Capítulo 6

Conclusão

Neste último capítulo, apresentaremos um resumo do trabalho realizado, as contribuições, e propostas de trabalhos futuros.

6.1 Resumo e Contribuições

As vantagens da aplicação de métodos formais têm sido cada vez mais evidenciadas no desenvolvimento de software. Na maioria dos países europeus, por exemplo, a aplicação de tais métodos a sistemas classificados como críticos, ou seja, aqueles que envolvem riscos de vida ou grande valor material, é exigida legalmente.

Neste trabalho, foi proposta uma estratégia de modelagem para desenvolvimento formal de sistemas que integra algumas técnicas formais. Esta estratégia consiste de três etapas. Na primeira, são utilizados diagramas de colaboraçao em UML-RT para expressar graficamente a arquitetura do sistema. Na segunda, é feita uma tradução formal dos diagramas em UML-RT para processos em CSP-OZ. E na última, é realizada uma tradução dos processos em CSP-OZ para CSP_M, usando a técnica de verificação de modelos (estendida) desenvolvida para CSP-OZ, em conjunto com o padrão de projeto que definimos. A extensão da estratégia para incluir a etapa de geração de código é discutida como tópico para trabalho futuro.

A estratégia de modelagem foi aplicada a um subconjunto de um sistema de prontuário eletrônico complexo e real, que deve ser classificado como crítico, pois envolve informações de pacientes, comunicação entre médicos, e controle de qualidade sobre os cuidados clínicos.

O sistema de prontuário eletrônico é baseado em um modelo orientado a objetos, o GEHR. Para verificar formalmente suas propriedades foi usada a técnica de verificação de modelos para CSP-OZ. Como essa técnica não contemplava o mecanismo de herança, foi proposto um padrão de projeto que permite representar conceitos de orientação a objetos como processos em CSP, utilizando uma álgebra de processos.

Para atender às necessidades do padrão, foi necessário realizar algumas adaptações na função que traduz a parte de Z de um processo e a coloca em paralelo com a parte de CSP.

Inicialmente, o padrão de projeto foi aplicado em um sistema bancário, o qual é simples, mas explora vários conceitos de orientação a objetos. Esta aplicação foi, primeiramente, especificada formalmente em CSP-OZ e, em seguida, traduzida de CSP-OZ para CSP_M, utilizando a técnica de verificação de modelos adaptada, descrita na Seção 3.5, juntamente com o padrão apresentado na Seção 4.3. A especificação do sistema bancário mostrou que nossa abordagem é capaz de modelar e analisar propriedades, em particular a ausência de *deadlock*, de sistemas simples baseados em orientação a objetos e concorrência, que utilizam álgebra de processos.

Poucos problemas foram encontrados durante a modelagem. O principal foi a limitação tecnológica, a qual não permitiu fazer testes mais complexos. Além disso, a ferramenta FDR possui uma interface pobre, pois as mensagens de erro são muito genéricas e o depurador não auxilia na detecção do problema (falta de usabilidade).

Alguns pontos devem ser destacados como resultados deste estudo científico. A principal contribuição da dissertação é a criação de um padrão de projeto em CSP que complementa a técnica de verificação de modelos proposta por Fischer [30], pois através dele é possível representar os conceitos de orientação a objetos em CSP, independentemente do uso particular feito nesta dissertação. Os principais conceitos de orientação a objetos considerados pela tradução são os seguintes:

1. herança: classes e subclasses;

2. variáveis com dois papéis: variáveis ora como valor ora como processo;

3. manipulação de processos: criação e remoção de processos dinamicamente;

4. ligações dinâmicas: escolha do objeto certo, de acordo com alguns critérios definidos na Seção 4.2, para a execução de uma determinada operação (tipicamente redefinida);

5. declarações polimórficas: uso de variáveis polimórficas;

6. redefinição de operações: alteração do corpo de operações herdadas, mantendo a interface;

Devemos esclarecer que o padrão ainda pode ser evoluido para incluir herança múltipla. Além disso, não fizemos nenhum teste utilizando mais de um nível de herança, devido às limitações da ferramenta utilizada (FDR).

Até o momento, não temos conhecimento de alguma abordagem para modelar e analisar sistemas orientados a objetos com concorrência em termos de álgebra de processos. Encontramos apenas duas estratégias de verificação de modelos orientadas a objetos, as quais não tratam herança: a primeira, proposta por Fischer e utilizada neste trabalho, que permite verificação automática de modelos em CSP-OZ. A segunda, proposta por Smith [48], permite verificação automática de modelos em Object-Z. As duas estratégias utilizam uma função que traduz a parte referente a Object-Z em processo. A principal distinção é que a segunda *não trata concorrência nem considera divergência de processos.* Segue a definição da função de tradução utilizada no trabalho de Smith:

```
OzSemantics(Ops,in,out,enable,effect,init,event) =
let
  OZ_PART =
     [] op:Ops @ enable(op)(s) & [] i:in(op) @
     not empty(effect(op)(s,i) &
     ([](o,s'): effect(op)(s,i) @ event(op,i,o) -> OZ_PART(s'))
  OZ_MAIN = [] s:init @ OZ_PART(s)
within OZ_MAIN
```

O processo `OZ_PART` recebe como parâmetro o estado corrente s do objeto. Este processo oferece uma escolha externa sobre todas as operações habilitadas. Os parâmetros de entrada são escolhidos pelo ambiente, através da escolha externa sobre `in(op)`. De acordo com a semântica de bloqueio de Object-Z, se `effect(op)(s,i)` for igual ao conjunto vazio, significa que o processo entra em *deadlock*. A alteração também requer o uso do operador escolha externa sobre todos os possíveis valores iniciais de `OZ_PART`, valores de saídas e estados após a execução da operação `op`.

A função `event(op,i,o)` traduz os parâmetros `op`, `i` e `o` em evento CSP válido. Ela é descrita da seguinte forma:

```
event(op,i,o) = (if i == {}
                 then (if o == {}
                       then op
                       else op.o)
```

```
else if o == {}
     then op.i
   else op.i.o)
```

A função `OzSemantics` impõe a seguinte restrição: um evento pode não ocorrer, mas só ocorre se for garantido o seu sucesso. Se nenhum evento puder ocorrer, acontece o *deadlock*. O objetivo deste trabalho de Smith foi criar uma técnica de verificação de modelos que consegue um melhor desempenho que as outras que utilizam a ferramenta FDR.

No decorrer da elaboração do padrão, vislumbramos duas outras abordagens que também consideram herança e que suprem as deficiências da estratégia de Fisher [30]. O fato de escolhermos a primeira abordagem como padrão não inviabiliza a utilização das duas outras. Como já dito, todas as três abordagens geram sobrecarga no processo final em CSP_M, devido à inserção de eventos extras. Entretanto, o benefício de implementar especificações orientadas a objetos em CSP parece compensar a existência de sobrecarga. As três abordagens foram apresentadas, graficamente, através de diagramas de seqüência em UML, que forneceu uma visão geral do comportamento das mesmas.

Como comentado na Seção 4.6, na área de semântica de linguagens de programação orientadas a objetos, existem trabalhos [12, 88], que utilizam álgebra de processos para definir a semântica da linguagem. Na área de processo de desenvolvimento de software, encontramos trabalhos [36, 73], que integram métodos formais.

6.2 Propostas de Trabalhos Futuros

A linguagem CSP-OZ permite herança múltipla, porque Object-Z possui este conceito. Porém, nem o padrão de projeto e nem as outras duas abordagens consideram herança múltipla. Um possível tópico de investigação é estender o padrão e possivelmente as outras abordagens para incorporá-la.

No padrão sugerido, a criação e utilização de objetos seguem a forma geral:

```
let
  procName = proc(objectType, entity)
  Flow = ...
within (procName [|{|...|}|] Flow)
```

Dependendo da quantidade de processos a serem criados, torna-se complexo manter o controle dessas criações, como aconteceu no nosso estudo de caso. Então, a utilização de uma ferramenta que possa fazer uma tradução automática, levando em consideração os elementos de orientação a objetos fornecidos pelo padrão, facilitaria muito a programação. Já existe uma ferramenta desenvolvida pelo Centro de Informática da UFPE, que faz a tradução de uma especificação em CSPZ para *scripts* de FDR [23]. Esta ferramenta pode ser estendida para incorporar orientação a objetos de acordo com o padrão de projeto.

Como já foi comentado na Seção 4.6, as três soluções foram apresentadas tanto no nível de código quanto em termos de diagramas de seqüência de UML, usando uma correspondência de um para um em uma abordagem pragmática. É importante formalizar a tradução dos diagramas de seqüência (UML) nos padrões de projeto em processos CSP-OZ, para não gerar qualquer erro na tradução.

Na tradução da classe Account de CSP-OZ para CSP_M (Seção 3.5), os conjuntos `Passw`, `Real` e `Number` receberam valores de forma ad-hoc. Como conseqüência, a especificação traduzida pode perder propriedades em relação a especificação original. É importante fazer uma reavaliação do mesmo exemplo utilizando a abordagem de verificação de modelos para sistemas infinitos.

Este trabalho limita a criação e utilização de uma instância de classe por vez. É necessário criar um mecanismo que permita que várias instâncias possam ser criadas e acessadas ao mesmo tempo.

Como o padrão utiliza semântica de cópia, o problema de mais de uma aplicação se referenciar o mesmo objeto passa a existir. Uma possível solução é ter uma estrutura que armazena os objetos (processos) já criados. Se houver uma demanda da aplicação de colocar um novo fluxo em paralelo com o objeto já criado, o padrão não deve criar uma nova cópia, mas sim controlar o acesso concorrente de alguma forma.

Existe um interesse real na formalização do sistema de prontuário eletrônico baseado no GEHR (Seção 1.1), que motiva a continuidade de sua especificação formal. Para completar o processo de desenvolvimento de software, pode-se fazer refinamentos sucessivos nesta especificação, utilizando, como base, por exemplo, a estratégia apresentada em [16], até alcançar classes em Java.

Apêndice A

Implementação do Sistema Bancário em CSP_M Utilizando o Padrão

Segue a apresentação da especificação do sistema bancário completa, utilizando o padrão de projeto. Neste sistema estamos utilizando os processos `BANK`, `ACCOUNT` e `SAVING`, que não utilizam métodos redefinidos.

A Função `FSeq` retorna um conjunto de seqüências, de tamanho s, com elementos do conjunto T. Seque a sua definição:

```
FSeq(T, 0) = {<>}
FSeq(T, 1) = union(FSeq(T, 0), {<z> | z<-T})
FSeq(T, s) = {z^z' | z<-FSeq(T, 1), z'<-FSeq(T, s-1)}
```

Função que adiciona mais um elemento a tupla.

```
makeContext((a,b,c,d),e) = (a,b,c,d,e)
```

Função que recupera a tupla de variáveis de estado.

```
recoverTuple(AccCtxt.(x,y,z,w)) = (x,y,z,w)
recoverTuple(SavCtxt.(x,y,z,w,t)) = (x,y,z,w,t)
```

Função que retorna o elemento de um conjunto unitário.

```
pick({x}) = x
```

Função que verifica se tem conta na coleção de contas.

```
hasAcc(accounts)=
  not empty({AccCtxt.(a,b,c,d) | AccCtxt.(a,b,c,d)<-accounts})
```

Função que verifica se tem poupança na coleção de contas.

```
hasSav(accounts)=
  not empty({SavCtxt.(a,b,c,d,e) | SavCtxt.(a,b,c,d,e)<-accounts})
```

Função que procura as contas, na coleção de contas, que possuem identificadores que estão na lista de identificadores.

```
findAcc(setAcc,{}) = {}
findAcc(setAcc,accounts) =
  {AccCtxt.(a,b,c,d) | AccCtxt.(a,b,c,d) <- accounts,member(a,setAcc)}
```

Função que procura as poupanças, na coleção de contas, que possuem identificadores que estão na lista de identificadores.

```
findSav(setSav,{}) = {}
findSav(setSav,accounts) =
  {SavCtxt.(a,b,c,d,e) | SavCtxt.(a,b,c,d,e) <- accounts,member(a,setSav)}
```

Função que seleciona as contas do conjunto de contas.

```
selectAcc(accounts) =
  {AccCtxt.(a,b,c,d) | AccCtxt.(a,b,c,d) <- accounts}
```

Função que seleciona as poupanças do conjunto de contas.

```
selectSav(accounts) =
  {SavCtxt.(a,b,c,d,e) | SavCtxt.(a,b,c,d,e) <- accounts}
```

Função que seleciona os identificadores do conjunto de contas (setAcc).

```
idsAcc(setAcc) = {n | n <- Number, AccCtxt.(x,y,z,w) <- setAcc, n == x}
```

Função que seleciona os identificadores do conjunto de poupanças (setSav).

```
idsSav(setSav) = {n | n <- Number, SavCtxt.(x,y,z,w,t) <- setSav, n == x}
```

A função checkAccSav (invariante de `BANK`) não permite que se tenha duas contas ou duas poupanças ou uma conta e uma poupança com o mesmo identificador. Isto significa que não se pode ter objetos da classe e de suas subclasses com o mesmo número de identificação. Implementamos essa restrição utilizando uma lista de identificadores não repetidos (sAcc), que fornece o próximo número disponível para a criação de um novo objeto.

```
checkAccSav(accounts,<>)=
   card(accounts) > 0 and
   empty(inter(idsAcc(selectAcc(accounts)),idsSav(selectSav(accounts))))
checkAccSav(accounts,sAcc)=
 if hasAcc(accounts) and not hasSav(accounts)
 then (if empty(inter(findAcc(set(sAcc),accounts),accounts))
       then true
       else false)
 else if hasSav(accounts) and not hasAcc(accounts)
      then (if empty(inter(findSav(set(sAcc),accounts),accounts))
            then true
            else false)
      else
       (if hasAcc(accounts) and hasSav(accounts)
```

```
        then (empty(inter(findAcc(set(sAcc),accounts),accounts)) and
              empty(inter(findSav(set(sAcc),accounts),accounts)) and
              empty(inter(idsAcc(selectAcc(accounts)),
                          idsSav(selectSav(accounts)))))
        else true)
```

Função que representa o invariante de ACCOUNT. Ela garante que o saldo da conta e da poupança não pode ficar negativo.

```
checkAcc(b) = if b >= 0 then true else false
```

Função que recupera o primeiro elemento de objeto conta (AccCtxt.(x,y,z,w)) e poupança (SavCtxt.(x,y,z,w,t)).

```
firstFind(AccCtxt.(x,y,z,w)) = x
firstFind(SavCtxt.(x,y,z,w,t)) = x
```

A função Semantics traduz a parte de Z para processo e o coloca em paralelo com a parte de CSP.

```
Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event) =
 let
   DIV = DIV
   Z_PART(s) =
     [] op:Ops @ enable(op)(s) & [] i: in(op) @
     (if empty(effect(op)(s,i))
      then (|~| o:out(op) @ event(op,i,o) -> DIV)
      else if op == terminate
           then (|~| (o,s'):effect(op)(s,i) @ event(op,i,o) -> SKIP)
           else (|~| (o,s'):effect(op)(s,i) @ event(op,i,o) -> Z_PART(s')))

   Z_MAIN = |~| s:init @ Z_PART(s)

within (Z_MAIN [|{| op | op <- CSPOps |}|] main)\{| op | op <- LocOps |}
```

Função que traduz os parâmetros op, i, o em eventos CSP válidos.

```
event(op,i,o) =
   (if i == {}
    then (if o == {}
          then op
          else op.o)
    else if o == {}
         then op.i
         else op.i.o)
```

A função proc cria processos dinamicamente. Ela recebe como parâmetros o tipo do processo e os valores de inicialização das variáveis de estado.

```
proc(TBANK,(cc,(sa,ss))) = BANK(cc,sa,ss)
proc(TACCOUNT,(n,b,p,v)) = ACCOUNT(n,b,p,v)
proc(TSAVING,(n,b,p,v,r)) = SAVING(n,b,p,v,r)
```

Conjunto de mensagens.

```
datatype Message =  notOkAccount | insuficientBalance | withoutReference
```

Conjunto de tipos de Objetos.

```
datatype ObjType = TACCOUNT | TBANK | TSAVING
```

conjunto de identificadores de objetos

```
nametype Number ={2,3}
```

Conjunto de valores do tipo Real.

```
nametype Real = {0,1}
```

Conjunto de taxas.

```
nametype Ratio = {0}
```

Conjunto de senhas.

```
nametype Passw = {1}
```

Conjunto de possíveis valores para o estado de ACCOUNT.

```
nametype AccountContext =
 {(n,b,p,v) | n <- Number, b <- Real, p <- Passw, v <- Real}
```

Conjunto de possíveis valores para o estado de SAVING.

```
nametype SavingContext =
 {(n,b,p,v,r) |  n <- Number, b <- Real, p <- Passw, v <- Real, r <- Ratio}
```

Conjunto de contas e poupanças.

```
datatype ObjectContext = AccCtxt.AccountContext |
                         SavCtxt.SavingContext
```

Conjunto de contas.

```
nametype ObjectContextAcc = {AccCtxt.acc | acc <- AccountContext}
```

Conjunto de poupanças.

```
nametype ObjectContextSav = {SavCtxt.sav | sav <- SavingContext}
```

Declaração dos canais.

```
channel getObjInd:number
channel notGetIndObj
channel createAccount:Passw
channel update:ObjectContext
channel add:ObjectContext
channel depositB:Real
channel withdrawB:Real
channel found:ObjectContext
channel notFound
channel message:Message
channel getBalanceB
channel createSaving:Passw
channel deposit,withdraw:Real
channel getBalance:Real
channel getStateAcc: AccountContext
channel getStateSav: SavingContext
channel withdrawOk,withdrawNotOk
channel getMoney
channel interestB:Ratio
channel interest:Ratio
channel enterAcc:Number
channel enterSav:Number
channel terminate
channel depositDup:Real
channel getBalanceDup:Real
channel recoverState:Ratio
channel exit
```

O processo SYSTEM é formado apenas pelo processo BANK.

```
SYSTEM =  let
            bkState = ({},<2,3>)
            procBank = proc(TBANK,bkState)
          within (procBank)
```

O processo BANK é responsável pelas operações bancárias como por exemplo abrir conta e poupança, depositar, sacar e consultar saldo. Ele possui como parâmetros o conjunto contas (cts) e uma lista de identificadores (sqId).

```
BANK(cts,sqId) =
 let
```

A constante local Ops contém o conjunto de nomes de canais usados pelo processo referente a parte de Z.

```
    Ops = {createAccount, enterAcc, add, getObjInd,notGetIndObj,
           message,createSaving, enterSav, found, notFound, update}
```

A constante LocOps contém apenas os nomes dos canais locais.

```
LocOps = {add,update,getObjInd,notGetIndObj,found,NotFound}
```

Abre uma nova conta.

```
main = createAccount?p  ->
         (getObjInd?ind -> add.(AccCtxt.(ind,0,p,0)) -> main
          []
          notGetIndObj -> message?m -> main)

       []
```

Entra com o número da conta.

```
          enterAcc?an ->
```

Achou a conta. É necessário adicionar o conjunto ObjectContextAcc porque o canal found só pode receber objetos do tipo conta.

```
          (found?obj:ObjectContextAcc ->
            (let
```

Como o objeto conta se encontra no formato AccCtxt.(a,b,c,d), então é necessário recuperar a tupla (a,b,c,d) para criar o processo ACCOUNT.

```
              e = recoverTuple(obj)
```

Cria um processo do tipo ACCOUNT.

```
              procAcc = proc(TACCOUNT,e)
```

Fluxo do processo procAcc.

```
              Flow = depositB?v -> deposit.v -> Flow
                     []
                     withdrawB?v ->  withdraw.v ->
                         (withdrawOk -> getMoney -> Flow
                          []
                           withdrawNotOk -> message?m -> Flow)
                     []
                     getBalanceB ->  getBalance?b -> Flow
                     []
```

Atualiza a coleção de contas do banco.

```
                     exit -> getStateAcc?st2 ->
                       update.(AccCtxt.st2) -> SKIP
        within (procAcc
                [|{|deposit,withdraw,withdrawOk,withdrawNotOk,
                getBalance,getStateAcc,exit|}|] Flow);main)
       []
       notFound -> message?m -> main)
      []
```

Abre uma nova poupança.

```
createSaving?p ->
 (getObjInd?ind -> add.(SavCtxt.(ind,0,p,0,0)) -> main
  []
  notGetIndObj -> message?m -> main)

[]
```

Entra com o número da poupança.

```
enterSav?sn ->
```

Achou a poupança.

```
(found?obj:ObjectContextSav ->
   (let
     e = recoverTuple(obj)
```

Cria um processo do tipo poupança.

```
     procSav = proc(TSAVING,s)
```

Fluxo do processo procSav.

```
      Flow = depositB?v -> deposit.v -> Flow
             []
             withdrawB?v -> withdraw.v ->
               (withdrawOk -> getMoney -> Flow
                []
                withdrawNotOk -> message?m -> Flow)
             []
             getBalanceB ->  getBalance?b -> Flow
             []
             interestB?r -> interest.r -> Flow
             []
```

Atualiza a coleção de contas do banco.

```
             exit -> getStateSav?st1 ->
              update.(SavCtxt.st1) -> SKIP
 within (procSav
         [|{|deposit,withdraw,withdrawOk,withdrawNotOk,
         getBalance,interest,getStateSav,exit|}|] Flow);
         main)
[]
notFound -> message?m -> main)
```

Conjunto de eventos de sincronização dos processos main e Z_MAIN.

```
CSPOps = Ops
```

A constante SizeSeq define o tamanho máximo para a lista de identificadores de contas e poupanças.

```
SizeSeq = 2
```

O conjunto SeqIds é resultado da aplicação da função FSeq, a qual cria um conjunto de seqüências do tipo do primeiro argumento e cada elemento terá cardinalidade máxima dada pelo segundo argumento.

```
SeqIds = FSeq({2,3},SizeSeq)
```

O estado de BANK possui os seguintes elementos: accounts, conjunto de contas e de poupanças; msg, mensagem que BANK pode apresentar ao ambiente; seqId, seqüência de identificadores utilizados na criação dos objetos conta e poupança; type, tipo do objeto utilizado; num, número de dentificador de conta e de poupança. A função enable, responsável pela habilitação e desabilitação das operações, não considera as variáveis de entrada, apenas as de estado. Como certas operações precisavam ser habilitadas e desabilitadas, então tivemos que adicionar no estado os elementos type e num, os quais não são inerentes a classe.

A função checkAccSav, definida acima, é responsável por garantir que os números dos identificadores sejam únicos. Isto é, essa função caracteriza o invariante do processo BANK.

```
state = {(accounts,msg,seqId,type,num)|
          accounts <- Set(ObjectContext),
          msg <- Message, seqId <- SeqIds,
          type <- ObjType,num <- Number,
          checkAccSav(accounts,seqId)}
```

A inicialização da parte de Z do processo BANK é dada pela constante init.

```
init = {(accounts',msg',seqId',type',num') |
        (accounts',msg',seqId',type',num') <- state,
         accounts' == cts, seqId'== sqId,
         checkAccSav(accounts',seqId')}
```

A função in define o conjunto de elementos entrada.

```
in(createAccount) = Passw
in(enterAcc) = Number
in(getObjInd) = {{}}
in(notGetIndObj)= {{}}
in(add) = ObjectContext
in(found) = {{}}
in(update) = ObjectContext
in(notFound) = {{}}
in(message) = {{}}
in(createSaving) = Passw
in(enterSav) = Number
```

A função out define o conjunto de elementos saída.

```
out(createAccount) = {{}}
out(enterAcc) = {{}}
out(getObjInd) = Number
out(notGetIndObj) = {{}}
out(add) = {{}}
out(found) = ObjectContext
out(update) = {{}}
out(notFound) = {{}}
out(message) = Message
out(createSaving) = {{}}
out(enterSav) = {{}}
```

A função enable testa se a operação está habilitada para os valores do estado.

```
enable(createAccount)((accounts,msg,seqId,type,num)) =  true
enable(enterAcc)((accounts,msg,seqId,type,num)) = true
enable(getIndObj)((accounts,msg,seqId,type,num)) =  not null(seqId)
enable(notGetIndObj)((accounts,msg,seqId,type,num)) =  null(seqId)
enable(add)((accounts,msg,seqId,type,num)) = true
enable(found)((accounts,msg,seqId,type,num)) =
 if type == TACCOUNT
 then let accs = selectAcc(accounts)
      within   {e | e <- accs, firstFind(e) == num} != {}
 else if type == TSAVING
      then let savs = selectSav(accounts)
           within{e | e <- savs, firstFind(e) == num} != {}
      else false
enable(update)((accounts,msg,seqId,type,num)) = true
enable(notFound)((accounts,msg,seqId,type,num)) =
  if type == TACCOUNT
  then let accs = selectAcc(accounts)
       within {e | e <- accs, firstFind(e) == num} == {}
  else if type == TSAVING
       then let savs = selectSav(accounts)
            within{e | e <- savs, firstFind(e) == num} == {}
       else false
enable(message)((accounts,msg,seqId,type,num)) = true
enable(createSaving)((accounts,msg,seqId,type,num)) =  true
enable(enterSav)((accounts,msg,seqId,type,num)) =  true
```

A função effect executa a operação habilitada pela função enable.

```
effect(createAccount)((accounts,msg,seqId,type,num),p) =
  {({},(accounts',msg',seqId',type',num')) |
   (accounts',msg',seqId',type',num') <- state, num'== num,
   type'==type,accounts'== accounts, msg'== msg, seqId'==seqId,
```

```
    checkAccSav(accounts',seqId')}

 effect(enterAcc)((accounts,msg,seqId,type,num),n) =
  {({},(accounts',msg',seqId',type',num')) |
   (accounts',msg',seqId',type',num') <- state, num'== n,
   type'== TACCOUNT, accounts'== accounts, msg'== msg,
   seqId'==seqId, checkAccSav(accounts',seqId')}

 effect(getIndObj)((accounts,msg,seqId,type,num),_) =
   {(o,(accounts',msg',seqId',type',num')) |
    (accounts',msg',seqId',type',num') <- state, num'== num,
    type'==type, accounts'== accounts, msg'== msg, o <- Number,
    o == head(seqId),seqId'== tail(seqId),
    checkAccSav(accounts',seqId')}

 effect(notGetIndObj)((accounts,msg,seqId,type,num),_) =
    {(o,(accounts',msg',seqId',type',num')) |
    (accounts',msg',seqId',type',num') <- state, num'== num,
    type'==type,accounts'== accounts,msg'== withoutReference,
    seqId'== seqId,checkAccSav(accounts',seqId')}

 effect(add)((accounts,msg,seqId,type,num),c) =
   {({},(accounts',msg',seqId',type',num')) |
    (accounts',msg',seqId',type',num')<- state, type'==type,
    accounts'== union(accounts,{c}),msg'== msg,seqId'== seqId,
    num'== num,checkAccSav(accounts',seqId')}

effect(found)((accounts,msg,seqId,type,num),_) =
  {(o,(accounts',msg',seqId',type',num')) |
   (accounts',msg',seqId',type',num') <- state, type'==type,
   accounts'== accounts, msg'== msg,  o <- ObjectContext,
   o == pick({ e | e <- accounts, firstFind(e) == num}),
   seqId'== seqId, num'== num,
   checkAccSav(accounts',seqId')}

effect(update)((accounts,msg,seqId,type,num),cs) =
 {({},(accounts',msg',seqId',type',num')) |
  (accounts',msg',seqId',type',num') <- state, num'==num,
   accounts'==  union(diff({ e | e <- accounts, type'==type,
   firstFind(e) == firstFind(cs)},accounts),{cs}),
   msg'== msg,seqId'== seqId,
   checkAccSav(accounts',seqId')}

effect(notFound)((accounts,msg,seqId,type,num),_) =
   {({},(accounts',msg',seqId',type',num')) |
    (accounts',msg',seqId',type',num') <- state,
```

```
      accounts'== accounts, msg' == notOkAccount, seqId'== seqId,
      num'== num,type'==type,checkAccSav(accounts',seqId')}

   effect(message)((accounts,msg,seqId,type,num),_) =
      {(o,(accounts',msg',seqId',type',num')) |
       (accounts',msg',seqId',type',num') <- state,o <- Message,
        type'==type, accounts'== accounts,msg'== msg,o == msg',
        seqId'== seqId, num'== num,checkAccSav(accounts',seqId')}

   effect(createSaving)((accounts,msg,seqId,type,num),p) =
      {({},(accounts',msg',seqId',type',num')) |
       (accounts',msg',seqId',type',num') <- state, type'==type,
       accounts'== accounts, msg'== msg, seqId'==seqId,
       num'== num,checkAccSav(accounts',seqId')}

   effect(enterSav)((accounts,msg,seqId,type,num),n) =
     {({},(accounts',msg',seqId',type',num')) |
      (accounts',msg',seqId',type',num') <- state, num'== n,
      type'== TSAVING, accounts'== accounts, msg'== msg,
      seqId'==seqId, checkAccSav(accounts',seqId')}
```

A função `Semantics` faz a tradução da parte de Z para processo e o coloca em paralelo com a parte de CSP.

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

O processo `ACCOUNT` modela a conta corrente.

```
ACCOUNT(nu,bl,ps,va) =
 let
```

Conjunto de operações usadas pelo processo da parte de Z.

```
    Ops = {deposit,getStateAcc,withdraw,withdrawOk,withdrawNotOk,
           getBalance,terminate,getBalanceDup,depositDup}
```

A constante `LocOps` contém apenas os nomes dos canais locais.

```
    LocOps = {}
```

O processo `main` possui dois eventos de depósito (`deposit?v` e `depositDup?b`) e dois de solicitar saldo (`getBalance?v` e `getBalanceDup?v`). Os eventos duplicatas, `depositDup?b` e `getBalanceDup?v`, só sincronizam com processos de suas subclasses, neste exemplo, com o `SAVING` que possui o mesmo identificador de `ACCOUNT`. Os outros dois eventos, `deposit?v` e `getBalance?v`, sincronizam com o processo `BANK`.

```
    main =  deposit?v -> main
            []
            withdraw?v -> (withdrawOk -> main
                           []
```

```
                    withdrawNotOk -> main)
          []
          getBalance?v -> main
          []
          exit -> getStateAcc?st -> terminate -> SKIP
          []
          getBalanceDup?v -> main
          []
          depositDup?b -> main
```

Conjunto de eventos de sincronização dos processos `main` e `Z_MAIN`.

```
CSPOps = Ops
```

Conjunto de possíveis valores para o estado de `ACCOUNT`.

```
state = {(num,bal,pw,vl) | num <- Number,
         bal <- Real, pw <- Passw,vl <- Real,checkAcc(bal)}
init = {(num',bal',pw',vl') | (num',bal',pw',vl') <- state,
        num' == nu,bal' == bl,pw'== ps,vl'== va,checkAcc(bal')}
```

A função `in` define o conjunto de elementos entrada.

```
in(deposit) = Real
in(getStateAcc) = {{}}
in(withdraw) = Real
in(withdrawOk) = {{}}
in(withdrawNotOk) = {{}}
in(getBalance) = {{}}
in(terminate) = {{}}
in(depositDup) = Real
in(getBalanceDup) = {{}}
```

A função `out` define o conjunto de elementos saída.

```
out(deposit) = {{}}
out(getStateAcc) = AccountContext
out(withdraw) = {{}}
out(withdrawOk) = {{}}
out(withdrawNotOk) = {{}}
out(getBalance) = Real
out(terminate) = {{}}
out(depositDup) = {{}}
out(getBalanceDup) = Real
```

A função enable testa se a operação está habilitada para os valores do estado.

```
enable(deposit)((num,bal,pw,vl))= true
enable(getStateAcc)((num,bal,pw,vl))= true
enable(withdraw)((num,bal,pw,vl))= true
enable(withdrawOk)((num,bal,pw,vl))= bal >= vl
enable(withdrawNotOk)((num,bal,pw,vl))= bal < vl
enable(getBalance)((num,bal,pw,vl))= true
enable(terminate)((num,bal,pw,vl))= true
enable(depositDup)((num,bal,pw,vl))= true
enable(getBalanceDup)((num,bal,pw,vl))= true
```

A função effect executa a operação habilitada pela função enable.

```
effect(deposit)((num,bal,pw,vl),v) =
 {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
  v <- Real, num' == num, bal'== bal + v, pw'== pw,vl'== vl,
  checkAcc(bal')}

effect(getStateAcc)((num,bal,pw,vl),_) =
 {(o,(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
   num' == num, bal'== bal, pw'== pw, o <- AccountContext,
   o == (num',bal',pw',vl'),vl'== vl, checkAcc(bal')}

effect(withdraw)((num,bal,pw,vl),v) =
 {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
  num' == num, bal'== bal, pw'== pw,vl'==v, checkAcc(bal')}

effect(withdrawOk)((num,bal,pw,vl),_) =
 {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
   num' == num, bal'== bal - vl, pw'== pw,vl'== vl,checkAcc(bal')}

effect(withdrawNotOk)((num,bal,pw,vl),_) =
 {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
  num' == num, bal'== bal, pw'== pw,vl'==vl,
  checkAcc(bal')}

effect(getBalance)((num,bal,pw,vl),_) =
 {(o,(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
 num' == num, bal'== bal, pw'== pw, vl'==vl, o <- Real,
 o == bal',checkAcc(bal')}

effect(terminate)((num,bal,pw,vl),_) =
 {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
   num' == num, bal'== bal,pw'== pw,vl'==vl,checkAcc(bal')}

effect(depositDup) ((num,bal,pw,vl),v) =
 {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
```

```
      v <- Real,num' == num, bal'== bal + v, pw'== pw,vl'==vl,
      checkAcc(bal')}

  effect(getBalanceDup)((num,bal,pw,vl),_) =
   {(o,(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
     num' == num, bal'== bal, pw'== pw, vl'== vl,o <- Real,
     o == bal',checkAcc(bal')}

within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

SAVING é o processo que modela poupança. Como ele é uma especialização de ACCOUNT, então ao ser criado, ele próprio cria um processo ACCOUNT com valores que fazem parte dos seus parâmetros. Os parâmetros de SAVING servem para inicializar as variáveis de estado dos dois processos.

```
SAVING(nu,bl,ps,va,rt) =
 let
```

Conjunto de operações usadas pelo processo da parte de Z.

```
  Ops = {recoverState,interest,terminate}
```

A constante LocOps contém apenas os nomes dos canais locais.

```
  LocOps = {recoverState}


  main =
   let
     e = (nu,bl,ps,va)
     procAcc = proc(TACCOUNT,e)
```

O processo Flow possui apenas os eventos específicos de poupança. Os eventos herdados não aparecem no Flow porque o seu processo ACCOUNT é quem os realiza.

```
     Flow =   exit -> getStateAcc?st1 -> recoverState?st ->
                 getStateSav!makeContext(st1,st) -> terminate -> SKIP
                    []
              interest?t -> getBalanceDup?b -> depositDup.(b*t) -> Flow

   within (procAcc [|{|getStateAcc,getBalanceDup,depositDup,
              terminate,exit|}|] Flow)\{|getBalanceDup,depositDup|}
```

Conjunto de eventos de sincronização dos processos main e Z_MAIN.

```
  CSPOps = Ops
```

O estado possui apenas um elemento, a taxa, porque os outros são herdados e manipulados pelo processo ACCOUNT.

```
state = {r | r <- Ratio}
init = {r' | r' <- state, r'== rt}
```

A função in define o conjunto de elementos entrada.

```
in(recoverState) = {{}}
in(interest) =  Ratio
in(terminate) = {{}}
```

A função out define o conjunto de elementos saída.

```
out(recoverState) = Ratio
out(interest) = {{}}
out(terminate) = {{}}
```

A função enable testa se a operação está habilitada para os valores do estado.

```
enable(recoverState)(r)= true
enable(interest)(r)= true
enable(terminate)(r)= true
```

A função effect executa a operação habilitada pela função enable.

```
effect(recoverState)(r,_) =
 {(o,r') | r' <- state, r'== r, o <- Ratio, o== r'}
effect(interest)(r,ra) =
 {({},r') | r' <- state, r'== ra}
effect(terminate)(r,_) =
 {({},r) | r' <- state, r'== r}

within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

Apêndice B

Implementação do Sistema Bancário em CSP_M Utilizando a Segunda Abordagem

Segue a apresentação da especificação do sistema bancário completa, utilizando um determinado processo que gerencia outros processos. Neste sistema estamos utilizando os processos `BANK`, `ACCOUNT` e `SPECACCOUNT`, que utilizam métodos redefinidos.

A Função `FSeq` retorna um conjunto de seqüências, de tamanho s, com elementos do conjunto T. Seque a sua definição:

```
FSeq(T, 0) = {<>}
FSeq(T, 1) = union(FSeq(T, 0), {<z> | z<-T})
FSeq(T, s) = {z^z' | z<-FSeq(T, 1), z'<-FSeq(T, s-1)}
```

Função que adiciona mais um elemento a tupla.

```
makeContext((a,b,c,d),e) = (a,b,c,d,e)
```

Função que recupera a tupla de variáveis de estado.

```
recoverTuple(AccCtxt.(x,y,z,w)) = (x,y,z,w)
recoverTuple(SpAccCtxt.(x,y,z,w,t)) = (x,y,z,w,t)
```

Função que retorna o elemento de um conjunto unitário.

```
pick({x}) = x
```

Função que verifica se tem conta especial na coleção de contas.

```
hasSpAcc(accounts)=
  not empty({SpAccCtxt.(a,b,c,d,e) | SpAccCtxt.(a,b,c,d,e)<-accounts})
```

Função que verifica se tem conta na coleção de contas.

```
hasAcc(accounts)=
  not empty({AccCtxt.(a,b,c,d) | AccCtxt.(a,b,c,d)<-accounts})
```

Função que procura as contas, na coleção de contas, que possuem identificadores que estão na lista de identificadores.

```
findAcc(setAcc,{}) = {}
findAcc(setAcc,accounts) =
  {AccCtxt.(a,b,c,d) | AccCtxt.(a,b,c,d) <- accounts, member(a,setAcc)}
```

Função que procura as contas especiais, na coleção de contas, que possuem identificadores que estão na lista de identificadores.

```
findSpAcc(setSpAcc,{}) = {}
findSpAcc(setSpAcc,accounts) =
 {SpAccCtxt.(a,b,c,d,e) | SpAccCtxt.(a,b,c,d,e) <- accounts,
  member(a,setSpAcc)}
```

Função que seleciona as contas do conjunto de contas.

```
selectAcc(accounts) =
 {AccCtxt.(a,b,c,d) | AccCtxt.(a,b,c,d) <- accounts}
```

Função que seleciona as contas especiais do conjunto de contas.

```
selectSpAcc(accounts) =
 {SpAccCtxt.(a,b,c,d,e) | SpAccCtxt.(a,b,c,d,e) <- accounts}
```

Função que seleciona os identificadores do conjunto de contas (setAcc).

```
idsAcc(setAcc) =
 {n | n <- Number, AccCtxt.(x,y,z,w) <- setAcc, n == x}
```

Função que seleciona os identificadores do conjunto de contas especiais(setSpAcc).

```
idsSpAcc(setSpAcc) =
 {n | n <- Number, SpAccCtxt.(x,y,z,w,t) <- setSpAcc, n == x}
```

A função `checkAccSpAcc` (invariante de `BANK` não permite que se tenha duas contas ou duas contas especiais ou uma conta e uma conta especial com o mesmo identificador. Isto significa que não se pode ter objetos da classe e de suas subclasses com o mesmo número de identificação. Implementamos essa restrição utilizando uma lista de identificadores não repetidos (sAcc), que fornece o próximo número disponível para a criação de um novo objeto.

```
checkAccSpAcc(accounts,<>)=
   card(accounts) > 0 and
   empty(inter(idsAcc(selectAcc(accounts)),idsSpAcc(selectSpAcc(accounts))))
checkAccSpAcc(accounts,sAcc)=
 if hasAcc(accounts) and not hasSpAcc(accounts)
 then (if empty(inter(findAcc(set(sAcc),accounts),accounts))
       then true
       else false)
 else if hasSpAcc(accounts) and not hasAcc(accounts)
```

```
      then (if empty(inter(findSpAcc(set(sAcc),accounts),accounts))
            then true
            else false)
      else (if hasAcc(accounts) and hasSpAcc(accounts)
            then (empty(inter(findAcc(set(sAcc),accounts),accounts)) and
                  empty(inter(findSpAcc(set(sAcc),accounts),accounts)) and
                  empty(inter(idsAcc(selectAcc(accounts)),
                              idsSpAcc(selectSpAcc(accounts)))))
            else true)
```

Função que representa o invariante de ACCOUNT. Ela garante que o saldo da conta e da conta especial não pode ficar negativo.

```
checkAcc(b)= if b >= 0 then true else false
```

Função que recupera o primeiro elemento de objeto conta (AccCtxt.(x,y,z,w)) e conta especial (SpAccCtxt.(x,y,z,w,t)).

```
firstFind(AccCtxt.(x,y,z,w)) = x
firstFind(SpAccCtxt.(x,y,z,w,t)) = x
```

A função Semantics traduz a parte de Z para processo e o coloca em paralelo com a parte de CSP.

```
Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event) =
 let
   DIV = DIV
   Z_PART(s) =
     [] op:Ops @ enable(op)(s) & [] i: in(op) @
     (if empty(effect(op)(s,i))
      then (|~| o:out(op) @ event(op,i,o) -> DIV)
      else if op == terminate
           then (|~| (o,s'):effect(op)(s,i) @ event(op,i,o) -> SKIP)
           else (|~| (o,s'):effect(op)(s,i) @ event(op,i,o) -> Z_PART(s')))

   Z_MAIN = |~| s:init @ Z_PART(s)

within (Z_MAIN [|{| op | op <- CSPOps |}|] main)\{| op | op <- LocOps |}
```

Função que traduz os parâmetros op, i, o em eventos CSP válidos.

```
event(op,i,o) =
   (if i == {}
    then (if o == {}
          then op
          else op.o)
    else if o == {}
         then op.i
         else op.i.o)
```

A função proc cria processos dinamicamente. Ela recebe como parâmetros o tipo do processo e os valores de inicialização das variáveis de estado.

```
proc(TBANK,(cc,sa)) = BANK(cc,sa)
proc(TACCOUNT,(n,b,p,v)) = ACCOUNT(n,b,p,v)
proc(TSPECACCOUNT,(n,b,p,v,bn)) = SPECACCOUNT(n,b,p,v,bn)
```

Conjunto de mensagens.

```
datatype Message =  notOkAccount | insuficientBalance | withoutReference
```

Conjunto de tipos de Objetos.

```
datatype ObjType = TACCOUNT | TBANK |TSPECACCOUNT
```

Conjunto de identificadores de objetos.

```
nametype Number = {2,3}
```

Conjunto de valores do tipo Real.

```
nametype Real = {0}
```

Conjunto de taxas.

```
nametype Ratio = {0}
```

Conjunto de senhas.

```
nametype Passw = {1}
```

Conjunto de possíveis valores para o estado de ACCOUNT.

```
nametype AccountContext =
 {(n,b,p,v) | n <- Number, b <- Real, p <- Passw, v <- Real}
```

Conjunto de possíveis valores para o estado de SPECACCOUNT.

```
nametype SpecAccountContext =
 {(n,b,p,v,bo) |  n <- Number, b <- Real, p <- Passw, v <- Real, bo <- Real}
```

Conjunto de contas e contas especiais.

```
datatype ObjectContext = AccCtxt.AccountContext |  SpAccCtxt.SpecAccountContext
```

Conjunto de contas.

```
nametype ObjectContextAcc = {AccCtxt.acc | acc <- AccountContext}
```

Conjunto de contas especiais.

```
nametype ObjectContextSpAcc = {SpAccCtxt.eacc | eacc <- SpecAccountContext}
```

Declaração dos canais.

```
channel getIndObj:Number
channel notGetIndObj
channel createAccount:Passw
channel update:ObjectContext
channel add:ObjectContext
channel depositB:Real
channel withdrawB:Real
channel found:ObjectContext
channel notFound
channel message:Message
channel getBalanceB
channel createSpecAccount:Passw
channel deposit,withdraw:Real
channel getBalance:Real
channel getStateAcc: AccountContext
channel getStateSpecAcc: SpecAccountContext
channel withdrawOk,withdrawNotOk
channel getMoney
channel bonusB
channel getBonus:Real
channel resetBonus
channel enterAcc:Number
channel enterSpecAcc:Number
channel terminate
channel depositDup:Real
channel exit
channel recoverState:Real
```

O processo SYSTEM é formado apenas pelo processo BANK.

```
SYSTEM =  let
            bkState = ({},<2,3>)
            procBank = proc(TBANK,bkState)
          within (procBank)
```

O processo BANK é responsável pelas operações bancárias como por exemplo abrir conta e poupança, depositar, sacar e consultar saldo. Ele possui como parâmetros o conjunto contas (cts) e uma lista de identificadores (sqId).

```
BANK(cts,sqId) =
 let
```

A constante local Ops contém o conjunto de nomes de canais usados pelo processo referente a parte de Z.

```
    Ops = {createAccount,enterAcc,add,getIndObj,notGetIndObj,message,
           createSpecAccount,enterSpecAcc,found,notFound,update}
```

A constante LocOps contém apenas os nomes dos canais locais.

```
LocOps = {add,update,getObjInd,notGetIndObj,found,NotFound}
```

Abre uma nova conta.

```
main = createAccount?p ->
        (getIndObj?ind -> add.(AccCtxt.(ind,0,p,0)) -> main
        []
        notGetIndObj -> message?m -> main)

       []
```

Entra com o número da conta.

```
       enterAcc?an ->
```

Achou a conta. É necessário adicionar o conjunto ObjectContextAcc porque o canal found só pode receber objetos do tipo conta.

```
        (found?obj:ObjectContextAcc ->
             (let
```

Como o objeto conta se encontra no formato AccCtxt.(a,b,c,d), então é necessário recuperar a tupla (a,b,c,d) para criar o processo ACCOUNT.

```
                e = recoverTuple(obj)
```

Cria um processo do tipo ACCOUNT.

```
                procAcc = proc(TACCOUNT,e)
```

Fluxo do processo procAcc.

```
                Flow = depositB?v -> deposit.v -> Flow
                       []
                       withdrawB?v ->  withdraw.v ->
                           (withdrawOk -> getMoney -> Flow
                            []
                            withdrawNotOk -> msg?m -> Flow)
                       []

                       getBalanceB ->  getBalance?b -> Flow

                       []
```

Atualiza a coleção de contas do banco.

```
                        exit -> getStateAcc?st2 ->
                         update.(AccCtxt.st2) -> SKIP

                within (procAcc [|{|deposit,withdraw,withdrawOk,
                        withdrawNotOk,getBalance,getStateAcc,exit|}|]
                        Flow);main)

               []
          notFound -> message?m -> main)

     []
```

Abre uma nova conta especial.

```
     createSpecAccount?p ->
      (getIndObj?ind -> add.(SpAccCtxt.(ind,0,p,0,0)) -> main
       []
       notGetIndObj -> message?m -> main)

     []
```

Entra com o número da conta especial.

```
     enterSpecAcc?ecn ->
```

Achou a conta.

```
      (found?obj:ObjectContextSpAcc ->
        (let
           e = recoverTuple(obj)
```

Cria um processo do tipo conta especial.

```
                procSpecAcc = proc(TSPECACCOUNT,e)
```

Fluxo do processo procSpecAcc.

```
                Flow = depositB?v -> deposit.v -> Flow
                       []
                       withdrawB?v -> withdraw.v ->
                         (withdrawOk -> getMoney -> Flow
                          []
                          withdrawNotOk -> msg?m -> Flow)
                       []
                       getBalanceB ->  getBalance?b -> Flow
                       []
                       bonusB -> getBonus?bo -> FlowBonus(bo)
                       []
```

Atualiza a coleção de contas do banco.

```
                        exit -> getStateSpecAcc?st1 ->
                          update.(SpAccCtxt.st1) -> SKIP
```

Processo responsável pelo fluxo bonus

```
                        FlowBonus(bo) = deposit.bo -> resetBonus -> Flow

        within (procSpecAcc [|{|deposit,withdraw,withdrawOk,
                withdrawNotOk,getBalance,getBonus,resetBonus,
                getStateSpecAcc,exit|}|] Flow);main)
      []
      notFound -> message?m -> main)
```

Conjunto de eventos de sincronização dos processos `main` e `Z_MAIN`.

```
CSPOps = Ops
```

A constante `SizeSeq` define o tamanho máximo para a lista de identificadores de contas e poupanças.

```
SizeSeq = 2
```

O conjunto `SeqIds` é resultado da aplicação da função `FSeq`, a qual cria um conjunto de seqüências do tipo do primeiro argumento e cada elemento terá cardinalidade máxima dada pelo segundo argumento.

```
SeqIds = FSeq({2,3},SizeSeq)
```

O estado de `BANK` possui os seguintes elementos: `accounts`, conjunto de contas e de poupanças; `msg`, mensagem que `BANK` pode apresentar ao ambiente; `seqId`, seqüência de identificadores utilizados na criação dos objetos conta e poupança; `type`, tipo do objeto utilizado; `num`, número de dentificador de conta e de poupança. A função `enable`, responsável pela habilitação e desabilitação das operações, não considera as variáveis de entrada, apenas as de estado. Como certas operações precisavam ser habilitadas e desabilitadas, então tivemos que adicionar no estado os elementos `type` e `num`, os quais não são inerentes a classe.

A função `checkAccSpcAcc`, definida acima, é responsável por garantir que os números dos identificadores sejam únicos. Isto é, essa função caracteriza o invariante do processo `BANK`.

```
state = {(accounts,msg,seqId,type,num)| accounts <- Set(ObjectContext),
          msg <- Message, seqId <- SeqIds,type <- ObjType,
          num <- Number,checkAccSpAcc(accounts,seqId)}
```

A inicialização da parte de Z do processo `BANK` é dada pela constante `init`.

```
init = {(accounts',msg',seqId',type',num') |
        (accounts',msg',seqId',type',num') <- state,accounts' == cts,
         seqId'== sqId, checkAccSpAcc(accounts',seqId')}
```

A função in define o conjunto de elementos entrada.

```
in(createAccount) = Passw
in(enterAcc) = Number
in(getIndObj) = {{}}
in(notGetIndObj)= {{}}
in(add) = ObjectContext
in(found) = {{}}
in(update) = ObjectContext
in(notFound) = {{}}
in(message) = {{}}
in(createSpecAccount) = Passw
in(enterSpecAcc) = Number
```

A função out define o conjunto de elementos saída.

```
out(createAccount) = {{}}
out(enterAcc) = {{}}
out(getIndObj) = Number
out(notGetIndObj) = {{}}
out(add) = {{}}
out(found) = ObjectContext
out(update) = {{}}
out(notFound) = {{}}
out(message) = Message
out(createSpecAccount) = {{}}
out(enterSpecAcc) = {{}}
```

A função enable testa se a operação está habilitada para os valores do estado.

```
enable(createAccount)((accounts,msg,seqId,type,num)) =  true
enable(enterAcc)((accounts,msg,seqId,type,num)) =  true
enable(getIndObj)((accounts,msg,seqId,type,num)) = not null(seqId)
enable(notGetIndObj)((accounts,msg,seqId,type,num)) = null(seqId)
enable(add)((accounts,msg,seqId,type,num)) = true
enable(found)((accounts,msg,seqId,type,num)) =
  if type == TACCOUNT
  then let accs = selectAcc(accounts)
       within   {e | e <- accs, firstFind(e) == num} != {}
  else if type == TSPECACCOUNT
       then let spaccs = selectSpAcc(accounts)
            within{e | e <- spaccs, firstFind(e) == num} != {}
       else false
enable(update)((accounts,msg,seqId,type,num)) = true
enable(notFound)((accounts,msg,seqId,type,num)) =
  if type == TACCOUNT
  then let accs = selectAcc(accounts)
       within   {e | e <- accs, firstFind(e) == num} == {}
```

```
    else if type == TSPECACCOUNT
         then let spaccs = selectSpAcc(accounts)
              within{e | e <- spaccs, firstFind(e) == num} == {}
         else false
  enable(message)((accounts,msg,seqId,type,num)) = true
  enable(createSpecAccount)((accounts,msg,seqId,type,num)) = true
  enable(enterSpecAcc)((accounts,msg,seqId,type,num)) =  true
```

A função effect executa a operação habilitada pela função enable.

```
  effect(createAccount)((accounts,msg,seqId,type,num),p) =
    {({},(accounts',msg',seqId',type',num')) |
      accounts',msg',seqId',type',num') <- state, num'== num,
      accounts'== accounts, msg'== msg, seqId'==seqId,
      type'== type,checkAccSpAcc(accounts',seqId')}

 effect(enterAcc)((accounts,msg,seqId,type,num),n) =
    {({},(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state, num'== n,
      accounts'== accounts, msg'== msg, seqId'==seqId,
      type'== TACCOUNT,checkAccSpAcc(accounts',seqId')}

  effect(getIndObj)((accounts,msg,seqId,type,num),_) =
    {(o,(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state,
     accounts'== accounts, msg'== msg, o <- Number,
     o == head(seqId),seqId'== tail(seqId), num'== num,
     type'== type,checkAccSpAcc(accounts',seqId')}

  effect(notGetIndObj)((accounts,msg,seqId,type,num),_) =
    {(o,(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state,
     accounts'== accounts, msg'== withoutReference,seqId'== seqId,
     num'== num,type'== type,checkAccSpAcc(accounts',seqId')}

  effect(add)((accounts,msg,seqId,type,num),c) =
    {({},(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state, num'== num,
      accounts'== union(accounts,{c}),msg'== msg,seqId'== seqId,
      type'== type,checkAccSpAcc(accounts',seqId')}

  effect(found)((accounts,msg,seqId,type,num),_) =
    {(o,(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state,
     accounts'== accounts, msg'== msg, o <- ObjectContext,
     o == pick({ e | e <- accounts, firstFind(e) == num}),
     seqId'== seqId, type'== type, num'== num,
     checkAccSpAcc(accounts',seqId')}
```

```
  effect(update)((accounts,msg,seqId,type,num),cs) =
    {({},(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state,
     accounts'==  union(diff({ e | e <- accounts,
     firstFind(e) == firstFind(cs)},accounts),{cs}),
     msg'== msg,seqId'== seqId,type'== type,
     num'== num,checkAccSpAcc(accounts',seqId')}

 effect(notFound)((accounts,msg,seqId,type,num),_) =
   {({},(accounts',msg',seqId',type',num')) |
    (accounts',msg',seqId',type',num') <- state, num'== num,
    accounts'== accounts, msg' == notOkAccount, seqId'== seqId,
    type'== type,checkAccSpAcc(accounts',seqId')}

effect(message)((accounts,msg,seqId,type,num),_) =
  {(o,(accounts',msg',seqId',type',num')) |
   (accounts',msg',seqId',type',num') <- state,o <- Message,
   num'== num,accounts'== accounts, msg'== msg,o == msg',
   seqId'== seqId,type'== type,checkAccSpAcc(accounts',seqId')}

effect(createSpecAccount)((accounts,msg,seqId,type,num),p) =
  {({},(accounts',msg',seqId',type',num')) |
   (accounts',msg',seqId',type',num') <- state, num'== num,
   accounts'== accounts, msg'== msg, seqId'==seqId,
   type'== type,checkAccSpAcc(accounts',seqId')}

effect(enterSpecAcc)((accounts,msg,seqId,type,num),n) =
  {({},(accounts',msg',seqId',type',num')) |
   (accounts',msg',seqId',type',num') <- state, num'== n,
   accounts'== accounts, msg'== msg, seqId'==seqId,
   type'== TSPECACCOUNT,checkAccSpAcc(accounts',seqId')}
```

A função `Semantics` faz a tradução da parte de Z para processo e o coloca em paralelo com a parte de CSP.

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

O processo `ACCOUNT` modela a conta corrente.

```
ACCOUNT(nu,bl,ps,va) =
 let
```

Conjunto de operações usadas pelo processo da parte de Z.

```
      Ops = {deposit,getStateAcc,withdraw,withdrawOk,withdrawNotOk,
             getBalance,terminate}
```

A constante LocOps contém apenas os nomes dos canais locais.

```
LocOps = {}


main =  deposit?v -> main
        []
        withdraw?v -> (withdrawOk -> main
                       []
                       withdrawNotOk -> main)
        []
        getBalance?v -> main
        []
        exit -> getStateAcc?st -> terminate -> SKIP
```

Conjunto de eventos de sincronização dos processos `main` e `Z_MAIN`.

```
CSPOps = Ops
```

Conjunto de possíveis valores para o estado de `ACCOUNT`.

```
state = {(num,bal,pw,vl) | num <- Number, bal <- Real,
          pw <- Passw,vl <- Real,checkAcc(bal)}
init = {(num',bal',pw',vl') | (num',bal',pw',vl') <- state,
         num' == nu, bal' == bl, pw'== ps,vl'== va,checkAcc(bal')}
```

A função `in` define o conjunto de elementos entrada.

```
in(deposit) = Real
in(getStateAcc) = {{}}
in(withdraw) = Real
in(withdrawOk) = {{}}
in(withdrawNotOk) = {{}}
in(getBalance) = {{}}
in(terminate) = {{}}
```

A função `out` define o conjunto de elementos saída.

```
out(deposit) = {{}}
out(getStateAcc) = AccountContext
out(withdraw) = {{}}
out(withdrawOk) = {{}}
out(withdrawNotOk) = {{}}
out(getBalance) = Real
out(terminate) = {{}}
```

A função enable testa se a operação está habilitada para os valores do estado.

```
enable(deposit)((num,bal,pw,vl))= true
enable(getStateAcc)((num,bal,pw,vl))= true
enable(withdraw)((num,bal,pw,vl))= true
enable(withdrawOk)((num,bal,pw,vl))= bal >= vl
enable(withdrawNotOk)((num,bal,pw,vl))= bal < vl
enable(getBalance)((num,bal,pw,vl))= true
enable(terminate)((num,bal,pw,vl))= true
```

A função effect executa a operação habilitada pela função enable.

```
effect(deposit)((num,bal,pw,vl),v) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
   v <- Real, num' == num, bal'== bal + v, pw'== pw,vl'== vl,
   checkAcc(bal')}

effect(getStateAcc)((num,bal,pw,vl),_) =
  {(o,(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
    num' == num, bal'== bal, pw'== pw,vl'== vl, o <- AccountContext,
    o == (num',bal',pw',vl'),checkAcc(bal')}

effect(withdraw)((num,bal,pw,vl),v) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
    v <- Real, num' == num, bal'== bal, pw' == pw,
    vl'== vl,checkAcc(bal')}

effect(withdrawOk)((num,bal,pw,vl),_) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
    num' == num, bal'== bal - vl, pw'== pw,vl'== vl,checkAcc(bal')}

effect(withdrawNotOk)((num,bal,pw,vl),_) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
   num' == num, bal'== bal, pw'== pw,vl'== vl,checkAcc(bal')}

effect(getBalance)((num,bal,pw,vl),_) =
  {(o,(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
    num' == num, bal'== bal, pw'== pw, vl'== vl,o <- Real,
    o == bal',checkAcc(bal')}

effect(terminate)((num,bal,pw,vl),_) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
   num' == num, bal'== bal, pw'== pw,vl'== vl,checkAcc(bal')}

within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

SPECACCOUNT é o processo que modela conta especial. Como ele é uma especialização de ACCOUNT, então ao ser criado, ele próprio cria um processo ACCOUNT com valores que

fazem parte dos seus parâmetros. Os parâmetros de SPECACCOUNT servem para inicializar as variáveis de estado dos dois processos.

```
SPECACCOUNT(nu,bl,ps,va,bn) =
 let
```

Conjunto de operações usadas pelo processo da parte de Z.

```
    Ops = {terminate,deposit,getBonus,resetBonus,recoverState}
```

A constante LocOps contém apenas os nomes dos canais locais.

```
    LocOps = {recoverState}


    main =
     let
       e = (nu,bl,ps,va)
       procAcc = proc(TACCOUNT,e)
```

O processo Flow possui todos os eventos herdados e mais os novos. Os eventos herdados e não redefinidos não possuem a parte de z, porque o processo da superclasse é que vai utilizar os dados. Porém, os eventos redefinidos serão tratados tanto na classe quanto na superclasse (divisão de trabalho).

```
       Flow =  deposit?v -> Flow
               []
               withdraw?v -> (withdrawOk -> Flow
                              []
                              withdrawNotOk -> Flow)
               []
               getBalance?v -> Flow
               []
               exit -> getStateAcc?st1 -> recoverState?st ->
                 getStateSpecAcc!makeContext(st1,st) -> terminate -> SKIP
               []
               getBonus?b -> FlowBonus

               FlowBonus = deposit?v -> resetBonus -> Flow

     within (procAcc [|{|deposit,withdraw,withdrawOk,withdrawNotOk,
                     getBalance,exit,getStateAcc,terminate|}|] Flow)
```

Conjunto de eventos de sincronização dos processos main e Z_MAIN.

```
    CSPOps = Ops
```

O estado possui apenas um elemento, o bônus, porque os outros são herdados e manipulados pelo processo ACCOUNT.

```
state = {bo | bo <- Real}
init = {bo' | bo' <- state, bo'== bn}
```

A constante `bonusRatio` é utilizada no momento do cálculo do bônus.

```
bonusRatio = 1
```

A função `in` define o conjunto de elementos entrada.

```
in(terminate) = {{}}
in(deposit) = Real
in(getBonus) = {{}}
in(resetBonus) = {{}}
in(recoverState) = {{}}
```

A função `out` define o conjunto de elementos saída.

```
out(terminate) = {{}}
out(deposit) = {{}}
out(getBonus) = Real
out(resetBonus) = {{}}
out(recoverState) = Real
```

A função `enable` testa se a operação está habilitada para os valores do estado.

```
enable(terminate)(bo)= true
enable(deposit)(bo)= true
enable(getBonus)(bo)= true
enable(resetBonus)(bo)= true
enable(recoverState)(bo)= true
```

A função `effect` executa a operação habilitada pela função enable.

```
effect(terminate)(bo,_) =
 {({},bo') | bo' <- state, bo'== bo}
effect(deposit)(bo,v) =
 {({},bo') | bo' <- state, bo'== bo + (v * bonusRatio)}
effect(getBonus)(bo,_) =
 {(o,bo') | bo' <- state, bo'== bo, o <- Real, o== bo'}
effect(resetBonus)(bo,_) =
 {({},bo') | bo' <- state,bo'== 0}
effect(recoverState)(bo,_) =
 {(o,bo') | bo' <- state, bo'== bo, o <- Real, o== bo'}

within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

Apêndice C

Implementação do Sistema Bancário em CSP_M Utilizando a Terceira Abordagem

Segue a apresentação da especificação do sistema bancário completa, utilizando cópia de processos. Neste sistema estamos utilizando os processos `BANK`, `ACCOUNT` e `SPECACCOUNT`, que utilizam métodos redefinidos.

Funções que recuperam um elemento na tupla

```
first4((a,b,c,d)) = a
second4((a,b,c,d)) = b
third4((a,b,c,d)) = c
forth4((a,b,c,d)) = d
```

A Função `FSeq` retorna um conjunto de seqüências, de tamanho s, com elementos do conjunto T. Seque a sua definição:

```
FSeq(T, 0) = {<>}
FSeq(T, 1) = union(FSeq(T, 0), {<z> | z<-T})
FSeq(T, s) = {z^z' | z<-FSeq(T, 1), z'<-FSeq(T, s-1)}
```

Função que adiciona mais um elemento a tupla.

```
makeContext((a,b,c,d),e) = (a,b,c,d,e)
```

Função que recupera a tupla de variáveis de estado.

```
recoverTuple(AccCtxt.(x,y,z,w)) = (x,y,z,w)
recoverTuple(SpAccCtxt.(x,y,z,w,t)) = (x,y,z,w,t)
```

Função que retorna o elemento de um conjunto unitário.

```
pick({x}) = x
```

Função que verifica se tem conta especial na coleção de contas.

```
hasSpAcc(accounts)=
 not empty({SpAccCtxt.(a,b,c,d,e) | SpAccCtxt.(a,b,c,d,e)<-accounts})
```

Função que verifica se tem conta na coleção de contas.

```
hasAcc(accounts)=
 not empty({AccCtxt.(a,b,c,d) | AccCtxt.(a,b,c,d)<-accounts})
```

Função que procura as contas, na coleção de contas, que possuem identificadores que estão na lista de identificadores.

```
findAcc(setAcc,{}) = {}
findAcc(setAcc,accounts) =
  {AccCtxt.(a,b,c,d) | AccCtxt.(a,b,c,d) <- accounts, member(a,setAcc)}
```

Função que procura as contas especiais, na coleção de contas, que possuem identificadores que estão na lista de identificadores.

```
findSpAcc(setSpAcc,{}) = {}
findSpAcc(setSpAcc,accounts) =
 {SpAccCtxt.(a,b,c,d,e) | SpAccCtxt.(a,b,c,d,e) <- accounts,
  member(a,setSpAcc)}
```

Função que seleciona as contas do conjunto de contas.

```
selectAcc(accounts) =
 {AccCtxt.(a,b,c,d) | AccCtxt.(a,b,c,d) <- accounts}
```

Função que seleciona as contas especiais do conjunto de contas.

```
selectSpAcc(accounts) =
 {SpAccCtxt.(a,b,c,d,e) | SpAccCtxt.(a,b,c,d,e) <- accounts}
```

Função que seleciona os identificadores do conjunto de contas (setAcc).

```
idsAcc(setAcc) =
  {n | n <- Number, AccCtxt.(x,y,z,w) <- setAcc, n == x}
```

Função que seleciona os identificadores do conjunto de contas especiais(setSpAcc).

```
idsSpAcc(setSpAcc) =
  {n | n <- Number, SpAccCtxt.(x,y,z,w,t) <- setSpAcc, n == x}
```

A função checkAccSpAcc (invariante de `BANK`) não permite que se tenha duas contas ou duas contas especiais ou uma conta e uma conta especial com o mesmo identificador. Isto significa que não se pode ter objetos da classe e de suas subclasses com o mesmo número de identificação. Implementamos essa restrição utilizando uma lista de identificadores não repetidos (sAcc), que fornece o próximo número disponível para a criação de um novo objeto.

```
checkAccSpAcc(accounts,<>)=
  card(accounts) > 0 and
  empty(inter(idsAcc(selectAcc(accounts)),idsSpAcc(selectSpAcc(accounts))))
checkAccSpAcc(accounts,sAcc)=
 if hasAcc(accounts) and not hasSpAcc(accounts)
 then (if empty(inter(findAcc(set(sAcc),accounts),accounts))
       then true
       else false)
 else if hasSpAcc(accounts) and not hasAcc(accounts)
      then (if empty(inter(findSpAcc(set(sAcc),accounts),accounts))
            then true
            else false)
      else (if hasAcc(accounts) and hasSpAcc(accounts)
            then (empty(inter(findAcc(set(sAcc),accounts),accounts)) and
                  empty(inter(findSpAcc(set(sAcc),accounts),accounts)) and
                  empty(inter(idsAcc(selectAcc(accounts)),
                              idsSpAcc(selectSpAcc(accounts)))))
            else true)
```

Função que representa o invariante de ACCOUNT. Ela garante que o saldo da conta e da conta especial não pode ficar negativo.

```
checkAcc(b)= if b >= 0 then true else false
```

Função que recupera o primeiro elemento de objeto conta (AccCtxt.(x,y,z,w)) e conta especial (SpAccCtxt.(x,y,z,w,t)).

```
firstFind(AccCtxt.(x,y,z,w)) = x
firstFind(SpAccCtxt.(x,y,z,w,t)) = x
```

A função Semantics traduz a parte de Z para processo e o coloca em paralelo com a parte de CSP.

```
Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event) =
 let
   DIV = DIV
   Z_PART(s) =
     [] op:Ops @ enable(op)(s) & [] i: in(op) @
     (if empty(effect(op)(s,i))
      then (|~| o:out(op) @ event(op,i,o) -> DIV)
      else if op == terminate
           then (|~| (o,s'):effect(op)(s,i) @ event(op,i,o) -> SKIP)
           else (|~| (o,s'):effect(op)(s,i) @ event(op,i,o) -> Z_PART(s')))

   Z_MAIN = |~| s:init @ Z_PART(s)

within (Z_MAIN [|{| op | op <- CSPOps |}|] main)\{| op | op <- LocOps |}
```

Função que traduz os parâmetros op, i, o em eventos CSP válidos.

```
event(op,i,o) =
   (if i == {}
    then (if o == {}
          then op
          else op.o)
    else if o == {}
         then op.i
         else op.i.o)
```

A função proc cria processos dinamicamente. Ela recebe como parâmetros o tipo do processo e os valores de inicialização das variáveis de estado.

```
proc(TBANK,(cc,sa)) = BANK(cc,sa)
proc(TACCOUNT,(n,b,p,v)) = ACCOUNT(n,b,p,v)
proc(TSPECACCOUNT,(n,b,p,v,bn)) = SPECACCOUNT(n,b,p,v,bn)
```

Conjunto de mensagens.

```
datatype Message =  notOkAccount | insuficientBalance | withoutReference
```

Conjunto de tipos de Objetos.

```
datatype ObjType = TACCOUNT | TBANK |TSPECACCOUNT
```

Conjunto de identificadores de objetos.

```
nametype Number = {2,3}
```

Conjunto de valores do tipo Real.

```
nametype Real = {0}
```

Conjunto de taxas.

```
nametype Ratio = {0}
```

Conjunto de senhas.

```
nametype Passw = {1}
```

Conjunto de possíveis valores para o estado de ACCOUNT.

```
nametype AccountContext =
  {(n,b,p,v) | n <- Number, b <- Real, p <- Passw, v <- Real}
```

Conjunto de possíveis valores para o estado de SPECACCOUNT.

```
nametype SpecAccountContext =
  {(n,b,p,v,bo) |  n <- Number, b <- Real, p <- Passw, v <- Real,
    bo <- Real}
```

Conjunto de contas e contas especiais.

```
datatype ObjectContext = AccCtxt.AccountContext |
                         SpAccCtxt.SpecAccountContext
```

Conjunto de contas.

```
nametype ObjectContextAcc = {AccCtxt.acc | acc <- AccountContext}
```

Conjunto de contas especiais.

```
nametype ObjectContextSpAcc = {SpAccCtxt.eacc | eacc <- SpecAccountContext}
```

Declaração dos canais.

```
channel getIndObj:Number
channel notGetIndObj
channel createAccount:Passw
channel update:ObjectContext
channel add:ObjectContext
channel depositB:Real
channel withdrawB:Real
channel found:ObjectContext
channel notFound
channel message:Message
channel getBalanceB
channel createSpecAccount:Passw
channel deposit,withdraw:Real
channel getBalance:Real
channel getStateAcc: AccountContext
channel getStateSpecAcc: SpecAccountContext
channel withdrawOk, withdrawNotOk
channel getMoney
channel bonusB
channel bonus
channel enterAcc:Number
channel enterSpecAcc:Number
channel terminate
channel depositDup:Real
channel exit
channel getState:AccountContext
channel setState:AccountContext
channel getBonus:Real
channel recoverState:Real
```

O processo SYSTEM é formado apenas pelo processo BANK.

```
SYSTEM =  let
            bkState = ({},<2,3>)
            procBank = proc(TBANK,bkState)
          within (procBank)
```

O processo BANK é responsável pelas operações bancárias como por exemplo abrir conta e poupança, depositar, sacar e consultar saldo. Ele possui como parâmetros o conjunto contas (cts) e uma lista de identificadores (sqId).

```
BANK(cts,sqId) =
 let
```

A constante local Ops contém o conjunto de nomes de canais usados pelo processo referente a parte de Z.

```
    Ops = {createAccount,enterAcc,add,getIndObj,notGetIndObj,message,
           createSpecAccount,enterSpecAcc,found,notFound,update}
```

A constante LocOps contém apenas os nomes dos canais locais.

```
    LocOps = {add,update,getObjInd,notGetIndObj,found,NotFound}
```

Abre uma nova conta.

```
    main = createAccount?p ->
              (getIndObj?ind -> add.(AccCtxt.(ind,0,p,0)) -> main
               []
               notGetIndObj -> message?m -> main)
           []
```

Entra com o número da conta.

```
           enterAcc?an ->
```

Achou a conta. É necessário adicionar o conjunto ObjectContextAcc porque o canal found só pode receber objetos do tipo conta.

```
            (found?obj:ObjectContextAcc ->
                 (let
```

Como o objeto conta se encontra no formato AccCtxt.(a,b,c,d), então é necessário recuperar a tupla (a,b,c,d) para criar o processo ACCOUNT.

```
                    e = recoverTuple(obj)
```

Cria um processo do tipo ACCOUNT

```
                    ProcAcc = proc(TACCOUNT,e)
```

Fluxo do processo ProcAcc.

```
                    Flow = depositB?v -> deposit.v -> Flow
                           []
                           withdrawB?v ->  withdraw.v ->
                             (withdrawOk -> getMoney -> Flow
                              []
                              withdrawNotOk -> message?m -> Flow)
                           []
                           getBalanceB ->  getBalance?b -> Flow
                           []
```

Atualiza a coleção de contas do banco.

```
                    exit -> getStateAcc?st2 ->
                      update.(AccCtxt.st2) -> SKIP

            within (ProcAcc [|{|deposit,withdraw,withdrawOk,
                    withdrawNotOk,getBalance,getStateAcc,exit|}|]
                    Flow);main)

            []
     notFound -> message?m -> main)
    []
```

Abre uma nova conta especial.

```
    createSpecAccount?p ->
      (getIndObj?ind -> add.(SpAccCtxt.(ind,0,p,0,0)) -> main
       []
       notGetIndObj -> message?m -> main)

    []
```

Entra com o número da conta especial.

```
    enterSpecAcc?ecn ->
```

Achou a conta.

```
     (found?obj:ObjectContextSpAcc ->
       (let
          e = recoverTuple(obj)
```

Cria um processo do tipo conta especial.

```
          procSpecAcc = proc(TSPECACCOUNT,e)
```

Fluxo do processo procSpecAcc.

```
          Flow = depositB?v -> deposit.v -> Flow
                 []
                 withdrawB?v -> withdraw.v ->
                    (withdrawOk -> getMoney -> Flow
                     []
                     withdrawNotOk -> message?m -> Flow)
                 []
                 getBalanceB ->  getBalance?b -> Flow
                 []
                 bonusB -> bonus -> Flow
                 []
```

Atualiza a coleção de contas do banco.

```
                    exit -> getStateSpecAcc?st1 ->
                      update.(SpAccCtxt.st1) -> SKIP

          within (procSpecAcc [|{|deposit,withdraw,withdrawOk,
                  withdrawNotOk,getBalance,bonus,
                  getStateSpecAcc,exit|}|] Flow);main)
        []
        notFound -> message?m -> main)
```

Conjunto de eventos de sincronização dos processos `main` e `Z_MAIN`.

```
CSPOps = Ops
```

A constante `SizeSeq` define o tamanho máximo para a lista de identificadores de contas e poupanças.

```
SizeSeq = 2
```

O conjunto `SeqIds` é resultado da aplicação da função `FSeq`, a qual cria um conjunto de seqüências do tipo do primeiro argumento e cada elemento terá cardinalidade máxima dada pelo segundo argumento.

```
SeqIds = FSeq({2,3},SizeSeq)
```

O estado de `BANK` possui os seguintes elementos: `accounts`, conjunto de contas e de poupanças; `msg`, mensagem que `BANK` pode apresentar ao ambiente; `seqId`, seqüência de identificadores utilizados na criação dos objetos conta e poupança; `type`, tipo do objeto utilizado; `num`, número de dentificador de conta e de poupança. A função `enable`, responsável pela habilitação e desabilitação das operações, não considera as variáveis de entrada, apenas as de estado. Como certas operações precisavam ser habilitadas e desabilitadas, então tivemos que adicionar no estado os elementos `type` e `num`, os quais não são inerentes a classe.

A função `checkAccSpcAcc`, definida acima, é responsável por garantir que os números dos identificadores sejam únicos. Isto é, essa função caracteriza o invariante do processo `BANK`.

```
state = {(accounts,msg,seqId,type,num)| accounts <- Set(ObjectContext),
          msg <- Message,seqId <- SeqIds,type <- ObjType,num <- Number,
          checkAccSpAcc(accounts,seqId)}
```

A inicialização da parte de Z do processo `BANK` é dada pela constante `init`.

```
init = {(accounts',msg',seqId',type',num') |
        (accounts',msg',seqId',type',num') <- state,accounts' == cts,
        seqId'== sqId, checkAccSpAcc(accounts',seqId')}
```

A função `in` define o conjunto de elementos entrada.

```
in(createAccount) = Passw
in(enterAcc) = Number
in(getIndObj) = {{}}
in(notGetIndObj)= {{}}
in(add) = ObjectContext
in(found) = {{}}
in(update) = ObjectContext
in(notFound) = {{}}
in(message) = {{}}
in(createSpecAccount) = Passw
in(enterSpecAcc) = Number
```

A função out define o conjunto de elementos saída.

```
out(createAccount) = {{}}
out(enterAcc) = {{}}
out(getIndObj) = Number
out(notGetIndObj) = {{}}
out(add) = {{}}
out(found) = ObjectContext
out(update) = {{}}
out(notFound) = {{}}
out(message) = Message
out(createSpecAccount) = {{}}
out(enterSpecAcc) = {{}}
```

A função enable testa se a operação está habilitada para os valores do estado.

```
enable(createAccount)((accounts,msg,seqId,type,num)) =  true
enable(enterAcc)((accounts,msg,seqId,type,num)) =  true
enable(getIndObj)((accounts,msg,seqId,type,num)) = not null(seqId)
enable(notGetIndObj)((accounts,msg,seqId,type,num)) = null(seqId)
enable(add)((accounts,msg,seqId,type,num)) = true
enable(found)((accounts,msg,seqId,type,num)) =
 if type == TACCOUNT
 then let accs = selectAcc(accounts)
      within   {e | e <- accs, firstFind(e) == num} != {}
 else if type == TSPECACCOUNT
      then let spaccs = selectSpAcc(accounts)
           within{e | e <- spaccs, firstFind(e) == num} != {}
      else false

enable(update)((accounts,msg,seqId,type,num)) = true
enable(notFound)((accounts,msg,seqId,type,num)) =
 if type == TACCOUNT
 then let accs = selectAcc(accounts)
      within   {e | e <- accs, firstFind(e) == num} == {}
 else if type == TSPECACCOUNT
```

```
        then let spaccs = selectSpAcc(accounts)
             within{e | e <- spaccs, firstFind(e) == num} == {}
        else false

  enable(message)((accounts,msg,seqId,type,num)) = true
  enable(createSpecAccount)((accounts,msg,seqId,type,num)) =  true
  enable(enterSpecAcc)((accounts,msg,seqId,type,num)) =  true
```

A função `effect` executa a operação habilitada pela função enable.

```
   effect(createAccount)((accounts,msg,seqId,type,num),p) =
     {({},(accounts',msg',seqId',type',num')) |
      (accounts',msg',seqId',type',num') <- state, num'== num,
      accounts'== accounts, msg'== msg, seqId'==seqId,
      type'== type,checkAccSpAcc(accounts',seqId')}

 effect(enterAcc)((accounts,msg,seqId,type,num),n) =
   {({},(accounts',msg',seqId',type',num')) |
    (accounts',msg',seqId',type',num') <- state, num'== n,
     accounts'== accounts, msg'== msg, seqId'==seqId,
     type'== TACCOUNT,checkAccSpAcc(accounts',seqId')}

  effect(getIndObj)((accounts,msg,seqId,type,num),_) =
    {(o,(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state,
     accounts'== accounts, msg'== msg, o <- Number,
     o == head(seqId),seqId'== tail(seqId), num'== num,
     type'== type,checkAccSpAcc(accounts',seqId')}

  effect(notGetIndObj)((accounts,msg,seqId,type,num),_) =
    {({},(accounts',msg',seqId',type',num')) |
    (accounts',msg',seqId',type',num') <- state,
    accounts'== accounts, msg'== withoutReference,
    seqId'== seqId, type'== type,num'== num,
    checkAccSpAcc(accounts',seqId')}

  effect(add)((accounts,msg,seqId,type,num),c) =
    {({},(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state, num'== num,
     accounts'== union(accounts,{c}),msg'== msg,seqId'== seqId,
     type'== type,checkAccSpAcc(accounts',seqId')}

  effect(found)((accounts,msg,seqId,type,num),_) =
    {(o,(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state,
     accounts'== accounts, msg'== msg, o <- ObjectContext,
     o == pick({ e | e <- accounts, firstFind(e) == num}),
     seqId'== seqId, type'== type, num'== num,
```

```
    checkAccSpAcc(accounts',seqId')}

  effect(update)((accounts,msg,seqId,type,num),cs) =
    {({},(accounts',msg',seqId',type',num')) |
     (accounts',msg',seqId',type',num') <- state,
     accounts'==  union(diff({ e | e <- accounts,
     firstFind(e) == firstFind(cs)},accounts),{cs}),
     msg'== msg,seqId'== seqId,type'== type,
     num'== num,checkAccSpAcc(accounts',seqId')}

 effect(notFound)((accounts,msg,seqId,type,num),_) =
   {({},(accounts',msg',seqId',type',num')) |
    (accounts',msg',seqId',type',num') <- state, num'== num,
    accounts'== accounts, msg' == notOkAccount, seqId'== seqId,
    type'== type,checkAccSpAcc(accounts',seqId')}

effect(message)((accounts,msg,seqId,type,num),_) =
  {(o,(accounts',msg',seqId',type',num')) |
   (accounts',msg',seqId',type',num') <- state,o <- Message,
   num'== num,accounts'== accounts, msg'== msg,o == msg',
   seqId'== seqId,type'== type,checkAccSpAcc(accounts',seqId')}

effect(createSpecAccount)((accounts,msg,seqId,type,num),p) =
  {({},(accounts',msg',seqId',type',num')) |
   (accounts',msg',seqId',type',num') <- state, num'== num,
   accounts'== accounts, msg'== msg, seqId'==seqId,
   type'== type,checkAccSpAcc(accounts',seqId')}

effect(enterSpecAcc)((accounts,msg,seqId,type,num),n) =
  {({},(accounts',msg',seqId',type',num')) |
   (accounts',msg',seqId',type',num') <- state, num'== n,
   accounts'== accounts, msg'== msg, seqId'==seqId,
   type'== TSPECACCOUNT,checkAccSpAcc(accounts',seqId')}
```

A função `Semantics` faz a tradução da parte de Z para processo e o coloca em paralelo com a parte de CSP.

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

O processo `ACCOUNT` modela a conta corrente.

```
ACCOUNT(nu,bl,ps,va) =
 let
```

Conjunto de operações usadas pelo processo da parte de Z.

```
      Ops = {deposit,getStateAcc,withdraw,withdrawOk,withdrawNotOk,
             getBalance,terminate}
```

A constante `LocOps` contém apenas os nomes dos canais locais.

```
LocOps = {}


main =  deposit?v -> main
        []
        withdraw?v -> (withdrawOk -> main
                       []
                       withdrawNotOk -> main)
        []
        getBalance?v -> main
        []
        exit -> getStateAcc?st -> terminate -> SKIP
        []
        getState?st -> setState?st -> main
```

Conjunto de eventos de sincronização dos processos `main` e `Z_MAIN`.

```
CSPOps = Ops
```

Conjunto de possíveis valores para o estado de `ACCOUNT`.

```
state = {(num,bal,pw,vl) | num <- Number, bal <- Real,
         pw <- Passw,vl <- Real,checkAcc(bal)}

init = {(num',bal',pw',vl') | (num',bal',pw',vl') <- state,
         num' == nu, bal' == bl, pw'== ps,vl'== va,checkAcc(bal')}
```

A função `in` define o conjunto de elementos entrada.

```
in(deposit) = Real
in(getStateAcc) = {{}}
in(withdraw) = Real
in(withdrawOk) = {{}}
in(withdrawNotOk) = {{}}
in(getBalance) = {{}}
in(terminate) = {{}}
in(getState) = {{}}
in(setState) = AccountContext
```

A função `out` define o conjunto de elementos saída.

```
out(deposit) = {{}}
out(getStateAcc) = AccountContext
out(withdraw) = {{}}
out(withdrawOk) = {{}}
out(withdrawNotOk) = {{}}
out(getBalance) = Real
out(terminate) = {{}}
out(getState) = AccountContext
out(setState) = {{}}
```

A função enable testa se a operação está habilitada para os valores do estado.

```
enable(deposit)((num,bal,pw,vl))= true
enable(getStateAcc)((num,bal,pw,vl))= true
enable(withdraw)((num,bal,pw,vl))= true
enable(withdrawOk)((num,bal,pw,vl))= bal >= vl
enable(withdrawNotOk)((num,bal,pw,vl))= bal < vl
enable(getBalance)((num,bal,pw,vl))= true
enable(terminate)((num,bal,pw,vl))= true
enable(getState)((num,bal,pw,vl))= true
enable(setState)((num,bal,pw,vl))= true
```

A função effect executa a operação habilitada pela função enable.

```
effect(deposit)((num,bal,pw,vl),v) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
    v <- Real, num' == num, bal'== bal + v, pw'== pw,vl'== vl,
    checkAcc(bal')}

effect(getStateAcc)((num,bal,pw,vl),_) =
  {(o,(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
    num' == num, bal'== bal, pw'== pw,vl'== vl, o <- AccountContext,
    o == (num',bal',pw',vl'),checkAcc(bal')}

effect(withdraw)((num,bal,pw,vl),v) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state, v <- Real,
   num' == num, bal'== bal, pw' == pw,vl'== vl,checkAcc(bal')}

effect(withdrawOk)((num,bal,pw,vl),_) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
   num' == num,bal'== bal - vl, pw'== pw,vl'== vl,checkAcc(bal')}

effect(withdrawNotOk)((num,bal,pw,vl),_) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
   num' == num, bal'== bal, pw'== pw,vl'== vl,checkAcc(bal')}

effect(getBalance)((num,bal,pw,vl),_) =
  {(o,(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
    num' == num, bal'== bal, pw'== pw, vl'== vl,o <- Real, o == bal',
    checkAcc(bal')}

effect(terminate)((num,bal,pw,vl),_) =
  {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
    num' == num, bal'== bal, pw'== pw,vl'== vl,checkAcc(bal')}

effect(getState)((num,bal,pw,vl),_) =
  {(o,(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
    num' == num, bal'== bal, pw'== pw, vl'== vl, o <- AccountContext,
```

```
      o == (num',bal',pw'),checkAcc(bal')}

  effect(setState)((num,bal,pw,vl),ctxt) =
    {({},(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
     num' == first4(ctxt), bal'== second4(ctxt), pw'== third4(ctxt),
     vl'== forth4(ctxt),checkAcc(bal')}

within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

SPECACCOUNT é o processo que modela conta especial. Como ele é uma especialização de ACCOUNT, então ao ser criado, ele próprio cria um processo ACCOUNT com valores que fazem parte dos seus parâmetros. Os parâmetros de SPECACCOUNT servem para inicializar as variáveis de estado dos dois processos.

```
SPECACCOUNT(nu,bl,ps,va,bn) =
 let
```

Conjunto de operações usadas pelo processo da parte de Z.

```
  Ops = {terminate,getBonus,recoverState}
```

A constante LocOps contém apenas os nomes dos canais locais.

```
  LocOps = {recoverState}


  main =
   let
     e = (nu,bl,ps,va)
     procAcc = proc(TACCOUNT,e)
```

O processo Flow possui todos os eventos herdados e mais os novos. Os eventos herdados e não redefinidos não possuem a parte de Z, porque o processo da superclasse é que vai utilizar os dados. Porém, os eventos redefinidos serão tratados tanto na classe quanto na superclasse através de sua cópia (divisão de trabalho).

```
     Flow =  deposit?v -> Flow
             []
             withdraw?v -> (withdrawOk -> Flow
                            []
                            withdrawNotOk -> Flow)
             []
             getBalance?v -> Flow
             []
             exit -> getStateAcc?st1 -> recoverState?st ->
               getStateSpecAcc!makeContext(st1,st) -> terminate -> SKIP
             []
             getState?st -> setState?st -> Flow
             []
             bonus -> getBonus?b -> BonusProc(b);Flow
```

```
BonusProc(b) = getState?st ->
               let
                 e = st
                 procAcc = proc(TACCOUNT,e)
                 Flow = deposit.b -> getState?st1 ->
                        setState.st1 -> exit ->
                        getStateAcc?st -> terminate  -> SKIP
               within (procAcc [|{|deposit,getState,setState,
                      exit,getStateAcc,terminate|}|] Flow)\
                      {|deposit,getState,exit,getStateAcc,
                        terminate|}

within (procAcc [|{|deposit,withdraw,withdrawOk,withdrawNotOk,
       getBalance,exit,getStateAcc,terminate,getState,setState|}|] Flow)
```

Conjunto de eventos de sincronização dos processos `main` e `Z_MAIN`.

```
CSPOps = Ops
```

O estado possui apenas um elemento, o bônus, porque os outros são herdados e manipulados pelo processo `ACCOUNT`.

```
state = {bo | bo <- Real}
init = {bo' | bo' <- state, bo'== bn}
```

A constante `bonusRatio` é utilizada no momento do cálculo do bônus.

```
bonusRatio = 1
```

A função `in` define o conjunto de elementos entrada.

```
in(terminate) = {{}}
in(getBonus) = {{}}
in(deposit) = Real
in(recoverState) = {{}}
```

A função `out` define o conjunto de elementos saída.

```
out(terminate) = {{}}
out(getBonus) = Real
out(deposit) = {{}}
out(recoverState) = Real
```

A função `enable` testa se a operação está habilitada para os valores do estado.

```
enable(terminate)(bo)= true
enable(getBonus)(bo)= true
enable(deposit)(bo)= true
enable(recoverState)(bo)= true
```

A função `effect` executa a operação habilitada pela função enable.

```
    effect(terminate)(bo,_) =
      {({},bo') | bo' <- state, bo'== bo}
    effect(getBonus)(bo,_) =
      {(o,bo') | bo' <- state,bo'== bo, o <- Real, o== bo, bo'==0}
    effect(deposit)(bo,v) =
      {({},bo') | bo' <- state,bo'== bo + (v * bonusRatio)}
    effect(recoverState)(bo,_) =
      {(o,bo') | bo' <- state, bo'== bo, o <- Real, o== bo'}

within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

Apêndice D

Especificação do Processo EHRS em CSP-OZ

Segue a apresentação do processo EHRS, que é um dos principais componentes do EHR_SERVER.

D.1 Tipos de Dados

Vamos começar apresentando o conjunto TOPIC, que possui os assuntos das transações criadas pelos médicos. Estes profissionais interagem com o sistema através de transações, e cada uma delas se refere a algum assunto ou tópico.

TOPIC::= patientIdentity | HCP | HCF | problemList |
familyHistory | contact | testResults | Others

O registro de paciente (EHR) inclui quatro tipos transações: *subject*, cujo tópico pode ser apenas *patientIdentity*, e que contém os dados cadastrais de paciente; *demographic*, que possui transações demográficas, tais como endereços de médicos (HCP) e de hospitais (HCF); *persistent*, que possui transações válidas para toda a vida do paciente, como, por exemplo, a lista de problemas (*problemList*) e a história familiar (*familyHistory*); e *eventual*, que possui transações válidas durante um determinado período, como, por exemplo, resultado de exame (*testResults*) e consulta (*contact*). O conjunto TRANSTYPE contém estes tipos de transações.

TRANSTYPE::= subject | demographic | persistent | eventual

Os motivos de falhas estão representados através do conjunto PURPOSE.

PURPOSE::= RecordRecovered | RecordNotRecovered | WithoutReference

A comunicação entre os sistemas de prontuário eletrônico é feita através de importação e exportação de transações. O conjunto EXT_COMM possui os tipos de comunicação.

EXT_COMM::= imp | exp

A especificação da classe EHRS utiliza uma grande variedade de tipos. Como estamos trabalhando em um nível abstrato, vários desses tipos estão sendo considerados como *given sets* porque não há necessidade de detalhar a sua estrutura interna. Além disso, renomeamos alguns nomes de tipos utilizados no modelo GEHR para não comprometer a legibilidade das classes CSP-OZ. Segue a definição dos *given sets* com os seus significados.

- HCA - conjunto de médicos.
- SRCE - conjunto de identificadores de prontuários.
- GEHR_ARCHID - conjunto de identificadores de arquétipos do sistema GEHR.
- TEXT - conjunto de textos.
- DATE_TIME - conjunto de data e hora.

[HCA, SRCE, DATE_TIME, GEHR_ARCHID, TEXT]

O conjunto EHRID representa os identificadores de registros de paciente.

EHRID == $\mathbb{N}$

Cada elemento do conjunto PERMISS determina o tipo do usuário que tem acesso a uma determinada transação. Segue a descrição dos elementos desse conjunto:

Perm_patient - Paciente
Perm_hcp_legally_responsible - Médico responsável pelo paciente
Perm_hcp_authorising - Médico responsável pela criação da transação
Perm_any_hcp - Qualquer médico
Perm_any_staff - Staff
Perm_anyone - Qualquer pessoa

PERMIS:: = Perm_patient | Perm_hcp_legally_responsible | Perm_hcp_authorising
Perm_any_hcp | Perm_any_staff | Perm_anyone

O conjunto PERMISSET contém subconjuntos do conjunto PERMIS, que definem os possíveis tipos de usuários que têm acesso a uma transação.

| PERMISSET: $\mathbb{P}$ PERMIS

Para facilitar o trabalho dos médicos, algumas organizações de saúde criaram codificadores, como, por exemplo, ICD-10 [43], SNOMED [44], etc, os quais codificam procedimentos médicos, doenças, laudos, materiais, exames, etc. O conjunto CODSYS contém os codificadores utilizados pelos médicos.

CODSYS = {ICD-10, ICPC, SNOMED, UMLS}

Quando se cria uma transação, deve-se informar o conjunto de codificadores utilizados na criação do conteúdo clínico. Esse conjunto é um dos elementos do conjunto CODSYSSET.

| CODSYSSET: $\mathbb{P}$ CODSYS

Como existem várias aplicações (APPLICATION) rodando ao mesmo tempo, para acessá-las é necessário o seu endereço. O conjunto APPLIC contém os endereços dessas aplicações.

APPLIC = {1,2,3}

Cada tipo de transação trabalha com um conjunto de conceitos. Estes conjuntos possuem uma grande quantidade de elementos. Por simplicidade, utilizaremos apenas alguns deles. O conjunto Concept_Subject_Set possui um único elemento, o conceito "patientId", que representa a identificação do paciente. O Concept_Demographic_Set contém "address", que representa endereços de profissionais e de entidades de saúde. O Concept_Persistent_Set possui dois conceitos, "bloodPressure", que representa a pressão sanguínea, e "headache", o sintoma dor de cabeça. Finalmente, o Concept_Eventual_Set possui também dois conceitos, "haemogram", que representa o exame hemograma e "consultation", a consulta médica. Segue a descrição destes conjuntos:

Concept_Subject_Set::= patientId
Concept_Demographic_Set::= address
Concept_Persistent_Set::= bloodPressure | headache
Concept_Eventual_Set::= haemogram | consultation

A especificação utiliza vários tipos que nós descrevemos através de esquemas. Segue a definição de cada um. TRANSD reúne dados necessários para a criação de uma nova transação. Ele possui os seguintes elementos: *sr*, identificador do prontuário fonte; *hca*, médico responsável pela criação da transação; *ehr_id*, identificador do registro de paciente; *tt*, tipo de transação; *ac*, conjunto de usuários que possui acesso a leitura das versões da transação; *am*, conjunto de usuários que possui direito de criação de versões da transação; *rs*, motivo da criação da transação; *gv*, versão do GEHR; *cs*, conjunto de codificadores utilizados na transação; *vid*, identificador da transação.

TRANSD

$sr : SRCE$
$hca : HCA$
$ehr\_id : \mathbb{N}$
$tt : TRANSTYPE$
$ac : PERMISSET$
$am : PERMISSET$
$rs : TEXT$
$gv : \mathbb{N}$
$cs : CODSYSSET$
$vid : DATE\_TIME$

VERSD reúne dados para a criação de uma nova versão de transação. Ele possui os seguintes elementos: *sr*, identificador do prontuário fonte; *hca*, médico responsável pela

criação da versão; *ehr_id*, identificador do registro de paciente; *tt*, tipo de transação; *rs*, motivo da criação da versão; *vid*, identificador da transação; *cs*, conjunto de codificadores utilizados nesta versão.

VERSD

$sr : SRCE$
$hca : HCA$
$ehr\_id : \mathbb{N}$
$tt : TRANSTYPE$
$rs : TEXT$
$vid : DATE\_TIME$
$cs : CODSYSSET$

TRANSIMPD reúne dados de todas as transações que se deseja importar. Ele possui os seguintes elementos: *hca*, médico responsável pela criação da versão ou da transação caso não exista no registro de paciente; *ehr_id*, identificador do registro de paciente; *rs*, motivo da criação da versão ou transação; *ex*, registro de importação; *ts*, lista de transações de outro prontuário para serem selecionadas e importadas pelo médico.

TRANSIMPD

$hca : HCA$
$ehr\_id : \mathbb{N}$
$tp : TOPIC$
$rs : TEXT$
$ex : EHR\_EXT$
$ts : \text{seq } VERS\_TRANS$

TRANSIMPORTED reúne dados de cada transação a ser importada. Ele possui os seguintes elementos: *sr*, identificador do prontuário fonte da informação; *tg*, identificador do prontuário receptor da informação; *hca*, médico responsável pela criação da versão ou da transação; *rs*, motivo da criação da transação; *tr*, transação importada.

TRANSIMPORTED

$sr : SRCE$
$tg : SRCE$
$hca : HCA$
$rs : TEXT$
$tr : VERS\_TRANS$

Além dos tipos citados acima, a especificação do processo EHRS utiliza várias classes que são apresentadas logo a seguir.

D.2 Pacientes e Arquétipos

A classe ARCHETYPED reúne características comuns que devem ser herdadas pelas classes EHR_CONT, ORG_ROOT e DEF_CONT. Ela possui os atributos *concept* e *archetype_id*

que representam o nome do conceito e o identificador do arquétipo relacionado ao conceito, respectivamente. Os parâmetros c e g são utilizados para inicializar as variáveis de estado.

Para não comprometer a legibilidade da especificação, abreviamos alguns nomes de classes e de atributos, como, por exemplo, EHR_CONTENT, ORGANISER_ROOT, DEFINITION_CONTENT e *archetype_id* foram abreviados para EHR_CONT, ORG_ROOT, DEF_CONT e *archId*, respectivamente.

ARCHETYPED(c : *TEXT*; g : *GEHR_ARCHID*)

concept : *TEXT*
archId : *GEHR_ARCHID*

INIT

concept = c
archId = g

A informação de paciente está estruturada em hierarquias de cabeçalhos, chamados de organizadores, os quais fornecem a estrutura navegacional para chegar aos itens de conteúdo (DEF_CONT). A classe ORG_ROOT é a organizadora de nível topo de uma hierarquia de organizadores. Ela é responsável pela organização dos itens de conteúdo (DEF_CONT). Como ORG_ROOT é subclasse de ARCHETYPED, então ela herda os atributos e métodos de ARCHETYPED. No momento, não é necessário o detalhamento dessa classe.

ORG_ROOT

inherit ARCHETYPED
...

Os itens de conteúdo são criados sob os organizadores citados acima. A classe DEF_CONT modela um item de conteúdo que está estruturado em algum formato, como, por exemplo, lista, árvore, tabela, etc. É nesta classe que se encontra as informações de paciente. Como DEF_CONT é subclasse de ARCHETYPED, então herda os atributos e métodos de ARCHETYPED. No momento, não é necessário o detalhamento dessa classe.

DEF_CONT

inherit ARCHETYPED
...

A classe EHR_CONT modela o conteúdo de uma versão de uma transação GEHR. Ela possui os atributos *content* que é o início (raiz) do conteúdo (dados do paciente) e *context*, o contexto da transação. Esta classe também é subclasse de ARCHETYPED. Os parâmetros desta classe servem para inicializar os atributos da própria classe e os herdados.

EHR_CONT(*c* : *TEXT*; *g* : *GEHR_ID*; *cx* : *DEF_CONT*; *ct* : *ORG_ROOT*)

inherit ARCHETYPED

context : *DEF_CONT*
content : *ORG_ROOT*

INIT

context = *cx* ∧ *content* = *ct*

D.3 Auditorias

A classe COMMIT_AUDIT modela a trilha de auditoria de uma transação realizada. Através dela, se sabe por quem, quando, e porque a informação foi adicionada no registro do paciente. Ela possui os seguintes atributos:

- *hca_authorizing*: médico que autoriza a transação.
- *time*: data e hora da criação da transação.
- *ehr_source*: identificador do prontuário eletrônico.
- *reason*: razão de criar a versão.
- *term_set*: conjunto de codificadores utilizado nas criação de transação.

COMMIT_AUDIT(*hca* : *HCA*; *tm* : *DATE_TIME*; *sr* : *SRCE*;
rs : *TEXT*; *cs* : *CODSYSSET*)

hca_authorizing : *HCA*
time : *DATE_TIME*
ehr_source : *SRCE*
reason : *TEXT*
term_set : *CODSYSSET*

INIT

hca_authorizing = *hca* ∧ *time* = *tm*
ehr_source = *sr* ∧ *reason* = *rs*
term_set = *cs*

A classe MERGE_AUDIT modela a trilha de auditoria de uma transação importada. Ela é subclasse de COMMIT_AUDIT e, além dos atributos e métodos herdados, possui os atributos *sender_ehr_source_id*, que é o identificador do prontuário que exportou as transações, e *original_audit*, que é a trilha de auditoria da transação no prontuário de origem.

MERGE_AUDIT(*tg* : *SRCE*; *hca* : *HCA*; *tm* : *DATE_TIME*;
sr : *SRCE*; *rs* : *TEXT*; *au* : *COMMIT_AUDIT*)

inherit *COMMIT_AUDIT*

sender_ehr_source_id : *SRCE*
original_audit : *COMMIT_AUDIT*

INIT

sender_ehr_source_id = *tg* ∧ *original_audit* = *au*

D.4 Transações e Versões de Transações

A classe TRANS modela uma versão de conteúdo de uma transação (VERS_TRANS), que corresponde a ação de um médico entrar com uma nova informação no registro do paciente. A classe VERS_TRANS está definida logo a seguir. Uma nova versão (TRANS) é gerada a partir da criação ou modificação de uma VERS_TRANS. Ela possui os atributos *content* e *audit*, que representam o conteúdo da versão e a informação de auditoria, respectivamente.

TRANS(*cont* : *EHR_CONT*; *aud* : *COMMIT_AUDIT*)

content : *EHR_CONT*
audit : *COMMIT_AUDIT*

INIT

content = *cont* ∧ *audit* = *aud*

A classe VERS_TRANS modela a transação lógica cuja informação existe dentro de versões, que correspondem as diferentes vezes em que a informação foi gerada. Ela é responsável por agrupar e controlar as versões. A classe possui os seguintes atributos:

- *access_rights*: conjunto de usuários que têm direito a leitura das versões da transação.
- *access_amend*: conjunto de usuários que têm direito de criar versões na transação.
- *time_created*: data e hora de criação.
- *gehr_version*: versão do GEHR utilizada pelo prontuário.
- *versions*: lista de versões (TRANS).

Esta classe possui um invariante que garante a existência de pelo menos uma versão (TRANS). Além disso, ela possui o evento *addVersion*, que adiciona uma versão na lista.

$VERS\_TRANS(acc, amd : \mathbb{P}\, PERMIS;\ tm : DATE\_TIME;\ gv : \mathbb{N};\ v1 : TRANS)$

$method\ addVersion : [v? : TRANS]$

$main = addVersion?v \rightarrow SKIP$

$access\_rights : \mathbb{P}\, PERMIS$

$access\_amend : \mathbb{P}\, PERMIS$

$time\_created : DATE\_TIME$

$gehr\_version : \mathbb{N}$

$versions : \text{seq}\ TRANS$

$versions \neq \varnothing$

INIT

$access\_rights' = acc \land access\_amend' = amd$

$time\_created' = tm \land gehr\_version' = gv$

$versions' = \langle v1 \rangle$

effect_addVersion

$\Delta(versions)$

$v? : TRANS$

$versions' = versions \frown \langle v? \rangle$

D.5 Registros

A classe EHR_BASE é a base comum de informação para as classes EHR e EHR_EXT que serão explicadas mais adiante. Ela possui na sua interface vários canais locais (*local_chan*), os quais são utilizados para a comunicação interna. Estes canais não são visíveis pelo ambiente.

O comportamento de EHR_BASE está descrito através de dois fluxos. O primeiro é definido pelo processo que recebe a solicitação de criação de uma transação (*createTrans1*), recupera o tipo da transação (*getTransType*), solicita o próximo tempo (*askForTime*), cria um objeto (c) do tipo COMMIT_AUDIT, que possui as informações de auditoria, como, por exemplo, o médico responsável pelas informações adicionadas na transação. Depois, ele cria um objeto (v) do tipo TRANS (versão da transação), o qual possui como parâmetros os objetos de auditoria (c) e a informação clínica do paciente (d). De acordo com o tipo da transação informado, o processo pode seguir um dos dois fluxos abaixo.

- Se o tipo de transação é igual a demographic ou eventual, ele cria uma transação, adiciona ela no registro (*addTransaction*) e finaliza o processo.
- Se o tipo de transação é igual a persistent, ele solicita a recuperação da transação (*recPersisTrans*). Se encontrar a transação, adiciona a versão na transação (*addVersion*), substitui a transação antiga pela nova (*replaceTrans*) e finaliza o

processo. Por outro lado, se não encontrar a transação, cria uma nova transação, adiciona ela no registro (*addTransaction*) e finaliza o processo.

O segundo fluxo é determinado pelo processo que recebe solicitação de recuperação das transações (permitidas) para que o usuário possa selecionar a desejada e criar uma versão dela (*recTransactions*); recebe solicitação de criação de uma versão (*createVersion1*), solicita o próximo tempo (*askForTime*), cria um objeto (c) do tipo COMMIT_AUDIT, cria um objeto do tipo TRANS, recupera a transação desejada (*getTrans*), adiciona a versão na transação (*addVersion*), substitui a transação antiga pela nova (*replaceTrans*) e finaliza o processo.

$$
\begin{array}{l}
EHR\_BASE(id : \mathbb{N};\ hca : HCA;\ tm : DATE\_TIME;\ sr : SRCE;\ patId : VERS\_TRANS) \\
local\_chan\ getTransType : [tt? : TRANSTYPE] \\
local\_chan\ getTrans : [tr? : VERS\_TRANS] \\
local\_chan\ replaceTrans : [t : VERS\_TRANS;\ tt : TRANSTYPE] \\
local\_chan\ addTransaction : [t : VERS\_TRANS;\ tt : TRANSTYPE] \\
local\_chan\ recPersisTrans : [d : EHR\_CONT;\ str? : \text{seq}\ VERS\_TRANS] \\
method\ createVersion1 : [d? : EHR\_CONT;\ cn? : VERSD] \\
method\ createTrans1 : [d? : EHR\_CONT;\ cn? : TRANSD] \\
method\ recTransactions : [perm? : PERMISS;\ trs! : \text{seq}\ VERS\_TRANS] \\
chan\ askForTime : [tm? : DATE\_TIME] \\
main = createTrans1?d?cn \rightarrow getTransType?tt \rightarrow askForTime?tm \rightarrow \\
\quad New\ c : COMMIT\_AUDIT(cn.hca, tm, cn.sr, cn.rs, cn.cs) \bullet \\
\quad New\ v : TRANS(d, c) \bullet \\
\quad if\ ((tt = demographic) \lor\ (tt = eventual)) \\
\quad then\ New\ t : VERS\_TRANS(cn.ac, cn.am, tm, cn.gv, v) \bullet \\
\quad\quad addTransaction.t.tt \rightarrow SKIP \\
\quad else\ recPersisTrans.d?str \rightarrow \\
\quad\quad (if\ not\ null(str) \\
\quad\quad then\ (head\ str \\
\quad\quad\quad [|\ \{|\ addVersion\ |\}\ |] \\
\quad\quad\quad (addVersion!v \rightarrow replaceTrans!(head\ str)!tt \rightarrow SKIP)) \\
\quad\quad else\ New\ t : VERS\_TRANS(cn.ac, cn.am, tm, cn.gv, v) \bullet \\
\quad\quad\quad (addTransaction.t.tt \rightarrow SKIP)) \\
\quad \Box \\
\quad recTransactions?perm!str \rightarrow createVersion1?d?cn \rightarrow askForTime?tm \rightarrow \\
\quad New\ c : COMMIT\_AUDIT(cn.hca, tm, cn.sr, cn.rs, cn.cs) \bullet \\
\quad New\ v : TRANS(d, c) \bullet \\
\quad (getTrans?tr \rightarrow \\
\quad\quad (tr \\
\quad\quad [|\ \{|\ addVersion\ |\}\ |] \\
\quad\quad (addVersion!v \rightarrow replaceTrans.tr.tt \rightarrow SKIP)))
\end{array}
$$

A classe EHR_BASE possui os seguintes atributos:

- *ehr_id*: identificador do registro de paciente.
- *hcp_created_by*: médico que criou o registro de paciente.

- *creation_time*: data e hora da criação do registro de paciente.
- *ehr_source*: identificador do prontuário eletrônico.
- *subject*: transação de cadastro de paciente.
- *demographic_entities*: seqüência de transações demográficas (endereços de médicos e instituições de saúde relacionados ao paciente). Na especificação estamos utilizando o nome *dem_entities*.
- *persistent_clinical_transactions*: seqüência de transações clínicas e persistentes do paciente. Na especificação estamos utilizando o nome *pers_clin_trans*.
- *event_clinical_transactions*: seqüência de transações clínicas e eventuais do paciente. Na especificação estamos utilizando o nome *event_clin_trans*.
- *tt*: tipo de transação do registro em que será adicionada a transação.
- *vid*: identificador da transação.

Esta classe possui um invariante que garante que o atributo *subject* só possui uma transação de conceito *patientId*; o *dem_entities* contém somente transações de conceito *address*; o *pers_cli_trans* possui apenas transações de conceitos *bloodPressure* e *headache*; o *event_cli_trans* contém somente transações de conceitos *haemogram* e *consultation*.

$$
\begin{array}{|l}
\textit{EHR\_BASE}(\textit{id} : \mathbb{N};\ \textit{hca} : \textit{HCA};\ \textit{tm} : \textit{DATE\_TIME};\ \textit{sr} : \textit{SRCE};\ \textit{patId} : \textit{VERS\_TRANS}) \\
\hline
\textit{ehr\_id} : \mathbb{N} \\
\textit{hcp\_created\_by} : \textit{HCA} \\
\textit{creation\_time} : \textit{DATE\_TIME} \\
\textit{ehr\_source} : \textit{SRCE} \\
\textit{subject} : \textit{VERS\_TRANS} \\
\textit{dem\_entities} : \text{seq } \textit{VERS\_TRANS} \\
\textit{pers\_clin\_trans} : \text{seq } \textit{VERS\_TRANS} \\
\textit{event\_clin\_trans} : \text{seq } \textit{VERS\_TRANS} \\
\textit{tt} : \textit{TRANSTYPE} \\
\textit{vid} : \textit{DATE\_TIME} \\
\hline
(\forall\, t : \textit{VERS\_TRANS} \mid t = \textit{subject} \bullet \textit{last}\ (t.\textit{versions}).\textit{concept} \in \textit{Concept\_Subject\_Set}) \\
(\forall\, t : \textit{ran}\ \textit{dem\_entities} \bullet \textit{last}\ (t.\textit{versions}).\textit{concept} \in \textit{Concept\_Demographic\_Set}) \\
(\forall\, t : \textit{ran}\ \textit{pers\_cli\_trans} \bullet \textit{last}\ (t.\textit{versions}).\textit{concept} \in \textit{Concept\_Persistent\_Set}) \\
(\forall\, t : \textit{ran}\ \textit{event\_cli\_trans} \bullet \textit{last}\ (t.\textit{versions}).\textit{concept} \in \textit{Concept\_Eventual\_Set}) \\
\hline
\textit{INIT} \\
\hline
\textit{ehr\_id} = \textit{id} \land \textit{hcp\_created\_by} = \textit{hca} \land \textit{creation\_time} = \textit{tm} \\
\textit{ehr\_source} = \textit{sr} \land \textit{subject} = \textit{patId} \land \textit{dem\_entities} = \varnothing \\
\textit{pers\_clin\_trans} = \varnothing \land \textit{event\_clin\_trans} = \varnothing \\
\hline
\end{array}
$$

Segue a descrição de cada operação da classe EHR_BASE.

- *effect_createTrans1*: recebe como parâmetro de entrada os dados do paciente (*d?*) e as informações necessárias para criar a transação desejada (*cn?*). Essa operação atualiza apenas a variável de estado *tt*. O restante dos valores de entrada são utilizados pela parte de CSP para criar a transação (VERS_TRANS).
- *effect_getTransType*: retorna o tipo da transação e não atualiza o estado.
- *effect_addTransaction*: adiciona a transação *t* na lista de transações de tipo *tt*.
- *effect_recPersisTrans*: verifica, na lista de transações persistentes, a existência de uma determinada transação, a partir de informações do paciente (*d*). Se encontrar, retorna a transação numa lista, senão, retorna a lista vazia (*str?*).
- *effect_replaceTrans*: substitui a transação antiga pela nova de acordo com o tipo da transação informado (*tt*).
- *effect_getTrans*: recupera a transação a partir do seu tipo de transação (*tt*).
- *effect_recTransactions*: recebe como parâmetro de entrada o tipo de usuário que deseja ter acesso as transações, e retorna uma lista das transações que podem ser alteradas (criar nova versão).
- *effect_createVersion1*: recebe como parâmetros de entrada os dados do paciente (*d?*) e as informações necessárias para criar a versão (*cn?*). A operação atualiza apenas as variáveis de estado tipo da transação (*tt*) e identificador da transação (*vid*). Os outros dados de entrada são utilizados pela parte de CSP para criar a nova versão (TRANS).

EHR_BASE(*id* : ℕ; *hca* : *HCA*; *tm* : *DATE_TIME*; *sr* : *SRCE*; *patId* : *VERS_TRANS*)

effect_createTrans1

$\Delta(tt)$
$d? : EHR\_CONT$
$cn? : TRANSD$

$tt' = cn?.tt$

effect_getTransType

$\Delta()$
$tt! : TRANSTYPE$

$tt! = tt$

effect_recPersisTrans

$\Delta()$
$d : EHR\_CONT$
$str? : \text{seq } VERS\_TRANS$

$(\exists\, t : \text{ran } pers\_clin\_trans \mid head\ (t.versions).content.concept = d.concept$
$str? =< t >) \lor$
$(\neg\ \exists\, t : \text{ran } pers\_clin\_trans \mid head\ (t.versions).content.concept = d?.concept$
$str? =<>)$

effect_getTrans

$\Delta()$

$tr? : VERS\_TRANS$

$((tt = subject \Rightarrow tr? = subject) \lor$
$(tt = demographic \Rightarrow \exists\, t : VERS\_TRANS \mid$
$t \in ran\ dem\_entities \land t.time\_created = vid \bullet tr? = t) \lor$
$(tt = persistent \Rightarrow \exists\, t : VERS\_TRANS \mid$
$t \in ran\ pers\_cli\_trans \land t.time\_created = vid \bullet tr? = t) \lor$
$(tt = eventual \Rightarrow \exists\, t : VERS\_TRANS \mid$
$t \in ran\ event\_cli\_trans \land t.time\_created = vid \bullet tr? = t))$

effect_createVersion1

$\Delta(tt, vid)$

$d? : EHR\_CONT$

$cn? : VERSD$

$tt' = cn?.tt$
$vid' = cn?.vid$

effect_addTransaction

$\Delta(subject, dem\_entities, pers\_clin\_trans, event\_clin\_trans)$

$t : VERS\_TRANS$

$tt : TRANSTYPE$

$tt = subject \Rightarrow (subject' = t \land dem\_entities' = dem\_entities$
$pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans) \lor$
$tt = demographic \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities \frown \langle t \rangle$
$pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans) \lor$
$tt = persistent \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities$
$pers\_clin\_trans' = pers\_clin\_trans \frown \langle t \rangle \land event\_clin\_trans' = event\_clin\_trans) \lor$
$tt = eventual \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities$
$pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans \frown \langle t \rangle)$

effect_recTransactions

$\Delta()$

$perm? : PERMISS$

$trs! : \text{seq}\ VERS\_TRANS$

$\exists\, tmp : \text{seq}\ VERS\_TRANS \bullet (perm? \in subject.access\_amend \Rightarrow tmp = subject)$
$\forall\, t : VERS\_TRANS \mid t \in ran\ dem\_entities \bullet$
$(perm? \in t.access\_amend \Rightarrow tmp = tmp \frown \langle t \rangle)$
$\forall\, t : VERS\_TRANS \mid t \in ran\ pers\_cli\_trans \bullet$
$(perm? \in t.access\_amend \Rightarrow tmp = tmp \frown \langle t \rangle)$
$\forall\, t : VERS\_TRANS \mid t \in ran\ event\_cli\_trans \bullet$
$(perm? \in t.access\_amend \Rightarrow tmp = tmp \frown \langle t \rangle) \land trs! = tmp$

$$
\begin{array}{|l}
\textit{effect\_replaceTrans} \\
\Delta(subject, dem\_entities, pers\_clin\_trans, event\_clin\_trans) \\
tr : VERS\_TRANS \\
tt : TRANSTYPE \\
\hline
tt = subject \Rightarrow (subject' = tr \land dem\_entities' = ehr.dem\_entities \\
pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans) \lor \\
tt = demographic \Rightarrow (subject' = subject \land \\
(\exists\, i : dom\ dem\_entities \mid dem\_entities(i).time\_created = tr.time\_created \bullet \\
dem\_entities' = dem\_entities \oplus \{i \mapsto tr\}) \land \\
pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans) \lor \\
tt = persistent \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities \\
(\exists\, i : dom\ pers\_clin\_trans \mid pers\_clin\_trans(i).time\_created = tr.time\_created \bullet \\
pers\_clin\_trans' = pers\_clin\_trans \oplus \{i \mapsto tr\}) \\
event\_clin\_trans' = event\_clin\_trans) \lor \\
tt = eventual \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities \\
pers\_clin\_trans' = pers\_clin\_trans \\
(\exists\, i : dom\ event\_clin\_trans \mid event\_clin\_trans(i).time\_created = tr.time\_created \bullet \\
event\_clin\_trans' = event\_clin\_trans \oplus \{i \mapsto tr\})
\end{array}
$$

A classe EHR modela um registro eletrônico de paciente. Após a declaração dos canais, encontramos três funções: *proj*, que aplicada a uma transação com todas as versões, retorna uma transação com apenas a última versão; *map*, que recebe uma função e aplica ela em todos os elementos da seqüência; e *topicTransType*, que aplicada a um tópico (assunto) retorna o tipo da transação.

Esta classe possui o comportamento herdado de EHR_BASE e o seu próprio, o qual fornece três opções de fluxos. O primeiro é definido pelo processo que recupera todas as transações de um determinado assunto (tópico), que são compatíveis com a versão do GEHR utilizada pelo sistema de prontuário solicitante (*recTransTopic*). Em seguida, adiciona o registro de importação e exportação (*addEhrExtract*) e finaliza o processo.

O segundo fluxo é determinado pelo processo que avisa ao EHRS (ver a seguir) que todas as suas transações de tópico *tp* possuem a versão do GEHR incompatível com a do prontuário solicitante (*incompVers*) e finaliza o processo.

O terceiro fluxo é definido pelo processo que recebe a solicitação de importação das transações de outro prontuário (*createTransImport1*) e passa a se comportar como IMP_TRANSS, que é um processo recursivo. O objetivo desse processo recursivo é de incluir no registro de paciente todas as transações recebidas em uma lista. O seu fluxo funciona da seguinte forma: enquanto a lista não estiver vazia (*seqNotNull*), solicita o próximo tempo (*askForTime*), recupera os dados da transação que se encontra na cabeça da lista (*getImpData*) e cria os objetos m do tipo MERGE_AUDIT, que possui as informações de auditoria, e do tipo EHR_CONT, que contém as informações do paciente, v do tipo TRANS, que é formado pelos objetos e e m, e finalmente t do tipo VERS_TRANS, que possui informações de direitos de acesso e a versão v.

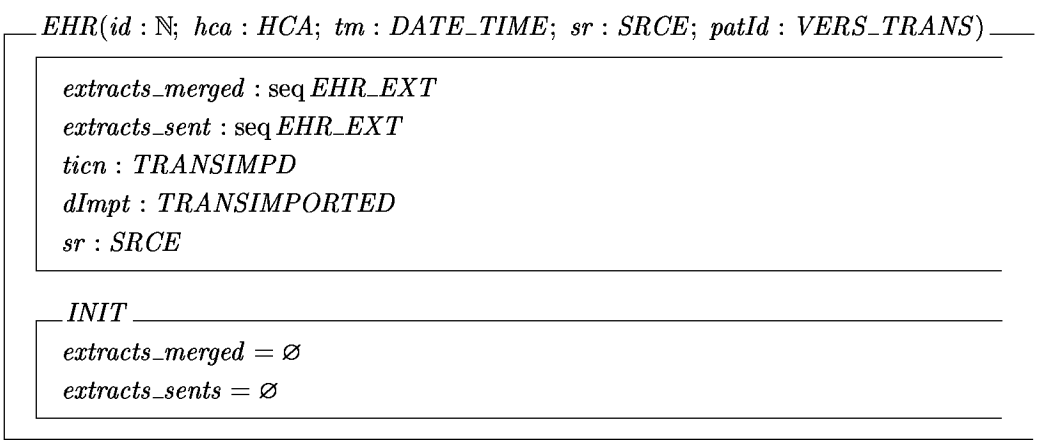
Após a criação dos objetos, o processo IMP_TRANSS, recupera o tipo da transação (*getTransType*), adiciona a transação no registro (*addTransation*), elimina a transação

que se encontra na cabeça da lista (*nextSeq*) e volta a se comportar como IMP_TRANSS. Quando a lista ficar vazia (*seqNull*), o processo adiciona o registro de importação e exportação no registro de paciente (*addEhrExtract*) e finaliza o processo.

$$
\begin{array}{l}
EHR(id : \mathbb{N};\ hca : HCA;\ tm : DATE\_TIME;\ sr : SRCE;\ patId : VERS\_TRANS) \\
inherit\ EHR\_BASE \\
method\ addEhrExtract : [ehrExt? : EHR\_EXT;\ comm? : EXT\_COMM] \\
method\ recTransTopic : [tp? : TOPIC;\ gv? : GEHR\_VERSION;\ str! : \mathrm{seq}\ VERS\_TRANS] \\
method\ incompVers, seqNull, seqNotNull, nextSeq : [] \\
method\ createTransImport1 : [sr? : SRCE;\ cn? : TRANSIMPD] \\
method\ getImpData : [dimpt? : TRANSIMPORTED] \\
\hline
proj : VERS\_TRANS \mapsto VERS\_TRANS \\
\hline
\forall t : VERS\_TRANS \bullet \\
proj(t).access\_rights = t.access\_rights \\
proj(t).access\_amend = t.access\_amend \\
proj(t).time\_created = t.time\_created \\
proj(t).gehr\_version = t.gehr\_version \\
proj(t).versions = \langle last\ t.versions \rangle \\
\hline
map : (X \mapsto Y) \mapsto \mathrm{seq}\ X \mapsto \mathrm{seq}\ Y \\
\hline
\forall f : X \mapsto Y, s : \mathrm{seq}\ X \bullet \\
\forall i : 1..\#s \bullet (map\ f\ s)(i) = f(s(i)) \\
\hline
topicTransType : TOPIC \mapsto TRANSTYPE \\
\hline
\forall tp : TOPIC, tt : TRANSTYPE \bullet topicTransType(tp) = tt \\
\hline
main = recTransTopic?tp?gv?!str \rightarrow addEhrExtract?ehrExt?comm \rightarrow SKIP \\
\qquad \Box \\
\qquad incompVers?tp?gv \rightarrow SKIP \\
\qquad \Box \\
\qquad createTransImport1?sr?cn \rightarrow IMP\_TRANSS \\
IMP\_TRANSS = \\
seqNotNull \rightarrow \\
\quad (askForTime?tm \rightarrow getImpData?dImp \rightarrow \\
\quad New\ m : MERGE\_AUDIT(dImp.tg, dImp.hca, tm, dImp.sr, dImp.rs, \\
\quad\ head(dImp.tr.versions).audit) \bullet \\
\quad New\ e : EHR\_CONT(head(dImp.tr.versions).content.concept, \\
\quad\ head(dImp.tr.versions).content.archId, \\
\quad\ head(dImp.tr.versions).content.content, \\
\quad\ head(dImp.tr.versions).content.context) \bullet \\
\quad New\ v : TRANS(e, m) \bullet \\
\quad New\ t : VERS\_TRANS(dImp.tr.access\_rights, dImp.tr.amend\_rights, tm, v) \bullet \\
\quad getTransType?tt \rightarrow addTransaction.t.tt \rightarrow nextSeq \rightarrow IMP\_TRANSS) \\
\Box \\
seqNull \rightarrow addEhrExtract?ehrExt?comm \rightarrow SKIP
\end{array}
$$

Além dos atributos herdados da classe EHR_BASE, EHR possui:

- *extracts_merged*: seqüência de todos os registros importados de outros prontuários.
- *extracts_sent*: seqüência de todos os registros exportados para outros prontuários.
- *ticn*: possui todas as transações que devem ser incluídas (importadas) no registro de paciente.
- *dimpt*: possui as informações necessárias para criar uma transação importada.
- *sr*: identificador do prontuário que se deseja importadas as transações.



Segue a descrição das operações da classe EHR.

- *effect_createTransImport1*: recebe como parâmetros de entrada o identificador do prontuário fonte das transações a serem importadas(*sr?*) e um registro com todas as transações (*cn?*). Essa operação atualiza as variáveis de estado prontuário fonte (*sr*), tipo de transação (*tt*) e registro com as informações das transações a serem importadas (*ticn*).
- enable_incompVers: a operação *incompVers* só possui o esquema enable porque nem atualiza estado e nem gera alguma saída. Ela recebe como parâmetros de entrada o tópico da transação (*tp?*) e a versão do GEHR utilizada pelo prontuário solicitante (*gv?*). O esquema enable só habilita esta operação se não existir nenhuma transação de tópico *tp?* que utiliza a mesma versão do GEHR do prontuário solicitante.
- *effect_addEhrExtract*: recebe como parâmetro de entrada o tipo da comunicação (*comm?*) e o registro de exportação e importação (*ehrExt?*). Se o tipo de comunicação é importação (imp), o registro de importação e exportação é adicionado na lista *extract_merged*, senão é adicionado na lista *extract_sents*.
- *effect_seqNotNull*: atualiza algumas variáveis de estado que são utilizadas pelas operações posteriores. A operação *seqNotNull* só fica habilitada enquanto houver transações na lista *ticn.ts*.

- enable_seqNull: a operação *seqNull* só possui o esquema enable porque nem altera estado e nem gera alguma saída. Ela só pode ser habilitada quando a lista de transações estiver vazia (*ticn.ts*).

- *effect_nextSeq*: retira a transação da cabeça da lista e atualiza a variável de estado *ticn* com o restante da lista.

- *effect_getImpData*: recupera os dados de uma transação para ser criada (importada) pela parte de CSP, através do construtor *New*.

- *effect_recTransTopic*: recebe como parâmetros de entrada o tópico da transação (*tp?*) e a versão do GEHR utilizada pelo prontuário solicitante (*gv?*). Essa operação gera como saída uma lista de transações que possuem tópico *tp?* e que utilizam a mesma versão do GEHR. As transações que participam da lista só possuem a última versão. O esquema *enable_recTransTopic* só habilita a operação se existir pelo menos uma transação de tópico *tp?* que utiliza a mesma versão do GEHR do prontuário solicitante.

$EHR(id : \mathbb{N};\ hca : HCA;\ tm : DATE\_TIME;\ sr : SRCE;\ patId : VERS\_TRANS)$

effect_createTransImport1

$\Delta()$
$sr? : SRCE$
$cn? : TRANSIMPD$

$sr' = sr? \land ticn' = cn?$
$tt' = topicTransType(cn?.tp)$

effect_getImpData

$\Delta()$
$dImp? : TRANSIMPORTED$

$dImp? = dImpt$

enable_incompVers

$tp? : TOPIC$
$gv? : GEHR\_VERSION$

$(tp? = patientIdentity \Rightarrow last(subject.versions).content.concept = tp? \land$
$subject.gehr\_version \neq gv?) \lor$
$(tp? = HCP \lor tp? = HCF \Rightarrow$
$\forall t : \text{ran } dem\_entities$
$\mid last(t.versions).content.concept = tp? \bullet t.gerh\_version \neq gv?) \lor$
$(tp? = familyHistory \lor tp? = problemList \Rightarrow$
$\forall t : \text{ran } pers\_clin\_trans$
$\mid last(t.versions).content.concept = tp? \bullet t.gerh\_version \neq gv?) \lor$
$(tp? = contact \lor tp? = testResults \Rightarrow$
$\forall t : \text{ran } event\_clin\_trans$
$\mid last(t.versions).content.concept = tp? \bullet t.gerh\_version \neq gv?)$

effect_addEhrExtract

$\Delta(extract\_merged, extract\_sents)$
$comm? : EXT\_COMM$
$ehrExt? : EHR\_EXT$

$comm? = imp \Rightarrow extract\_merged' = extract\_merged \frown ehrExt?$
$comm? = exp \Rightarrow extract\_sents' = extract\_sents \frown ehrExt?$

enable_recTransTopic

$tp? : TOPIC$
$gv? : GEHR\_VERSION$
$trseq! : \text{seq } VERS\_TRANS$

$(tp? = patientIdentity \Rightarrow last(subject.versions).content.concept = tp?$
$subject.gehr\_version = gv?) \lor$
$(tp? = HCP \lor tp? = HCF \Rightarrow$
$\exists t : \text{ran } dem\_entities \mid last(t.versions).content.concept = tp?$
$\land\ t.gehr\_version = gv?) \lor$
$(tp? = familyHistory \lor tp? = problemList \Rightarrow$
$\exists t : \text{ran } pers\_cli\_trans \mid last(t.versions).content.concept = tp?$
$\land\ t.gehr\_version = gv?) \lor$
$(tp? = contact \lor tp? = testResults \Rightarrow$
$\exists t : \text{ran } event\_cli\_trans \mid last(t.versions).content.concept = tp?$
$\land\ t.gehr\_version = gv?)$

effect_recTransTopic

$\Delta()$
$tp? : TOPIC$
$gv? : GEHR\_VERSION$
$trseq! : \text{seq } VERS\_TRANS$

$\exists tmptrseq : \text{seq } VERS\_TRANS \bullet$
$((tp? = patientIdentity \land last(subject.versions).content.concept = tp? \land$
$subject.gehr\_version = gv?) \Rightarrow tmptrseq = \langle proj(subject)\rangle) \lor$
$(tp? = HCP \lor tp? = HCF \Rightarrow \exists x \bullet x = dem\_entities\upharpoonright$
$\{t \mid t : \text{ran } dem\_entities \bullet last(t.versions).content.concept = tp?$
$\land\ t.gehr\_version = gv?\} \land tmptrseq = map\ proj\ x) \lor$
$(tp? = familyHistory \lor tp? = problemList \Rightarrow$
$\exists x \bullet x = pers\_cli\_trans\upharpoonright$
$\{t \mid t : \text{ran } pers\_cli\_trans \bullet last(t.versions).content.concept = tp?$
$\land\ t.gehr\_version = gv?\} \land tmptrseq = map\ proj\ x) \lor$
$(tp? = contact \lor tp? = testResults \Rightarrow$
$\exists x \bullet x = event\_cli\_trans\upharpoonright$
$\{t \mid t : \text{ran } event\_cli\_trans \bullet last(t.versions).content.concept = tp?$
$\land\ t.gehr\_version = gv?\} \land tmptrseq = map\ proj\ x)) \land$
$tmptrseq = tmptrseq \frown \langle proj(subject)\rangle \land trseq! = tmptrseq$

enable_seqNull

$ticn.ts = \langle\rangle$

effect_nextSeq

$\Delta(ticn)$

$ticn'.ts = tail\ ticn.ts$

enable_seqNotNull

$ticn.ts \neq \langle\rangle$

effect_seqNotNull

$\Delta(dImpt)$

$dImpt'.sr = sr$
$dImpt'.hca = ticn.hca$
$dImpt'.tg = ehr\_source$
$dImpt'.rs = ticn.rs$
$dImpt'.tr = head\ \ ticn.ts$

A classe EHR_EXT modela o registro de importação e exportação. Ela possui uma estrutura semelhante a da classe EHR. Além dos atributos herdados da classe EHR_BASE, possui *destination_ehr_source*, que é o identificador do prontuário fornecedor do registro de importação, e *hcp_authorising_acquisition*, o médico que autoriza o recebimento do registro.

EHR_EXT($id : \mathbb{N}$; $hca : HCA$; $tm : DATE\_TIME$; $sr : SRCE$; $tg : SRCE$; $hcatg : HCA$; $patId : VERS\_TRANS$)
inherit EHR_BASE

$destination\_ehr\_source : SRCE$
$hcp\_authorising\_acquisition : HCA$

INIT

$destination\_ehr\_source = tg$
$hcp\_authorising\_acquisition = hcatg$

A classe EHRS é quem possui a coleção de registros de pacientes. Ela é responsável por:

- recuperar registros de pacientes;
- solicitar a criação de transações e versões de transações;
- requisitar a criação de transações importadas;
- solicitar transações para serem exportadas;

Após a declaração dos canais, encontramos o *main* que é formado por uma escolha externa de quatro processos: CREATE_TRANS, CREATE_VERSION, EXPORT_TRANS e IMPORT_TRANS.

$$
\begin{array}{l}
EHRS(sqId : \mathrm{seq}\, EHRID;\ sr : SRCE) \\
method\ notGetEhr, notGetIndObj, recNotRecvd : [] \\
method\ getIndObj : [r? : EHRID] \\
method\ updateEhr, addEhr : [ehr : EHR] \\
method\ getEhr, recRecvd : [ehr? : EHR] \\
method\ getTransType : [tt? : TRANSTYPE] \\
chan\ createTrans : [app? : APPLIC;\ d? : EHR\_CONT;\ cn? : TRANSD] \\
chan\ createTrans1 : [d! : EHR\_CONT;\ cn! : TRANSD] \\
chan\ askForTime : [tm? : DATE\_TIME] \\
chan\ failEhr : [app! : APPLIC;\ p! : PURPOSE] \\
chan\ createTransImport : [app? : APPLIC;\ sr? : SRCE;\ cn? : TRANSIMPD] \\
chan\ createTransImport1 : [sr! : SRCE;\ cn! : TRANSIMPD] \\
chan\ selectTrans : [pid? : EHR\_CONT;\ tp? : TOPIC;\ gv? : \mathbb{N}] \\
chan\ transRecvd : [trseq! : \mathrm{seq}\ VERS\_TRANS] \\
chan\ extExported : [ehrExt? : EHR\_EXT] \\
chan\ createVersion : [app? : APPLIC;\ ehrid? : EHRID;\ perm? : PERMISS] \\
chan\ createVersion1 : [d! : EHR\_CONT;\ cn! : VERSD] \\
chan\ transNotRecvd, incompVers, incompGehrVers : [] \\
chan\ addEhrExtract : [ehrExt! : EHR\_EXT;\ comm! : EXT\_COMM] \\
chan\ recTransactions : [trs? : \mathrm{seq}\ VERS\_TRANS] \\
chan\ showTrans : [app! : APPLIC;\ trs! \mathrm{seq}\ VERS\_TRANS] \\
chan\ recVersData : [app : APPLIC;\ d? : EHR\_CONT;\ cn? : VERSD] \\
chan\ addTransaction : [t! : VERS\_TRANS;\ tt! : TRANSTYPE] \\
main = CREATE\_TRANS \\
\qquad \Box \\
\qquad CREATE\_VERSION \\
\qquad \Box \\
\qquad EXPORT\_TRANS \\
\qquad \Box \\
\qquad IMPORT\_TRANS
\end{array}
$$

O processo CREATE_TRANS é responsável pela criação de transação. Ele recebe a solicitação de criação de uma transação (*createTrans*), recupera o tipo da transação (*getTransType*) e segue um dos dois fluxos:

1. Recupera o registro do paciente (*getEhr*). Se o tipo da transação for igual a *subject*, o processo avisa que o registro já existe e por isso não é possível criar essa transação (*failEhr*), e volta a se comportar como *main*. Por outro lado, se o tipo da transação for diferente de *subject*, o processo CREATE_TRANS solicita ao processo *ehr* a criação da transação (*createTrans1*), atualiza a sua coleção de registros (*updateEhr*), finaliza o processo, e volta a se comportar como *main*.

2. Não recupera o registro (*notGetEhr*). Se o tipo da transação for igual a *subject*, o processo pode seguir um dos dois caminhos:

   - Consegue um próximo índice disponível (*getIndObj*), solicita o próximo tempo (*askForTime*), cria os objetos necessários para criar o objeto *e* do tipo EHR,

adiciona o registro na coleção de registros (*addEhr*), e volta a se comportar como *main*.

- Não consegue um próximo índice disponível (*notGetIndObj*), avisa que não foi possível criar um registro (*failEhr*), e volta a se comportar como *main*.

Por outro lado, se o tipo da transação não for igual a *subject*, o processo avisa que não foi possível criar a transação porque o registro do paciente não existe (*failEhr*), e volta a se comportar como *main*.

```
EHRS(sqId : seq EHRID; sr : SRCE)
CREATE_TRANS = createTrans?app?d?cn → getTransType?tt →
(getEhr?ehr →
  if (tt == subject)
  then failEhr!app!p → main
  else
    ((ehr
    [| {| createTrans1 |} |]
    (createTrans1!d!cn → update.ehr → SKIP)); main)
□
notGetEhr →
  (if(tt == subject)
  then(getIndObj?ind → askForTime?tm →
          (New c : COMMIT_AUDIT(cn.hca, tm, cn.sr, cn.rs, cn.cs) •
          New v : TRANS(d, c) •
          New t : VERS_TRANS(cn.ac, cn.am, tm, cn.gv, v) •
          New e : EHR(ind, cn.hca, tm, cn.sr, t) •
          addEhr.e → main)
        □
        notGetIndObj → failEhr!app!p → main)
  else failEhr!app!p → main))
```

O processo CREATE_VERSION é responsável pela criação de versão. Ele recebe a solicitação de criação de versão (*createVersion*) e segue um dos dois fluxos:

1. Recupera o registro (*getEhr*), solicita ao *ehr* as transações permitidas para o tipo de usuário *perm* (recTransactions), envia as transações para a aplicação selecionar a desejada (*showTrans*), recebe da aplicação a nova versão da transação selecionada (*recVersData*), solicita ao registro a criação da nova versão (*createVersion1*), atualiza a coleção de registros (*updateEhr*), finaliza o processo, e volta a se comportar como *main*.

2. Não recupera o registro (*notGetEhr*), avisa que o registro não existe (*failEhr*), e volta a se comportar como *main*.

$EHRS(sqId : \text{seq}\, EHRID;\ sr : SRCE)$

$CREATE\_VERSION = createVersion?app?ehrid?perm \rightarrow$
$(getEhr?ehr \rightarrow$
$(ehr$
$[|\ \{|\ recTransactions, createVersion1\ |\}\ |]$
$(recTransactions!perm?trs \rightarrow showTrans!app!trs \rightarrow recVersData.app?d?cn \rightarrow$
$createVersion1!d!cn \rightarrow update.ehr \rightarrow SKIP));\ main$
$\Box$
$notGetEhr \rightarrow failEhr!app!p \rightarrow main)$

O processo EXPORT_TRANS é responsável pela seleção das transações a serem exportadas. Ele recebe solicitação de exportação de transações (*selectTrans*) e segue um dos dois fluxos:

1. Recupera o registro do paciente a partir do *pid*, que são informações cadastrais (*recRecvd*). Dependendo do assunto e da versão do GEHR recebidos, o processo pode seguir um dos dois caminhos:
   - Recebe informação de que não existem transações de um determinado tópico que utilizam a mesma versão do GEHR do prontuário solicitante (*incompVers*); avisa ao processo EHRS_EXTRACT que as versões são incompatíveis (*incompGehrVers*), finaliza o processo, e volta a se comportar como *main*.
   - Recebe transações de tópico informado que utilizam a mesma versão do GEHR do prontuário solicitante (*recTransTopic*), repassa as transações para o processo EHRS_EXTRACT (*transRecvd*), recebe o registro de exportação (*extExported*), adiciona o registro de exportação no registro do paciente (*addEhrExtract*), atualiza a coleção de registros (*updateEhr*), finaliza o processo, e volta a se comportar como *main*.
2. Não recupera o registro do paciente a partir do *pid* (*recNotRecvd*), avisa ao processo EHRS_EXTRACT que não pode recuperar as transações (*transNotRecvd*), e volta a se comportar como *main*.

$EHRS(sqId : \text{seq}\, EHRID;\ sr : SRCE)$

$EXPORT\_TRANS = selectTrans?pid?tp?gv \rightarrow$
$(recRecvd?ehr \rightarrow$
$(ehr$
$[|\ \{|\ incompVers, recTransTopic, addEhrExtract\ |\}\ |]$
$((incompVers \rightarrow incompGehrVers \rightarrow SKIP)$
$\Box$
$(recTransTopic!tp!gv?str \rightarrow transRecvd!str \rightarrow extExported?ehrExt \rightarrow$
$addEhrExtract!ehrExt!comm \rightarrow updateEhr.ehr \rightarrow SKIP)));\ main$
$\Box$
$recNotRecvd \rightarrow transNotRecvd \rightarrow main)$

O processo IMPORT_TRANS é responsável pela solicitação de criação das transações importadas. Ele recebe a solicitação de criação de transações (*createTransImport*) e segue um dos dois fluxos:

1. Recupera o registro do paciente (*getEhr*), solicita a criação das transações (*createTransImport1*) e a inclusão do registro de importação e exportação (*addEhrExtract*); atualiza a coleção de registros (*updateEhr*), finaliza o processo, e volta a se comportar como *main*.

2. Não recupera o registro (*notGetEhr*), avisa que o registro não existe (*failEhr*) e volta a se comportar como *main*.

$$
\begin{array}{l}
EHRS(sqId : \mathrm{seq}\, EHRID;\ sr : SRCE) \\
IMPORT\_TRANS = createTransImport?app?sr?cn \rightarrow \\
\quad (getEhr?ehr \rightarrow \\
\quad\quad (ehr \\
\quad\quad [|\ \{|\ createTransImport1, addEhrExtract\ |\}\ |] \\
\quad\quad (createTransImport1!sr!cn \rightarrow addEhrExtract!ehrExt!comm \rightarrow \\
\quad\quad\quad updateEhr.ehr \rightarrow SKIP));\ main \\
\quad \Box \\
\quad notGetEhr \rightarrow failEhr!app!p \rightarrow main)
\end{array}
$$

A classe EHRS possui os seguintes atributos:

- *ehrs*: coleção de registros de paciente.
- *ehr_id*: identificador do registro.
- *app*: endereço da aplicação.
- *tt*: tipo de transação.
- *s*: identificador do prontuário local.
- *p*: motivo da falha.
- *pid*: informações de identificação do paciente.
- *ehrExt*: registro de importação e exportação.
- *comm*: tipo de comunicação (importação ou exportação)

Além dos atributos, a classe possui um invariante que garante a unicidade dos identificadores de registros de paciente.

$EHRS(sqId : \text{seq}\, EHRID;\ sr : SRCE)$

$ehrs : \mathbb{P}\, EHR$
$seqIds : \text{seq}\, EHRID$
$app : APPLIC$
$ehr\_id : \mathbb{N}$
$tt : TRANSTYPE$
$s : SRCE$
$p : PURPOSE$
$pid : EHR\_CONT$
$ehrExt : EHR\_EXT$
$comm : EXT\_COMM$

$\forall e1, e2 : ehrs \bullet e1.ehr\_id = e2.ehr\_id \Rightarrow e1 = e2$

INIT

$ehrs = \varnothing$
$seqIds = sqId$
$s = sr$

Segue a descrição de cada operação da classe EHRS:

- *effect_updateEhr*: substitui o registro antigo (e) pelo novo (ehr).
- *effect_selectTrans*: recebe como parâmetros de entrada a identificação do paciente (pid?), o tópico da transação ($tp?$) e a versão do gehr ($gv?$). A operação atualiza apenas a variável de estado *pid*.
- *effect_addEhrExtract*: gera como saída o registro de importação e exportação ($ehrExt$), e o tipo da comunicação ($comm$). Esta operação não altera variáveis de estado.
- *effect_createTransImport*: recebe como parâmetros de entrada o endereço da aplicação ($app?$), o identificador do prontuário solicitante ($sr?$) e um registro com todas as transações ($cn?$). A operação *createTransImport* atualiza as variáveis de estado endereço da aplicação (app), identificador do registro do paciente (ehr_id), tipo de comunicação ($comm$) e registro de importação e exportação ($cn?.ex$).
- *effect_getEhr*: recupera o registro do paciente. Esta operação só é habilitada se existir o registro na coleção de registros.
- *effect_notGetEhr*: atualiza o motivo da falha. A operação *notGetEhr* só fica habilitada quando o registro desejado não é encontrado na coleção de registros.
- *effect_failEhr*: gera como saída o motivo da falha.
- *effect_addTransaction*: gera como saída o tipo da transação.

- *effect_extExported*: recebe como parâmetro de entrada o registro de importação e exportação (*ehrExt?*). Esta operação atualiza as variáveis de estado registro de importação e exportação (*ehrExt*), e tipo de comunicação (*comm*).
- *effect_createVersion*: recebe como parâmetros de entrada o endereço da aplicação (*app?*), o identificador do registro (*ehrid?*) e o tipo de usuário (*perm?*). Ela atualiza as variáveis de estado endereço da aplicação (*app*) e identificador do registro do paciente (*ehr_id*).
- *effect_createTrans*: recebe como parâmetros de entrada o identificador da aplicação (*app?*), os dados do paciente (*d?*) e as informações para criar a transação (cn?). Esta operação atualiza as variáveis de estado endereço da aplicação (*app*), identificador do registro do paciente (*ehr_id*) e o tipo da transação (*tt*).
- *effect_getTransType*: recupera o tipo da transação.
- *effect_addEhr*: adiciona o novo registro (*ehr*) na coleção de registros.
- *effect_recRecvd*: recupera o registro do paciente a partir do *pid*. Essa operação só está habilitada se encontrar o registro na coleção de registros.
- enable_recNotRecvd: a operação *recNotRecvd* só possui o esquema enable, porque nem atualiza o estado e nem gera alguma saída. Ela só está habilitada se não encontrar o registro do paciente.
- *effect_getIndObj*: recebe um próximo índice disponível (primeiro da lista) e atualiza a variável de estado *seqIds* com a lista sem o primeiro elemento. Esta operação só fica habilitada enquanto existir elementos na lista de identificadores.
- *effect_notGetIndObj*: A operação *notGetIndObj* atualiza a variável de estado *p* informando que não há referência disponível. Esta operação só é habilitada quando não exisitr mais elementos na lista de identificadores.

EHRS(*sqId* : seq *EHRID*; *sr* : *SRCE*)

effect_updateEhr

$\Delta(ehrs)$
$ehr : EHR$

$(\exists\, e : ehrs \bullet$
$e.ehr\_id = ehr.ehr\_id$
$ehrs' = (ehrs \setminus \{e\}) \cup \{ehr\}$

effect_selectTrans

$\Delta(pid)$
$pid? : EHR\_CONT$
$tp? : TOPIC$
$gv? : \mathbb{N}$

$pid' = pid?$

effect_addEhrExtract

$\Delta()$
$ehrExt! : EHR\_EXT$
$comm! : EXT\_COMM$

$ehrExt! = ehrExt$
$comm! = comm$

effect_createTransImport

$\Delta(app, ehr\_id, comm, ehrExt)$
$app? : APPLIC$
$sr? : SRCE$
$tg? : SRCE$
$cn? : TRANSIMPD$

$app' = app? \land ehr\_id' = cn?.ehr\_id$
$comm' = imp \land ehrExt' = cn?.ex$

enable_getEhr

$\exists\, e : ehrs \bullet e.ehr\_id = ehr\_id$

effect_getEhr

$\Delta(p)$

$ehr? : EHR$

$\exists\, e : ehrs \bullet e.ehr\_id = ehr\_id$

$ehr? = e \land p' = RecordRecovered$

enable_notGetEhr

$\nexists e : ehrs \bullet e.ehr\_id = ehr\_id$

effect_notGetEhr

$\Delta(p)$

$p' = RecordNotRecovered$

effect_failEhr

$\Delta()$

$app! : APPLIC$

$p! : PURPOSE$

$app! = app$

$p! = p$

effect_addTransaction

$\Delta()$

$t! : VERS\_TRANS$

$tt! : TRANSTYPE$

$tt! = tt$

effect_extExported

$\Delta(ehrExt, comm)$

$extEhr? : EHR\_EXT$

$ehrExt' = extEhr?$

$comm' = exp$

effect_createVersion

$\Delta()$

$app? : APPLIC$

$ehrid? : EHRID$

$perm? : PERMISS$

$app' = app?$

$ehr\_id' = ehrid?$

effect_createTrans

$\Delta(ehr\_id, tt, app)$

$app? : APPLIC$

$d? : EHR\_CONT$

$cn? : TRANSD$

$ehr\_id' = cn?.ehr\_id$

$tt' = cn?.tt \land app' = app?$

effect_getTransType

$\Delta()$

$tt? : TRANSTYPE$

$tt? = tt$

effect_addEhr

$\Delta(ehrs)$

$ehr : EHR$

$ehrs' = ehrs \cup \{ehr\}$

enable_recRecvd

$ehr? : EHR$

$\exists\, ehr : ehrs \bullet last(ehr.subject.versions).content.content = pid.content$

effect_recRecvd

$\Delta()$

$ehr? : EHR$

$\exists\, ehr : ehrs \bullet last(ehr.subject.versions).content.content = pid.content$

$ehr? = ehr$

enable_recNotRecvd

$\nexists ehr : ehrs \bullet$

$last(ehr.subject.versions).content.content = pid.content$

enable_getIndObj

$seqIds \neq <>$

effect_getIndObj

$\Delta(seqIds)$

$ind? : EHRID$

$ind? = head\ seqIds$

$seqIds' = tail\ seqIds$

enable_notGetIndObj

$seqIds = <>$

effect_notGetIndObj

$\Delta(p)$

$p' = WithoutReference$

Apêndice E

Implementação do Processo CREATE_TRANS em CSP_M Utilizando o Padrão

Segue a apresentação da implementação do processo CREATE_TRANS, que faz parte do componente EHRS, para mostrar a utilização do padrão na especificação de sistemas orientados a objetos.

E.1 Funções

Apresentamos a seguir todas as funções utilizadas pela especificação. Começamos com as funções auxiliares que recuperam elementos que se encontram numa deteminada posição da tupla.

```
first2((a,b)) = a
second2((a,b)) = b

first3((a,b,c)) = a
second3((a,b,c)) = b
third3((a,b,c)) = c

first4((a,b,c,d)) = a
second4((a,b,c,d)) = b
third4((a,b,c,d)) = c
forth4((a,b,c,d)) = d

first5((a,b,c,d,e)) = a
second5((a,b,c,d,e)) = b
third5((a,b,c,d,e)) = c
forth5((a,b,c,d,e)) = d
fifth5((a,b,c,d,e)) = e

first10((a,b,c,d,e,f,g,h,i,j)) = a
```

```
second10((a,b,c,d,e,f,g,h,i,j)) = b
third10((a,b,c,d,e,f,g,h,i,j)) = c
forth10((a,b,c,d,e,f,g,h,i,j)) = d
fifth10((a,b,c,d,e,f,g,h,i,j)) = e
sixth10((a,b,c,d,e,f,g,h,i,j)) = f
seventh10((a,b,c,d,e,f,g,h,i,j)) = g
eighth10((a,b,c,d,e,f,g,h,i,j)) = h
nineth10((a,b,c,d,e,f,g,h,i,j)) = i
tenth10((a,b,c,d,e,f,g,h,i,j)) = j

first12((a,b,c,d,e,f,g,h,i,j,k,l)) = a
second12((a,b,c,d,e,f,g,h,i,j,k,l)) = b
third12((a,b,c,d,e,f,g,h,i,j,k,l)) = c
forth12((a,b,c,d,e,f,g,h,i,j,k,l)) = d
fifth12((a,b,c,d,e,f,g,h,i,j,k,l)) = e
sixth12((a,b,c,d,e,f,g,h,i,j,k,l)) = f
seventh12((a,b,c,d,e,f,g,h,i,j,k,l)) = g
eighth12((a,b,c,d,e,f,g,h,i,j,k,l)) = h
nineth12((a,b,c,d,e,f,g,h,i,j,k,l)) = i
tenth12((a,b,c,d,e,f,g,h,i,j,k,l)) = j
eleventh12((a,b,c,d,e,f,g,h,i,j,k,l)) = k
twelveth12((a,b,c,d,e,f,g,h,i,j,k,l)) = l
```

A função `FSeq` retorna um conjunto de seqüências, de tamanho `s`, com elementos do conjunto `T`. Segue a sua definição:

```
FSeq(T, 0) = {<>}
FSeq(T, 1) = union(FSeq(T, 0), {<z> | z<-T})
FSeq(T, s) = {z^z' | z<-FSeq(T, 1), z'<-FSeq(T, s-1)}
```

Para adicionar uma dupla no final de uma tupla de dez elementos, usamos a função `makeContext`.

```
makeContext((a,b,c,d,e,f,g,h,i,j),(k,l)) = (a,b,c,d,e,f,g,h,i,j,k,l)
```

Para recuperar os registros, na coleção de registros, que possuem seus identificadores no conjunto de identificadores, aplicamos a função `findEhrs`.

```
findEhrs(setId,ehrs) ={ e | e <- ehrs,member(ehrIdent(e),setId)}
```

A função `checkEhrs` não permite que se tenha dois registros com o mesmo identificador. Ela caracteriza o invariante de EHRS. Implementamos essa restrição utilizando duas outras funções: `findEhrs` (apresentada anteriormente) que retorna todos os registros cujos identificadores pertencem ao conjunto de identificadores, e `empty` (função própria de FDR) que testa se o conjunto está vazio.

```
checkEhrs(ehrs,<>)= true
checkEhrs({},sId)= true
checkEhrs(ehrs,sId)= empty(findEhrs(set(sId),ehrs))
```

Para substituir a transação antiga pela nova (*tr*), usamos a função `changeTransaction`.

```
changeTransaction(tr,<>,seqAux) = seqAux
changeTransaction(tr,seqTrans,seqAux) =
   if (transId(head(seqTrans)) == transId(tr))
   then changeTransaction(tr,<>,seqAux^<tr>^tail(seqTrans))
   else changeTransaction(tr,tail(seqTrans),seqAux^<head(seqTrans)>)
```

A recuperação de uma transação, que se encontra numa lista de transações, é feita através da função `findTrans`.

```
findTrans(seqTrans,vid) =
   if(transId(head(seqTrans)) == vid)
   then head(seqTrans)
   else findTrans(tail(seqTrans),vid)
```

Ao invés de utilizarmos diretamente as funções auxiliares nas operações, preferimos definir outras funções que as utilizam para não comprometer a legibilidade da especificação. Definimos as funções `transConcept`, `transId`, `ehrIdent`, `subjTrans`, `demTrans`, `persTrans`, `evenTrans` e `ehrContConcept`, que recuperam, respectivamente, o conceito utilizado na transação (transação como parâmetro), o identificador da transação, o identificador do registro, a transação do tipo `subject`, a seqüência de transações dos tipos demográficas, persistentes e eventuais, e o conceito utilizado na transação (contéudo do registro como parâmetro).

```
transConcept(tr) = first4(first2(head(fifth5(tr))))
transId(tr) = third5(tr)
ehrIdent(ehr) = first12(ehr)
subjTrans(ehr) = fifth12(ehr)
demTrans(ehr) = sixth12(ehr)
persTrans(ehr) = seventh12(ehr)
evenTrans(ehr) = eighth12(ehr)
ehrContConcept(ehrcont) = first4(ehrcont)
```

O esquema TRANSD possui as informações necessárias para se criar uma transação. Para recuperar o prontuário fonte, o médico, o identificador do registro do paciente, o tipo de transação, o direito de leitura a transação, o direito de correção da transação, o motivo de criar a transação, a versão do GEHR, o conjunto de codificadores utilizados na transação, o identificador da transação, e os dados do paciente, utilizamos as respectivas funções `srTransd`, `hcaTransd`, `ehridTransd`, `ttTransd`, `acTransd`, `amTransd`, `rsTransd`, `gvTransd`, `csTransd`, `vidTransd` e `dataTransd`.

```
srTransd(dcn) = first10(second2(dcn))
hcaTransd(dcn) = second10(second2(dcn))
ehridTransd(dcn) = third10(second2(dcn))
ttTransd(dcn) = forth10(second2(dcn))
acTransd(dcn) = fifth10(second2(dcn))
amTransd(dcn) = sixth10(second2(dcn))
rsTransd(dcn) = seventh10(second2(dcn))
```

```
gvTransd(dcn) = eighth10(second2(dcn))
csTransd(dcn) = nineth10(second2(dcn))
vidTransd(dcn) = tenth10(second2(dcn))
dataTransd(dcn) = first2(dcn)
```

O conjunto AppDataCn possui tuplas cujos elementos são acessados através das funções application, recordId, transactionType, patientData.

```
application(apdcn) = first3(apdcn)
recordId(apdcn) = third10(third3(apdcn))
transactionType(apdcn) = forth10(third3(apdcn))
patientData(apdcn) = second3(apdcn)
```

O tipo TrTransType possui dois elementos: a transação e o tipo da transação. Para recuperá-los utilizamos as funções trTrTransType e ttTrTransType, respectivamente.

```
trTrTransType(trtt) = first2(trtt)
ttTrTransType(trtt) = second2(trtt)
```

O conjunto EhrExtEc possui duplas de elementos. Para acessar o elemento tipo de comunicação, utilizamos a função commehrext, e para acessar o registro de importação/exportação, a função ehrehrext.

```
commehrext(ce) = second2(ce)
ehrehrext(ce) = first2(ce)
```

A função checkTransactions garante que cada tipo de transação possui transações de conceitos específicos. Por exemplo, o tipo subject só pode ter uma transação cujo conceito é patientId; demographic pode apenas ter transações de conceito address; persistent pode ter transações de conceitos bloodPressure e headache; eventual permite ter transações de conceitos haemogram e consultation. Segue a sua definição:

```
checkTransactions(s,d,p,e) =
   (member(transConcept(s),Concept_Subject_Set)) and
   empty({x | x <- d,
         not member(transConcept(x),Concept_Demographic_Set)}) and
   empty({x | x <- p,
         not member(transConcept(x),Concept_Persistent_Set)}) and
   empty({x | x <- e,
         not member(transConcept(x),Concept_Eventual_Set)})
```

Os conjuntos Concept_Subject_Set, Concept_Demographic_Set, Concept_Persistent_Set e Concept_Eventual_Set estão definidos na Seção E.2. Para criar processos dinamicamente, utilizamos a função proc. Seus parâmetros de entrada são o tipo do processo e os valores de inicialização das variáveis de estado. Observamos que os parâmetros reais desta função determinam o processo que será ativado.

```
proc(TEHRS,(setEhr,seqId,sr)) =
   EHRS(setEhr,seqId,sr)
```

```
proc(TEHRBASE,(ehrid,ct,sr,sub,dement,perstr,evettrans,extmerg,extsent)) =
   EHR_BASE(ehrid,ct,sr,sub,dement,perstr,evettrans,extmerg,extsent)
proc(TEHR,(ehrid,ct,sr,sub,dement,perstr,evettrans,extmerg,extsent)) =
   EHR(ehrid,ct,sr,sub,dement,perstr,evettrans,extmerg,extsent)
proc(TVERSTRANS,(ar,aa,tc,gv,vs)) =
   VERS_TRANS(ar,aa,tc,gv,vs)
```

E.2 Tipos de Dados

Os conjuntos descritos nesta seção devem conter poucos elementos para evitar a explosão de estados. Vamos começar apresentando o conjunto `OBJTYPE` que possui as constantes `TEHRS`, `TEHR`, `TEHRBASE` e `TVERSTRANS`, as quais representam os tipos de objetos. Essas constantes são passadas como parâmetro da função `proc`, que é responsável pela criação de processos como explicado na Seção E.1.

```
datatype OBJTYPE = TEHRS | TEHR | TEHRBASE | TVERSTRANS
```

Como existem várias aplicações rodando independentemente umas das outras, para acessá-las é necessário do seu endereço. Os endereços se encontram no conjunto `APPLIC`.

```
nametype APPLIC = {1,2,3}
```

O conjunto `PURPOSE` possui as constantes `RecordNotRecovered`, `WithoutReference` e `RecordRecovered` que representam os motivos das falhas.

```
datatype PURPOSE =  RecordNotRecovered | WithoutReference |
                    RecordRecovered
```

Todo registro de paciente possui um identificador. Esses identificadores se encontram no conjunto `EHRID`.

```
nametype EHRID = {1,2}
```

As transações estão organizadas de acordo com os seguintes tipos de transação: "subject", "demographic", "persistent" e "eventual". Esses tipos são representados pelas constantes `subject`, `demographic`, `persistent` e `eventual`, que pertencem ao conjunto `TRANSTYPE`.

```
datatype TRANSTYPE =  subject |demographic |persistent |eventual
```

Cada tipo de transação possui assuntos específicos, os quais se encontram no conjunto `TOPIC`.

```
datatype TOPIC = patientIdentity | HCP         | HCF         |problemList |
                 familyHistory   | contact     |testResults | Others
```

A comunicação entre os sistemas é feita através de importação e exportação de transações. As constantes que representam esses tipos são elementos do conjunto `EXT_COMM`.

```
datatype EXT_COMM = imp |exp
```

Como em CSP_M não é permitido declarar tipos básicos como *given sets*, então tivemos que instanciá-los para conjunto finitos, como é o caso de SRCE, TEXT e GEHR_ARCHID que estão definidos logo a seguir.

Os endereços de prontuários pertencem ao conjunto SRCE.

```
nametype SRCE = {1}
```

O conjunto TEXT representa os elementos do tipo texto.

```
nametype TEXT = {1,2,3,4}
```

Os identificadores de arquétipos do GEHR são elementos do conjunto GEHR_ARCHID.

```
nametype GEHR_ARCHID = {1}
```

Os conjuntos DEF_CONT e ORG_ROOT possuem os itens de conteúdo e os organizadores de nível topo, respectivamente.

```
nametype DEF_CONT = {1}
nametype ORG_ROOT = {1}
```

Todas as permissões de acesso às transações se encontram no conjunto PERMIS. O PERMISSET é um dos sub-conjuntos do PERMIS.

```
PERMIS = {1,2}
PERMISSET = {{1},{2}}
```

O conjunto CODSYS possui os codificadores ICD, ICPC e SNOMED da área de saúde. O CODYSSET é um dos sub-conjuntos do CODSYS. Tivemos que restringir os valores destes conjuntos para evitar a explosão de estados.

```
nametype ICD = {1}
nametype ICPC = {2}
nametype SNOMED ={3}
nametype CODSYS = {{1},{2},{3}}
nametype CODSYSSET = {{{1},{2},{3}}}
```

O conjunto ARCHETYPED contém tuplas, onde o primeiro elemento é o conceito e o segundo é o identificador do arquétipo relacionado ao conceito.

```
nametype ARCHETYPED = {(concept,archId) | concept <- TEXT,
                       archId <- GEHR_ARCHID}
```

Os conteúdos de registros se encontram no conjunto EHR_CONT.

```
nametype EHR_CONT = {(concept,archId,context,content) |
                     (concept,archId) <- ARCHETYPED,
                      context <- DEF_CONT, content <- ORG_ROOT}
```

Os médicos pertencem ao conjunto HCA.

```
nametype HCA = {1,2}
```

As informações de auditoria são elementos do conjunto COMMIT_AUDIT.

```
nametype COMMIT_AUDIT = {(hca,tm,sr,rs,termSet) | hca <- HCA,
                         tm <- DATE_TIME,sr <- SRCE, rs <- TEXT,
                         termSet <- CODSSET}
```

O conjunto TRANS contém versões.

```
TRANS = {(cont,audit) | cont <- EHR_CONT, audit <- COMMIT_AUDIT}
```

A constante SizeSeq define o tamanho máximo para a lista de versões. SEQ_TRANS é o conjunto de seqüências de versões (TRANS).

```
sizeSeq = 2
SEQ_TRANS = FSeq(TRANS,sizeSeq)
```

O conjunto VersTransContext possui os valores para a entidade referente ao processo VERS_TRANS (transações).

```
VersTransContext  = {(ar,am,tm,gv,vs) | ar <- PERMISSET,
                     aa <- PERMISSET,tm <- DATE_TIME,
                     gv <- GEHR_VERSION, vs <- SEQ_TRANS}
```

A partir do conjunto de transações, obtemos o conjunto de seqüências de transações (SEQ_VERS_TRANS).

```
SEQ_VERS_TRANS  = Fseq(VersTransContext,sizeSeq)
```

Os Identificadores de transações se encontram no conjunto DATE_TIME.

```
nametype DATE_TIME  = {1,2}
```

As versões do GEHR pertencem ao conjunto GEHR_VERSION.

```
nametype GEHR_VERSION = {1,2}
```

O conjunto TRANSD possui as informações necessárias para a criação das transações.

```
nametype TRANSD = {(sr,hca,ehrid,tt,ac,am,rs,gv,cs,vid) | sr <- SRCE,
                   hca <- HCA, ehrid <- EHRID, tt <- TRANSTYPE,
                   ac <- PERMISSET, am <- PERMISSET,rs <- TEXT,
                   vid <- DATE_TIME,gv <- GEHR_VERSION, cs <- CODSYSSET}
```

Como a técnica de verificação de modelos proposta por Fischer só considera uma entrada e uma saída para cada canal, então tivemos que reunir as informações de entrada em uma tupla e as de saída em outra. Os conjuntos AppDataCn, DataCn, AppPurp, TrTransType, que desempenham esse papel, possuem os possíveis valores para os canais createTrans, recTrData, failEhr e replaceTrans, respectivamente.

```
nametype AppDataCn = {(app,d,cn) | app <- APPLIC, data <- EHR_CONT,
                      cn <- TRANSD}
nametype DataCn = {(d,cn) | d <- EHR_CONT, cn <- TRANSD}
nametype AppPurp = {(app,p) | app <- APPLIC, p <- PURPOSE}
nametype TrTransType = {(tr,tt) | tr <- VersTransContext, tt <- TRANSTYPE}
```

Cada tipo de transação utiliza um conjunto de conceitos, que se encontra logo a seguir.

```
datatype Concept_Subject_Set = patientId
datatype Concept_Demographic_Set = address
datatype Concept_Persistent_Set = bloodPressure | headache
datatype Concept_Eventual_Set = haemogram | consultation
```

Os valores que a entidade referente ao processo EHR_BASE pode receber, se encontram no conjunto EhrBaseContext.

```
nametype EhrBaseContext =
 {(ehr_id,hcp_created_by,creation_time,ehr_source,subject,dem_entities,
   pers_cli_trans,event_cli_trans,tt,vid) |
   ehr_id <- EHRID, hcp_created_by <- HCA,creation_time <- DATE_TIME,
   ehr_source <- SRCE, subject <- VersTransContext,
   dem_entities <- SEQ_VERS_TRANS,pers_cli_trans <- SEQ_VERS_TRANS,
   event_cli_trans <- SEQ_VERS_TRANS,tt <- TRANSTYPE, vid <- DATE_TIME,
   checkTransactions(subject,set(dem_entities),set(pers_cli_trans),
                     set(event_cli_trans))}
```

O conjunto EhrExtContext possui valores para a entidade referente ao processo EHR_EXT (registro de importação e exportação).

```
nametype EhrExtContext = {(ehr_id,hcp_created_by,creation_time,ehr_source,
   subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid,
   destination_ehr_source,hcp_authorizing) | ehr_id <- EHRID,
   hcp_created_by <- HCA,creation_time <- DATE_TIME,ehr_source <- SRCE,
   subject <- VersTransContext,dem_entities <- SEQ_VERS_TRANS,
   pers_cli_trans <- SEQ_VERS_TRANS,event_cli_trans <- SEQ_VERS_TRANS,
   tt <- TRANSTYPE, vid <- DATE_TIME,
   destination_ehr_source <- SRCE,hcp_authorizing <- HCA,
   checkTransactions(subject,set(dem_entities),
                     set(pers_cli_trans),set(event_cli_trans))}
```

SEQ_EHR_EXT é o conjunto de seqüências de elementos de EhrExtContext.

```
SEQ_EHR_EXT = FSeq(EhrExtContext,2)
```

Os possíveis valores utilizados pelo canal getStateEhr se encontram no conjunto Ehr.

```
nametype Ehr = {(ehr_id,hcp_created_by,creation_time,ehr_source,subject,
     dem_entities,pers_cli_trans,event_cli_trans,tt,vid,extracts_merged,
     extracts_sents) | ehr_id <- EHRID, hcp_created_by <- HCA,
     creation_time <- DATE_TIME,ehr_source <- SRCE,
     subject <- VersTransContext,dem_entities <- SEQ_VERS_TRANS,
     pers_cli_trans <- SEQ_VERS_TRANS,event_cli_trans <- SEQ_VERS_TRANS,
     tt <- TRANSTYPE, vid <- DATE_TIME, extracts_merged <- SEQ_EHR_EXT,
     extracts_sents <- SEQ_EHR_EXT,
     checkTransactions(subject,set(dem_entities),set(pers_cli_trans),
                       set(event_cli_trans))}
```

O conjunto EhrContext possui os valores para a entidade referente ao processo EHR.

```
nametype EhrContext = {(extracts_merged,extracts_sents) |
                        extracts_merged <- SEQ_EHR_EXT,
                        extracts_sents <- SEQ_EHR_EXT}
```

EhrExtEc contém os possíveis valores utilizados pelo canal addEhrExtract.

```
nametype EhrExtEc = {(ehrext,comm) | ehrcont <- EhrExtContext,
                     comm <- EXT_COMM}
```

Segue a declaração dos canais.

```
channel createTrans:AppDataCn
channel createTrans1:DataCn
channel askForTime:DATE_TIME
channel getEhr:Ehr
channel getTransType:TRANSTYPE
channel addVersion:TRANS
channel updateEhr:Ehr
channel addTransaction:TrTransType
channel recPersisTrans:EHR_CONT.SEQ_VERS_TRANS
channel failEhr:AppPurp
channel replaceTrans:TrTransType
channel notGetEhr
channel getIndObj: EHRID
channel notGetIndObj
channel addEhr:Ehr
channel recoverState:EhrContext
channel getStateEhrBase: EhrBaseContext
channel getStateEhr:Ehr
channel getStateVersTrans:VersTransContext
channel terminate
channel exit
```

E.3 Transações e Versões de Transações

O processo VERS_TRANS, o qual é uma abreviação de VERSIONED_TRANSACTION, modela uma transação. Ele é responsável por adicionar novas versões. Os parâmetros ar, aa, tc, gv e vs são utilizados para inicializar as suas variáveis de estado. Segua a definição de VERS_TRANS.

```
VERS_TRANS(ar,aa,tc,gv,vs) =
```

Na sua definição, várias constantes locais são introduzidas. Elas são apresentadas a seguir. A constante local Ops contém o conjunto de nomes de canais usados pelo processo referente a parte de Z.

```
 let
    Ops = {addVersion,getStateVersTrans,terminate}
```

A constante LocOps contém apenas os nomes dos canais locais.

```
LocOps = {}
```

O comportamento do processo VERS_TRANS está definido pelo processo main abaixo.

```
main = addVersion?v -> exit -> getStateVersTrans?st -> terminate -> SKIP
```

A constante local CSPOps contém o conjunto de nomes de canais usados pelo processo referente a parte de CSP. Neste caso, os nomes de canais são os mesmos utilizados pelo processo referente a parte de Z.

```
CSPOps = Ops
```

O estado possui seus componentes descritos no apêndice E.

```
state = {(access_rights,access_amend,time_created,gehr_version,
         versions) |access_rights <- PERMISSET,access_amend <- PERMISSET,
          time_created <- DATE_TIME,gehr_version <- GEHR_VERSION,
          versions <- SEQ_TRANS,(#versions > 0)}
```

O estado inicial da parte de Z do processo VERS_TRANS é dada pela constante init.

```
init = {(access_rights',access_amend',time_created',gehr_version',
         versions')|(access_rights',access_amend',time_created',
         gehr_version',versions') <- state,access_rights'== ar,
         access_amend'== aa,time_created' == tc,gehr_version' == gv,
         versions' == vs, (#versions' > 0)}
```

A parte de Z é descrita por um conjunto de funções que são utilizadas pela função Semantics (Seção 3.5). Portanto, a seguir apresentamos as equações in, out, enable e effect referentes aos canais do processo VERS_TRANS. A função in aplicada a operação addVersion define que os valores de entrada desta operação devem pertencer ao conjunto TRANS (abreviação de TRANSACTION). Por outro lado, in aplicada as operações getStateVersTrans e terminate determina que elas não possuem valores de entrada ({}).

```
in(addVersion) = TRANS
in(getStateVersTrans) = {{}}
in(terminate) = {{}}
```

A função out aplicada a operação getStateVersTrans define que os valores de saída devem pertencer ao conjunto VersTransContext. Esta função aplicada as operações addVersion e terminate não geram valores de saída ({}).

```
out(addVersion) = {{}}
out(getStateVersTrans) = VersTransContext
out(terminate) = {{}}
```

A função enable habilita e desabilita operações. Neste caso todas as operações estão habilitadas.

```
enable(addVersion)((access_rights,access_amend,time_created,
 gehr_version,versions))= true
enable(getStateVersTrans)((access_rights,access_amend,time_created,
 gehr_version,versions))= true
enable(terminate)((access_rights,access_amend,time_created,
 gehr_version,versions))= true
```

A função `effect` executa a operação habilitada passada como parâmetro. Segue a descrição da função `effect` para cada operação.

A função `effect(addVersion)` adiciona uma versão na lista de versões da transação.

```
effect(addVersion)((access_rights,access_amend,time_created,
       gehr_version,versions),v) =
       {({},(access_rights',access_amend',time_created',
         gehr_version',versions')) |
         (access_rights',access_amend',time_created',
         gehr_version',versions') <- state,
         access_rights' == access_rights,
         access_amend' == access_amend,
         time_created' == time_created,
         gehr_version' == gehr_version,
         versions' == versions ^ <v>, (#versions' > 0)}
```

A função `effect(getStateVersTrans)` recupera o estado da transação e o atribui a variável de saída `o`.

```
effect(getStateVersTrans)((access_rights,access_amend,time_created,
       gehr_version,versions),_) =
       {(o,(access_rights',access_amend',time_created',
         gehr_version',versions')) |
        (access_rights',access_amend',time_created',
         gehr_version',versions') <- state,
        access_rights' == access_rights,
        access_amend' == access_amend,
        time_created' == time_created,
        gehr_version' == gehr_version,
        versions' == versions, o <- VersTransContext,
        o == (access_rights',access_amend',time_created',
              gehr_version',versions'),(#versions' > 0)}
```

A função `effect(terminate)` nem altera o estado e nem gera qualquer saída.

```
effect(terminate)((access_rights,access_amend,time_created,
       gehr_version,versions),_) =
       {({},(access_rights',access_amend',time_created',
        gehr_version',versions')) |
        (access_rights',access_amend',time_created',
```

```
        gehr_version',versions') <- state,
        access_rights' == access_rights,
        access_amend' == access_amend,
        time_created' == time_created,
        gehr_version' == gehr_version,
        versions' == versions, (#versions' > 0)}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

E.4 Registros

O processo `EHR_BASE` é responsável pela manutenção das transações. Segue a sua definição.

```
EHR_BASE(id,hca,tm,sr,sub,dem,pers,event) =

 let
    Ops = {createTrans1,addTransaction,recPersisTrans,replaceTrans,
           getStateEhrBase,terminate,getTransType}
```

O conjunto `LocOps` possui os nomes dos canais locais.

```
    LocOps = {getTransType,addTransaction,replaceTrans}
```

O comportamento do processo `EHR_BASE` está definido pelo processo `main` abaixo.

```
    main = createTrans1?dcn -> getTransType?tt -> askForTime?tm ->
           (let
              c = (hcaTransd(dcn),tm,srTransd(dcn),rsTransd(dcn),
                   csTransd(dcn))
              v = (dataTransd(dcn),c)
            within
               (if ((tt == demographic) or (tt == eventual))
                then(let
                       t = (acTransd(dcn),amTransd(dcn),tm,
                            gvTransd(dcn),v)
                     within(addTransaction!(t,tt) -> exit ->
                            getStateEhrBase?st -> terminate -> SKIP))
                else
                   recPersisTrans!(dataTransd(dcn))?str ->
                   (if not null(str)
                    then ((let
                             e = head(str)
                             procVersTrans = proc(TVERSTRANS,e)
                             Flow = addVersion!v -> exit ->
                                    getStateVersTrans?t ->
```

```
                                replaceTrans!(t,tt) ->
                                getStateEhrBase?st ->
                                terminate -> SKIP
                        within(procVersTrans [|{|addVersion,exit,
                                terminate,getStateVersTrans|}|]
                                Flow)))
                else
                    (let
                        t = (acTransd(dcn),amTransd(dcn),tm,
                             gvTransd(dcn),v)
                     within(addTransaction!(t,tt) ->
                             exit -> getStateEhrBase?st ->
                             terminate -> SKIP)))))


CSPOps = Ops
```

O estado de EHR_BASE possui vários atributos, que estão descritos no apêndice E. Além deles, o estado possui a função checkTransactions, que garante que cada tipo de transação só recebe transações de conceitos específicos.

```
state = {(ehr_id,hcp_created_by,creation_time,ehr_source,subject,
          dem_entities,pers_cli_trans,event_cli_trans) |
          ehr_id <- EHRID, hcp_created_by <- HCA,tt<-TRANSTYPE,
          vid<-DATE_TIME,creation_time <- DATE_TIME,ehr_source <- SRCE,
          subject <- VersTransContext,dem_entities <- SEQ_VERS_TRANS,
          pers_cli_trans <- SEQ_VERS_TRANS,
          event_cli_trans <- SEQ_VERS_TRANS,
          checkTransactions(subject,set(dem_entities),
            set(pers_cli_trans),set(event_cli_trans))}
```

O estado inicial da parte de Z do processo EHR_BASE é dada pela constante init.

```
init = {(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid') |
        (ehr_id',hcp_created_by',creation_time',ehr_source',
        subject',dem_entities',pers_cli_trans',event_cli_trans',
        tt',vid') <- state,ehr_id' == ehrid, hcp_created_by' == hcp,
        creation_time' == ct,ehr_source' == sr, subject' == sub,
        dem_entities' == dement,pers_cli_trans'== perstr,
        event_cli_trans == evettrans,
        checkTransactions(subject',set(dem_entities'),
             set(pers_cli_trans'),set(event_cli_trans'))}
```

A função in aplicada às operações createTrans1, addTransaction, recPersisTrans e replaceTrans define que os valores de entrada devem pertecer aos conjuntos DataCn, TrTransType, EHR_CONT e TrTransType, respectivamente. A aplicação desta função no restante das operações determina que elas não possuem valores de entrada.

```
in(createTrans1) = DataCn
in(addTransaction) = TrTransType
in(recPersisTrans) = EHR_CONT
in(replaceTrans)= TrTransType
in(getStateEhrBase)= {{}}
in(terminate) = {{}}
in(getTransType) = {{}}
```

A função out aplicada às operações recPersisTrans, getStateEhrBase e getTransType determina que os valores de saída pertencem aos respectivos conjuntos SEQ_VERS_TRANS, EhrBaseContext e TRANSTYPE. Aplicando in as outras operações não geram valores de saída.

```
out(createTrans1) = {{}}
out(addTransaction) = {{}}
out(recPersisTrans) = SEQ_VERS_TRANS
out(replaceTrans) = {{}}
out(getStateEhrBase)= EhrBaseContext
out(terminate) = {{}}
out(getTransType) = TRANSTYPE
```

A função enable habilita e desabilita as operações. Neste caso, todas as operações se encontram habilitadas.

```
enable(createTrans1)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid))= true
enable(addTransaction)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid))= true
enable(recPersisTrans)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid))= true
enable(replaceTrans)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid))= true
enable(getStateEhrBase)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid))= true
enable(terminate)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid))= true
enable(getTransType) = ((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid))= true
```

A função effect executa as operações que estão habilitadas pelo enable. Segue a descrição da função effect para cada operação.

A função effect(createTrans1) atualiza apenas a variável de estado tipo de transação (tt). Esta operação utiliza a função ttTransd para recuperar o valor do tipo da transação que se encontra na tupla i (parâmetro de entrada).

```
effect(createTrans1)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid),i) =
```

```
{({},(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
 dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
 (ehr_id',hcp_created_by',creation_time',ehr_source',
  subject',dem_entities',pers_cli_trans',event_cli_trans',
  tt',vid') <- state,ehr_id'==ehr_id,hcp_created_by'==hcp_created_by,
  creation_time'==creation_time,ehr_source'== ehr_source,
  subject'== subject,dem_entities' ==dem_entities,
  pers_cli_trans'==pers_cli_trans,event_cli_trans'==event_cli_trans,
  tt'= ttTransd(i),vid'== vid,
  checkTransactions(subject',set(dem_entities'),set(pers_cli_trans'),
                    set(event_cli_trans'))}
```

Para adicionar uma transação no registro de paciente, utiliza-se a função `effect` aplicada ao parâmetro `addTransaction`, a qual usa duas outras funções: a `ttTrTransType` que recupera o valor do tipo da transação na tupla `i` (parâmetro de entrada), e a `trTrTransType` que recupera a transação.

```
effect(addTransaction)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid),i) =
  if ttTrTransType(i) == subject
  then {({},(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
        (ehr_id',hcp_created_by',creation_time',ehr_source',
        subject',dem_entities',pers_cli_trans',event_cli_trans',
        tt',vid') <- state,ehr_id'==ehr_id, hcp_created_by'==hcp_created_by,
         creation_time'==creation_time,ehr_source'== ehr_source,
         subject'== trTrTransType(i),dem_entities'== dem_entities,
         pers_cli_trans' == pers_cli_trans,
         event_cli_trans'==event_cli_trans,tt'==tt, vid'==vid,
         checkTransactions(subject',set(dem_entities'),set(pers_cli_trans'),
                           set(event_cli_trans'))}
  else
    if ttTrTransType(i) == demographic
    then {({},(ehr_id',hcp_created_by',creation_time',ehr_source',
          subject',dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
          (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
          dem_entities',pers_cli_trans',event_cli_trans',tt',vid') <- state,
          ehr_id'==ehr_id, hcp_created_by'==hcp_created_by,
          creation_time'==creation_time,ehr_source'== ehr_source,
          subject'== subject',
          dem_entities' == dem_entities ^ <trTrTransType(i)>,
          pers_cli_trans' == pers_cli_trans,
          event_cli_trans' == event_cli_trans, tt'==tt, vid'==vid,
          checkTransactions(subject',set(dem_entities'),
             set(pers_cli_trans'),set(event_cli_trans'))}
    else if ttTrTransType(i) == persistent
         then {({},(ehr_id',hcp_created_by',creation_time',ehr_source',
               subject',dem_entities',pers_cli_trans',event_cli_trans',
```

```
            tt',vid')) | (ehr_id',hcp_created_by',creation_time',
            ehr_source',subject',dem_entities',pers_cli_trans',
            event_cli_trans',tt',vid') <- state,
            ehr_id'==ehr_id, hcp_created_by'==hcp_created_by,
            creation_time'==creation_time,ehr_source'== ehr_source,
            subject'== subject', dem_entities' == dem_entities,
            pers_cli_trans' == pers_cli_trans ^ <trTrTransType(i)>,
            event_cli_trans' == event_cli_trans,tt'==tt,vid'==vid,
            checkTransactions(subject',set(dem_entities'),
                 set(pers_cli_trans'),set(event_cli_trans'))}
      else  {({},(ehr_id',hcp_created_by',creation_time',ehr_source',
            subject',dem_entities',pers_cli_trans',event_cli_trans',
            tt',vid')) |(ehr_id',hcp_created_by',creation_time',
             ehr_source',subject',dem_entities',pers_cli_trans',
             event_cli_trans',tt',vid') <- state, ehr_id'==ehr_id,
             hcp_created_by'==hcp_created_by,
             creation_time' == creation_time,ehr_source'== ehr_source,
             subject'== subject',dem_entities' == dem_entities,
             pers_cli_trans' == pers_cli_trans,
             event_cli_trans' == event_cli_trans ^ <trTrTransType(i)>,
             tt'== tt,vid'==vid,
             checkTransactions(subject',set(dem_entities'),
               set(pers_cli_trans'),set(event_cli_trans'))}
```

A função `effect(recPersisTrans)` verifica se existe a transação na lista de transações persistentes. Se encontrar, retorna a lista com a transação. Senão, retorna a lista vazia.

```
effect(recPersisTrans)((ehr_id,hcp_created_by,creation_time,ehr_source,
     subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid),i) =
  let
   s_pers_cli_trans = set(pers_cli_trans)
  within
    (if not empty({tr | tr <- s_pers_cli_trans,
       transConcept(tr) == ehrContConcept(i)})
    then {(o,(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
          dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
          (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
          dem_entities',pers_cli_trans',event_cli_trans',tt',vid') <- state,
          ehr_id' == ehr_id, hcp_created_by' == hcp_created_by,
          creation_time' == creation_time, ehr_source' == ehr_source,
          subject' == subject, dem_entities' == dem_entities,
          pers_cli_trans' == pers_cli_trans,event_cli_trans'==event_cli_trans,
          tt'==tt,vid'==vid,o <- SEQ_VERS_TRANS, o == <tr>,
          checkTransactions(subject',set(dem_entities'),set(pers_cli_trans'),
                            set(event_cli_trans'))}
    else {(o,(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
          dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
          (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
```

```
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid') <- state,
        ehr_id' == ehr_id, hcp_created_by' == hcp_created_by,
        creation_time' == creation_time, ehr_source' == ehr_source,
        subject' == subject, dem_entities' == dem_entities,
        pers_cli_trans' == pers_cli_trans,event_cli_trans'== event_cli_trans,
        tt'==tt,vid'==vid,o <- SEQ_VERS_TRANS, o == <>,
        checkTransactions(subject',set(dem_entities'),set(pers_cli_trans'),
                          set(event_cli_trans'))})
```

Para substituir a transação antiga pela nova, utiliza-se a função `effect(replaceTrans)`, a qual usa duas outras funções: a `ttTrTransType` que recupera o tipo da transação e a `changeTransaction` que realiza a substituição.

```
effect(replaceTrans)((ehr_id,hcp_created_by,creation_time,ehr_source,subject,
  dem_entities,pers_cli_trans,event_cli_trans,tt,vid),i) =
  if (ttTrTransType(i) == subject)
  then {({},(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
        (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid') <- state,
        ehr_id' == ehr_id, hcp_created_by'== hcp_created_by,
        creation_time'== creation_time,ehr_source'== ehr_source,
        subject'== trTrTransType(i),dem_entities'== dem_entities,
        pers_cli_trans' == pers_cli_trans,
        event_cli_trans'== event_cli_trans,tt'==tt, vid'==vid,
        checkTransactions(subject',set(dem_entities'),
             set(pers_cli_trans'),set(event_cli_trans'))}
  else
     if ttTrTransType(i) == demographic
     then {({},(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
           dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
           (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
            dem_entities',pers_cli_trans',event_cli_trans',tt',vid') <- state,
            ehr_id'==ehr_id, hcp_created_by'==hcp_created_by,
            creation_time'==creation_time,ehr_source'== ehr_source,
            subject'== subject',
            dem_entities' ==
              changeTransaction(trTrTransType(i),dem_entities,<>),
            pers_cli_trans' == pers_cli_trans,
            event_cli_trans' == event_cli_trans,tt'== tt,vid'==vid,
            checkTransactions(subject',set(dem_entities'),set(pers_cli_trans'),
                              set(event_cli_trans'))}

     else if (ttTrTransType(i) == persistent)
          then {({},(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
                dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
                (ehr_id',hcp_created_by',creation_time',ehr_source',
                 subject',dem_entities',pers_cli_trans',event_cli_trans',
```

```
                tt',vid') <- state,ehr_id'==ehr_id,
                hcp_created_by'==hcp_created_by,
                creation_time'==creation_time,ehr_source'== ehr_source,
                subject'== subject',dem_entities' == dem_entities,
                pers_cli_trans' ==
                  changeTransaction(trTrTransType(i),pers_cli_trans,<>),
                event_cli_trans' == event_cli_trans, tt'==tt, vid'==vid,
                checkTransactions(subject',set(dem_entities'),
                    set(pers_cli_trans'),set(event_cli_trans'))}
        else  {({},(ehr_id',hcp_created_by',creation_time',ehr_source',
                subject',dem_entities',pers_cli_trans',event_cli_trans',
                tt',vid')) |(ehr_id',hcp_created_by',creation_time',
                ehr_source',subject',dem_entities',pers_cli_trans',
                event_cli_trans',tt',vid') <- state,ehr_id'==ehr_id,
                hcp_created_by'==hcp_created_by,
                creation_time'==creation_time,ehr_source'== ehr_source,
                subject'== subject',dem_entities' == dem_entities,
                pers_cli_trans' == pers_cli_trans,tt'==t,vid'==vid,
                event_cli_trans' ==
                   changeTransaction(trTrTransType(i),event_cli_trans,<>),
                 checkTransactions(subject',set(dem_entities'),
                   set(pers_cli_trans'),set(event_cli_trans'))}
```

A tupla de valores que representa o estado do processo `EHR_BASE` é recuperada e atribuída a variável de saída `o` através da função `effect(getStateEhrBase)`.

```
effect(getStateEhrBase)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid),_) =
  {(o,(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
  dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
  (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
  dem_entities',pers_cli_trans',event_cli_trans',tt',vid') <- state,
  ehr_id'== ehr_id, hcp_created_by'== hcp_created_by,
  creation_time' == creation_time, ehr_source'== ehr_source,
  subject'== subject',dem_entities'== dem_entities,
  pers_cli_trans' == pers_cli_trans, tt'==tt, vid'==vid,
  event_cli_trans' == event_cli_trans, o <- EhrBaseContext,
  o == (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid'),
  checkTransactions(subject',set(dem_entities'),set(pers_cli_trans'),
                    set(event_cli_trans'))}
```

A função `effect(terminate)` não altera o estado e nem gera qualquer saída.

```
effect(terminate)((ehr_id,hcp_created_by,creation_time,ehr_source,
        subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid),_) =
        {({},(ehr_id',hcp_created_by',creation_time',ehr_source',
        subject',dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
```

```
        (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid') <- state,
        ehr_id'==ehr_id,hcp_created_by'==hcp_created_by,
        creation_time'==creation_time,ehr_source'== ehr_source,
        subject'== subject', dem_entities' == dem_entities,
        pers_cli_trans' == pers_cli_trans,event_cli_trans' == event_cli_trans,
        tt'==tt, vid'==vid,
        checkTransactions(subject',set(dem_entities'),set(pers_cli_trans'),
             set(event_cli_trans'))}
```

A função `effect(getTransType)` recupera o tipo da transação e o atribui a variável de saída `o`.

```
effect(getTransType)((ehr_id,hcp_created_by,creation_time,ehr_source,
    subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid),_) =
    {(o,(ehr_id',hcp_created_by',creation_time',ehr_source',
    subject',dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
    ehr_id',hcp_created_by',creation_time',ehr_source',
    subject',dem_entities',pers_cli_trans',event_cli_trans',tt',vid') <- state,
    ehr_id'== ehr_id, hcp_created_by'== hcp_created_by,
    creation_time' == creation_time, ehr_source'== ehr_source,
    subject'== subject',dem_entities'== dem_entities,
    pers_cli_trans' == pers_cli_trans, tt'== tt, vid'==vid,
    event_cli_trans' == event_cli_trans, o <- TRANSTYPE,o == tt',
    checkTransactions(subject',set(dem_entities'),
    set(pers_cli_trans'),set(event_cli_trans'))}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

A classe `EHR` é uma subclasse da classe `EHR_BASE`. Ela herda todos atributos, operações e comportamento da sua superclasse. Segue a sua definição.

```
EHR(ehrid,hcp,ct,sr,sub,dem,perstr,eventtr,extmerg,extsent) =
  let
    Ops = {recPersisTrans,replaceTrans,recoverState,terminate}
```

Os nomes dos canais locais do processo `EHR` pertencem ao conjunto `LocOps`.

```
    LocOps = {recoverState}
```

O processo `main` é formado pelo paralelismo dos processos `procEhrBase` e `Flow`. O `Flow` possui apenas os eventos específicos de `EHR`; os herdados não aparecem nele porque o processo `EHR_BASE` é quem os realiza.

```
    main =
     let
       e = (ehrid,hcp,ct,sr,sub,dem,perstr,eventtr)
       procEhrBase = proc(TEHRBASE,e)
```

```
    Flow = exit -> getStateEhrBase?st1 -> recoverState?st ->
           getStateEhr!makeContext(st1,st) -> terminate -> SKIP
 within(procEhrBase [|{|getStateEhrBase,terminate,exit|}|] Flow)

CSPOps = Ops
```

O estado possui dois elementos, os quais estão descritos no apêndice E.

```
state = {(extracts_merged,extracts_sents) |
          extracts_merged  <- SEQ_EHR_EXT,
          extracts_sents <- SEQ_EHR_EXT}
```

O estado inicial da parte de Z do processo `EHR` é dado pela constante `init`.

```
init = {(extracts_merged',extracts_sents')|
        (extracts_merged',extracts_sents') <- state,
         extracts_merged' == extmerg,
         extracts_sents' == extsent}
```

A função `in` aplicada nas operações `recoverState` e `terminate` determina que elas não possuem valores de entrada.

```
in(recoverState) = {{}}
in(terminate) = {{}}
```

A função `out` aplicada à operação `recoverState` determina que os valores de saída pertencem ao conjunto `EhrContext`. Esta função aplicada as outras operações não geram valores de saída.

```
out(recoverState) = EhrContext
out(terminate) = {{}}
```

A função enable habilita e desabilita as operações. Neste caso, todas elas estão habilitadas.

```
enable(recoverState)((extracts_merged,extracts_sents))= true
enable(terminate)((extracts_merged,extracts_sents))= true
```

A função effect executa as operações habilitadas. Segue a descrição da função `effect` aplicada a cada operação.

A função `effect(recoverState)` recupera o estado e o atribui a variável de saída `o`.

```
effect(recoverState)((extracts_merged,extracts_sents),_) =
 {(o,(extracts_merged',extracts_sents')) |
  (extracts_merged',extracts_sents') <- state,
   extracts_merged' == extracts_merged,
   extracts_sents' == extracts_sents,
   o <- EhrContext, o == (extracts_merged',extracts_sents')}
```

A função `effect(terminate)` não altera o estado e nem gera qualquer saída.

```
effect(terminate)((extracts_merged,extracts_sents),_) =
 {({},(extracts_merged',extracts_sents')) |
  (extracts_merged',extracts_sents') <- state,
  extracts_merged' == extracts_merged,
  extracts_sents' == extracts_sents}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

O processo `EHRS` é responsável pela manutenção de registros de pacientes. No nosso caso, ele realiza apenas a inclusão de transações, porque só estamos tratanto o processo `CREATE_TRANS`.

```
EHRS(setEhr,seqId,s) =
 let
    Ops = {createTrans, getTransType, getEhr,notGetEhr,
           addEhr, updateEhr, getIndObj,notGetIndObj,failEhr}
```

A constante `LocOps` contém apenas os nomes dos canais locais.

```
    LocOps = {getTransType,getEhr,updateEhr}
```

O processo `main` possui apenas o processo `CREATE_TRANS`, porque o escolhemos para aplicar o padrão de projeto. Segue a descrição do processo `CREATE_TRANS`.

```
main = CREATE_TRANS
CREATE_TRANS = createTrans?appdcn  -> getTransType?tt ->
     (getEhr?ehr ->
      if (tt = subject)
      then failEhr?appp -> main
      else
        ((let
           e = ehr
           procEhr = proc(TEHR,e)
           Flow = createTrans1!patientData(appdcn) ->
                  exit -> getStateEhr?st ->updateEhr.st -> SKIP
       within (procEhr [|{|createTrans1,getStateEhr,exit|}|] Flow));main)

     []
     notGetEhr ->
       (if (tt == subject)
        then (getIndObj?ind -> askForTime?tm ->
              (let
                 c = (hcaTransd(appdcn),tm,srTransd(appdcn),
                       rsTransd(appdcn),csTransd(appdcn))
                 v = (dataTransd(appdcn),c)
                 t = (acTransd(appdcn),amTransd(appdcn),
                      tm,gvTransd(appdcn),v)
```

```
                e = (ind,hcaTransd(appdcn),tm,srTransd(appdcn),
                     t,<>,<>,<>,<>,<>)
               within(addEhr.e -> main))
          []
          notGetIndObj -> failEhr?appp -> main)
    else failEhr?appp -> main))

CSPOps = Ops
```

O conjunto SeqIds é resultado da aplicação da função FSeq, a qual cria um conjunto de seqüências cujos elementos têm o tipo dado pelo primeiro argumento e cujo tamanho máximo é dado pelo segundo argumento.

```
SeqIds = FSeq(EHRID,sizeSeq)
```

O estado de EHRS possui vários atributos, que estão descritos no apêndice E. Além dos atributos, o estado possui a função checkEhrs que garante a unicidade dos identificadores de registros de paciente. Esta função caracteriza o invariante do processo EHRS.

```
state = {(ehrs,seqId,app,ehrid,tt,sr,p)|
          ehrs <- Set(Ehr),seqId <- SeqIds, app <- APPLIC,
          ehrid <- EHRID, tt <- TRANSTYPE,sr <- SRCE,
          p <- PURPOSE,checkEhrs(ehrs,seqId)}
```

O estado inicial da parte de Z do processo EHRS é dado pela constante init.

```
init = {(ehrs',seqId',app',ehrid',tt',sr',p') |
        (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
        ehrs' == setEhr, seqId'== sqId,
        sr'== s, checkEhrs(ehrs',seqId')}
```

A função in define o conjunto de elementos de entrada para cada operação.

```
in(createTrans) = AppDataCn
in(getTransType) = {{}}
in(getEhr) = {{}}
in(notGetEhr) = {{}}
in(addEhr) = Ehr
in(updateEhr) = Ehr
in(getObjInd) = {{}}
in(notGetObjInd)= {{}}
in(failEhr) = {{}}
```

A função out define o conjunto de elementos de saída para cada operação.

```
out(createTrans) = {{}}
out(getTransType) = TRANSTYPE
out(getEhr) = Ehr
out(notGetEhr) = {{}}
out(addEhr) = {{}}
```

```
out(updateEhr) = {{}}
out(getObjInd) = EHRID
out(notGetObjInd) = {{}}
out(failEhr) = AppPurp
```

A função `enable` habilita as operações que tiverem o predicado verdadeiro. Vamos comentar apenas as operações `getEhr`, `notGetEhr`, `getObjInd` e `notGetObjInd`, porque as demais possuem o predicado verdadeiro. A função `enable(getEhr)` só habilita a operação `getEhr` se encontrar o registro de paciente na coleção de registros. Já a função `enable(notGetEhr)` só habilita `notGetEhr` se não encontrar o registro na coleção de registros. A função `enable(getObjInd)` habilita a operação `getObjInd` se a lista de identificadores tiver elementos. E para finalizar, `enable(notGetObjInd)` habilita `notGetObjInd` se a lista de identificadores estiver vazia.

```
enable(createTrans)((ehrs,seqId,app,ehrid,tt,sr,p)) = true
enable(getTransType)((ehrs,seqId,app,ehrid,tt,sr,p)) = true
enable(getEhr)((ehrs,seqId,app,ehrid,tt,sr,p)) =
  not empty({e | e <- ehrs,ehrIdent(e) == ehrid})
enable(notGetEhr)((ehrs,seqId,app,ehrid,tt,sr,p)) =
  empty({e | e <- ehrs, ehrIdent(e) == ehrid})
enable(addEhr)((ehrs,seqId,app,ehrid,tt,sr,p)) = true
enable(updateEhr)((ehrs,seqId,app,ehrid,tt,sr,p)) = true
enable(getObjInd)((ehrs,seqId,app,ehrid,tt,sr,p)) = not null(seqId)
enable(notGetObjInd)((ehrs,seqId,app,ehrid,tt,sr,p)) = null(seqId)
enable(failEhr)((ehrs,seqId,app,ehrid,tt,sr,p)) = true
```

A função `effect` executa as operações que foram habilitadas pela função `enable`. No corpo de cada `effect`, encontramos a função `checkEhrs`, que garante que, após a realização da operação, os identificadores dos registros de paciente continuam sendo únicos. Segue a descrição da aplicação da função `effect` para cada operação.

A função `effect(createTrans)` atualiza apenas as variáveis de estado `app`, `ehrid` e `tt` em função do parâmetro de entrada `i`. Ela não gera qualquer saída.

```
effect(createTrans)((ehrs,seqId,app,ehrid,tt,sr,p),i) =
    {({},(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
     ehrs'== ehrs, seqId'== seqId, app'== application(i),
     ehrid'== recordId(i), tt'== transactionType(i),sr'== sr,
     p'== p,checkEhrs(ehrs',seqId')}
```

A função `effect(getTransType)` gera o valor do tipo da transação como saída (`o`). Essa função não atualiza variáveis de estado.

```
effect(getTransType)((ehrs,seqId,app,ehrid,tt,sr,p),_) =
    {(o,(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
     ehrs'== ehrs, seqId'== seqId, app'== app, ehrid'== ehrid,
     tt'== tt,sr'== sr, o <- TRANSTYPE, o == tt',p'== p,
     checkEhrs(ehrs',seqId')}
```

Para recuperar o registro de paciente e atribuí-lo à variável de saída o, utiliza-se a função effect(getEhr).

```
effect(getEhr)((ehrs,seqId,app,ehrid,tt,sr,p),_) =
    {(o,(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
     ehrs'== ehrs, seqId'== seqId, app'== app, ehrid'== ehrid,
     sr'== sr,tt'== tt,p'== p, o <- Ehr, o == pick({e | e <- ehrs,
     ehrIdent(e) == ehrid'}),checkEhrs(ehrs',seqId')}
```

Caso o registro não tenha sido encontrado, a função effect(notGetEhr) é executada. Ela não altera o estado e nem gera qualquer saída.

```
effect(notGetEhr)((ehrs,seqId,app,ehrid,tt,sr,p),_) =
    {({},(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,ehrs'== ehrs,
      seqId'== seqId, app'== app, ehrid'== ehrid,sr'== sr,
      tt'== tt, p'== p, checkEhrs(ehrs',seqId')}
```

Para adicionar um registro na coleção de registros, utiliza-se a função effect(addEhr).

```
effect(addEhr)((ehrs,seqId,app,ehrid,tt,sr,p),i) =
    {({},(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
     ehrs'== union(ehrs,{i}), seqId'== seqId, app'== app,
     ehrid'== ehrid,
     tt'== tt, sr'== sr,p'== p,checkEhrs(ehrs',seqId')}
```

A substituição do registro antigo do paciente pelo atual (i) é realizada pela função effect(updateEhr). Os outros atributos permanecem inalterados.

```
effect(updateEhr)((ehrs,seqId,app,ehrid,tt,sr,p),i) =
    {({},(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
     ehrs'== union(diff({ e | e <- ehrs,
     ehrIdent(e) == ehrIdent(i)},ehrs),{i}),
     seqId'== seqId, app'== app, ehrid'== ehrid, tt'== tt,
     sr'== sr,p'== p,
     checkEhrs(ehrs',seqId')}
```

A função effect(getObjInd) fornece o próximo índice disponível e atualiza a lista de índices.

```
effect(getObjInd)((ehrs,seqId,app,ehrid,tt,sr,p),_) =
    {(o,(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
      ehrs'== ehrs, app'== app, ehrid'== ehrid, tt'== tt,
      sr'== sr,o <- EHRID, o == head(seqId), seqId'== tail(seqId),
      p'== p,checkEhrs(ehrs',seqId')}
```

Se não existir um índice disponível, a função `effect(notGetObjInd)` é executada. Ela não altera o estado e nem gera qualquer saída.

```
effect(notGetObjInd)((ehrs,seqId,app,ehrid,tt,sr,p),_) =
    {({},(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,ehrs'== ehrs,
      seqId'== seqId, app'== app, ehrid'== ehrid, sr'== sr,tt'== tt,
      p'== p, checkEhrs(ehrs',seqId')}
```

A função `effect(failEhr)` gera uma dupla, como saída (o), onde o primeiro elemento é o endereço da aplicação e o segundo o motivo da falha.

```
effect(failEhr)((ehrs,seqId,app,ehrid,tt,sr,p),_) =
    {(o,(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
      ehrs'== ehrs, app'== app, ehrid'== ehrid,tt'== tt,
      sr'== sr, o <- AppPurp,o == (app',p'),seqId'== seqId,
      p'== p, checkEhrs(ehrs',seqId')}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

O processo `SYSTEM` define a arquitetura do sistema. Ele cria o processo `EHRS` através da função `proc`. Essa função recebe dois parâmetros: o primeiro é o tipo do processo a ser criado, e o segundo, os valores para a inicialização das variáveis de estado do processo `EHRS`.

```
SYSTEM =
  let
    ehrsState = ({},<1,2>,1)
    procEhrs = proc(TEHRS,ehrsState)
  within (procEhrs)
```

Bibliografia

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, Abr 1996.

[2] Cristopher Alexander. *A Pattern Language.* Oxford University Press, 1977.

[3] Cristopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[4] R. Allen and D. Garlen. A formal basis for archetectural connection. In *ACM Transactions on Software Engineering and Methodology*, pages 213–249. ACM, 1997. volume 6.

[5] A.W.Roscoe. An alternative order for the failures model. *Journal of Logic and Computation*, 2(5):557–578, 1992.

[6] A.W.Roscoe. An unbound non-determinism in csp. *Journal of Logic and Computation*, pages 131–172, 1993.

[7] Geoff Barrett. Model checking in practice: the T9000 virtual channel processor. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 129–147. Springer-Verlag, 1993.

[8] Thomas Beale. The gehr archetype system requirements. Technical report, The GEHR Foundation, Aug 2000. Rev 3.1 draft B.

[9] Thomas Beale. Archetypes - constraint-based domain models for future-proof information systems. Technical report, The GEHR Foundation, Aug 2001. Rev 2.2.1.

[10] Thomas Beale. The gehr object model architecture. Technical report, The GEHR Foundation, Mar 2001. Revision 4.1 draft E.

[11] K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. Technical report, Apple Computer and Tektronix, Inc., 1987. Techical Report,No. CR-87-43.

[12] Cliff B.Jones. Process-algebraic foundations for an object-based design notation. Technical report, University of Manchester, 1993. Technical Report,UMCS-93-10-1.

[13] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[14] F. Buschmann. *A System of Patterns.* Wiley, 1996.

[15] C.A.Hoare, S.D.Brookes, and A.W.Roscoe. A theory of communicating sequential processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, 1984.

[16] A. L. C. Cavalcanti and A. C. A. Sampaio. From csp-oz to java with processes. In *In IPDPS (International Parallel and Distributed processing Symposium) 2002 Workshop on Formal Methods for Parallel Programming*, 2002.

[17] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[18] F. Derepas, P. Gastin, and D. Plainfossé. Avoiding state explosion for distributed systems with timestamps. In J. N. Oliveira and P. Zave, editors, *Proceedings of the 10th Symposium on Formal Methods for Increasing Software Productivity (FME'01)*, number 2021 in LNCS, pages 119–134. SPRINGER, 2001.

[19] DICOM. Digital imaging and communications in medicine. Disponível em http://-www.nema.org/dicom/.

[20] Final Committee Draft, CD 13568.2, and ISO Panel JTC1/SC22/WG19(Rapporteur Group for Z). Z notation. Disponível em http://www.comlab.ox.ac.acuk/oucl/groups/zstandards/, Aug 1999.

[21] R. Duke, G.Rose, and G.Smith. Object-z: A specification language advocated for the description of standards. In *Computer Standards and Interfaces*, pages 17:511–533, 1995.

[22] S. Dupuy, Y. Ledru, and M. Chbbre-Peccoud. Translating the omt dynamic model into object–z. In *ZUM'98: The Z Formal Specifications Notation*, pages 347–366. Springer-Verlag, 1998. LNCS volume 1493.

[23] Adalberto Farias, Alexandre Mota, and Augusto Sampaio. From cspz to cspm: A transformational java tool. In *In Proceedings of IV Workshop on Formal Methods*, pages 1–10, 2001.

[24] C. Fischer. *Combination and Implementation of Processes and Data From CSP-OZ to Java.* PhD thesis, Oldenburg University, January 2000.

[25] C. Fischer, E. Olderog, and H. Wehrheim. A csp view on uml–rt structure diagrams. In *LNCS*, pages 91–108. Springer-Verlag, 2001. volume 2029.

[26] Clemens Fischer. Combining csp and z. Technical report, University of Oldenburg, 1996. Tecnical Report.

[27] Clemens Fischer. *CSP-OZ - a combination of CSP and Object-Z.* Chapman and Hall, 1997. 423-438.

[28] Clemens Fischer. Csp-oz: A combination of object-z and csp. Technical report, University of Oldenburg, 1997. TRCF-97-2.

[29] Clemens Fischer. Csp-oz:a combination of object-z and csp. In *Formal Methdos for Open Object-Based Distributed Systems(FMOODS'97)*, pages 423–438. Chapman and Hall, 1997. volume 2.

[30] Clemens Fischer and Heike Wehrheim. Model-checking CSP-OZ specifications with FDR. In *Proceedings of the 1st International Conference on Integrated Formal Methods (IFM)*, pages 315–334, 1999.

[31] Formal Systems(Europe) Ltd. *Failures–Divergence Refinement*, oct 1997. Revision 2.0.

[32] M. Fowler. *Analysis Patterns.* Addison Wesley, 1997.

[33] C. Harald Gall, R. René Klösh, and T. Roland Mittermeir. Application patterns in re-engineering:identifying and using reusable concepts. In *Proceedings of the 6th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pages 1099–1106, 1996.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[35] G.Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide.* Addison Wesley, 1999.

[36] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. Technical report, Computer.org, 2002. IEEE Software, vol.7, n.5.

[37] HL7. Health level seven. Disponível em http://www.hl7.org/.

[38] HL7. Message development framework. Disponível em http://medinfo.rochester.edu./hl7/v3.0/mdf_toc.htm.

[39] C.A.R Hoare. Communicating sequential processes. In *CACM*, pages 21(8):666–677. CACM, 1978.

[40] C.A.R Hoare. *Communicating sequential processes.* Prentice-Hall International, 1985.

[41] D. Ingram, D. Lloyd, D. Kalra, and *et al.* Clinical functional specifications. Technical report, Centre for Health Informatics and Multiprofessional Education, and University College London, May 1993. GEHR Deliverable 7.

[42] D. Ingram, D. Lloyd, D. Kalra, and *et al.* Gehr architecture 1.0. Technical report, Centre for Health Informatics and Multiprofessional Education, and University College London, Jun 1995. Deliverables 19, 20, 24, Final Release.

[43] SNOMED International. The international statistical classification of diseases and related health problems, tenth revision. Disponível em http://who.int/whosis/icd10/.

[44] SNOMED International. Systematized nomenclature of medicine. Disponível em http://snomed.org.

[45] ISO. International organization for standardization. Disponível em http://iso.ch.iso./en/ISOOnline.frontpage/.

[46] JIRA. Japan industries association of radiological systems. Disponível em http://-www.jira-net.or./jp./e/.

[47] C.B. Jones. *Communicating sequential processes.* Prentice-Hall International, 1985.

[48] G. Kassel and G. Smith. Model checking object-z classes: Some experiments with fdr. In *In 8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, pages 445–452, 2001.

[49] S. Kim and D. Carrington. Formalizing the uml class diagram using object-z. In *UML'99: The Unified Modelling Language - Beyond the Standard*, pages 83–98. Springer-Verlag, 1999. LNCS volume 1723.

[50] K.Walden and J.Nerson. *Seamless Object-Oriented Software Architecture.* Prentice-Hall, 1995.

[51] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall PTR, 2002.

[52] David Lloyd and Dimitrios Sotiriou. Architectures. In *2expn EU-CEN Workshop Electronic Healthcare Record*, pages 7–8, 1997.

[53] Medicom UK Ltd. Medicom uk. Disponível em http://www.medicom.co/uk/.

[54] B.P. Mahony and J.S. Dong. Blendind object-z and timed csp: An introduction to tcoz. In *In 20th International Conference on Software Engineering(ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.

[55] K. McMillan and J. Schwalbe. Formal verication of the gigamax cache consistency protocol, 1991.

[56] Bertand Meyer. *Eiffel - The Language.* Prentice-Hall, 1992.

[57] Microsoft. Activex. Disponível em http://microsoft.com./com/tech/activex.asp.

[58] R. Milner. *Communicating and Concurrency.* Prentice-Hall, 1989.

[59] M. Modell. *Data Analysis, Data Modelling and Classification.* McGraw-Hill, first edition, 1992.

[60] A. Mota and A. Sampaio. Model checking csp-z. In *In Proceedings of the European Joint Conference on Theory and Practice of Software*, pages 205–220. Springer-Verlag, 1998. volume 1382.

[61] Alexandre Mota and Augusto Sampaio. Model checking by refinement: Speeding up analysis. a ser submetido.

[62] I-Med Clinical Networks. I-med clinical networks. Disponível em http://www.ogc.be/hometelecare/hometelenet/articles/practb5001a.html.

[63] OMG. Clincal observation access service. Disponível em http://cgi.omg.org/pub/docs/corbamed/99–03–25.zip.

[64] OMG. Clinical image access service. Disponível em http://cgi.omg.org/pub/docs/corbamed/99–12–01.pdf.

[65] OMG. Common object request broker architecture. Disponível em http://omg.org/.

[66] OMG. Corbamed. Disponível em http://omg.org/homepages/corbamed/.

[67] OMG. Health care information locator service. Disponível em http://cgi.omg.org/pub/docs/corbamed/99–11–18.pdf.

[68] OMG. Person identification service. Disponível em http://cgi.omg.org/pub/docs/corbamed/98–02–29.rtf.

[69] OMG. Resource access decision. Disponível em http://www.omg.org/pub/docs/corbamed/99–04–04.pdf.

[70] OMG. Terminology query service. Disponível em http://cgi.omg.org/pub/docs/corbamed/98–03–22.pdf.

[71] Health one. Hdmp health one. Disponível em http://www.hdmp.com/defaultmain.asp.

[72] S. Pruba, R. Dixon, and N. Harris. *Black Sea TeleDiab:Diabetes Computer System with Communication Technology for Black Sea Region.* 1998. Vol.3(3), pp. 193-196.

[73] R.A.Kemmerer. Integrating formal methods into the development process. Technical report, University of California, Santa Barbara, 1990. IEEE Software.

[74] A.W. Roscoe. Model-checking csp. In *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 353–378. Prentice-Hall, 1994.

[75] B. Scattergood. *The Semantics and Implementation of Machine–Readable CSP.* PhD thesis, Programming Research Group, Oxford University, February 1998.

[76] Gunther Schadow. Hl7 process modelling with petri nets. Disponível em http://aurora.rg.iupui.edu/g̃unther/petri.htm.

[77] Douglas C. Schmidt. *Design Patterns in Communications Software.* Cambridge University Press, 2001.

[78] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern Oriented-Software Architecture.* Wiley & Sons, 2000.

[79] Brain Selic and Jim Rumbaugh. Using uml for modeling complex real-time systems. Whitepaper, Mar 1998.

[80] Peter Shloeffel, Thomas Beale, Sam Heard, and David Rowed. Background and overview of the good electronic healthcare record. Disponível em http://gehr.org/Documents/BackgroundOverview_of_GEHR.htm.

[81] Peter Shloeffel, Thomas Beale, Sam Heard, and David Rowed. The good electronic healthcare record. Disponível em http://gehr.org/.

[82] G. Smith. *An Object-Oriented Approach to Formal Specifications.* PhD thesis, Department of Computer Science, University of Queensland, Oct 1992.

[83] G. Smith. A semantic integration of object-z and csp for the specification of concurrent systems. In *Formal Methods Europe(FME97)*, pages 62–81. Springer-Verlag, 1997. LNCS volume 1313.

[84] A.S. Tenenbaum. 1996. 3rd edition.

[85] Antti Valmari. The state explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[86] W3C. World wide web consortium. Disponível em http://www.w3.org/.

[87] W3C. Xml - extensible markup language. Disponível em http://www.w3.org/-XML/.

[88] D.J. Walker. Pi-calculus semantics of object-oriented programming languages. Technical report, Computer Science Department,Edinburgh University, 1990. Technical Report,ECS-LFCS-90-122.

[89] Heike Wehrheim. Subtyping patterns for active objects. In *Proceedings 8ter Workshop des GI Arbeitskreises GROOM (Grundlagen objekt-orientierter Modellierung): Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme*, 2000.