusto Cezar Alves Sampaio, como requisito parcial para a obtenção do grau de Bacharel em Ciência da

Orientador: Augusto Cezar Alves Sampaio

Co-orientador: Sidney Carvalho Nogueira

## Folha de Aprovação

# Geração de Modelos de Teste em CSP a Partir de Casos de Uso

---

Gleibson Rodrigo Silva de Oliveira

Aprovado em 23 de Junho de 2009

Banca Examinadora:

_____

Prof. Augusto Sampaio

(Orientador)

_____

Prof. Alexandre Cabral Mota

(Avaliador)

*“Never send a human do a machine job”*

***Agent Smith***

*“O sertanejo é antes de tudo, um forte”*

***filme Alto da Compadecida***

# Agradecimentos

Dedico este trabalho a meus pais, que lutaram desde cedo para me oferecer a educação que todo pai e mãe sonham para seus filhos, e conseguiram.

Agradeço aos amigos de verdade que encontrei durante o curso, que me renderam boas noitadas de farras e conversas sérias, e as vezes os dois ao mesmo tempo.

Um abraço ao meu amigo Glerter e boa sorte em sua nova jornada. Sucesso!
Agradecer também a minha namorada Nathalia pelos choques de realidade, me fazendo colocar os pés no chão novamente, e por compreender meus momentos de ausência.
Agradecer a Augusto Sampaio, meu orientador, pela paciência, seriedade e atenção. Também a Sidney Nogueira, pela enorme paciência e dedicação durante todo o trabalho.
Agradeço a oportunidade de encontrar pessoas tão incríveis que encontrei durante minha graduação. Com certeza o período de maior aprendizado na minha vida, onde até mesmo decepções me ensinaram muito mais do que eu podia esperar.

Em memória de minha tia Virgínia.

# Resumo

Testes ainda é a mais utilizada atividade de verificação e validação de software. Porém, esta atividade ainda acarreta um custo de cerca de 40% do gasto com o desenvolvimento do software. Neste cenário, pesquisas na literatura apontam para a automação deste processo, na tentativa de promover a redução deste custo. Este trabalho de pesquisa segue a abordagem de geração automática de testes segundo modelos formais, ou simplesmente Model Based Testing (MBT).

A proposta deste trabalho é estender uma estratégia de automação de testes a partir de casos de uso, implementando a inclusão de dados no mesmo, pois a estratégia original considera apenas aspectos de controle, bem como de relacionamentos entre casos de uso (noções de includes e extends de UML) Com relação aos dados, são oferecidos mecanismos para declaração de tipos e variáveis, leitura e escrita, entrada e saída.

A pesquisa é realizada no contexto da cooperação entre Centro de Informática – UFPE e uma fabricante de telefones celulares. Para escopo deste trabalho, o projeto de cooperação será referenciado simplesmente como Cin-Test. A estratégia atual encontra-se em processo de implantação e a extensão deve ser conservativa, sendo permitida a inclusão, ou não, de dados durante o processo. Ao final deste trabalho, é gerada a extensão à ferramenta desenvolvida no projeto de cooperação citado.

# Conteúdo

# Índice de imagens

# 1. Introdução

A crescente velocidade de desenvolvimento e distribuição de produtos de software torna evidente a preocupação com a qualidade do produto final. Com o aumento da velocidade, o cliente final exige mais qualidade e menos tempo de espera pelo produto entregue. Neste cenário, não há espaço para grandes falhas ou promessas não cumpridas pelo software, acarretando, no pior caso, a perda do cliente.

Descrito o cenário, faz-se necessário adotar políticas de verificação e validação do software. Essas atividades (V&V) continuam a ser bastante custosas dentro do desenvolvimento do produto. Em alguns projetos equivalem a 40% do tempo de desenvolvimento [01]. Portanto, surgem iniciativas para automatizar o processo, em particular os testes, que continuam sendo a técnica de verificação e validação mais comumente utilizada.

Tomamos, neste trabalho, a atividade de testes de software partindo da especificação do caso de teste até a execução do mesmo, incluindo rastreabilidade com casos de uso e requisitos e critérios de cobertura. Segundo [02] a atividade de teste pode ser dividida em cinco partes: Planejamento e controle, design, construção e execução, avaliação das saídas e fechamento das atividades. Para simplificação, neste trabalho condensaremos as cinco atividades em apenas três: criação do caso de teste (planejamento, controle, design e construção), execução dos testes e avaliação das saídas (avaliação e fechamento). Este trabalho foca na primeira etapa, a criação do caso de teste.

Segundo [02], a atividade de construção do caso de teste compreende as seguintes atividades:

- Planejamento dos testes;
- Acompanhamento das atividades de teste;
- Análise de riscos dos testes;
- Transformação de objetivos dos testes em casos de testes tangíveis;
- Priorizar os testes;
- Identificar dados de teste para suportar a execução;
- Desenvolver os procedimentos dos testes;

- Procedimentos para comparação de resultados desejados com resultados obtidos.

A execução dos testes e avaliação dos resultados são atividades que podem ser realizadas por profissionais de áreas distintas da engenharia de software. O mesmo não acontece com a confecção dos testes, tornando-a uma atividade mais delicada, pois objetivos de negócio e qualidade do produto devem guiar os testes. É também uma atividade mais custosa, devido ao tipo do profissional, o tamanho do esforço empregado e ao tempo empregados na atividade, e ainda passível de erros, por ser uma atividade repetitiva, falhas humanas podem ser inseridas durante o processo.

Nas recentes pesquisas na literatura, um tema bastante presente é a geração automática de casos de teste. Considerando este tema, encontramos duas alternativas: geração baseada em modelos UML [07,08,09,10] e geração baseada em modelos com notação formal [03,04,05,06,27].

A utilização de modelos UML para geração de testes é bastante adotada na indústria devido a larga utilização da notação UML para descrever software. Porém, para avaliarmos se uma implementação é válida de acordo com a especificação e para comparar os testes com os resultados obtidos de sua execução, precisamos de uma noção mais precisa de corretude. Utilizando métodos formais, podemos conseguir isso [11].

Neste trabalho de pesquisa a linguagem de especificação formal utilizada é CSP (*CommunicatingSequentialProcess)*, que segundo [12] é um formalismo bem estabelecido para modelagem e verificação de sistemas concorrentes. Um modelo denotacional consolidado e uma ferramenta eficaz têm incentivado muito trabalho sobre o raciocínio algébrico e modelos de verificação utilizando esta linguagem.

Este trabalho possui um modelo CSP como produto intermediário do processo de geração de testes. É mostrado mais adiante, o apoio ferramental existente no contexto desta pesquisa. Este apoio conta com uma ferramenta que toma como entrada os documentos de caso de uso escritos em uma notação controlada e gera um modelo intermediário CSP destes casos de uso. A mesma ferramenta utiliza este modelo formal para gerar os cenários de testes e os casos de testes.

Entretanto, a existência de uma tecnologia não é fator determinístico no sucesso de sua utilização: há uma série de elementos externos envolvidos que se combinam para definir sua aceitação. Estes elementos incluem ambientes de desenvolvimento, ferramentas de suporte, metodologias associadas e volume de documentação [13].

Considerando as restrições, foi desenvolvida uma ferramenta que automatiza a geração de testes. Esta foi construída no contexto da pesquisa em cooperação entre Centro de Informática – CIn – UFPE e uma fabricante de telefones celulares [05]. Diversos conceitos, ferramentas e metodologias apresentadas neste trabalho estão incluídas no contexto da cooperação Cin-UFPE e uma fabricante de telefones celulares. Estes conceitos serão identificados ao longo do trabalho como pertencentes ao projeto de cooperação. A ferramenta citada gera como produto intermediário um modelo CSP extraído dos documentos de casos de uso, e a partir deste modelo são gerados os casos de teste.

Porém, a ferramenta constrói modelos que não levam em consideração a existência de dados de entrada, de estado ou desaída, devido à ausência destes no  teplate de especificação do caso de uso. Essa restrição da ferramenta resulta em modelos que representam apenas o fluxo do caso de uso, limitando as possibilidades apenas para geração de testes de exploração de caminhos. Com a inclusão de dados é possível definir, de maneira clara os limites da aplicação e de estruturas internas. Conhecendo esses limites é possível aplicar mais técnicas de geração de testes a partir do modelo, resultando no aumento do número de cenários de testes e na precisão dos mesmos, utilizando os dados para casos de testes de limites da solução.

## 1.1 Objetivos

Analisar e implementar uma extensão à atual ferramenta para geração de modelos formais, utilizando CSP, que leva em consideração os dados de entrada. Para que isso seja viável, é necessário analisar como esses dados serão dispostos dentro do template de espeficicação do caso de

uso, que serve de entrada para a ferramenta. Será necessária a extensão do template bem como o processamento do mesmo para geração do novo modelo CSP. Com relação aos dados, a notação corrrente dos templates é estendida com construções para declarar tipos e variáveis, ler e escrever valores de variáveis e entrada e saída de valores.

Os documentos de caso de uso atuais possuem uma estrutura xml agregada que compõem o caso de uso lido pela ferramenta. Esse xml deve ser estendido para que seja possível adicionar tipos de relacionamento entre os casos de uso, promovendo aumento do reuso, e ainda os dados de entrada. Atualmente, o reuso de fluxos de casos de uso dá-se através dos direcionamentos do início e final do caso de uso, representados pelas palavras FROM STEP e TO STEP, repectivamente. Esta abordagem permite apenas o reuso de fluxos inteiros, não sendo possível o aproveitamento de apenas parte do fluxo de um caso de uso específico. Mais detalhes sobre esse template serão encontrados nos capítulos adiante.

O escopo da solução proposta começa na disposição destes novos elementos no documento de caso de uso, e consequentemente no documento xml, até a geração do modelo CSP. Este trabalho não propõe novas técnicas para geração de casos de teste a partir do novo modelo com dados. Este tópico é abordado na seção de conclusões e trabalhos futuros.

## 1.2 Estrutura do trabalho

Este trabalho está estruturado da seguinte forma:

- Capítulo 2: Revisão da literatura, onde são apresentados os principais conceitos para entendimento de todo este trabalho de pesquisa. Aqui também são apresentadas as principais pesquisas encontradas na área de MBT.

- Capítulo 3: Apresentação do modelo adotado pela fabricante de telefones celulares, onde é feito um breve resumo do modelo adotado com base nas publicações recentes a respeito.
- Capítulo 4: Apresentação do modelo de extensão, onde podemos encontrar todas as sugestões e implementações deste trabalho. O desenvolvimento da solução deu-se sobre uma ferramenta existente.
- Capítulo 5: Conclusões, resultados obtidos e Trabalhos futuros, onde são encontradas as principais conclusões e benefícios deste trabalho. Também são encontrados os principais trabalhos futuros na continuação deste trabalho.

# 2. Testes de Software baseados em modelos

Testes de software necessitam do uso de um modelo para guiar os esforços na seleção dos testes e na verificação dos mesmos. Na maioria dos casos, esses modelos são implícitos e existem apenas na cabeça dos testadores, que aplicam este conhecimento de maneira *ad-hoc.* Estes modelos encapsulam o comportamento da aplicação, permitindo aos testadores entender as capacidades do sistema. Quando estes modelos são escritos, eles se tornam compartilháveis e reusáveis. Neste caso, os testadores iniciam a metodologia de testes baseados em modelos, MBT (*ModelBasedTests*) como é conhecida na literatura.

Testes baseados em modelos têm ganhado *bastante* atenção com a popularização do uso de modelos, em particular UML, no contexto da engenharia de software. Atualmente, existe uma série de modelos usados para como base para casos de teste.

Neste capítulo são introduzidos os conceitos básicos acerta de MBT, que darão suporte ao entendimento do conteúdo deste trabalho. Aqui serão apresentados conceitos básicos e também uma revisão mais aprofundada da literatura de testes baseados em modelos. Serão explanadas as técnicas presentes na literatura e as principais adoções dessas técnicas na indústria.

## 2.1 Definição dos conceitos de geração de testes e modelos formais

Abriremos a discussão sobre testes baseados em modelos com um pequeno conjunto de definições comuns da literatura. Alguns dos modelos mais populares são abordados. Os conceitos aqui presentes estão contextualizados com a pesquisa deste trabalho. Conceitos mais específicos de cada técnica serão mostrados na seção 2.2, onde será feito um apanhado geral sobre as recentes pesquisas na área.

### 2.1.1. Modelos

De acordo com a pesquisa de [14], um modelo é uma descrição do comportamento de um sistema. Comportamento pode ser descrito como sequencias de entradas aceitas pelo sistema, ações, condições e lógica externa, ou o fluxo dos dados através dos módulos e rotinas da aplicação. Logo, um modelo para ser útil para grupos de testadores e para múltiplas tarefas de teste deve compreender o modelo mental dos testadores e ser escrito de uma maneira clara e simples. Geralmente prefere-se um modelo formal a um modelo prático.

Existem numerosos tipos de modelos, onde cada um descreve diferentes aspectos da aplicação. Por exemplo, controle de fluxo (controlflow), fluxo de dados (data flow) e grafo de dependência expressam como a implementação se comporta representando a estrutura do código fonte. Tabelas de decisão e máquinas de estado (state machines), por outro lado, são usadas para descrever comportamento de chamadas de funções, blackbox. Considerando MBT, a comunidade atualmente tende a pensar em termos de modelos de caixa preta (blackbox).

Não falaremos em detalhes sobre cada tipo de modelo de software que pode ser adotado em MBT. Porém, falaremos dos principais modelos e conceitos que ajudarão a entender o desenrolar deste trabalho. Termos mais gerais, que abrangem o contexto desta pesquisa, são demonstrados a seguir.

### 2.1.2. *Finite State Machine* (FSM) e *Labeled Transition Systems* (LTS)

Um ponto de partida para analisar a relação entre métodos formais e geração de testes é considerar que ambos são baseados em modelos. Notações formais como máquinas de estados finitas,

FSM (*FiniteState Machine)* [15], e sistemas rotulados por transição, LTS (*LabelledState Machine*) [16], fornecem modelos não ambíguos das funcionalidades do sistema.

Segundo [17], uma máquina de estados finita (FSM) é uma 6-upla $(S, I, A, R, \Delta, T)$, onde $S$ é um conjunto de estados finito, $I \subset S$ um conjunto de estados iniciais, $A$ é um alfabeto finito de símbolos de entrada, e $R$ um conjunto de possíveis saídas e respostas. O conjunto $\Delta \subset S \times A$ é o domínio da transição $T$, que representa uma função de $\Delta$ para $S \times R$. A relação de transição define como a máquina de estados reage ao receber uma entrada $a \in A$ quando o estado é $s \in S$, assumindo que $(s, a) \in \Delta$. A situação $(s, a) \notin \Delta$ significa que o símbolo de entrada não pode ser recebido naquele determinado estado. Quando $T(s, a) = (s', r)$, o sistema atualiza o estado para s' e responde uma saída *r*. Se *T* não é uma função, mas mantém uma associação de cada par estado-entrada com um conjunto não vazio de pares estado-resposta, podemos dizer que a máquina de estados (FSM) é não determinística, e as saídas são interpretadas como um conjunto de possíveis respostas para um dado de entrada em um determinado estado.

Em um modelo baseado em uma máquina de estados finita de um software, o estado normalmente representa algum aspecto do fluxo de controle do programa. Por exemplo, o estado de um modelo baseado em FSM em uma abordagem *white-box* representa um bloco básico de código que está sendo executado ou até mesmo o registrador de eventos do sistema.

Em [18], encontramos que uma máquina de estados pode ser representada por um diagrama de transição de estados. O resultado desta representação é um grafo direcionado onde os vértices correspondem aos estados da máquina e as arestas correspondem

às transições dos estados. Cada aresta é rotulada com os dados de entrada e saída associados à transição.

[17] define sistemas rotulados por transição (LTS) como sendo uma 4-upla $(S, L, T, s_0)$, onde S é um conjunto contável e não vazio de estados, *L* é um conjunto contável de rótulos, $T \subseteq S \times (L \cup \{\tau\}) \times S$ é a relação de transição e $s_0 \in S$ é o estado inicial.

O rótulo $L$ representa as interações observáveis do sistema, o rótulo especial $\tau \notin L$ representa uma ação interna não observável. É denotado a classe de todas as transições rotuladas do sistema sobre $L$ por $\mathcal{LTS}(L)$. O *trace* (caminho) é uma sequencia finita de ações observáveis. O conjunto de todos os *traces* sobre $L$ é denotado por $L^*$, com $\varepsilon$ denotando uma sequencia vazia. Se $\sigma_1, \sigma_2 \in L^*$ então $\sigma_1 \cdot \sigma_2$ é a concatenação de $\sigma_1$ e $\sigma_2$. Com $|\sigma|$ representando o tamanho do *trace* $\sigma$, por exemplo, número finito de ocorrências da ação em $\sigma$.

### 2.1.3. *Controlled Natural Language* (CNL)

Visando prover maior formalismo e controle a todo o ciclo de geração de testes; especificações (casos de uso e documentos de requisitos) não são construídas de maneira *ad-hoc.* Algumas abordagens na literatura têm explorado o mapeamento entre especificações formais a partir dos requisitos. Este esforço resulta em linguagens de especificação controladas, para que possam ser processadas por ferramentas de automação.

A linguagem controlada PENG (*ProcessableEnglish*) [19], define o mapeamento entre inglês e lógica de primeira ordem para verificar a consistência dos requisitos. Uma iniciativa bastante similar é ACE (*AttemptoControlledEnglish*) [20] que também envolve processamento de linguagem natural para verificação e validação através de análises

lógicas. Outros trabalhos pesquisam sobre mapeamento entre inglês e sistemas rotulados por transição (LTS) [03] e [21].

Como regra geral, as abordagens para tornar uma linguagem de especificação de documentos mais formal acaba resultando em linguagens controladas. São definidos subconjuntos do inglês, ou outro idioma, para serem controlados e o domínio da aplicação normalmente é refletido neste subconjunto. Uma gama de ferramentas foi desenvolvida para processar essas linguagens naturais. Na seção seguinte mostraremos, de maneira mais aprofundada, as pesquisas com o objetivo de construir uma linguagem natural controlada para processamento e automação.

No contexto deste trabalho de pesquisa, casos de uso são descritos como entidades (atores) realizando ações sequenciais para atingir um determinado objetivo. Logo, casos de uso são especificados como uma sequencia de passos formando o cenário de uso do sistema, e a linguagem natural é usada para descrever as ações que um ator realiza em um passo. Este formato permite que os casos de uso sejam auditados com maior facilidade [03]. É definida então uma CNL (*Controlled Natural Language*) - Linguagem Natural Controlada – que representa um subconjunto do inglês, para permitir a transformação automática para modelos formais CSP.

## 2.2 *CommunicatingSequential Processes* (CSP)

Para especificação formal do modelo é necessário a utilização de uma linguagem formal, e não apenas de uma linguagem controlada. Na literatura, CSP (*CommunicatingSequential Processes*) é o formalismo adotado como padrão em diversos contextos.

CSP [22] [23] é uma álgebra de processos que pode representar máquinas de estado e seus dados de uma maneira abstrata e sucinta. CSP

é uma notação útil para especificar e projetar comportamento de sistemas concorrentes e distribuídos de hardware e software. Conta com modelos semânticos capazes de representar o comportamento de ângulos variados, pelos quais ferramentas podem verificar propriedades como: ausência de *deadlock*, ausência de *livelock*, comportamento determinístico e refinamentos entre processos.

Segundo [04], esta notação formal é utilizada pelo seu poder de descrever aspectos complexos do sistema, como concorrência, em uma notação abstrata, mas, bastante próxima da implementação da solução. Permite ainda a descrição do software em termos de processos que operam independentemente (paralelismo), e interagem (comunicativo) uns com os outros e com o ambiente. A relação entre os processos pode ser feita utilizando os operadores algébricos que permitem a definição de composições complexas de processos. O comportamento dos processos de CSP por si só é descrito em termos de sequência de eventos, que são operações atômicas e instantâneas. Através do mecanismo de troca de mensagens, é introduzido o conceito de canal (representado pela palavra-chave *channel*). Canais podem transmitir mensagens e dados de um determinado tipo (*datatype*).

## 2.3 Tipos de geração de testes a partir de modelos

Aqui serão apresentadas as principais técnicas presentes na literatura de geração de casos de teste. O ponto em comum destas técnicas é o uso de modelos (formais e não formais) para representação da aplicação.

### 2.3.1 Utilização de modelos não formais

Na literatura, o uso de modelos não formais é bastante frequente em pesquisas. A principal ferramenta adotada nestes casos é UML. Os principais benefícios ao adotarmos UML como linguagens de modelagem dos modelos de entrada do MBT são:

- Grande adoção na indústria;
- UML provê modelos genéricos, compatíveis com quase todo tipo de aplicação;
- Possui ferramentas consolidadas no marcado para automação.

Abaixo segue um resumo das principais pesquisas na área de MBT utilizando modelos UML.

#### 2.3.1.1. MBT a partir de conjunto de modelos UML

Esta técnica representa os estudos de [24]. Consiste na construção de UML comportamental para geração de casos de teste e execução de *scripts* com base nos critérios de cobertura do modelo.

UML 2.0 contém um grande número de diagramas e notações definidas por um meta-modelo com alguma liberdade para permitir diferentes interpretações semânticas dos diagramas por diferentes ferramentas. Logo, para um modelo usado em MBT, foi necessário selecionar um subconjunto de UML e deixar clara a semântica do subconjunto escolhido, para que ferramentas de MBT possam interpretar o modelo.

Cada modelo para testes aborda de maneira diferente os diagramas mencionados, suportando vários tipos e definindo um subconjunto fechado destes diagramas que podem ser usados nestes modelos. Foi necessário, então, definir a parte de dados do modelo (diagramas de classe e instâncias são comumente usados para isto) e o aspecto comportamental do modelo.

**Modelando UML para gerar testes:**

Para utilizar UML como base no MBT, é necessário que o modelo descreva o comportamento do sistema. Dentro dos principais diagramas abordados, existem pequenas adaptações e convenções utilizadas para a geração de testes.

- **Diagrama de classes**: representa o modelo de dados do comportamento do sistema;
- **Diagrama de instâncias**: Usado para configurar o estado inicial do sistema para geração de testes, definindo objetos, valores iniciais dos seus atributos e a associação entre objetos;
- **Diagrama de estados(máquina de estados):** Este tipo de diagrama em UML é associado ao fluxo principal do sistema e formaliza o comportamento esperado das classes usando transições entre estados para responder a eventos do usuário;
- ***ObjectConstraintLanguage*(OCL):** Comumente usado para representar o comportamento do diagrama de classes e formalizar o comportamento esperado das operações utilizando-se de pré-condições e pós-condições. Também encontramos OCL com o diagrama de estados para formalizar a transição entre estados. As guardas e os efeitos das transições são expressos em predicados OCL.

Como em qualquer outra abordagem MBT, o modelo não pode ser ambíguo. A natureza de UML o torna ambíguo como dito no início da seção 2.3.1, porém, esta ambiguidade é removida com o uso de OCL. Expressões OCL associadas com operações de classes e máquinas de estados fornecem o formalismo necessário para a modelagem segundo MBT.

Abaixo seguem os exemplos do diagrama de classes e da máquina de estados do exemplo retirado de [24], ambos com notação OCL incluída.

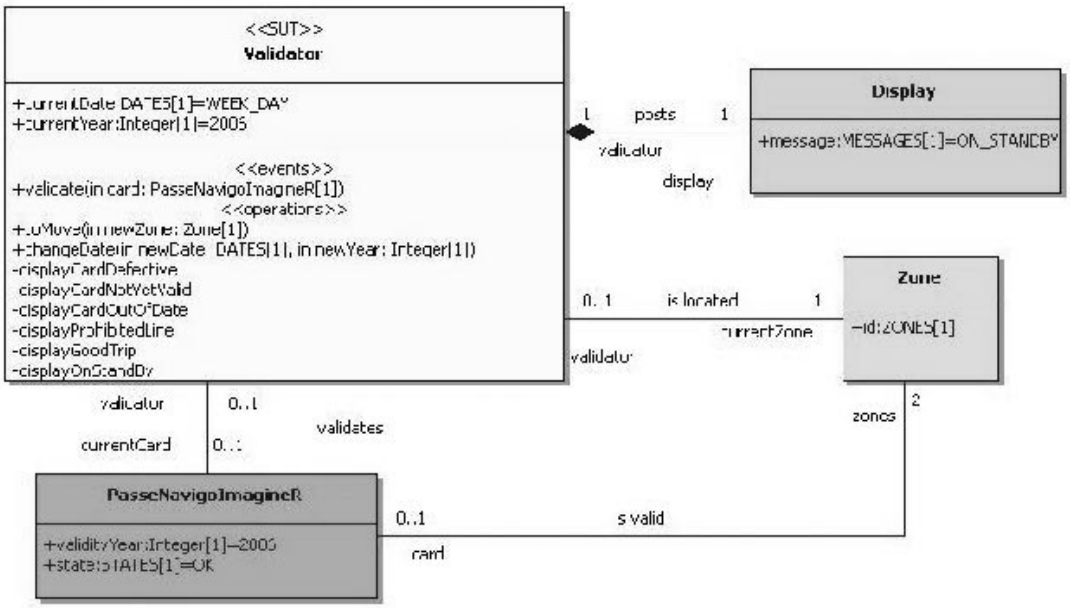



Figura 1–Diagrama de classes com notação OCL

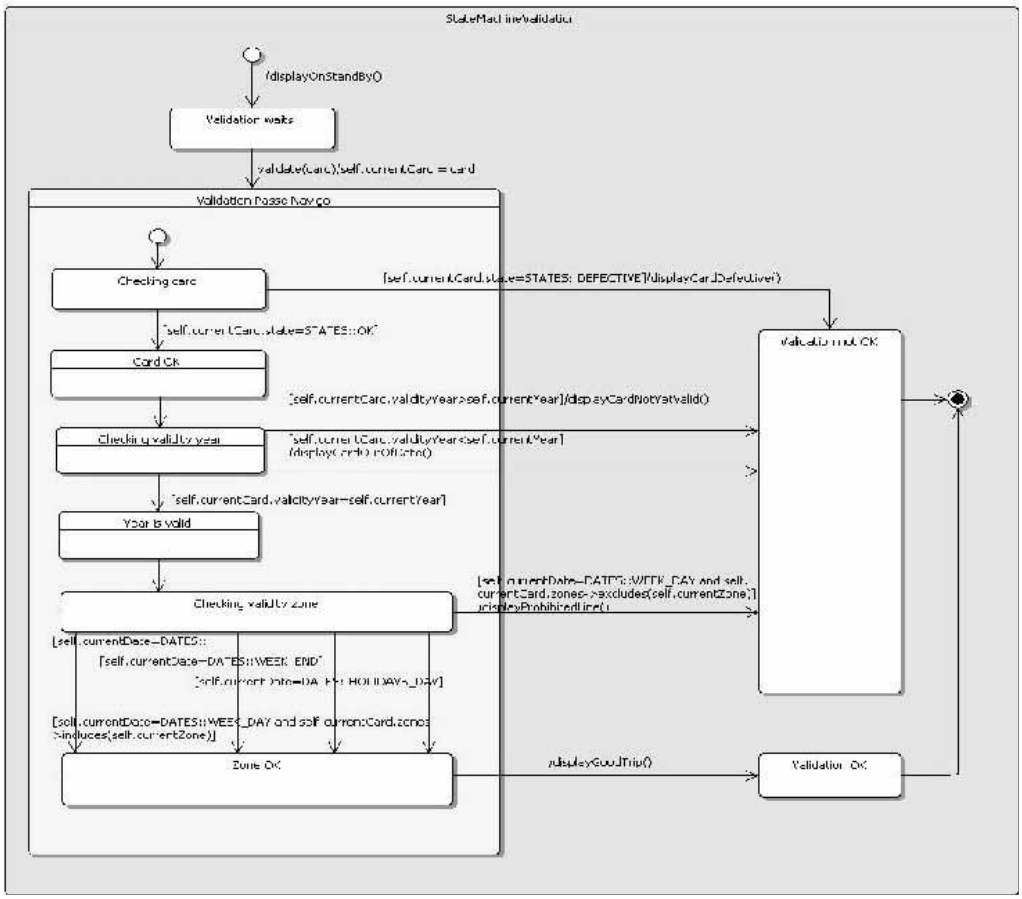



Figura 2 - Máquina de estados com OCL

### 2.3.1.2. MBT a partir de diagramas de estadosUML

**Utilizando FSM e ControlFlow**

Outra pesquisa semelhante é encontrada em [25]. Esta pesquisa toma como base diagramas UML de estados. A principal contribuição deste trabalho é mostrar como diagramas de estados UML podem ser transformados em uma notação onde possam ser aplicadas técnicas tradicionais de análise de fluxo. A metodologia apresentada no artigo transforma diagramas de estados em máquinas de estados finitas estendidas (EFSM) a fim de nivelar a

hierarquia e os estados concorrentes e eliminar a comunicação *broadcast.*

Diagramas de controle de fluxo em UML são identificados em termos de caminhos nas EFSM. O segundo passo do trabalho apresentado transforma EFSM em gráficos de fluxo. Todas as associações entre definições e uso empregadas em diagramas de estados UML podem ser identificadas nos gráficos de fluxo resultantes. A transformação torna viável a aplicação de técnicas convencionais de análise de fluxo de dados (*data flow*) para geração de casos de teste.

As figuras abaixo representam respectivamente, um diagrama de estados UML e uma EFSM resultante.

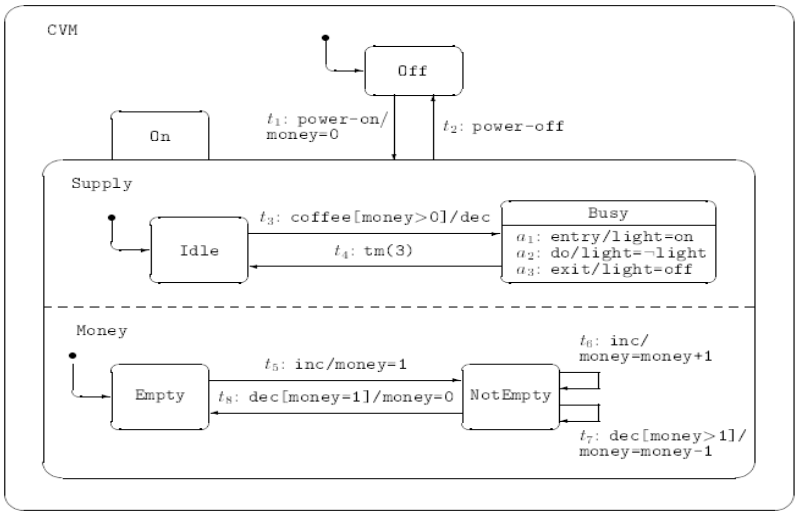
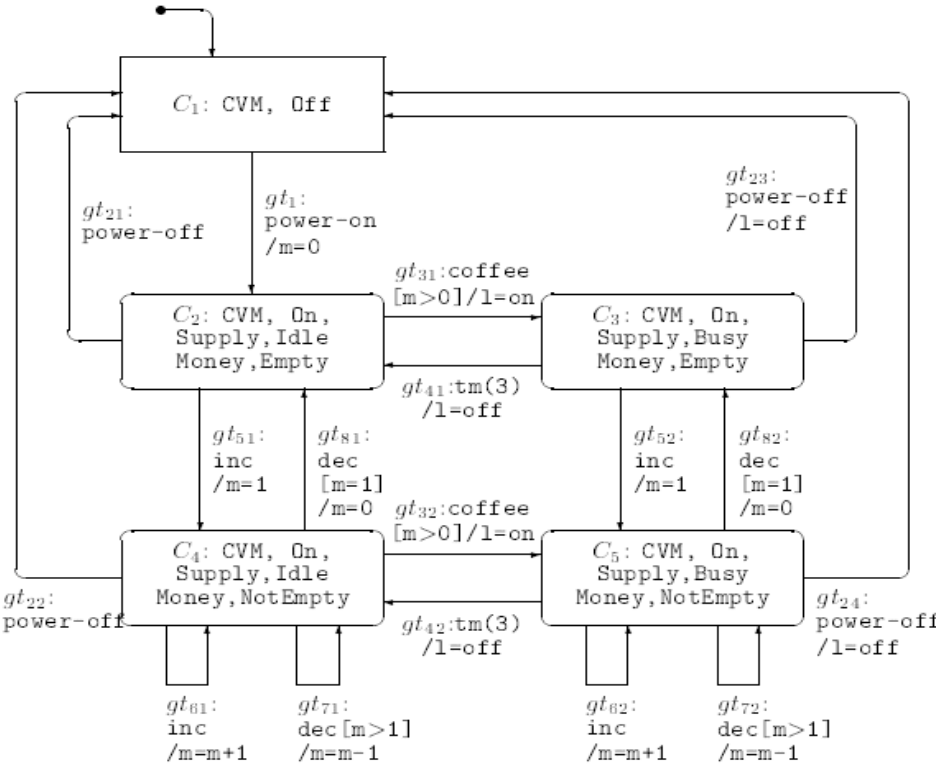



Figura 3 - Diagrama de estados UML detalhado




$$gt_{a1} = (C_3, \lambda, \texttt{true}, \texttt{l=}\neg\texttt{l}, C_3) \qquad gt_{a2} = (C_5, \lambda, \texttt{true}, \texttt{l=}\neg\texttt{l}, C_5)$$

Figura4 - Extended Finite State Machine (EFSM)

Da EFSM resultante deriva-se o seguinte grafo de fluxo.

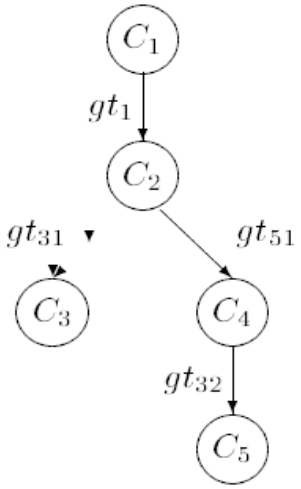



Figura5 - Gráfico de Fluxotradicional

A partir deste grafo de fluxo, são gerados os casos de testes para o sistema, obedecendo a critérios de cobertura.

## AGEDIS – Uma ferramenta para MBT

Outra pesquisa que conta como entrada um modelo semelhante à pesquisa citada anteriormente está presente em [26]. Esta abordagem realizada pelo laboratório de pesquisa da IBM utiliza como entrada diagramas de estados UML controlados. Esses controles representam a semântica do sistema. A figura 6 mostra como esses diagramas são estruturados.

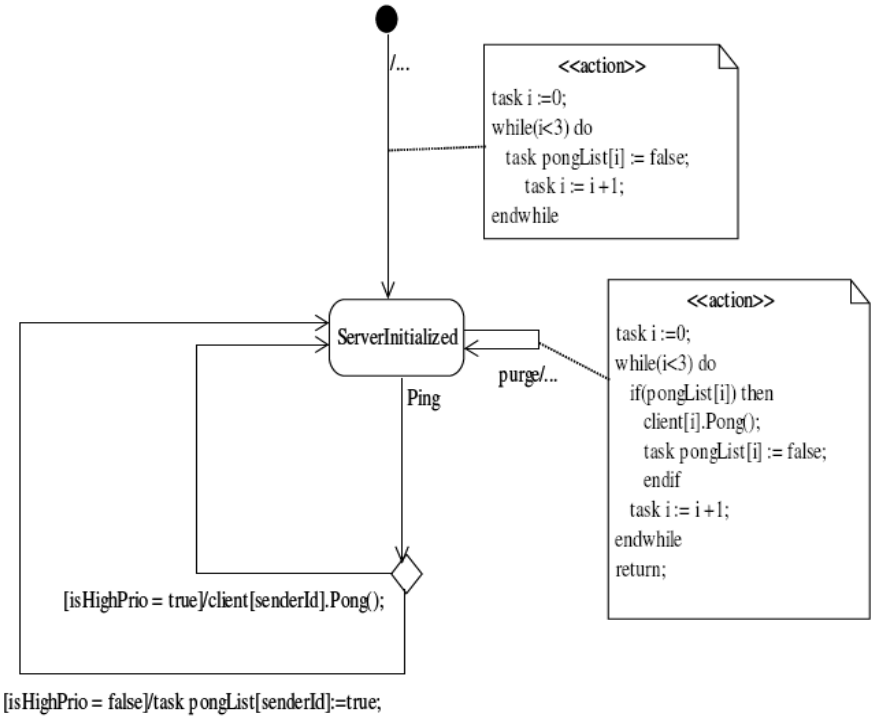



Figura6 - Diagrama de estadoscontrolado

Depois de construídos os diagramas, os testes são gerados, observando o propósito requerido. Para alcançar um determinado propósito, os controladores são observados para o sistema chegar ao estado pretendido pelo propósito especificado. Estes propósitos guiam a geração dos testes.

Ainda nesta linha de pesquisa, outra pesquisa [01] utiliza como entrada sistemas rotulados por transição (LTS), que são extraídos de modelos UML tradicionais utilizando uma ferramenta específica. Este LTS resultante representa a especificação dinâmica da implementação sob teste. Casos de testes são gerados a partir de LTS considerando os propósitos de testes.

Estes propósitos são abstrações dos casos de teste. Como primeira aproximação, podemos observá-los como sequencias de eventos incompletas. Utilizando o exemplo de [01], da máquina de café, um propósito de teste pode ser representado pela saída do café. Isto corresponde a qualquer teste que eventualmente englobe

o estado “servir café”. Vemos a representação do LTS da máquina de café, um propósito de teste e um teste gerado na figura 7, onde “?” representa a entrada de dados e “!” representa qualquer saída.

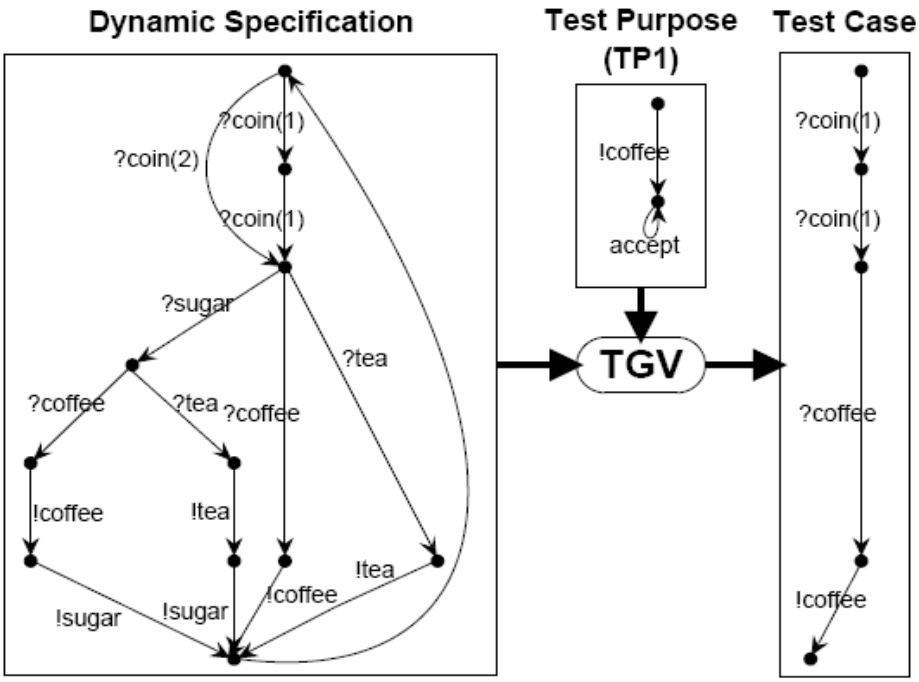



Figura 7 - Especificação, propósito de teste e caso de teste em LTS

## 2.3.2 Utilização de modelos formais

Nesta seção falaremos sobre a utilização de modelos formais na especificação do software. As principais pesquisas na literatura envolvem modelos algébricos como modelos formais de software. Na seção a seguir falaremos sobre esses modelos e sobre as pesquisas recentes.

### 2.1.1.1. Testes utilizando modelos algébricos

Dentre os benefícios de utilizarmos modelos algébricos no MBT, o principal deles é o alcance de um maior formalismo, aproximando o modelo da implementação real do sistema. Ainda contamos com a base matemática para provar a corretude do modelo, do sistema e eventuais transformações sofridas por ambos.

Grande parte dos modelos algébricos encontrados na literatura utiliza a linguagem formal CSP. Em [03], é definida uma linguagem de especificação de requisitos para que, a partir dos requisitos, seja possível gerar um modelo CSP de acordo com o que foi especificado no documento. Este esforço resulta em linguagens controladas, semelhante à [19] e [20]. Essa transformação do documento em modelos, em alguns casos, é feita de maneira automática, com o auxílio de ferramentas específicas.

Como visto em [03], [04] e [05], existem abordagens que tomam como entrada os documentos de requisitos e casos de uso construídos em uma linguagem controlada. Este documento é processado para obtermos os dados necessários para construção do modelo CSP. Este modelo resultante é utilizado para geração de casos de teste.

A mesma abordagem é encontrada em [27], onde todos os modelos de entrada são especificados em uma combinação de CSP e CASL, que combinam processos (CSP) como especificações de tipos de dados (CASL). Neste caso, as entradas no processo MBT são os próprios modelos formais. A idéia principal da pesquisa é descrever sistemas reativos na forma de processo baseados em operadores CSP, onde a comunicação destes processos são os valores dos tipos de dados, que são especificados em CASL.

A seção 3 mostra o modelo utilizado no projeto entre a fabricante de telefones celulares e Centro de Informática – Cin –

UFPE. O modelo é descrito em detalhes e serão apontados os pontos de melhoria do mesmo, assim como sugestões para aumentar a qualidade do processo.

# 3. Modelo utilizado no mercado

Neste capítulo mostraremos a aplicação prática de um subconjunto das técnicas demonstradas na revisão da literatura, do capítulo anterior. As pesquisas realizadas no contexto da parceria Centro de Informática e uma fabricante de telefones celulares, ou simplesmente Cin-Test, resultaram na implementação de uma estratégia para especificação de requisitos e casos de uso, geração de casos de teste e seleção de casos de teste. A estratégia obtida conta com um apoio ferramental que será descrito a seguir.

As principais informações sobre o modelo utilizado na Cin-Test foram obtidas em contato com seus integrantes e a partir das pesquisas [03], [04] e [05] que foram publicadas em congressos e conferências.

## 3.1 Visão geral da estratégia

A abordagem utilizada engloba as atividades de especificação, geração de modelos formais e geração de testes. O resumo completo da estratégia é mostrada na figura abaixo. Todos os pontos do processo abaixo serão explicados no decorrer desta seção.

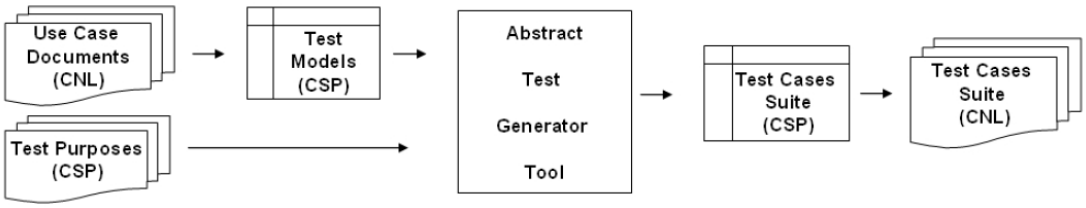



Figura 8 - Visão geral do modelo utilizado como base

A interação entre usuários e sistema e entre o sistema e seus componentes é documentada na especificação dos casos de uso. Estes casos de uso possuem um modelo (*template*) específico, com informações

sobre rastreamento de requisitos, descrição do caso de uso e uma maneira de interagir com o sistema.

Antes da definição dos casos de uso, os requisitos são definidos de maneira abstrata, representando o objetivo do sistema. A partir destes requisitos, casos de uso a nível de usuário (*userview*) são criados baseados na análise dos requisitos. Este primeiro conjunto define como os atores interagem com o sistema. Depois desta definição inicial, são criados os casos de uso a nível de componentes (*componentview*), baseados nos primeiros casos de uso definidos. Esta segunda definição representa como os componentes do sistema se comunicam.

O foco da abordagem está na construção dos documentos de caso de uso (*componente userview*), pois representam entradas importantes para outras áreas do desenvolvimento. É preciso garantir, ainda, que estes casos de uso criados representem de maneira clara os requisitos do sistema. Logo, para escrita desses casos de uso é utilizada uma linguagem CNL (*Controlled Natural Language*). Esta linguagem representa um subconjunto do inglês relevante para o domínio da aplicação. Com a CNL é possível evitar a inclusão de sentenças ambíguas na definição dos casos de uso, resultando em um documento com maior qualidade e evitando problemas na geração do modelo formal.

Cada sentença do caso de uso é traduzida como um evento CSP. Cada caso de uso define parte da especificação formal do sistema. A presença de fluxo alternativo e de exceção no caso de uso é capturado pelo CSP com o operador de escolha externa, permitindo a combinação de processos. São gerados dois modelos, o modelo que representa a visão do usuário (*userviewmodel*) e outro que representa a visão do sistema (*componentviewmodel*).

Baseado nos modelos gerados, a relação entre casos de uso a nível de usuário e casos de uso a nível de componentes é estabelecida pelo mapeamento do modelo mais abstrato para o mais concreto. Esta relação é realizada para provar que o modelo do sistema (componentes) é um refinamento do modelo do usuário. Além de habilitar a relação entre as

duas visões (usuário e componentes), permite também a definição de múltiplos níveis de abstração.

### 3.1.1. Casos de Uso e CNL

Especificações de casos de uso capturam o comportamento do sistema em diversos níveis de abstração. Portanto, dependendo da necessidade do desenvolvedor, casos de uso são criados para diferentes propósitos. No contexto do projeto Cin-Test, são utilizados dois *templates* para especificação de casos de uso a nível de usuário e de componentes. Ambos os *templates* definem fluxo de execução que determina a interação entre usuário e sistema. A CNL, é usada para escrever os passos habilitando validações e transformações através de ferramentas.

**Caso de uso a nível de usuário**

Casos de uso a nível de usuário especificam comportamento do sistema quando um único usuário executa o mesmo. Ele especifica operações do usuário e respostas esperadas do sistema. A figura abaixo apresenta um caso de uso exemplo. Os tópicos seguintes explicam o que cada campo do *template* significa e como ele deve ser preenchido. Um exemplo de caso de uso a nível de usuário é mostrado na figura 9.

- **Feature**

Casos de uso são agrupados inicialmente por features (funcionalidades). Este agrupamento é conveniente para propósitos de organização; não é obrigatório para o contexto da aplicação.

- **RelatedRequirement(s)**

A lista de requisitos relacionados é usada para propósitos de rastreamento. Quando um requisito sofre uma mudança, é possível saber quais casos de uso são afetados pela mudança.

- **Description**

Uma descrição simples do propósito do caso de uso.

- **ExecutionFlow**

Um caso de uso especifica diferentes cenários, dependendo da entrada do usuário e suas ações. Logo, cada fluxo de execução representa um possível caminho que o caso de uso por percorrer. Abaixo estão descritas cada parte de um fluxo de execução.

- **Step**

Cada passo do caso de uso é representado pela tripla: ação do usuário (*useraction*), estado do sistema (*system state*) e resposta do sistema (*system response*). O campo *useraction*descreve uma operação realizada pelo usuário. O campo *system state*representa a atual configuração do sistema imediatamente antes da execução da ação do usuário. Por fim, o campo *system response*é a descrição do resultado da operação após a ação do usuário ocorrer, baseado no atual estado do sistema.

- **FlowTypes**

Fluxos de execução são categorizados em três tipos: fluxo principal (*mainflow*), fluxo alternativo (*alternativeflow*) e fluxo de exceção (*exception flow*). O primeiro representa o caminho feliz do caso de uso, com a execução de todos os passos funcionando como esperado. O segundo representa uma escolha, que pode ou não envolver o usuário. Durante o fluxo principal da aplicação é possível realizar outras operações, e quando isso acontece, o fluxo alternativo é seguindo pela aplicação. O terceiro e último tipo de fluxo representa

cenários de erro causados por dados de entrada inválidos ou estados críticos do sistema.

Fluxos alternativos e de exceção estão intimamente ligados à escolhas do usuário e condições do estado do sistema. O último pode fazer o sistema responder de maneira diferente à mesma entrada do usuário, em diferentes momentos.

- **Referências entre fluxos**

Como visto no tópico anterior, existem situações que o usuário ou até mesmo o sistema podem escolher entre dois caminhos diferentes. Neste caso, como visto, faz-se necessária a definição de diversos fluxos, cada um contendo início e fim. Os estados iniciais dos fluxos são representados, no *template*, por "**FROM_STEPS**" e o estado final, representado por "**TO_STEP**". Um caso de uso pode ter mais de um estado inicial, indicando que o mesmo foi disparado de um outro caso de uso. Diferentemente, o estado final do caso de uso recebe apenas um valor.

No fluxo principal, quando o estado inicial do caso de uso é definido por **START**, significa que o mesmo pode ser executado independente de qualquer outro caso de uso. Quando o estado inicial referencia campos de outros casos de uso, significa que o mesmo é iniciado após uma sequencia de ações do caso de uso referenciado. Sobre o estado final, quando é preenchido com **END** significa que o caso de uso terminou de maneira correta e que o usuário pode executar outro caso de uso agora. Estes dois campos são essenciais para definir a navegação da aplicação, permitindo ainda o reuso de outros fluxos.

**UC_02** - Incoming message moved to the Important Messages folder

**Related requirement(s):** REQ_1302, REQ_1326
**Description:** User accepts an incoming message and moves it to the Important Messages folder.

**Main Flow** **From Steps:** START **To Step:** END

| Step Id | User Action | System State | System Response |
|---|---|---|---|
| 1M | Read incoming message. | | Message content is displayed. |
| 2M | Open the menu. | "Important Messages" feature is on. | "Move to Important Messages" option is displayed. |
| 3M | Select "Move to Important Messages" option. | Message storage is not full. | "Message moved to Important Messages folder" is displayed. |
| 4M | Wait for at most 2 seconds. | | The next message is highlighted. |

**Exception Flow** **From Steps:** 2M **To Step:** END

| Step Id | User Action | System State | System Response |
|---|---|---|---|
| 1E | Select "Move to Important Messages" option. | Message storage is full. | "Memory required" dialog is displayed. |
| 2E | Confirm memory information dialog. | | Message content is displayed. |

Figura 9 - Caso de uso a nível de usuário

## Caso de uso a nível de componentes

Estes casos de uso especificam o comportamento do sistema na interação do usuário com os componentes do sistema. Neste nível, o sistema é decomposto em componentes que, concorrentemente, processam a requisição do usuário e se comunicam entre si. A figura abaixo mostra um caso de uso à nível de componentes.

No exemplo mostrado na figura 10, os componentes **MESSAGE STORAGE APP** e **DISPLAY APP** são responsáveis por guardar e mostrar as mensagens para o usuário, respectivamente. Estes componentes definem o nível arquitetural de abstração do caso de uso à nível de usuário mostrado na figura anterior. Em outras palavras, para cada caso de uso a nível de usuário existirá um caso de uso a nível de componentes, representando o refinamento do caso de uso anterior. No nível de componentes, os passos do caso de uso são decompostos em troca de mensagens.

Normalmente, casos de uso descrevem apenas o comportamento do sistema sem descrever seu comportamento interno. Os casos de uso a nível de componentes quebram essa convenção, e atualmente

são usados para detalhar os casos de uso a nível de usuário, que segue a idéia tradicional de caso de uso.

Na visão de componentes é necessário definir o componente que está invocando a ação e o que está provendo o serviço. Isto representa o processo de troca de mensagens composto por *sender*, *recievere* *message*. O usuário é atualmente visto como um componente, que pode enviar e receber mensagens de outros componentes. Um componente pode ainda enviar mensagem para si mesmo. Esta possibilidade em particular permite a definição de cenários concorrentes, que representa um requisito não funcional. Então, componentes podem trocar mensagens e compartilhar recursos, o que não é possível em modelos regulares de casos de uso.

**Main Flow**
**From Steps:** START
**To Step:** END

| Step Id | Sender | Message | System State | Receiver |
|---|---|---|---|---|
| 1M | User | Read incoming message. | | Message App |
| 2M | Message App | Open incoming message. | | Message Viewer |
| 3M | User | Open the Menu. | | Message App |
| 4M | Message App | Display Menu. | "Important Messages" feature is on. | Menu Controller |
| 5M | Menu Controller | "Move to Important Messages" option is displayed. | | User |
| 6M | User | Select the "Move to Hot Messages" option. | | Message App |
| 7M | Message App | "Move to Important Messages" option. | | Menu Controller |
| 8M | Menu Controller | Save message at "Important Messages" folder. | Message storage is not full. | Message Storage App |
| 9M | Message Storage App | "Message moved to Important Messages folder" is displayed. | | User |
| 10M | User | Wait for at most 2 seconds. | | User |
| 11M | Message App | The next inbox message is highlighted. | | List App |
| 12M | List App | Available message is selected. | | User |

**Exception Flow**
**From Steps:** 7M
**To Step:** END

| Step Id | Sender | Message | System Response | Receiver |
|---|---|---|---|---|
| 1E | Menu Controller | Save message at "Important Message" folder. | Message storage is full. | Message Storage App |
| 2E | Message Storage App | "Memory required" message is displayed. | | Display App |
| 3E | User | Confirm memory information dialog. | | Message App |
| 4E | Message App | Message content is displayed. | | User |

Figura 10 - Caso de uso a nível de componente

### 3.1.2. Geração do modelo CSP

**Modelo a nível de usuário**

Cada passo do fluxo de execução do caso de uso é transformado em um processo CSP. O nome deste processo é definido pela composição do identificador do passo e do identificador do caso de uso, formando uma chave única em todos os passos de todos os casos de uso do sistema. O corpo contém eventos de controle (*steps*, *conditions*, *expectedresults*) que delimitam os eventos gerados pela ação do usuário, estado do sistema e resposta do sistema, presentes no caso de uso à nível de usuário. Como já foi explicado, cada fluxo possui um campo **FROM_STEP** e um campo **TO_STEP** que determinam quando os fluxos começam e terminam. Eles deve referenciar um passo de um outro caso de uso ou as palavras-chave **START** e **END**.

A figura abaixo representa o CSP gerado para o caso de uso. Ele contém o processo **System**, que é o processo principal, e outros processo que se referem a fluxos de outros casos de uso. O processo **System** se refere ao processo **UC_02_1M** e outro fluxo de execução com o campo **FROM_STEP** contendo a palavra chave **START**. O processo **UC_02_2M** é definido como uma escolha externa do CSP entre o restante da execução do fluxo principal, o processo **UC_02_3M**, e o fluxo de exceção, **UC_02_1E**. O processo **UC_02_4M** finaliza com o processo **SKIP**, uma vez que o campo **TO_STEP** está preenchido com **END**.

## Modelo a nível de componentes

O modelo a nível de componentes é um pouco diferente do modelo à nível de usuário. Os canais dos componentes contém informação sobre os componentes envolvidos na troca de mensagens e seus nomes acrescidos do sufixo **Comp**, tornando os alfabetos dos modelos a nível de componente e usuários diferentes. Os tipos de dados usados em ambos os níveis são os mesmos, visto que os casos de uso se referem a elementos do mesmo domínio.

Na figura abaixo, o processo mais acima representa o modelo à nível de componentes, que é definido por execução paralela dos componentes do sistema, incluindo o usuário. Eles são compostos par-a-par usando o operador de paralelismo de CSP ( **[]** ). Cada componente aceita um conjunto de eventos para sincronização; **UserChannels** e **MessageAppChannels** são exemplos de elementos dos alfabetos usados na composição.

A figura abaixo mostra parte do processo **USER_P** de um dos casos de uso. Eventos **readComp.USER.MESSAGE_APP** e **isstateComp.MENU_CONTROLER.USER** são exemplos de comunicação entre componentes de usuários e de sistema. De maneira similar ao nível do usuário, se existem fluxos alternativos ou de exceção o operador de escolha externa é usado para capturar as alternativas. Na figura, o **USER_UC_02** contém um operador de escolha externa entre os processos **USER_UC_02_9M** e **USER_UC_02_3E** para denotar o fluxo de exceção.

```
USER_P =
  -- Scenario Case: Incoming message is moved to the Important Messages folder
  USER_UC_02
  [] ...

USER_UC_02 =
  -- Message: Read incoming message.
  readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE,{}) ->
  -- Message: Open the CSM.
  openComp.USER.MESSAGE_APP.DTOPE_MENU.(CSM_MENU_LIST,{}) ->
  -- Message: "Move to Important Messages" option is displayed.
  isstateComp.MENU_CONTROLLER.USER.DTISS_MENUITEM_STATEVALUE.
              (MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}).(DISPLAYED_VALUE,{}) ->
  -- Message: Select the "Move to Important Messages" option.
  selectComp.USER.MESSAGE_APP.DTSEL_MENUITEM.(MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}) ->
  (USER_UC_02_9M [] USER_UC_02_3E)

USER_UC_02_9M =
  -- Message: "Message moved to Important Message folder" is displayed.
  isstateComp.MESSAGE_STORAGE_APP.USER.DTISS_DIALOG_STATEVALUE.
             (MESSAGE_MOVED_TO_IMPORTANT_MESSAGE_FOLDER,{}}).(DISPLAYED_VALUE,{}) ->
  -- Message: Wait for at most 2 seconds.
  waitComp.USER.USER.DTWAI_ITEM.(SECOND,{AT_MOST.2}) ->
  -- Message: The next inbox message is highlighted.
  isstateComp.USER.LIST_APP.DTISS_SENDABLEITEM_STATEVALUE.
             (INBOX_MESSAGE,{NEXT}).(HIGHLIGHTED_VALUE,{})->
  -- Message: Available message is selected.
  isstateComp.LIST_APP.USER.DTISS_SENDABLEITEM_STATEVALUE.(MESSAGE,{}).(AVAILABLE_VALUE,{}) ->
  USER_P
```

Figura 11 - Exemplo de processo CSP usuário

### 3.1.3. Geração dos testes a partir do modelo CSP

Dado um modelo de teste S e uma propriedade $\Phi$, pode-se obter os traces de S que satisfaçam Φ (traces de S que conduzam a uma terminação com sucesso). Chama-se traces de cenários de testes, ts, quando $\Phi$ descreve critérios de seleção de testes. Um cenário de teste é o elemento central usado para construir casos de testes CSP. Esta seleção mostra como gerar cenários de teste como contra-exemplos das verificações de refinamento.

Considerando que temos um modelo S, que é alvo dos testes que queremos selecionar, então definimos S' como sendo o modelo S com a adição de eventos de marcação depois que satisfaçam Φ. A idéia é realizar verificações de refinamento na forma de $S' \sqsubseteq_T S$ que gerem cenários de testes como contra exemplos.

## 3.2 Pontos de melhoria identificados

Pela abordagem descrita na seção anterior percebe-se que o projeto cobre todo o ciclo de geração de teste, partindo dos documentos de requisitos e casos de uso, finalizando com a geração e seleção dos casos de teste. Porém, algumas destas etapas merecem um pouco mais de atenção. Na especificação dos casos de uso, nenhum dos *templates* dos documentos fornece uma maneira de reusar um caso de uso por inteiro ou mesmo parte do fluxo. Em modelos UML é possível reusar um caso de uso e até mesmo estender o comportamento do mesmo. A atual abordagem apenas compõe casos de uso, e não os relaciona.

Outro ponto observado é que ao contrário de [27], a pesquisa não incorpora dados nos modelos para geração de testes. Esses dados não estão presentes em nenhuma das etapas, documentos, modelos e geração de testes. A inclusão de dados potencializa o testes gerados, permitindo uma maior gama de cenários de testes, melhora a qualidade do documento de casos de uso e ainda aproxima o modelo formal da implementação.

# 4. Proposta de Extensão

Aqui é explicada a proposta de extensão do modelo utilizado pelo CIn-Test descrita na seção anterior. Esta proposta de extensão deve preservar todos os conceitos do modelo antigo, devido ao fato do CIn-Test já utilizar o processo. A extensão é totalmente conservativa, apenas adicionando elementos novos, para potencializar o modelo. Nas subseções seguintes são demonstradas as oportunidades de extensão e a extensão propriamente dita nas diversas etapas do processo.

## 4.1 Motivação

Os principais pontos de melhoria observados no processo foram:

- Proposta de relacionamento entre casos de uso
- Inclusão dos dados nos modelos gerados

Essas propostas de extensões foram observadas como necessidades de um subconjunto de integrantes do projeto Cin-Test e ainda com pesquisas na literatura de MBT sobre a utilização dessas abordagens.

Promovendo o relacionamento dos casos de uso, de maneira similar à encontrada nos diagramas de UML, provemos um maior reuso de casos de uso e reaproveitamento de fluxos de uma maneira clara e bem definida. Esta proposta de extensão nos relacionamentos dos casos de uso refletem em várias atividades do processo, que são descritas nas subseções seguintes.

Com a inclusão de dados nos modelos gerados, os testes gerados no final do ciclo são fortalecidos. Com os dados pode-se definir entradas de dados lpara realização de testes nas fronteiras da aplicação e até mesmo condições de execução dentro do sistema. Ainda é possível um maior detalhamento nos testes gerados, aumentando a precisão na cobertura do sistema. Assim como a primeira proposta, a inclusão de dados no modelo

reflete em várias atividades do processo, que serão descritas nas subseções seguintes.

## 4.2 Extensão na definição do Caso de Uso

Para inclusão das extensões propostas, a abordagem utilizada é a modificação no *template* do caso de uso, pois, com isso, todas as áreas subsequentes são adaptadas e corretamente afetadas.

### 4.2.1 Relacionamento entre casos de uso

Em diagramas de casos de uso presentes em UML, observamos que o relacionamento entre casos de uso presentes do diagrama são distribuídos em dois tipos básicos: inclusão e extensão. Esses tipos são tomados como base para a definição do relacionamento neste trabalho. A grosso modo, poderíamos apenas importar a abordagem de UML para os documentos de casos de uso. Porém, isso não é possível devido a uma característica que diferencia o modelo UML dos documentos de caso de uso. No diagrama, os casos de uso são representados de maneira atômica e indivisível. Isto não é verdade para os documentos apresentados neste pesquisa. Logo, é necessária a evolução dos conceitos. A figura abaixo mostra como ficaria confuso o modelo sem adaptações. Na figura, não é definido claramente se o UC1 é incluído após os passos do UC2 ou antes desses passos.

**UC2 – Moving Messages from Inbox to Important Messages**

**Includes** UC1

**Main Flow**

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1M | Select "Move to Important Messages" option. | Message storage has enough space. | "Message moved to Important Messages folder" is displayed. |

Figura 12 - Alternativa de inclusão de casos de uso

O mesmo acontece no processo de extensão, apresentado na figura abaixo. Admitindo que o UC2 tenha 5 passos, o *template* não deixa claro em que ponto o comportamento do UC2 é reaproveitado. Caso fosse definido que após o último passo do caso de uso estendido, neste caso UC2, seria feita a extensão; então, não seria possível reusar parte de um fluxo de UC2, somente o mesmo completo.

**UC3 – Delete Important Messages**

Extends UC2

**Main Flow**

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1M | Select important message(s). | | Message(s) are highlighted. |
| 2M | Clean up selected message(s). | | Clean is performed. |

Figura 13 - Alternativa para extensão de caso de uso

O ponto principal desta limitação é que ao importarmos os conceitos de UML simplesmente, incluiríamos ou estenderíamos todo o caso de uso. Isto representa uma limitação no mundo real, onde

observamos com frenquencia a necessidade de reuso de apenas parte do comportamento de um caso de uso. Logo, a proposta implica na definição do ponto de extensão e do ponto de inclusão de dentro do próprio caso de uso.

Esta limitação é superada com a definição e um ponto onde ocorre a inclusão e um ponto onde ocorre a extensão. Definidos estes pontos seria possível reusar parte de um fluxo de caso de uso, estendendo exatamente de um ponto em particular e seria possível formalizar a inclusão, eliminando a ambiguidade do local da inclusão. Neste cenário, ficamos com o seguinte *template*.

**UC2 – Moving Messages from Inbox to Important Messages**

**Includes** UC1@START
**Extension points** Clean up: 1A

**Main Flow**

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1M | Select "Move to Important Messages" option. | Message storage has enough space. | "Message moved to Important Messages folder" is displayed. |

**Alternative Flows**

From Step: START
To Step: START

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1A | Select "Move to Important Messages" option. | Message storage has not enough space. | "Message storage has not enough space" is displayed. "Clean Up request" is displayed for messages |

Figura 14 - Modelo final de inclusão e ponto de extensão no documento de caso de uso

Na figura, podemos observar a presença do mecanismo de localização da inclusão "@START", informando que o caso de uso é incluído antes da execução do passo START. É possível observar também o campo *extensionpoint*, necessário para identificar os

pontos do caso de uso que são extensíveis. Com essa identificação, a extensão se dá pelo uso do rótulo definido no ponto de extensão, neste caso "*Cleanup"*, eliminando a preocupação, por parte do extensor, com o identificador do passo exato onde ocorre a extensão. Com isso, o processo de extensão se comporta como mostrado na figura abaixo. O caso de uso UC3 estende o comportamento do UC2 exatamente no ponto "*Cleanup"*, que foi definido em UC2 como sendo o passo 1A do fluxo alternativo de UC2.

**UC3 – Delete Important Messages**

Extends (Important Messages Folder is not Empty, UC2@Clean up)

**Main Flow**

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1M | Select important message(s). | | Message(s) are highlighted. |
| 2M | Clean up selected message(s). | | Clean is performed. |

Figura 15 - Modelo final para extensão no documento de caso de uso

### 4.2.2 Inclusão de dados de entrada no processo

O relacionamento dos casos de uso promove reuso, mas não aumenta as possibilidades dos cenários de testes nem a precisão dos mesmos. Para este fim, tem-se a proposta de inclusão de dados no modelo, a fim de definir claramente os limites da aplicação, aumentando, consequentemente, o número de cenários de testes possíveis.

A inclusão de dados é feita mediante declaração de variáveis, tipos de dados, constantes, valores de entrada, valores de saída e guardas. As variáveis, tipos de dados e constantes declaradas podem estar no escopo de *feature*, dito global, como no escopo de caso de uso, dito local. Esses elementos são declarados antes do início do

comportamento da *feature* ou caso de uso. Os valores de entrada e os valores de saída são declarados localmente para processamento de condições (guardas). Essas guardas incrementam o campo System State do template do documento.

Como exemplo, temos a figura abaixo demonstrando a adição dos campos no template do caso de uso.

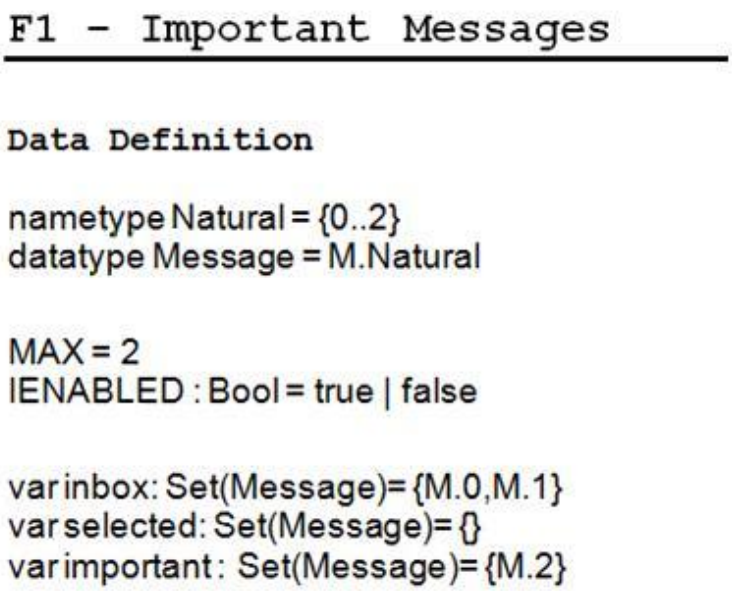



Figura 16 - Inclusão de dados na feature

Declaração de tipos globais (***Natural*** e ***Message***), declaração de constantes a serem utilizadas em todo o comportamento da *feature*, normalmente representando restrições dos casos de uso (***MAX***, ***IENABLE***) e declaração de variáveis (***inbox***, ***selected***, ***important***).

UC3 – Delete Important Messages

Extends (Interaction is enabled, UC2@Clean up)

**Data Definitions**

var selected: Set(Message) = {}

**Main Flow**

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1M | Select important message(s). **[ x: IP(important)– {{}} ]** | Important Messages Folder is not Empty. **[ #important > 0 ]** | Message(s) are highlighted. **[ selected:=x ]** |
| 2M | Clean up selected message(s). | | Clean is performed. **[ important := important– selected) ]** |

Figura 17 - Inclusão de dados no caso de uso

Declaração de caso de uso com acréscimo de variáveis locais (***selected***). Observa-se que a variável já foi definida globalmente (no nível de *feature*), porém, como foi redefinida; o valor da redefinição é válido apenas para dentro do caso de uso. Adição de valores de entrada no campo ***action***, onde podemos encontrar o uso de uma variável local ***x***. Inclusão de uma guarda no campo ***System State***, criando uma condição de realização baseada nos dados de entrada e não somente no estado do sistema. Inclusão de valores de saída no campo ***System Response***, aumentando o detalhamento da resposta do sistema ao usuário. Este ponto aproxima o modelo da implementação final.

Ao final da extensão, o template de caso de uso é expandido para suportar dados nos campos **dataDefinition**, onde serão incluídas as variáveis e constantes da feature e do próprio caso de uso, e nos campos **action**, contendo a inicialização das variáveis para execução do passo, **system state**, contendo a guarda do sistema, através da verificação dos valor das variáveis, e ainda, no campo **system response**, onde a memória (variáveis) possui seu valor atualizado e as variáveis de saída são mostradas ao usuário. Com isto, temos a

evolução do modelo de acordo com as figuras abaixo. As marcações na figura 19 representam o acréscimo de informações desenvolvido por este trabalho.

F1 – Important Messages

UC2 – Moving Messages from Inbox to Important Messages

**Main Flow**

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1M | Select "Move to Important Messages" option. | Message storage has enough space. | "Message moved to Important Messages folder" is displayed. |

**Alternative Flows**

From Step: START
To Step: START

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1A | Select "Move to Important Messages" option. | Message storage has not enough space. | "Message storage has not enough space" is displayed. "Clean Up request" is displayed for messages |

Figura 18 - Template de documento sem dados e sem relacionamentos

F1 - Important Messages

Data Definition

nametype Natural = {0..2}
datatype Message = M.Natural

MAX = 2
IENABLED : Bool = true | false

var inbox: Set(Message)= {M.0,M.1}
var selected: Set(Message)= {}
var important : Set(Message)= {M.2}

UC2 - Moving Messages from Inbox to Important Messages

**Includes** UC1@START
**Extension points** Clean up: 1A

Main Flow

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1M | Select "Move to Important Messages" option. | Message storage has enough space. [#(important ∪ selected) <= MAX ] | "Message moved to Important Messages folder" is displayed. [inbox:= inbox – selected, important:= important U selected)] |

Alternative Flows

From Step: START
To Step: START

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1A | Select "Move to Important Messages" option. | Message storage has not enough space. [#(important ∪ selected) > MAX ] | "Message storage has not enough space" is displayed. "Clean Up request" is displayed for [!(important ∪ selected) – MAX ] messages |

Figura 19 - Template do documento após a implementação da extensão

## 4.3 Extensão na CNL

Não foi realizada uma proposta formal sobre a extensão da CNL neste trabalho devido ao foco do mesmo. Por este motivo, encontram-se no caso de uso elementos que pertencem à sintaxe de CSP. Este ponto representa um fator negativo, tendo em vista que o responsável pela confecção dos

documentos de caso de uso deverá entender parte da sintaxe utilizada em CSP. A extensão realizada na CNL apenas permite que estes novos elementos estejam presentes no documento de caso de uso. Uma CNL mais suave para o usuário é proposta como tema de trabalhos futuros, ao final deste documento.

Com a formalização de uma CNL mais suave, promovendo a remoção de elementos CSP do documento de caso de uso, a confecção do mesmo diminui em complexidade e qualidade do documento aumenta consideravelmente. Elementos como ***var selected: Set(Message) = {}*** não devem estar presentes no modelo final.

## 4.4 Geração do modelo CSP

A abordagem de geração do modelo CSP foi mantida inalterada do modelo original. A extensão que foi proposta é conservativa, o que significa que modelos sem a presença de dados ou relacionamentos entre casos de uso ainda seguem o fluxo normal na geração de modelos e casos de testes.

A ferramenta é constituída de diversos plugins que utilizam a estrutura da IDE Eclipse PDE. A arquitetura da mesma é modular, estando presente em um destes módulos a leitura do documento e xml dos casos de uso e também a geração do modelo CSP. O desenvolvimento foi focado exclusivamente neste módulo afim de tornar a extensão conservativa.

Com a extensão do template do documento de casos de uso, devido à inclusão de dados e de relacionamentos, a aplicação também precisou sofrer alterações para correta geração do modelo. Para a inclusão dos relacionamentos entre casos de uso, foram criadas três classes entidades: ***UCExtension***, ***UCInclude*** e ***UCExtensionPoint***, representando extensão, inclusão e ponto de extensão, respectivamente. Estas classes foram agrupadas em uma entidade chamada ***UCRelations***, para fins de organização de arquitetura. A figura abaixo mostra atual arquitetura, lembrando que os relacionamentos são no nível de caso de uso, não sendo

possível relacionamentos entre *features*; sendo portanto, adicionado o atributo ***UCRelations*** na classe entidade ***UseCase***.

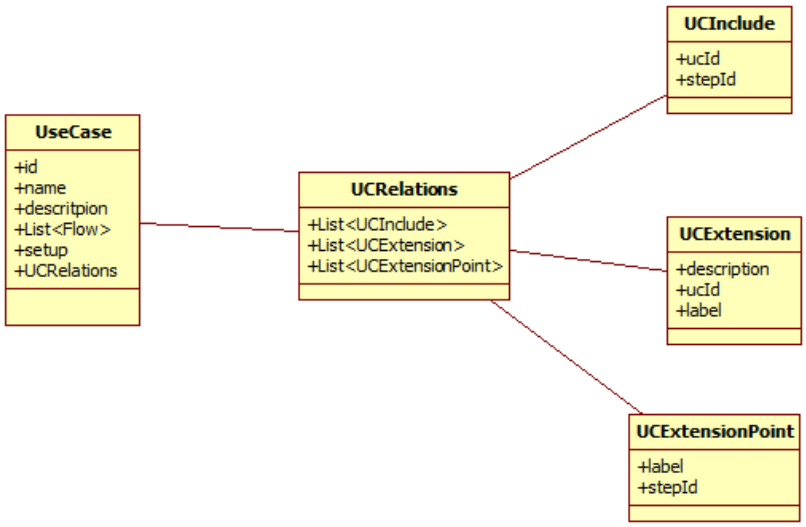



Figura 20 - Modelo UML da inclusão dos dados nos UseCases

Sobre a inclusão de dados, as modificações na arquitetura foram mais numerosas. Inicialmente, os tipos de dados foram divididos em dois grandes grupos: definição de dados e dados do passo.

Na definição dos dados, podemos encontrar elementos como tipo simples (***DataSimpleType***), que representa a definição simples de um tipo de dado, definição de constantes (***DataConstant***), que representa uma constante utilizada em todo o escopo válido e ainda declaração de variáveis, que são utilizadas, em sua maioria, nas guardas presentes nos passos do caso de uso, mas também podem ter seus valores alterados nos passos dos casos de uso. Ainda sobre a inclusão de dados, temos a definição de tipos internos (***DataInternalType***), que são os tipos complexos (não nativos) das variáveis definidas e, por fim, a definição do conceito de memória (***DataMemory***), que agrupa todas as variáveis definidas. Esta memória é associada à *feature* ou caso de uso, o que determina seu escopo. Na figura 19 podemos encontrar todos esses elementos. Sob o título de **Data Definition** encontramos, respectivamente, a definição de tipos simples (***nametype Natural = {0..2}***), a declaração de constantes (***MAX = 2***) e variáveis (***var inbox:Set(Message)={M.0, M.1}***). Por fim,

dentro da declaração da variável, encontramos a definição de um tipo complexo, que resulta em um tipo interno (***Set(Message)***).

Devido às suas características, os tipos simples e tipos internos herdam da mesma classe abstrata DataType. Ainda para fins de organização da arquitetura, todas essas classes que representam dados são agrupadas em uma classe entidade chamada **Data Definition**. Esta classe passa a ser um atributo de Feature e UseCase, como mostrado na figura abaixo. Todas as classes apresentadas na figura 21, com exceção da classe ***UseCase*** foram adicionadas no processo de desenvolvimento, para prevêr os dados.

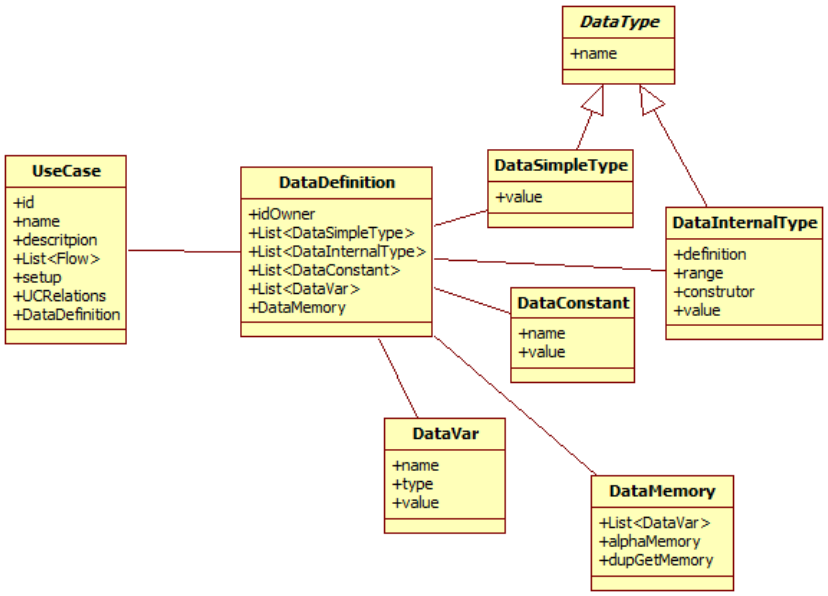



Figura 21 - Modelo UML da inclusão de dados nos UseCases

No grupo de dados do passo, encontramos uma divisão simples, realizada ainda no template do caso de uso. O uso de variáveis dentro do passo de um caso de uso é realizado nas colunas **action**, **system state** e **system response**. Na primeira coluna, são realizadas as obtenções dos valores das variáveis necessárias dentro do passo. Na segunda coluna, está presente a guarda do passo, onde encontramos uma expressão no formato de condição. Na terceira e última coluna, encontramos a atualização da memória, com os dados de saída.

Neste contexto, as informações sobre variáveis devem estar presentes na própria definição do fluxo, a classe ***FlowStep***, como mostrado na figura abaixo. A classe ***DataAction***, representa a utilização de variáveis locais (escopo de passo), a classe ***DataGuard*** representa a guarda do passo, presente na coluna ***System State*** e a classe ***DataOutput*** representa a saída dos valores e atualização da memória. As classes adicionadas ao projeto neste ponto foram: ***DataAction***, ***DataGuard*** e ***DataOutput***. A classe ***FlowStep*** é nativa do software.

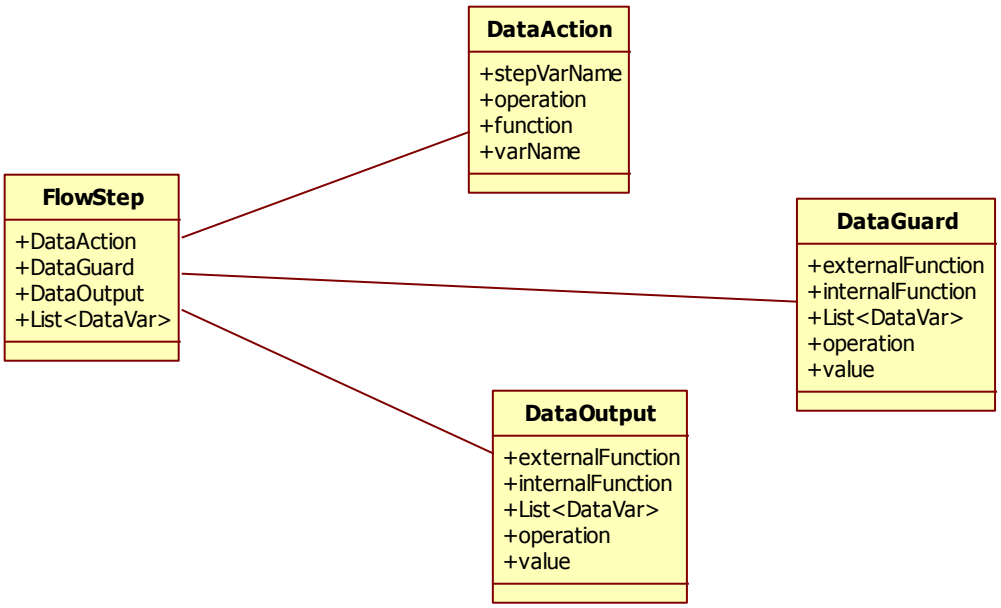



Figura 22 - Modelo UML da utilização de dados no passo (Step)

Pela ordem, a sequencia de ações realizadas na execução do passo são: inicialização de variáveis, verificação da guarda, realização da ação, atualização do estado do sistema e atualização da memória. Estas informações são transferidas nesta estrutura para o processo CSP gerado para o passo do caso de uso.

Dentro do contexto conservativo, o template para o modelo CSP é modificado de acordo com as figuras abaixo.

```
FID_UCID_FLOW = let

    FID_UCID_s#n =
      action#n -> condition#n -> response#n -> SKIP;
      continuations_s#n

within FID_UCID_START
```

Figura 23 - Template CSP para modelo sem dados e sem relacionamentos

```
FID_UCID_FLOW(label#k) = let

    FID_UCID_s#n = uc#i;
      action#n -> condition#n -> response#n -> SKIP;
      (label#k [] SKIP);
      continuations_s#n
    within FID_UCID_START

FID_UCID_Ext_uc#j_label#j(label#k) = c#j -> FID_UCID(label#k)
```

Figura 24 - Template CSP para modelo com relacionamento e sem dados

Esta primeira figura representa o *template* CSP anterior à extensão do relacionamento entre casos de uso realizado por este trabalho. Para suportar a inclusão de dados, os seguintes campos foram adicionados no template, como mostra a figura acima:

- **label#k:** representando o conjunto de *labels* que indicam os pontos de extensão do caso de uso;
- **uc#i:** representando a lista de inclusão de casos de uso importados (incluídos) pelo presente caso de uso;
- **(label#k [] SKIP):** representando a extensão do comportamento de outro caso de uso;
- A última linha representa a definição formal do ponto de extensão do presente caso de uso.

Como continuidade das contribuições apresentadas neste trabalho, a figura abaixo mostra as modificações sofridas no template após a inclusão dos dados.

```
FID_UCID_Ext_uc#j_label#j = get?... -> g#j -> FID_UCID

channel in_FID_UCID_s#n_iname : inputType
channel out_FID_UCID_s#n_oname: outType

FID_UCID_FLOW(label#k) = let

    FID_UCID_s#n = uc#i;
     get?... ->
     guard#n &
     action#n -> in_FID_UCID_s#n_iname?iname : range(iname) ->
     condition#n -> response#n -> outputs#n ->
     set!... -> mem_update -> SKIP;
     (label#k [] SKIP);
     continuations_s#n

within FID_UCID_START
```

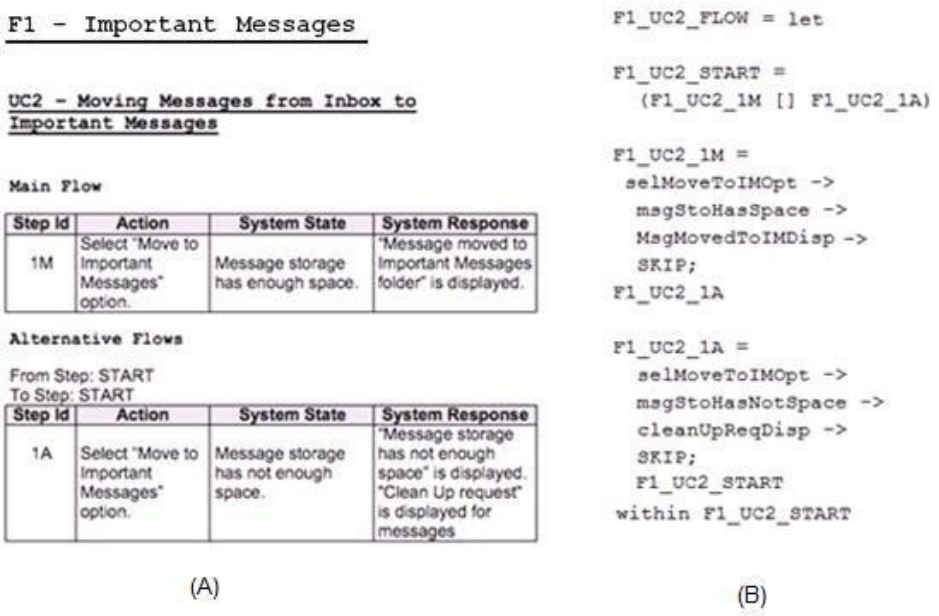
Figura25 - Template CSP completo

Os seguintes campos foram adicionados:

- **channel**: definindo os canais de dados de entrada e saída do caso de uso;
- **get?..:** obtendo todas as variáveis necessárias para a realização do caso de uso;
- **in_FID_UCID_s#n_iname?iname:** alterando as variáveis que representam os valores de entrada no caso de uso;
- **set!..:** alterando o valor das variáveis na memória após a realização do caso de uso.

Em um exemplo prático é feita a comparação entre as duas alternativas:

1 – Antigo modelo adotado no Cin-Test (sem dados e sem relacionamentos)



F1 – Important Messages

UC2 – Moving Messages from Inbox to Important Messages

Main Flow

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1M | Select "Move to Important Messages" option. | Message storage has enough space. | "Message moved to Important Messages folder" is displayed. |

Alternative Flows

From Step: START
To Step: START

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1A | Select "Move to Important Messages" option. | Message storage has not enough space. | "Message storage has not enough space" is displayed. "Clean Up request" is displayed for messages |

(A)

```
F1_UC2_FLOW = let

F1_UC2_START =
  (F1_UC2_1M [] F1_UC2_1A)

F1_UC2_1M =
 selMoveToIMOpt ->
  msgStoHasSpace ->
  MsgMovedToIMDisp ->
  SKIP;
F1_UC2_1A

F1_UC2_1A =
  selMoveToIMOpt ->
  msgStoHasNotSpace ->
  cleanUpReqDisp ->
  SKIP;
  F1_UC2_START
within F1_UC2_START
```

(B)

Figura 26 - Caso de uso e CSP sem dados

O modelo apresentado na figura 26 não conta com relacionamentos mais complexos, como inclusões e extensões. As possibilidades de relacionamentos entre os casos de uso são limitadas às cláusulas FROM STEP e TO STEP, que como mostrado no decorrer deste trabalho, restingem o poder dos relacionamentos. Não estão inclusos nenhum tipo de dado em nenhum escopo.

2 – Modelo estendido

Para uma melhor visualização do modelo resultante, faremos a análise do modelo em partes. A primeira parte, apresentada na figura 27, representa a inclusão de variáveis e constantes dentro de escopo de *feature*. Neste exemplo a declaração de variáveis é realizada apenas no escopo de *feature*, porém, o mesmo pode ser feito no escopo de caso de uso. No CSP gerado, apresentado na figura 27-B, observamos que as variáveis (***inbox***, ***selected*** e ***important***) são apresentadas no topo, como ***datatype var***. O segundo bloco do CSP representa a definição de um tipo de dado não nativo, nomeado anteriormente de tipo interno. Esse tipo representa um agrupamento (***Set***) de

mensagens. Para este tipo é definido um *range*, um construtor e um valor. Sob a marcação ***–types***, são apresentados os tipos simples, Natural e Messagem, e as constantes sob a marcação ***--constants***. A definição da memória é feita pelo mapeamento do nome da variável declarada anteriormente, associada com o seu valor de inicialização, observado no final da figura 27-B. Para inicialização das variáveis da memória, é utilizado o tipo interno (***Messages***) já definido.

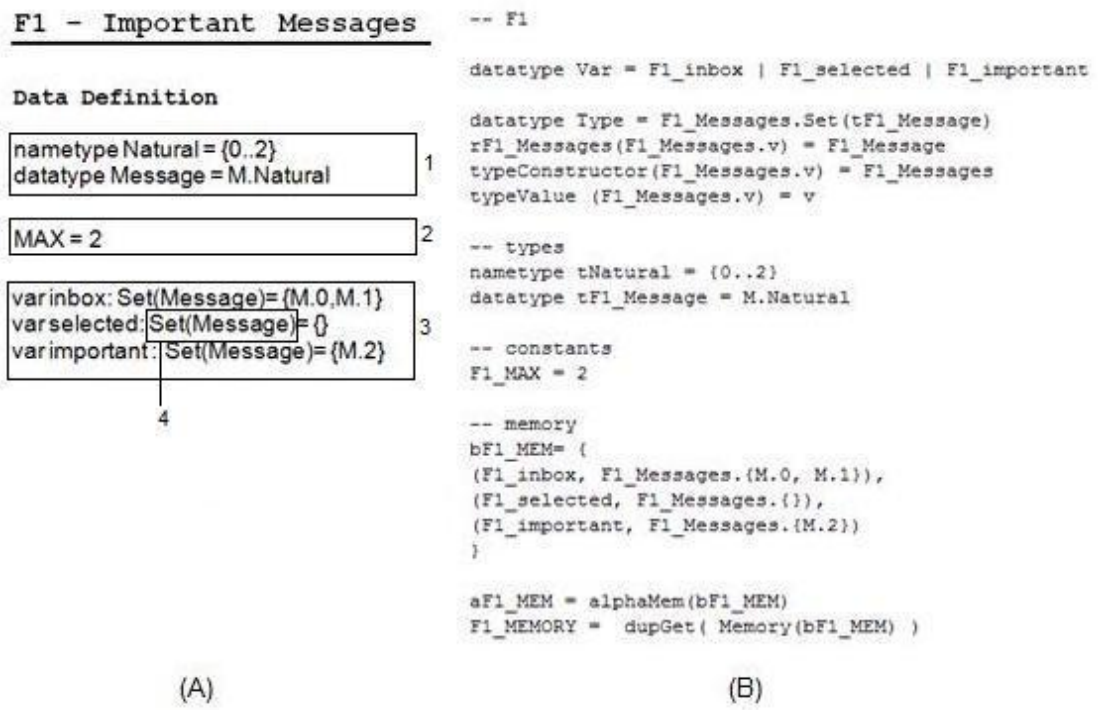



Figura 27 - Declaração de dados no escopo de feature

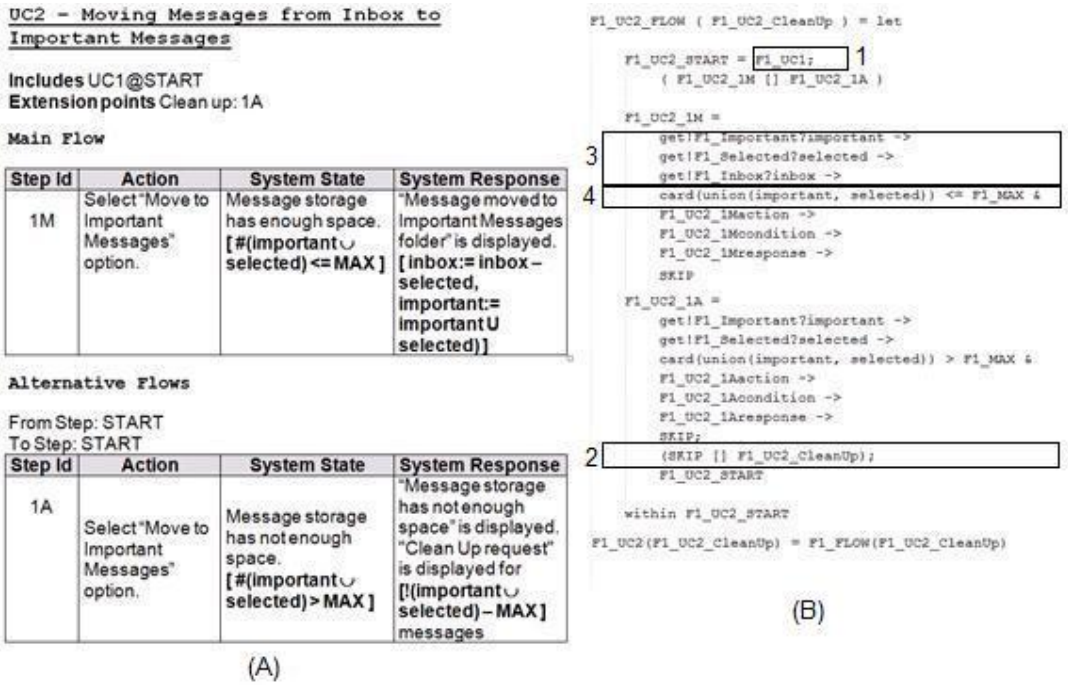
De acordo com a implementação descrita anteriormente, pode-se fazer uma rápida associação dos termos do caso de uso com as classes implementadas.

- Tipo Simples (***DataSimpleType***):
  - **nametype Natural = {0..2}** (marcação 1 da figura 27-A)
  - **datatype Message = M.Natural** (marcação 1 da figura 27-A)
- Constantes (***DataConstant***):
  - **MAX = 2** (marcação 2 da figura 27-A)
- Variável (***DataVar***):
  - **var inbox : Set(Message) = {M.0, M.1}** (marcação 3 da figura 27-A)

    - **var selected: Set(Message) = {}** (marcação 3 da figura 27-A)
    - **var important : Set(Message) = {M.2}** (marcação 3 da figura 27-A)
- Tipo Interno (***DataInternalType***):
    - **Set(Message)** (marcação 4 da figura 27-A)

Na segunda parte da nossa análise, podemos observar a inclusão dos dados e dos realcionamentos no template do caso de uso. No exemplo mostrado na figura 28, podemos observar a presença de ***Includes*** e ***ExtensionPoint*** para relacionamentos, respectivamente nas marcações 1 e 2 da figura 28-B.

Neste exemplo em particular, não existe dados presentes na coluna action. Esta coluna representa a obtenção de valores para as variáveis locais do passo (Step). Na segunda coluna, temos o estado do sistema, onde é incluída a guarda. Esta guarda é representada por uma expressão booleana, que ao ser avaliada como verdadeira, o passo é executado. No exemplo, no passo 1M, nota-se a presença da seguinte guarda: ***[#(important U selected) <= MAX],*** onde as variáveis ***important*** e ***selected***, e a constante ***MAX*** foram declaradas anteriormente, no escopo da feature. Para realização da guarda, é necessário a utilização dos valores das variáveis, obtidos através do conteúdo da marcação 3 na figura 28-B. Esta obtenção é realizada observando-se escopo local (caso de uso) e global (feature). Observa-se também a ausência do escopo de passo (Step) devido a ausência de dados na coluna ***action*** (***DataAction***). A realização da guarda é representada no conteúdo da marcação 4 da figura 28-B.



UC2 – Moving Messages from Inbox to Important Messages

**Includes** UC1@START
**Extension points** Clean up: 1A

**Main Flow**

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1M | Select "Move to Important Messages" option. | Message storage has enough space. **[#(important ∪ selected) <= MAX ]** | "Message moved to Important Messages folder" is displayed. **[ inbox:= inbox – selected, important:= important U selected)]** |

**Alternative Flows**

From Step: START
To Step: START

| Step Id | Action | System State | System Response |
|---|---|---|---|
| 1A | Select "Move to Important Messages" option. | Message storage has not enough space. **[#(important ∪ selected) > MAX ]** | "Message storage has not enough space" is displayed. "Clean Up request" is displayed for **[!(important ∪ selected) – MAX ]** messages |

(A)

```
F1_UC2_FLOW ( F1_UC2_CleanUp ) = let

    F1_UC2_START = F1_UC1;                    1
        ( F1_UC2_1M [] F1_UC2_1A )

    F1_UC2_1M =
        get!F1_Important?important ->
        get!F1_Selected?selected ->
        get!F1_Inbox?inbox ->
        card(union(important, selected)) <= F1_MAX &
        F1_UC2_1Maction ->
        F1_UC2_1Mcondition ->
        F1_UC2_1Mresponse ->
        SKIP
    F1_UC2_1A =
        get!F1_Important?important ->
        get!F1_Selected?selected ->
        card(union(important, selected)) > F1_MAX &
        F1_UC2_1Aaction ->
        F1_UC2_1Acondition ->
        F1_UC2_1Aresponse ->
        SKIP;
        (SKIP [] F1_UC2_CleanUp);
        F1_UC2_START

    within F1_UC2_START
F1_UC2(F1_UC2_CleanUp) = F1_FLOW(F1_UC2_CleanUp)
```

(B)

Figura 28 - Caso de Uso de CSP com dados

De acordo com a implementação a associação dos termos do caso de uso com as classes implementadas é:

- Relacionamentos (***UCRelations***)
  - Inclusão (***UCInclude***):
    - ***UC1@START*** (marcação 1 da figura 28-B)
  - Ponto de extensao (***UCExtensionPoint***):
    - ***CleanUp: 1A*** (marcação 2 da figura 28-B)
- Guarda (***DataGuard***):
  - ***[#(important U selected) <= MAX]*** (marcação 4 da figura 28-B)
  - ***[#(important U selected) > MAX]*** (marcação 4 da figura 28-B)
- Saída (***DataOutput***):
  - ***[!(important U selected) - MAX]*** (sem representação no CSP)

Neste último ponto, podemos observar a ausência da saída de dados no modelo CSP. Este ponto mostra a necessidade de continuidade do trabalho, que devido ao curto tempo, não foi completamente adicionado a parte de dados no modelo CSP. A inclusão de dados no passo também é outro ponto de melhoria, sendo necessária uma maior generalização do mesmo, para abranger mais casos. O presente resultado do desenvolvimento, abrange poucos casos.

No novo modelo, além das inclusões de dados e relacionamentos, podemos observar que cada item do passo foi transformado em um identificador único. Uma tabela de referência para mapeamento entre o identificador gerado e o texto completo do passo é salva no mesmo arquivo, para que o modelo seja legível para usuários humanos.

# 5. Conclusões e Trabalhos Futuros

## 5.1 Principais contribuições

O principal resultado deste trabalho é a inclusão de dados no modelo final, que reflete no aumento do poder dos casos de testes gerados. Com a inclusão dos dados, o número de possiblidades de tipos de testes aumentam assim como o poder do teste propriamente dito. A seleção de testes ganha um critério adicional. Anteriormente apenas o fluxo era considerado, agora, a carga e os limites da aplicação podem ser considerados na seleção dos testes finais.

Dentre os principais benefícios obtidos com este trabalho, podemos listar:

- **Reuso de comportamentos de casos de uso**

  Através do modelo de relacionamentos entre casos de uso, é permitido o reuso de todo, ou parte, do fluxo de outro caso de uso, bem como a composição do mesmo através de inclusões.

- **Aumento na qualidade do documento**

  Através do formalismo adotado, a qualidade do documento aumenta e consequentemente do modelo também. A proximidade entre o documento e a implementação aumenta bastante com a inclusão de dados para representar os limites da solução.

- **Aumento no detalhamento dos casos de uso**

  Com a definição de limites claros e adição de guardas, o comportamento do caso de uso é detalhado de uma maneira mais completa.

- **Geração de modelo com inclusão de dados**

  O modelo gerado através da extensão proposta conta com a inclusão de dados, o que aumenta o número de cenários de testes possíveis.

- **Aumento da proximidade do modelo com a implementação**

  A inclusão dos dados e dos relacionamentos entre casos de uso torna o modelo bastante próximo do comportamento real da aplicação.

- **Aumento das possibilidades de cenários de testes**

  Com a inclusão dos dados, o número de cenários de testes aumenta de maneira significativa. Com isso, os testes gerados tornam-se mais específicos e técnicas de seleção de testes podem ser aplicadas de maneira mais precisa.

## 5.2 Limitações da solução construídas

A solução proposta por este trabalho versa apenas sobre a primeira metade do fluxo de automação de testes, a descrição dos casos de uso e a geração do modelo formal, usando CSP. Não é abordado neste trabalho nenhuma estratégia para melhoria da geração ou seleção dos casos de testes. A melhoria proposta na geração dos testes é reflexo da melhoria obtida na geração do modelo; modelo com dados resultando em testes com dados.

## 5.3 Trabalhos futuros

Devido ao curto tempo para conclusão deste trabalho, foi concebida apenas a extensão da CNL para os relacionamentos entre os casos de uso, o mesmo não acontecendo para a inclusão de dados. No capítulo 4 deste trabalho pode-se observar que a inclusão dos dados nos casos de uso é feita através de sentenças que incluem instruções CSP (como, por exemplo, ***var inbox: Set(Message)= {M.0,M.1}***). Um trabalho futuro de relevante contribuição para a pesquisa como um todo é o desenvolvimento da extensão da CNL já utilizada para que a mesma seja capaz de prover a inclusão de dados de uma maneira simples (sem uso de CSP) para o responsável pela confecção dos casos de uso.

Um outro trabalho possível dentro da linha desta pesquisa é a conversão dos documentos de casos de uso em diagramas UML, em especial, diagramas de sequência estendidos. Sendo possível a conversão nos dois sentidos (UseCase – UML – UseCase) as possibilidades e facilidades na especificação dos documentos de caso de uso aumentam.

# Referências


[01]Ledru, Y., du Bousquet, L., Maury, P.B.O., Oriat, C., Potet, M., Potet, L.: **Test purposes: adapting the notion of specification to testing**.

[02]ISTQB - International Software Testing Qualifications Board: **Certified Tester – Foundation Level Syllabus, version 2007**

[03] Cabral, G., Sampaio, A.: **Formal Specification generation from requirement documents.**

[04] Cabral, G. Sampaio, A.: **Automated Formal Specification Genaration and Refinament from Requirement Documents**.

[05] Nogueira, S.C., Sampaio, A., Mota, A.: **Guided Test Generation from CSP Models.**

[06] Bouge, L., Choquet, N., Fribourg, L., Gaudel, M.C.: **Test Sets Generation from Algebric Specifications Using Logic Programs.**

[07] Hong, H.S., Kwon, Y.R., Cha, S.D.: **Testing of Object-Oriented Programs Based on Finite State Machines.**

[08] Kung, D., Suchack, N., Gao, J., Hsia, P., Toyoshima, Y., Chen, C.: **On Object State Testing**

[09] McGregor, J.D., Dyer, D.M.: **Selecting Functional Test Case for Object-Oriented Software.**

[10] Turner, C.D., Robson, D.J.: **The State-based Testing of Object-Oriented Programs.**

[11] Belinfant, A., Feenstra, J., de Vries,G.R., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heenrik, L.: **Formal Test Automation: A Simple Experiment**

[12] Cavalcanti, A., Gaudel, M.C.: **Testing from Refinament in CSP**

[13] Braga, A.B.: **Automatização de um conjunto completo de transformações para UML-RT**

[14] El-Far, I.K., Whittaker, J.A.: **Model-based Software Testing**

[15] Friedman, G., Hartman, A., Nagin, K., Shiran, T.: **Projected State Machine Coverage for Software Testing**

[16] Nogueira, S.C.: **Geração Automática de Casos de Test CSP Orientada por Propósitos.**

[17] Tretmans, J.: **Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation**

[18] Lee, D., Yannakakis, M.: **Principles and Methods of Testing Finite State Machines**

[19] Schwitter, R., Ljungberg, A., Hood D..: **ECOLE - a look-ahead editor for a controlled language.**

[20] Fuchs, N., Schwertel, U., Schwitter, R.:**Attempto Controlled English - not just another logic specification language.**

[21] Holt, A.: **Formal verification with natural language specifications: guidelines, experiments and lessons so far.**

[22] Hoare, C.A.R.:**Communicationg Sequential Processes.**

[23] Roscoe, A.W., Hoare, C.A.R., Bird, R.: **The Theory and Pratice of Concurrency**

[24] Bernard, E., Bouquet, F., Charbonnier, A., Legeard, B., Peureux, F., Utting, M., Torreborre, E.:**Model-based Testing from UML Models**

[25] Kim, Y.G., Hong, H.S., Cho, S.M., Bae, D.H., Cha, S.D.: **Test Cases Generation from UML State Diagrams**

[26] Hartman, A., Nagin, K.: **The AGEDIS Tools for Model Based Testing**

[27] Kahsai, T., Roggenbach, M., Schlingloff, B.H.: **Specification-based testing for refinament**