Integração de verificadores formais para agentes móveis

André Gustavo Andrade

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO DE MESTRE
EM
CIÊNCIAS

ÁREA DE CONCENTRAÇÃO: Ciência da Computação

ORIENTADORA: Profa. Dra. Ana Cristina Vieira de Melo

Durante a elaboração deste trabalho o autor recebeu auxilio financeiro do CNPq

São Paulo, Janeiro de 2007

Integração de verificadores formais para agentes móveis

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por André Gustavo Andrade e aprovada pela Comissão Julgadora

São Paulo, 12 de junho de 2007

Banca Examinadora:

Profa. Dra. Ana Cristina Vieira de Melo (orientadora) - IME/USP
Prof. Dr. Flávio Soares Correa da Silva - IME/USP
Prof. Dr. Álvaro Freitas Moreira - UFRGS

Agradecimentos

Dedicatória

Dedico esta tese à minha família.

Agradecimentos

Gostaria de agradecer em primeiro lugar a minha família: minha mãe, meu pai e meu irmão. Eles, mais do que os outros, me apoiaram sempre e conviveram comigo neste período de conturbações naturais que acompanha a escrita de uma tese.

Em seguida gostaria de agradecer a todos os meus amigos, que de todas as formas possíveis contribuíram para este trabalho. Sem o apoio de todos eles certamente este trabalho não seria possível.

E em terceiro lugar, à minha orientadora, Ana Cristina Vieira de Melo, cujo trabalho constante e os constantes puxões de orelha permitiram-me crescer. Por muitas vezes, ela, além de sua tarefa de orientação técnica, teve de assumir o papel de conselheira e até mesmo de uma segunda mãe.

Ainda, com muito carinho, gostaria de agradecer a meus professores que de alguma maneira contribuiram para que eu chegasse até aqui.

Nominalmente, gostaria ainda de agradecer a: *Yolanda Aparecida Juanini Andrade, Francisco Márcio Andrade, Eduardo Alexandre Andrade, Ana Cristina Vieira de Melo, Julio Cesar Avero, Rodrigo Zanato Tripodi, Rafael Rojas Pretel, Rodrigo Moreira Barbosa, Ana Beatriz Graciano,Thiago Schumacher Barcelos, Paulo Feofiloff, Zara Issa Abud, Leliane Nunes de Barros e Leyla Naomi*

Resumo

A maioria das ferramentas existentes para verificação formal de agentes móveis diferem no poder de expressar agentes, bem como nas capacidades de verificação. Assim, mesmo considerando uma única teoria para agentes móveis, as ferramentas correspondentes não forem construídas para comunicarem-se, e por este motivo, não podem operar em conjunto. Dado que essas ferramentas oferecem serviços (capacidades) distintos, a integração e cooperação das mesmas podem proporcionar um ambiente com um conjunto mais amplo de capacidades de verificação.

No entanto, algumas dificuldades surgem quando tentamos usufruir do serviço conjunto das ferramentas. As ferramentas utilizam-se, normalmente, de subconjuntos distintos de uma mesma teoria e a expressividade de cada subconjunto pode ser diferente. Quando isso ocorre é necessário mapear não apenas sintaticamente mas também semanticamente estes subconjuntos.

Neste trabalho mostraremos técnicas para realizar o mapeamento entre dois subconjuntos de π-*calculus* utilizados por duas ferramentas existentes, a VTUBAINA e o HAL/Jack. Mostraremos como estas técnicas são utilizadas para usufruir das capacidades conjuntas destas ferramentas, e como a utilização deste recurso permite ampliar as capacidades isoladas de cada ferramenta.

Um software, denominado Hydra foi desenvolvido como protótipo da implementação dos mapeamentos propostos neste trabalho.

Abstract

The majority of the existing tools for formal verification of mobile agents differ in the power of expressing agents, as well the verification capabilities. Therefore, even considering a single theory for mobile agents, the designed tools were not built to communicate with each other, and for this reason they cannot operate together. Once these tools offer different services (capabilities), the integration and coordination of these tools can provide an environment with a greater set of verification capabilities.

However, some difficulties appear when we try to use the joint services from these tools. They usually use distinct subsets of the same theory and the expressivity of each subgroup can be different. When it occurs it is necessary to map these subsets not only syntactically but also semantically.

In this paper we will present techniques to do the mappings between two subsets of π-*calculus* used by two existing tools, VTUBAINA and HAL/Jack. We will demonstrate how these techniques are used to provide us the joint capabilities of both tools, and how the use of this technique can extend the isolated capabilities of each one.

A software program, named Hydra was developed as a prototype of the implementation of the mappings proposed on this work.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Integração de Verificadores Formais

Com o crescimento da comunicação celular, serviços via satélite e redes sem fio, novos sistemas requerem recursos e informações acessados em qualquer lugar, através dos dispositivos móveis. Dada a importância das informações relevantes ao funcionamento desses sistemas, cresceu também a necessidade de se formalizar tais sistemas e verificar propriedades. Devido às dificuldades de se prever comportamento de sistemas concorrentes, álgebras de processos, tais como o CCS[1] [11] e o CSP[2] [8], surgiram como teoria para sistemas concorrentes, e posteriormente, baseadas nas anteriores, surgiram álgebras para agentes móveis, como a que utilizarei como base do trabalho de pesquisa, o *π-calculus* [13].

π-calculus [13] é um cálculo de processos que visa a especificação e verificação das comunicações entre agentes móveis [16, 20, 23, 5]. Ele tem como base o CCS e introduz novos elementos que permitem a mobilidade de agentes.

O desenvolvimento formal de agentes móveis possui duas etapas primordiais: a especificação e a verificação dos agentes.

A especificação de um sistema de agentes pode ser efetuada com a utilização de álgebras de processos, e tem como objetivo descrever o comportamento do sistema através da especificação formal do comportamento de cada um dos agentes.

Podemos, por exemplo, descrever o comportamento de telefones celulares em

[1]Calculus of Communicating Systems
[2]Communicating Sequential Processes

relação às antenas, expressando sua capacidade de mobilidade e ao mesmo tempo o comportamento de sua comunicação.

A verificação de um sistema tem como objetivo a verificação de propriedades do sistema ou de equivalência entre dois sistemas. Poderíamos estar interessados em verificar, por exemplo, se o sistema de telefonia móvel que modelamos é correto, ou se é equivalente a um certo outro modelo para telefonia móvel.

A tarefa de verificação formal de sistemas é, normalmente, exaustiva e complexa e, portanto, propensa a erros. Isto motiva a comunidade de desenvolvimento formal a criar ferramentas para a verificação de sistemas.

Como as ferramentas foram desenvolvidas independentemente, e em sua grande maioria, para demonstrar alguma nova teoria a respeito de agentes móveis [4, 22, 1], cada uma delas trabalha com um subconjunto diferente de π-*calculus*. Além de trabalharem com subconjuntos diferentes, usam uma sintaxe e semântica próprias e, em geral, verificam apenas algumas das equivalências e propriedades existentes.

Porém, normalmente precisamos verificar várias propriedades, e/ou equivalências, diferentes quando trabalhamos com agentes móveis e muitas vezes uma única ferramenta não atende a todas as verificações requisitadas. Isso faz com que o mesmo processo tenha de ser reescrito várias vezes, com sintaxes diferentes e algumas vezes passando até mesmo por reescrituras semânticas, quando precisamos submeter o sistema a mais de uma ferramenta. Essas conversões manuais entre as ferramentas propiciam a introdução de erros dificilmente detectáveis.

Este trabalho utiliza-se de álgebras de processos (linguagens compostas de teoria e cálculo com capacidade de exprimir os eventos possíveis de serem realizados por um processo e os comportamentos subseqüentes do mesmo) especialmente π-*calculus* [13], uma álgebra de processos baseada em CCS [11] (Calculus of Communicating Systems) e duas ferramentas de verificação formal existentes para esta álgebra: A HAL [4] (History Dependant Automata Laboratory) e a VTUBAINA [1] (A Verification Tool for Up-to Bisimulation and Automata INtegration Automatization).

A pesquisa em questão, visa estudar como integrar ferramentas de verificação formal de maneira automática, permitindo que o usuário, ao especificar um processo na linguagem de entrada de uma dada ferramenta, possa utilizar-se da mesma

e de outras para efetuar as verificações de interesse contribuindo para a automação do processo.

Veremos também que a integração das ferramentas não contribui apenas com a facilidade de uso mas também com um aumento do poder de verificação isolado das ferramentas, permitindo que, trabalhando em conjunto, haja um acréscimo de funcionalidades.

Porém, a integração entre ferramentas não é uma tarefa trivial, visto que cada uma delas utiliza-se de linguagens sintaticamente diferentes e de um conjunto restrito de π-*calculus*. Desta forma, a integração envolve problemas como mapeamentos sintáticos e semânticos entre as linguagens das ferramentas, que serão estudados adiante.

1.2 Organização do texto

Este trabalho está dividido da seguinte forma:

- O primeiro capítulo é esta introdução, que objetiva descrever o trabalho realizado.
- O segundo capítulo é uma introdução a π-*calculus*, o cálculo de processos utilizado pelas ferramentas que pretendemos integrar.
- O terceiro capítulo introduz as ferramentas utilizadas na pesquisa, suas linguagens, e as capacidades de cada uma delas.
- O quarto capítulo aborda os mapeamentos sintáticos e semânticos entre as ferramentas viabilizando a integração das mesmas.
- O quinto capítulo aborda o protótipo desenvolvido como implementação dos mapeamentos estudados no capítulo anterior
- O último capítulo conclui o trabalho resumindo os benefícios da técnica estudada e abordando possíveis trabalhos futuros.
- No apêndice encontra-se uma pesquisa preliminar, listando as ferramentas de verificação e simulação existentes.

Capítulo 2

π-calculus

Este capítulo discorre brevemente sobre *π-calculus* [12] para que, posteriormente, possamos relacionar as ferramentas com a teoria e as ferramentas entre si. Com isso, podemos mostrar qual subconjunto do cálculo cada uma das ferramentas implementa e as funcionalidades que oferecem.

2.1 Sintaxe de *π-calculus*

π-calculus é um cálculo de processos que permite descrever e analisar o comportamento de comunicações efetuadas por agentes móveis. Sua capacidade de expressar mobilidade, através de reconfiguração dinâmica dos processos, estende a capacidade de seu predecessor CCS, o qual é capaz de expressar concorrência mas não mobilidade.

Definição 1 *A sintaxe de* π-calculus *pode ser definida da seguinte forma [13]:*

$$P \stackrel{def}{=} \mathbf{O} \mid \alpha.P \mid (\nu x)\ P \mid (P_1 \mid P_2) \mid P_1 + P_2 \mid !\alpha.P$$

$$\alpha \stackrel{def}{=} a(b) \mid \overline{a}b \mid \tau$$

Onde P representa processos e α representa ações. As operações envolvendo processos podem ser assim descritas:

- O: Representa o processo nulo (ou inativo). Um processo que não apresenta qualquer possibilidade de comunicação ou reconfiguração.

- $\alpha.P$: Noção de sequência. O processo executa a ação α e posteriormente passa a se comportar como P.
- $(\nu x)\ P$: Restrição de um nome x ao processo P, visível somente a P, pertencendo ao conjunto de nomes restritos $\mathbf{bn}(P)$ (do inglês, bound names)
- $P_1 \mid P_2$: Dois processos, P_1 e P_2, estão executando paralelamente.
- $P_1 + P_2$: Ou o processo P_1 é executado ou o processo P_2 é executado, sendo este um ou exclusivo.
- $!P$: Representa uma quantidade infinita de processos da forma P rodando paralelamente. Ou seja: $!P \stackrel{def}{=} !P \mid P$.

As ações em π-*calculus* podem ser as seguintes:

- $a(b)$: Significa que por um canal de nome a é recebida uma informação que será unificada com o nome b. Neste caso, b é um nome restrito **bn**
- $\overline{a}b$: Significa que por um canal de nome a é enviado um nome b. Este nome pode ser um canal ou uma informação, sendo ambos indistintos para o cálculo.
- τ: Representa uma ação silenciosa, interna ao processo.

Para um dado processo, excetuando-se casos onde temos ações de entrada $a(b)$ ou restrições explícitas (νb) , todos os demais nomes do processo pertencem ao conjunto de nomes livres **fn** (do inglês, free names).

Definição 2 *Chamaremos o conjunto das ações de entrada de $\mathcal{I}$ e o conjunto das ações de saída de $\mathcal{O}$*

Definição 3 *Chamaremos de subject de uma ação α o nome do canal através do qual um nome está sendo enviado ou recebido, e representaremos por $ch(\alpha)$.*

Definição 4 *Chamaremos de object de uma ação α o nome recebido ou enviado através de um canal, e representaremos por $obj(\alpha)$.*

Definição 5 *Definimos uma substituição σ de x por y em um processo P como a troca de todos os nomes x naquele processo pelo nome y e denotamos por $P\{x/y\}$*

2.1.1 Exemplo

Telefonia Celular

Considere o problema na telefonia celular onde temos uma estação de controle que decide para qual das várias antenas retransmissoras deve enviar os sinais relativos à comunicação de um certo telefone celular, o qual eventualmente pode mudar a localização e, por isso, mudar de antena. Quando isso acontece, é necessário redirecionar os sinais para a nova antena. Utilizaremos π-*calculus* para modelar este cenário, que é uma adaptação do exemplo que aparece em [12]

O sistema tem por agentes: antenas, que podem estar ligadas ou desligadas, um controlador que efetua a sinalização da necessidade de troca de antenas e um carro no qual está localizado o telefone móvel.

Definiremos a antena como um processo capaz de se comportar de duas maneiras diferentes: *Ant_ligada* quando estiver se comunicando com o veículo, e *Ant_desligada* quando estiver ociosa.

$\forall i \in 1, 2$ vale:

$$Ant\_ligada_i(fala_i, troca_i, ganha_i, perde_i) \stackrel{def}{=}$$
$$fala_i.Ant\_ligada(fala_i, troca_i, ganha_i, perde_i)$$
$$+perde(t, s).\overline{troca}(s, t).Ant\_desligada(ganha_i, perde_i)$$

O processo Ant_ligada_i assim definido, mostra-se capaz de tomar alternativamente dois comportamentos: ou continuar na mesma antena, evoluindo para o mesmo processo, ou então gerar um sinal de perda passando a se comportar como uma antena desligada assim definida:

$$Ant\_desligada_i(ganha_i, perde_i) \stackrel{def}{=} ganha_i(t, s).Ant\_ligada(t, s, ganha_i, perde_i)$$

A antena desligada só possui um comportamento possível: esperar ganhar um agente para se ligar a ela e tornar-se uma antena ligada.

Além das antenas, precisamos também modelar o controle central:

$$Controle_1 \stackrel{def}{=} \overline{perde_1}(fala_2, troca_2).\overline{ganha_2}(fala_2, troca_2).Controle_2$$
$$Controle_2 \stackrel{def}{=} \overline{perde_2}(fala_1, troca_1).\overline{ganha_1}(fala_1, troca_1).Controle_1$$

O controle central cuida de gerar os sinais de ganho e perda de agentes para as antenas.

O carro pode ser modelado da seguinte maneira:

$$Carro(fala, troca) \stackrel{def}{=} \overline{fala}.Carro(fala, troca) + troca(t, s).Carro(t, s)$$

Comportando-se de duas formas alternativamente: ou efetua a fala e permanece o mesmo processo, ou efetua uma troca de antena, permanecendo no mesmo processo.

Quando o controle percebe a perda do sinal, é gerado um sinal *perde*, capturado pela antena ligada a qual o telefone está ligado. O controlador também gera em seguida o sinal *ganha* para a próxima antena. A antena a qual o telefone está ligado recebe o sinal de perda, envia o sinal *troca* para o telefone e desliga-se. A nova antena, que estava desligada, recebe o sinal de ganho e liga-se. O carro recebe o sinal de troca e reconfigura-se para agora estar conectado a esta nova antena.

2.2 Semântica de π-*calculus*

Uma forma de definir a semântica de π-*calculus* é inicialmente compreender a noção de congruência estrutural, ou seja, agentes que intuitivamente tem o mesmo comportamento e que podem ser identificados como tal mediante sua estrutura.

Entretanto a congruência estrutural não é suficiente para definir equivalências em álgebras de processos. Faz-se necessária também uma semântica operacional para cada um dos operadores, que neste trabalho será apresentada por um LTS(*Labelled Transition System*) baseado na semântica *early*

2.2.1 Congruência Estrutural

As semânticas para π-*calculus* podem ser definidas utilizando-se o conceito de congruência estrutural. A congruência estrutural caracteriza processos sintaticamente

Conversão Alpha[1]	Se P e Q são variantes por conversão-alpha então $P \equiv Q$
Processo *vazio*	$P \mid \mathbf{O} \equiv P, P + \mathbf{O} \equiv P$
Comutativa	$P \mid Q \equiv Q \mid P, P + Q \equiv Q + P$
Associativa	$(P \mid Q) \mid R \equiv P \mid (Q \mid R), (P + Q) + R \equiv P + (Q + R)$
Extensão de escopo	$(\nu x)\ (\nu y)\ P \equiv (\nu y)\ (\nu x)\ P$
	$(\nu x)\ P \equiv P$ **if** $x \notin \mathbf{fn}(P)$
	$P \mid (\nu x)\ Q \equiv (\nu x)\ (P \mid Q)$ **if** $x \notin \mathbf{fn}(P)$
	$P + (\nu x)\ Q \equiv (\nu x)\ (P + Q)$ **if** $x \notin \mathbf{fn}(P)$
Replicação	$!\alpha.P \mid \alpha.P \equiv !\alpha.P$
	$!\alpha.P \mid !\alpha.P \equiv !\alpha.P$

Tabela 2.1: Leis de congruência estrutural

semelhantes contendo comportamentos iguais.

Dois processos P e Q são estruturalmente congruentes (denotado por $P \equiv Q$) quando seguem as leis descritas na Tabela 2.2.1

2.2.2 Semântica Early

A semântica operacional descreve como ocorrem as transições em *π-calculus*. Existem três principais tipos de semântica para *π-calculus*: a *early*, a *late* e a *open*. No decorrer deste trabalho adotaremos a semântica early, porque as ferramentas com as quais trabalharemos adotam esta semântica.

Podemos observar as regras da semântica *early* na Tabela 2.2.2.

- STRUCT: $\dfrac{P' \equiv P, \quad P \xrightarrow{\alpha} Q, \quad Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$

  Na regra STRUCT se P é congruente a P' e Q é congruente a Q' e através da ação α, P evolui para Q, então P' através da mesma ação α evolui para Q'. Ou seja, processos estruturalmente congruentes podem ser intercambiados.

- SILENT: $\tau.P \xrightarrow{\tau} P$

  Denota uma ação silenciosa, imperceptível para o observador.

- IN: $x(y).P \xrightarrow{x(w)} P\{w/y\} \quad w \notin \mathbf{fn}((\nu y)\ P)$

  Ação de entrada. Caso exista uma ação de entrada no canal x prefixando o processo P, e esta ação ocorra, tendo como entrada w, então deve-se trocar

STRUCT: $$\frac{P' \equiv P,\ \ P \xrightarrow{\alpha} Q,\ \ Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$$

SILENT: $$\tau.P \xrightarrow{\tau} P$$

IN: $$x(y).P \xrightarrow{x(w)} P\{w/y\} \quad w \notin \mathbf{fn}((\nu y)\ P)$$

OUT: $$\overline{x}y.P \xrightarrow{\overline{x}y} P$$

PAR: $$\frac{P \xrightarrow{\alpha} P',\ \ \mathbf{bn}(\alpha) \cap \mathbf{fn}(Q) = \{\,\}}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

COM: $$\frac{P \xrightarrow{a(x)} P',\ \ Q \xrightarrow{\overline{a}u} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

OPEN: $$\frac{P \xrightarrow{\overline{x}y} P',\ \ x \neq y}{(\nu y)\ P \xrightarrow{\overline{x}(w)} P'\{w/y\}}\ w \notin \mathbf{fn}((\nu y)\ P')$$

RES: $$\frac{P \xrightarrow{\alpha} P',\ \ x \notin \alpha}{(\nu x)\ P \xrightarrow{\alpha} (\nu x)\ P'}$$

CLOSE: $$\frac{(\nu w)\ P \xrightarrow{\overline{x}w} P' \quad Q \xrightarrow{x(w)} Q'}{(\nu w)\ P \mid Q \xrightarrow{\tau} (\nu w)\ (P' \mid Q')}$$

BANG: $$\frac{\alpha.P \xrightarrow{\alpha} P'}{!\alpha.P \xrightarrow{\alpha} !\alpha.P \mid P'}$$

Tabela 2.2: Semântica Early

as ocorrências de y em P por w quando este último for diferente dos nomes livres de $(\nu y)\ P$.

- OUT: $\overline{x}y.P \xrightarrow{\overline{x}y} P$

  Ação de saída. Prefixa P de maneira que, quando ocorrer (se ocorrer), faz com que o processo evolua para P.

- PAR: $\dfrac{P \xrightarrow{\alpha} P',\ \ \mathbf{bn}(\alpha) \cap \mathbf{fn}(Q) = \{\,\}}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$

  Caso P evolua para P' através de α e não haja comunicação entre P e Q, então Q continuará o mesmo enquanto P evolui para P' se estiverem executando em paralelo.

- COM: $\dfrac{P \xrightarrow{a(x)} P',\ \ Q \xrightarrow{\overline{a}u} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$

  Efetua a comunicação entre os processos P e Q através das regras IN e OUT. O processo Q envia para o processo P através do canal a o nome u.

- OPEN: $\dfrac{P \xrightarrow{\overline{x}y} P',\ \ x \neq y}{(\nu y)\ P \xrightarrow{\overline{x}(w)} P'\{w/y\}}\ w \notin \mathbf{fn}((\nu y)\ P')$

  Denota como um nome restrito em P pode deixar de sê-lo em P' através da substituição.

- RES: $\dfrac{P \xrightarrow{\alpha} P',\ \ x \notin \alpha}{(\nu x)\ P \xrightarrow{\alpha} (\nu x)\ P'}$

  Restrição. Denota que um nome restrito em P continua restrito em P' depois de efetuada uma ação desde que este nome não faça parte da mesma.

- CLOSE: $\dfrac{(\nu w)\ P \xrightarrow{\overline{x}w} P' \quad Q \xrightarrow{x(w)} Q'}{(\nu w)\ P \mid Q \xrightarrow{\tau} (\nu w)\ (P' \mid Q')}$

  Esta regra denota a extrusão de escopo, ou seja, como o nome w foi comunicado de P para Q tornando-se restrito aos dois processos.

- BANG: $\dfrac{\alpha.P \xrightarrow{\alpha} P'}{!\alpha.P \xrightarrow{\alpha} !\alpha.P \mid P'}$

  Denota o comportamento da replicação. Ou seja, depois de efetuada uma ação em um processo replicante, ainda existem infinitas cópias desse processo em paralelo.

2.2.3 Relações de equivalência

Vamos apresentar agora uma das equivalências de comportamento mais fundamentais denominada *strong* bi-similaridade.

Bi-simulações

São relações de equivalências comportamentais entre processos , de maneira a saber qual o nível de semelhança existe entre eles, sendo assim definidas [18]:

Definição 6 *Uma Bi-simulação* $\sim$ *é uma relação binária simétrica* $\mathcal{R}$ *sobre processos, onde* P *e* Q *são bi-similares se* $P\mathcal{R}Q$ *e* $P \xrightarrow{\alpha} P'$ *implica que* $\exists Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}Q'$

Intuitivamente isto significa que se P pode fazer uma ação, então Q pode fazer a mesma ação, e os processos derivados seguem essa mesma relação.

Definição 7 *Uma strong bi-simulação é uma relação binária simétrica* $\mathcal{R}$ *sobre processos satisfazendo:* $P\mathcal{R}Q$ *e* $P \xrightarrow{\alpha} P'$ *onde* $\mathbf{bn}(\alpha)$ *são novos, implica que:*

- *Se* $\alpha = a(x)$ *então* $\exists Q' : Q \xrightarrow{\alpha} Q' \wedge \forall u : P'\{u/x\}\mathcal{R}Q\{u/x\}'$
- *Se* α *não for uma entrada, então* $\exists Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}Q'$

Ou seja, a *strong* bi-simulação é a união de todas as possíveis bi-simulações, e é denotada por $P \sim .Q$

A verificação de bi-simulação entre um par de processos (P, Q) é realizada gerando todos os pares de processos a partir de (P, Q) que também deverão estar presentes em $\mathcal{R}$. Porém, muitos pares de processos gerados podem ser estruturalmente congruentes, de maneira que a relação contém processos desnecessários para representá-la.

Definiremos agora as bi-simulações *up-to*, que reduzem o tamanho das relações, mas para isso, nos utilizaremos de composições sobre relações binárias da seguinte forma: $\sim \mathcal{S} \sim$. Por exemplo, $P \sim \mathcal{S} \sim Q$ significa que existem P' e Q' tal que, $P \sim P', P'\mathcal{S}Q', Q' \sim Q$.

Definição 8 *Uma bi-simulação up-to é uma relação $\mathcal{S}$ sendo que $P\mathcal{S}Q$ implica em:*

Se $P \xrightarrow{\alpha} P'$, então existe $Q', Q \xrightarrow{\alpha} Q' \wedge P' \sim \mathcal{S} \sim Q'$ Se $Q \xrightarrow{\alpha} Q'$, então existe $P', P \xrightarrow{\alpha} P' \wedge P' \sim \mathcal{S} \sim Q'$

Definição 9 *Dizemos que uma relação $\mathcal{R}$ progride para uma relação $\mathcal{S}$, sendo denotado por $\mathcal{R} \rightarrowtail \mathcal{S}$, se: $(P,Q) \in \mathcal{R} \wedge P \xrightarrow{\alpha} P'$ implica que existe Q' tal que $Q \xrightarrow{\alpha} Q' \wedge (P',Q') \in \mathcal{S}$*

Definição 10 *Uma relação $\mathcal{R}$ é uma bi-simulação se $\mathcal{R} \rightarrowtail \mathcal{R}$. Existem algumas funções de relações para relações que tem a propriedade de garantirem uma bi-simulação, ou seja, $\mathcal{R} \rightarrowtail \mathcal{F}(\mathcal{R})$ implica em $\mathcal{R} \subseteq \sim$.*

Definição 11 *Dada uma função $\mathcal{F}$ sobre relações, dizemos que uma relação $\mathcal{R}$ é uma bi-simulação up-to $\mathcal{F}$ se e somente se, para cada P e Q tal que $P\mathcal{R}Q$, e $P \xrightarrow{\alpha} P'$ existe Q' : $Q \xrightarrow{\alpha} Q'$ e $P'\mathcal{F}(\mathcal{R})Q'$, e de forma simétrica para todas as transições de Q.*

As bi-simulações *up-to* mais freqüentes são:

- Bi-simulações *up-to* nomes livres: Seja σ uma substituição injetora nos $\mathbf{fn}(P_0, Q_0)$, então $\mathcal{F}_{sub}(\mathcal{R}) \stackrel{def}{=} \{(P,Q); \exists (P_0,Q_0) \in \mathcal{R}$, tal que, $P \equiv P_0\sigma \wedge Q \equiv Q_0\sigma\}$
- Bi-simulações *up-to* restrições:
  $\mathcal{F}_{rest}(\mathcal{R}) \stackrel{def}{=} \{(P,Q); \exists (P_0,Q_0) \in \mathcal{R}, \exists x$, tal que, $P \equiv (\nu x)\ P_0 \wedge Q \equiv (\nu x)\ Q_0\}$
- Bi-simulações *up-to* composição paralela:
  $\mathcal{F}_{par}(\mathcal{R}) \stackrel{def}{=} \{(P,Q); \exists (P_0,Q_0) \in \mathcal{R}, \exists T$, tal que, $P \equiv P_0 \mid T \wedge Q \equiv Q_0 \mid T\}$

Congruência

Infelizmente, a bi-simulação não é preservada pela prefixação, desta forma, para definirmos a congruência em termos de bi-simulação é necessário que dois processos sejam bi-similares para toda substituição. Assim, definimos [18]:

Definição 12 *Dois agentes P e Q são strong congruentes, denotados por $P \simeq Q$ se e somente se $P\sigma \sim Q\sigma$ para toda substituição σ*

2.3 Forma Normal

A forma normal de um processo descrito em π-*calculus* garante que processos estruturalmente congruentes, possuam a mesma forma normal, facilitando a verificação das conguências. Em [21], Sangiorgi descreve uma forma normal para processos descritos em π-*calculus*.

Definição 13 *Um processo P em sua forma normal, pode ser descrito através da seguinte sintaxe:*

$$(\nu\vec{x})\left((\alpha_1.P_1)^{m_1} \mid \cdots \mid (\alpha_n.P_n)^{m_n}\right), \quad \textit{onde:}$$

$$n \geq 1, m_i \in \mathbb{N} \cup \{\omega\}$$

$$\forall i : x_i \in \mathbf{fn}\left((\alpha_1.P_1)^{m_1} \mid \cdots \mid (\alpha_n.P_n)^{m_n}\right)$$

$$\forall i, j \in \mathbb{N}_+ \wedge i \neq j \Rightarrow (\alpha_i.P_i \neq \alpha_j.P_j)$$

Onde $\alpha.P^1 = \alpha.P$, $\alpha.P^m = \alpha.P|\alpha.P^{m-1}$ *e* $\alpha.P^\omega = !\alpha.P$.

As formas normais são usadas para comparar processos e para reescrevê-los, podendo torná-los mais simples.

Por exemplo, Hirschkoff, em [7], apresenta um sistema de normalização, e prova que se dois processos P e Q são equivalentes, $P \equiv Q$ então a forma normal de P é igual a forma normal de Q excetuando-se os nomes restritos (que podem ser renomeados sem alteração semântica para que fiquem literalmente iguais).

2.4 Verificação de Modelos

Também é possível verificar propriedades sobre processos descritos em π-*calculus* através da utilização de π-logic [14], que é uma lógica temporal para π-*calculus*.

Definição 14 *A sintaxe de π-logic pode ser assim descrita:*

$$\phi ::= true| \sim \phi|\phi_1 \& \phi_2|EX\{\mu\}\phi| < \mu > \phi|EF\phi$$

E possui a seguinte interpretação:

- $P \models true$ sempre é válido
- $P \models\sim \phi$ é válido se e somente se não valer $P \models \phi$
- $P \models \phi_1 \& \phi_2$ se e somente se $P \models \phi_1$ e $P \models \phi_2$.
- $P \models EX\{\mu\}\phi$ se e somente se existir um P_1 tal que $P \rightarrow^{\mu} P_1$ e $P_1 \models \phi$. Ou seja, se no estado atual, ocorrendo μ, no próximo estado ϕ seja válido.
- $P \models < \mu > \phi$ se e somente se existem $P_0, .., P_n$, $n \geq 1$ tais que $P = P_0 \rightarrow^{\tau} P_1 \rightarrow^{\tau} ... \rightarrow^{\tau} P_{n-1} \rightarrow^{\mu} P_n$ e $P_n \models \phi$. Ou seja, se depois de 1 ou mais ações silenciosas, ocorre uma ação μ que leve a um estado onde ϕ seja válido.
- $P \models EF\phi$ se e somente existem $P_0, ..., P_n$ e $\mu_1, ..., \mu_n$, com $n \geq 0$, tais que $P = P_0 \rightarrow^{\mu_1} P_1 ... \rightarrow^{\mu_n} P_n$ e $P_n \models \phi$. Ou seja, se depois de 0 ou mais ações quaisquer, atinge-se um estado onde ϕ seja válido.

E usualmente, são definidas as seguintes abreviações:

- $\phi_2 | \phi_2$ abrevia $\sim (\sim \phi_1 \& \sim \phi_2)$. Representa uma maneira de expressar o operador lógico **ou** através da lei de De Morgan.
- $[\mu]\phi$ abrevia $\sim < \mu > \sim \phi$. No estado atual, não vale depois de 1 ou mais ações silenciosas e uma ação μ chegue-se a um estado onde não vale ϕ
- $AG\phi$ abrevia $\sim EF \sim \phi$. Este é o operador *always*, e significa que ϕ vale agora e sempre no futuro.

Assim, podemos verificar propriedades em processos descritos em π-*calculus*. Por exemplo, tomemos os seguintes processos:

$$\text{Buf} \stackrel{def}{=} in(x).\text{Buf1}$$
$$\text{Buf1} \stackrel{def}{=} \overline{out}x.\text{Buf}$$

Queremos verificar se Buf respeita a seguinte propriedade: sempre que receber um nome m em algum momento m possa ser enviado através do canal *out*.

Essa propriedade pode ser assim descrita em π-logic: $AG([in(m)]EF < \overline{out}m > true)$

2.5 Ferramentas para Verificação de π-*calculus*

Hoje em dia existem variadas ferramentas de verificação formal que adotam π-*calculus* como teoria, mas as capacidades e mesmo as linguagens de entrada diferem bastante.

Podemos dentre elas citar a VTUBAINA, capaz de verificar equivalências, bi-simulações e simulação; a HAL, que além de verificar bi-simulação, é capaz de verificar propriedades e o MWB(Mobile WorkBench) que também executa verificação de equivalências e propriedades.

Existem outras ferramentas[2] capazes de simulação como, por exemplo, a *Pict*, uma linguagem de programação, que utiliza como entrada uma linguagem funcional baseada em π-*calculus* e que é capaz de simular processos. Ela traduz a linguagem de entrada para um programa na linguagem C, que utiliza suas bibliotecas e então compila-o para efetuar a simulação.

Ainda nas linguagens de programação, capazes de simular agentes, encontra-se a *Tyco*, uma linguagem orientada a objetos baseada em π-*calculus* utilizada principalmente no desenvolvimento de aplicações concorrentes.

Outras ferramentas podem ser vistas no Apêndice A, onde são estudadas com maior detalhamento.

2.6 Considerações Finais

Neste capítulo introduzimos π-*calculus* mostrando a sintaxe, semântica, algumas equivalências e bi-simulações que serão importantes para a compreensão da linguagem de entrada e capacidade das ferramentas a serem integradas. Apresentamos também π-logic como linguagem lógica para a verificação de propriedades para sistemas de processos descritos em π-*calculus*.

Apesar de π-*calculus* possuir um conjunto de combinadores, nem todas as ferramentas contemplam todos e, por isso, existe a dificuldade na integração das mesmas, que do contrário, limitaria-se a uma tradução puramente sintática.

No capítulo seguinte será mostrada a integração entre duas ferramentas de veri-

[2]Para referências sobre as ferramentas, consultar o Apêndice

ficação para π-*calculus* e o que é necessário para fazer com que elas cooperem entre si.

Capítulo 3

HAL e VTUBAINA

3.1 Introdução

Estas ferramentas (HAL e VTUBAINA) foram escolhidas dentre outras, estudadas no apêndice, pelos seguintes motivos:

- Disponibilidade: são ferramentas livres e possuem ampla documentação.
- São ferramentas utilizadas na verificação formal de processos, sendo a HAL bastante utilizada e conhecida.
- Utilizam-se de abordagens diferentes para a verificação.
- Possuem capacidades diferentes de verificação, complementando-se mutuamente.

3.2 Definições Preliminares

3.2.1 HD-Autômato

HD-Autômato (History Dependent Automata)[15], é um LTS (Label Transition System) utilizado para descrever as transições possíveis de um sistema de processos de maneira mais compacta, onde processos que diferem apenas por uma substituição de seus nomes livres são agrupados em um mesmo nó do autômato.

Definição 15 *Dado um processo* P, $clsmap(P) = (Q, \sigma)$, *onde* Q *representa a classe dos processos diferentes de* P *por uma substituição bijetiva* $\sigma : \mathbf{fn}(Q) \rightarrow \mathbf{fn}(P)$, *ou seja,* $P \equiv Q\sigma$.

Definição 16 *Um Autômato Compactado é um Sistema de Transição LTS (Labelled Transition System) definido pelos seguintes componentes:*

- *um conjunto de estados* S;
- *um conjunto* L *de rótulos para as transições;*
- *um estado inicial* $s_0 \in S$;
- *uma função* $\eta : S \rightarrow \mathcal{N}$, *que associa cada estado a um conjunto de nomes;*
- *um conjunto de transições* $T \subseteq S \times L \times S$, *representado por* $\xrightarrow{l,\sigma}$, *onde* $l \in L$ *e* σ *é uma função bijetiva, tal que* $\sigma : \mathcal{N} \rightarrow \mathcal{N}$.

Dado um estado $s_1 \in S$ *associado ao processo* P, *se* $P \xrightarrow{\alpha} P'$ *e* $clsmap(P') = (Q, \sigma)$, *então temos* $s_1 \xrightarrow{\alpha,\sigma} s_2 \in T$, *onde* s_2 *estará associado ao processo* Q, *e* $\eta(s_1) = \eta(s_2)\sigma$.

3.3 HAL

HAL(History Dependant Automata Laboratory)[1] é uma ferramenta integrada de especificação, verificação e análise de processos distribuídos e concorrentes através do uso de HD-Autômatos que englobe um conjunto de ferramentas que permite reescrever processos em π-*calculus* na forma de autômatos para posteriormente utilizar o Jack (uma ferramenta de verificação para CCS, capaz de verificar autômatos também) para efetuar a verificação.

Assim sendo, a HAL é uma iniciativa de integrar um verificador já existente para não ter de reescrevê-lo.

Suas capacidades são implementadas através de 5 módulos:

[1]Disponível para download, juntamente com manual em: http://rep1.iei.pi.cnr.it/projects/JACK/hal.html (última visualização: 16/07/2005)

pi-to-hd : Responsável por transformar os processos descritos em π-*calculus* em um hd-autômato.

hd-reduce : Reduz o número de estados de um hd-autômato de um processo, e conseqüentemente reduz o tempo e o número de passos necessários na verificação.

hd-to-aut : Transforma um hd-autômato em um autômato descrito numa linguagem reconhecida pelo Jack (FC2[2]).

pi-to-actl : Transforma sentenças em π-logic [14] para cláusulas na lógica ACTL [19], reconhecida pelo Jack.

jack : Faz a verificação de bi-simulação early nos autômatos fornecidos ou a checagem de modelo de acordo com as cláusulas lógicas.

Dá suporte a verificação de equivalência comportamental através de bi-simulação early, bem como de propriedades que possam ser expressas através de lógica temporal(π-logic). Entretanto, limita-se a processos com número finito de estados.

3.3.1 Linguagem

A HAL utiliza o seguinte subconjunto de π-*calculus* como linguagem de entrada:

$$P \stackrel{def}{=} \mathbf{O} \mid \alpha.P \mid (\nu x)\ P \mid (P_1 \mid P_2) \mid (P_1 + P_2)$$

$$\alpha \stackrel{def}{=} a(b) \mid \overline{a}b \mid \tau$$

Caracterizando, assim, o sub-conjunto de processos não replicantes em π-*calculus*, ou seja, não possui o operador !.

A linguagem utilizada pode ser vista na Tabela 3.3.1 e no exemplo a seguir:

```
define Buf(in,out)=in?(x).Buf1(in,out,x)
define Buf1(in,out,x)=out!x.Buf(in,out)
```

que seria expresso em π-*calculus* da seguinte maneira: Buf $\stackrel{def}{=}$ $in(x)$.Buf1 e Buf1 $\stackrel{def}{=}$ $\overline{out}x$.Buf.

[2]Para mais informações sobre este formato, consultar o site da HAL

Nome	***π-calculus***	**HAL**
Definição	$P \stackrel{def}{=} Q$	P(**fn**(Q))=Q
Nulo	**O**	
Seqüência	$\alpha.P$	α.P
Restrição	$(\nu x)\ P$	(x)P
Paralelo	$P\|Q$	P|Q
Alternativa	$P+Q$	P+Q
Ação Entrada	$a(b)$	a?(b)
Ação Saída	$\overline{a}b$	a!b
Ação Silenciosa	τ	tau

Tabela 3.1: Relação entre o subconjunto de *π-calculus* e a sintaxe na HAL

3.3.2 Capacidades

A HAL é composta de 5 módulos básicos que se integram de modo a prover um ambiente de especificação e verificação para agentes móveis. Isso é feito através da interligação da saída de certos módulos (em arquivos) com a entrada de outros, provendo as seguintes capacidades:

- Verificação de bi-simulação (early) entre os processos.
- Checagem de propriedades expressas através de fórmulas em lógica temporal (π-logic).
- Geração de autômatos (LTS) que representam os comportamentos dos processos.
- Redução de autômatos, para tornar a verificação mais eficiente.

O fato de lidar com processos com número finitos de estados é justamente o que impede a implementação do operador replicação na ferramenta, uma vez que este pode levar a números infinitos de estados.

3.3.3 Exemplo

Tomemos dois processos:

$$\text{Buf} \stackrel{def}{=} in(x).\text{Buf1}$$

$$\text{Buf1} \stackrel{def}{=} \overline{out}x.\text{Buf}$$

Queremos verificar se Buf e Buf1 são equivalentes. Além disso, gostaríamos de verificar também a propriedade de que uma vez recebida uma mensagem pelo canal *in*, ela sairá eventualmente pelo canal *out* antes de uma mensagem recebida posteriormente, ou seja, a ordem das mensagens é preservada:

Primeiramente descrevemos os dois processos na linguagem da HAL:

```
define Buf(in,out)=in?(x).Buf1(in,out,x)
define Buf1(in,out,x)=out!x.Buf(in,out)

build Buf
build Buf1
```

E fazemos o mesmo com a propriedade a ser verificada:

```
doubleorder=AG([in?m][in?k]<out!m>true)
```

Utilizando o módulo **pi-to-hd**, são gerados os hd-autômatos referentes aos dois processos, como vemos nas Figuras 3.1 e 3.2.

Para tentar reduzir o número de estados dos autômatos, poderíamos utilizar o módulo **hd-reduce**, gerando os autômatos das Figuras 3.3 e 3.4

Agora nos utilizamos do módulo **hd-to-aut** para transformar os hd-autômatos reduzidos para autômatos que o jack possa verificar, e utilizando o **jack**, obtemos que os dois processos não são equivalentes.

Para verificar a propriedade, utilizaremos o módulo **pi-to-actl**, que transforma as sentenças escritas em π-logic para lógica ACTL, aceita pelo jack, e por fim, utilizamos o próprio **jack** para verificarmos a propriedade, utilizando para isso a formula gerada e o autômato de Buf, concluindo que Buf respeita a propriedade.

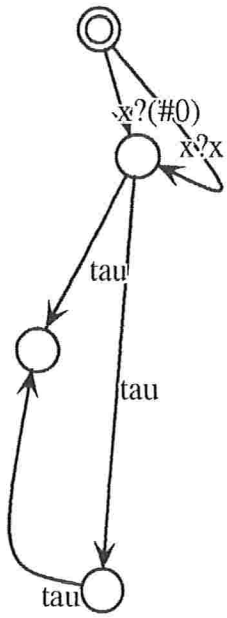

Figura 3.1: HD-autômato para o processo Buf

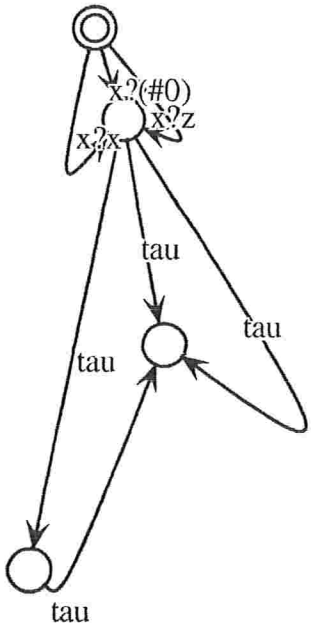

Figura 3.2: HD-autômato para o processo Buf1

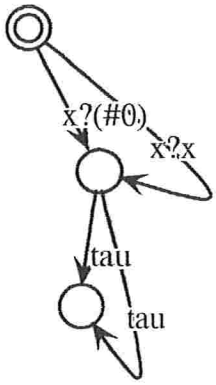

Figura 3.3: HD-autômato reduzido para o processo Buf

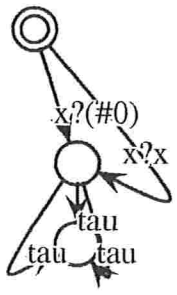

Figura 3.4: HD-autômato reduzido para o processo Buf1

Nome	π-*calculus*	VTUBAINA
Definição	$P \stackrel{def}{=} Q$	P=Q
Nulo	O	
Seqüência	$\alpha.P$	α.P
Restrição	$(\nu x)\ P$	(^ x)P
Paralelo	$P \mid Q$	P|Q
Replicação	$!P$	!P
Ação Entrada	$a(b)$	a(b)
Ação Saída	$\overline{a}b$	a<b>
Ação Silenciosa	τ	tau

Tabela 3.2: Relação entre o subconjunto de π-*calculus* e a sintaxe na VTUBAINA

3.3.4 Limitações

A grande limitação da HAL é a impossibilidade de verificar bi-simulações envolvendo processos com replicação.

Outra limitação (operacional) é a restrição quanto ao ambiente de execução: os aplicativos rodam apenas em ambiente unix, e como o código fonte não é aberto, é impossível recompilá-lo para qualquer outra plataforma. Não há também, por parte dos autores das mesmas, a iniciativa ou promessa da disponibilização da ferramenta para qualquer outra plataforma.

3.4 VTUBAINA

3.4.1 Linguagem

A VTUBAINA trabalha com o seguinte subconjunto de π-*calculus*:

$$P \stackrel{def}{=} \mathbf{O} \mid \alpha.P \mid (\nu x)\ P \mid (P_1 \mid P_2) \mid !\alpha.P$$

$$\alpha \stackrel{def}{=} a(b) \mid \overline{a}b \mid \tau$$

A linguagem utilizada pode ser vista na Tabela 3.4.1 e no exemplo a seguir:

```
define Buf=in(x).Buf1
define Buf1=out<x>.Buf
```

que seria expresso em π-*calculus*da seguinte maneira: Buf $\stackrel{def}{=}$ $in(x)$.Buf1 e Buf1 $\stackrel{def}{=}$ $\overline{out}x$.Buf.

3.4.2 Capacidades

A ferramenta VTUBAINA é o resultado dos estudos sobre técnicas de verificação para π-*calculus* e, por isso, implementa tanto as técnicas de verificação por refinamento de partições [17],verificação *on-the-fly*[3], sistemas de normalização, e prova por bi-simulações *up-to*. A ferramenta efetua as seguintes tarefas:

- Simula as transições de um processo;
- Detecta e visualiza fn() e *nomes restritos* de um processo;
- Aplica apenas o sistema de normalização em um processo, visualizando passo a passo, as regras aplicadas e os resultados gerados na expressão;
- Verifica congruência estrutural entre dois processos;
- Verifica bi-simulações *up-to* entre dois processos;
- Gera o autômato reduzido de transição do processo com desdobramento utilizando apenas os an();
- Imprime o autômato final em formato .fc2 (o mesmo formato utilizado pelo Jack) (Jack tool Set[6]);
- Imprime o autômato final em formato .dot (dotty - Graph Editor[9, 10]).

3.4.3 Exemplo

Tome dois processos:

$$P = \overline{a}x|a(c).\overline{c}j$$

$$Q = \overline{a}x|a(c).\overline{c}j|(\nu d)\ d(g).!\overline{a}x$$

Para verificar se estes processos são bi-similares, descrevemos os processos na linguagem da VTUBAINA:

```
**********************************************************************

V T U B A I N A

a Verification Tool for Up-to Bisimulation and
Automata INtegration Automatization


**********************************************************************

VTUBAINA> agent P=a<x>|a(c).c<j>
Agent P was built successfully!
VTUBAINA> agent Q=a<x>|a(c).c<j>|(^d)(d(g).!a<x>)
Agent Q was built successfully!
VTUBAINA>
```

Agora vamos verificar se eles são bi-similares através do comando **eq**

```
VTUBAINA> eq P Q
Agents P,Q are bisimilar!
VTUBAINA>
```

Suponha agora que queremos simular o comportamento dos dois processos para ter certeza de que realmente se comportam igualmente quando ocorre a ação silenciosa:

```
VTUBAINA> simul P

[0] - Break simulation
[1-n] - Choose transition
1) tau action (free): a<x> com a(#0)
2) extern output action: a<x>
3) extern input action: a(#0)
choice: 1
```

```
P = x<j>
1) extern output action: x<j>
choice: 1
P =

Simulation finished! No more transitions can be made.
VTUBAINA> simul Q

[0] - Break simulation
[1-n] - Choose transition
1) tau action (free): a<x> com a(#2)
2) extern output action: a<x>
3) extern input action: a(#2)
choice: 1
Q = x<j>
1) extern output action: x<j>
choice: 1
Q =

Simulation finished! No more transitions can be made.
VTUBAINA>
```

Ainda utilizando a VTUBAINA, podemos explorar o porquê dos processos serem bi-similares. Se utilizarmos o comando **normalize** para obter a normalização do processo Q e compararmos com o processo P, veremos que eles são não apenas bi-similares, mas também estruturalmente congruentes (na forma normalizada).

```
VTUBAINA> normalize Q

NORMALISATION RULE 6 ACHIEVED:
Q = a<x> | a(#2).#2<j> | (^#0)#0(#1).!a<x>
           -----------   ^^^^^------------

NORMALISATION RULE 6 ACHIEVED:
```

```
Q = a<x> | (^#0)(#0(#1).!a<x> | a(#2).#2<j>)
    ----   ^^^^^----------------------------
Q = (^#0)(a<x> | #0(#1).!a<x> | a(#2).#2<j>)

NORMALISATION RULE 9 ACHIEVED: Q = (^#0)(a<x> | a(#2).#2<j>)

NORMALISATION RULE 5 ACHIEVED:
Q = (^#0)(a<x> | a(#2).#2<j>)
    ^^^^^--------------------
Q = a<x> | a(#2).#2<j>
Agent Q normalized successfully!
VTUBAINA> view P
Agent (SOURCE)  P = a<x>|a(c).c<j>
Agent (CURRENT) P = a<x> | a(#0).#0<j>
VTUBAINA> view Q
Agent (SOURCE)  Q = a<x>|a(c).c<j>|(^d)(d(g).!a<x>)
Agent (CURRENT) Q = a<x> | a(#2).#2<j>
VTUBAINA>
```

Uma vez que fazendo a substituição de $\{\#0/\#2\}$ obteremos exatamente os mesmos processos.

3.4.4 Limitações

A VTUBAINA apesar de aceitar processos com replicações, possui como limitação não aceitar processos com alternativa.

Assim como a HAL, a VTUBAINA também possui uma limitação (operacional) relativa aos sistemas operacionais. Existem versões disponíveis apenas para *Linux* e *Windows*.

3.5 Potencial de Integração

Ambas as ferramentas são capazes de executar verificação de bi-simulação *early*, e de gerar os autômatos representativos de cada processo. Também são capazes de

simular de maneira assistida as transições em um dado processo.

Contudo, apenas a HAL é capaz de verificar propriedades, ao passo que somente a VTUBAINA é capaz de verificar congruência estrutural, verificar bi-simulações *up-to*, calcular a forma normal e visualizar fn() e bn() de um processo.

Assim, quando integramos as duas ferramentas em um ambiente, temos três ganhos:

- A capacidade de escolha da melhor ferramenta para utilizar nos casos onde as capacidades oferecidas sejam as mesmas.
- A possibilidade de utilizar recursos de outra ferramenta para processos especificados em uma linguagem diferente. Por exemplo, posso especificar um sistema na linguagem da VTUBAINA e posteriormente querer verificar propriedades com a HAL.
- O aumento das capacidades isoladas de cada uma das ferramentas, de maneira que processos que seriam antes recusados como entrada, podem ser transformados por uma das ferramentas em uma entrada aceitável. Por exemplo, um processo pode conter replicação, que a HAL não aceita. Mas, ao passar pela VTUBAINA, a forma normal do processo pode não conter replicação, o que o tornaria aceitável para a HAL.

3.6 Exemplo de Integração

Suponha, como caso de uso, um software que possui como parte importante um *buffer* modelado da seguinte forma:

$$\text{Buf} \stackrel{def}{=} in(x).\text{Buf1}$$

$$\text{Buf1} \stackrel{def}{=} \overline{out}x.\text{Buf}$$

que foi especificado da seguinte maneira, utilizando a VTUBAINA:

```
Buf=in(x).Buf1
Buf1=out<x>.Buf
```

O comportamento deste buffer foi simulado utilizando a especificação do software. Além da simulação, tal *buffer* demonstrou possuir o comportamento esperado, e possuir equivalência com o seguinte processo:

$$\text{BufGenerico} \stackrel{def}{=} in(x).\overline{out}x.\text{BufGenerico}$$

Por razão de eficiência, em um dado momento, gostaríamos de aumentar o tamanho desse buffer para conter não apenas uma mensagem, mas duas mensagens, e modelamos um novo buffer da seguinte forma:

$$\text{NovoBuf} \stackrel{def}{=} in(x).\text{NovoBuf1}(x)$$

$$\text{NovoBuf1(x)} \stackrel{def}{=} in(y).\text{NovoBuf2(x,y)} + \overline{out}x.\text{NovoBuf}$$

$$\text{NovoBuf2(x,y)} \stackrel{def}{=} \overline{out}x.\text{NovoBuf1(y)}$$

Infelizmente, a alternativa não faz parte do fragmento de π-*calculus* aceito pela VTUBAINA. A solução, portanto, será integrar a VTUBAINA à HAL, a qual aceita a alternativa na sua linguagem. Mapeando o buffer que já havíamos testado com a VTUBAINA, obtemos de maneira automática e livre de erros, uma especificação que pode ser comparada com a especificação do novo *buffer* .

Feito isso, descobriríamos que os dois buffers não são equivalentes, o que é esperado, pois um deles é capaz de armazenar uma mensagem a mais. Entretanto ,por exemplo, mesmo não possuindo comportamentos equivalentes, estes agentes podem ainda preservar propriedades semelhantes para a aplicação em questão, nos interessa apenas que o *buffer* repasse as mensagens na ordem correta. Então, poderíamos checar a seguinte propriedade na HAL:

```
define doubleorder=AG([in?m][in?k]<out!m>true)
```

Descobriríamos, assim, que ambos os buffers satisfazem a dada propriedade. Mas, isso só seria possível mediante a integração das ferramentas. A VTUBAINA desempenhando um papel fundamental na simulação do software em seu estágio inicial e a HAL fornecendo a verificação da equivalência e de propriedades.

3.7 Considerações Finais

Neste capítulo apresentamos as capacidades e as limitações das duas ferramentas, bem como a sintaxe e semântica de suas linguagens.

Apresentamos também a possibilidade de integração entre a VTUBAINA e a HAL desde que exista uma maneira de converter os processos especificados na linguagem de uma para a linguagem da outra, e mostramos que esta integração pode, inclusive, aumentar o potencial de verificação que as ferramentas possuíam separadamente.

No capítulo seguinte, nos ocuparemos de definir os mapeamentos sintáticos e semânticos entre as ferramentas.

Capítulo 4

Mapeamentos Sintáticos e Semânticos

4.1 Introdução

Este capítulo apresenta um estudo teórico sobre a integração de duas ferramentas de verificação formal: VTUBAINA e HAL. O estudo aborda o mapeamento sintático entre operadores das ferramentas, bem como o mapeamento semântico entre operadores distintos já que as ferramentas utilizam subconjuntos distintos de π-*calculus*.

A integração dessas ferramentas é um objetivo almejável visto que reescrever as ferramentas para que efetuem novas verificações é muito trabalhoso (porque as técnicas são trabalhosas de serem implementadas) e, acima de tudo, porque as ferramentas criadas não estariam verificadas formalmente, o que as tornaria mais propensas a erros. Hoje em dia, quando é necessário recorrer a mais de uma ferramenta para fazer verificações, dado que cada uma possui capacidades diferentes, é necessário traduzir manualmente a especificação do sistema para as linguagens de entrada de cada ferramenta, o que é propenso a erros.

Assim, o desenvolvimento de um ambiente integrado não só minimizaria o custo da verificação, como diminuiria o trabalho do usuário, uma vez que o ambiente se encarregaria de fazer os mapeamentos entre uma ferramenta e outra de forma automatizada, minimizando mais uma vez a probabilidade de erros.

Este trabalho é um estudo sobre a integração de verificadores formais para π-*calculus*, especificamente.

4.1.1 Exemplo de Integração

Tomemos como exemplo o seguinte processo:

$$P = Y \mid (\nu x)\ !x(b).Q \mid Z$$

Sendo Y e Q processos sem alternativas ou replicações. Por questões de eficiência, poderíamos verificar se este processo P é bi-similar ao processo $P' = Y \mid !Q \mid Z$ na HAL.

Porém a HAL seria incapaz de efetuar a verificação uma vez que a replicação não faz parte do subconjunto de π-*calculus* com o qual ele trabalha. Mas, se utilizarmos a VTUBAINA para extrair a forma normal do processo P, teríamos como saída: $P = Y \mid Z$, um processo que a HAL consegue verificar. Isso significa que, neste caso, podemos usar a VTUBAINA como pré-processador para a HAL que só poderá ser usada para verificações desse processo mediante tal pré-processamento. Esta e outras formas de integração das ferramentas, serão os objeto de estudo deste capítulo.

4.2 Definições Preliminares

Para a integração das ferramentas, são necessários dois tipos de mapeamentos: o sintático e o semântico. O sintático porque as ferramentas utilizam linguagens com sintaxes diferentes. Desta forma, é necessário traduzir os processos descritos na linguagem de origem para a linguagem destino. O semântico porque as duas ferramentas não lidam com o mesmo subconjunto de π-*calculus*. Assim, em alguns casos, é preciso reescrever o processo não apenas com mudanças sintáticas, porque o processo resultante não seria aceito pela ferramenta, mas com alterações na estrutura semântica quando isso é possível.

Para fazer a integração entre as ferramentas, faremos os mapeamentos sintático e semântico entre a HAL e a VTUBAINA. Mapearemos as capacidades de cada uma delas em termos do que cada uma delas pode receber como entrada ou produzir como saída, mediante suas capacidades de processamento.

Definição 17 *Utilizaremos os termos assim definidos durante o mapeamento de*

entradas e saídas:

Processos: *São processos descritos na linguagem de entrada da ferramenta da qual estamos tratando. Denotamos por ProcessoHAL os processos descritos na linguagem da HAL e por ProcessoVTUB os processos descritos na linguagem da VTUBAINA.*

HD-Autômatos: *HD autômato de algum processo, descrito na linguagem da ferramenta.*

FC2: *Autômato descrito na linguagem FC2, usada como entrada para o Jack.*

DOT: *Autômato descrito na linguagem DOT, normalmente utilizada para a impressão de autômatos pela ferramenta GraphViz.*

Tela: *Informações enviadas para a tela. Como este item é muito variável, em cada caso serão apresentados mais detalhes.*

Propriedade: *Sentenças lógicas descritas em ACTL.*

4.3 Análise do Potencial de Cooperação das Ferramentas

As ferramentas básicas da HAL possuem as seguinte capacidades:

pi-to-hd : ProcessoHAL -> HD-Autômato

hd-reduce : HD-Autômato -> HD-Autômato reduzido

hd-to-aut : HD-Autômato -> FC2

pi-to-actl : π-logic -> ACTL

atg : FC2 -> Tela (Autômato é exibido na tela graficamente)

hd-atg : HD-Autômato -> Tela (Exibindo o autômato descrito)

amc : ProcessoHAL, Propriedade -> Tela ou ProcessoHAL, ProcessoHAL -> Tela (Informando se a sentença lógica é satisfeita pelo processo. Em caso negativo, apresenta um contra-exemplo)

A VTUBAINA possui as seguintes capacidades:

eq : ProcessoVTUB, ProcessoVTUB -> Tela (Exibe se os processos são ou não estruturalmente equivalentes)

onthefly : ProcessoVTUB, ProcessoVTUB -> Tela (Exibe se os processos são ou não *early* bissimilares)

uptos : ProcessoVTUB, ProcessoVTUB -> Tela (Exibe se os processos são ou não bissimilares via simulação up-to)

savefc2 : ProcessoVTUB -> FC2

savedot : ProcessoVTUB -> DOT

simul : ProcessoVTUB -> Tela (Simula a evolução de um processo de maneira assistida pelo usuário, de forma que este possa interagir para escolher a próxima ação a ser executada)

normalize : ProcessoVTUB -> ProcessoVTUB

cong-est : ProcessoVTUB, ProcessoVTUB -> Tela (Verifica se os processos são estruturalmente congruentes)

Dadas as entradas e saídas de cada uma das ferramentas, podemos ver que não existem possibilidades de comunicação a partir da HAL para a VTUBAINA. Isto porque nenhum tipo de saída emitida pela HAL é compatível com qualquer tipo de entrada requerida pela VTUBAINA. Mas o contrário pode ser feito de duas formas: através do FC2, que é um formato de arquivo padrão, ou através de processos. Em outras palavras, as ferramentas podem ser utilizadas em conjunto através do comando **normalize** da VTUBAINA.

Retomando o exemplo do começo do capítulo, $P = Y \mid (\nu x)\ !x(b).Q \mid Z$, sendo Y e Q processos sem alternativas ou replicações. Reescrevendo-o na linguagem da VTUBAINA, poderíamos utilizar a VTUBAINA e sua função **normalize**, cujo resultado seria o processo $P = Y \mid Q$, mas na linguagem da VTUBAINA. Para conseguir repassar este processo para que a HAL verifique, é necessário traduzi-lo para a linguagem com a qual a HAL possa lidar.

Nome	π-*calculus*	HAL
Definição	$P \stackrel{def}{=} Q$	P(**fn**(Q))=Q
Nulo	O	
Seqüência	$\alpha.P$	α.P
Restrição	$(\nu x)\ P$	(x)P
Paralelo	$P\|Q$	P|Q
Alternativa	$P+Q$	P+Q
Ação Entrada	$a(b)$	a?(b)
Ação Saída	$\overline{a}b$	a!b
Ação Silenciosa	τ	tau

Tabela 4.1: Relação entre o subconjunto de π-*calculus* e a sintaxe na HAL

Como a HAL e a VTUBAINA usam subconjuntos diferentes de π-*calculus* como linguagens de entrada, nem todo processo que a VTUBAINA admite para processamento, e pode inclusive expressar através de uma saída, é reconhecido pela HAL.

Desta forma, é preciso uma conversão não apenas sintática, mas uma análise semântica dos processos para determinar como fazer a conversão entre as ferramentas e o que é mais importante: se é possível fazer esta conversão.

4.3.1 Mapeamentos sintáticos e semânticos

Como discutido anteriormente, a integração das ferramentas só é possível mediante os mapeamentos entre entrada e saída (sintática) e entre domínios (semânticos) de processos. As seções que seguem mostram estes dois tipos de mapeamentos separadamente.

Definição 18 *A HAL utiliza-se do seguinte subconjunto:*

$$P \stackrel{def}{=} \mathrm{O} \mid \alpha.P \mid (\nu x)\ P \mid (P_1 \mid P_2) \mid (P_1 + P_2)$$

$$\alpha \stackrel{def}{=} a(b) \mid \overline{a}b \mid \tau$$

Tomemos como exemplo o processo $Q = a(x).\overline{b}y.\overline{b}x|(\nu z)\ \overline{a}z$. Sua representação na linguagem utilizada pela HAL seria:

```
Q(a,b,y)=a?(x).b!y.b!x|(z)a!z
```

Nome	π-*calculus*	VTUBAINA
Definição	$P \stackrel{def}{=} Q$	P=Q
Nulo	**O**	
Seqüência	$\alpha.P$	α.P
Restrição	$(\nu x)\ P$	(^ x)P
Paralelo	$P\|Q$	P|Q
Replicação	$!P$	!P
Ação Entrada	$a(b)$	a(b)
Ação Saída	$\overline{a}b$	a<b>
Ação Silenciosa	τ	tau

Tabela 4.2: Relação entre o subconjunto de π-*calculus* e a sintaxe na VTUBAINA

Por outro lado, a VTUBAINA trabalha com o seguinte sub-conjunto de π-*calculus*:

Definição 19 *Subconjunto de* π-calculus *utilizado pela VTUBAINA:*

$$P \stackrel{def}{=} \mathbf{O} \mid \alpha.P \mid (\nu x)\ P \mid (P_1 \mid P_2) \mid !\alpha.P$$

$$\alpha \stackrel{def}{=} a(b) \mid \overline{a}b \mid \tau$$

O mesmo processo Q do exemplo anterior seria assim descrito na VTUBAINA:

```
Q=a(x).b<y>.b<x>|(^z)(a<z>)
```

Note que o processo Q pode ser especificado tanto na VTUBAINA quanto na HAL porque usa apenas operadores contidos no subconjunto de π-*calculus* implementado nas duas ferramentas em questão. Em outros casos um processo pode ser verificado por apenas uma das ferramentas. Apesar do processo poder ser utilizado em ambas as ferramentas, as linguagens de entrada são diferentes.

Para solucionar o problema, definiremos a função $\mathbf{map}_{\mathrm{Vtub}\rightarrow\mathrm{Hal}}()$ que faz o mapeamento de processos escritos no padrão de entrada da VTUBAINA para processos reconhecidos pela HAL. Definimos também $\mathbf{map}_{\mathrm{Hal}\rightarrow\mathrm{Vtub}}()$ como a função inversa de $\mathbf{map}_{\mathrm{Vtub}\rightarrow\mathrm{Hal}}()$, ou seja, responsável por mapear os processos da HAL para a VTUBAINA. Assim, temos:

Definição 20 $\mathbf{map}_{Vtub \rightarrow Hal}()$ *é uma função que mapeia os processos da VTUBAINA para processos da HAL:*

$$\mathbf{map}_{Vtub \rightarrow Hal}() : P_{VTUBAINA} \rightarrow P_{HAL}$$

A função $\mathbf{map}_{Vtub \rightarrow Hal}()$ *é sintaticamente definida como:*

$$\mathbf{map}_{Vtub \rightarrow Hal}(tau) \rightarrow tau \tag{4.1}$$

$$\mathbf{map}_{Vtub \rightarrow Hal}(a(x)) \rightarrow a?(x) \tag{4.2}$$

$$\mathbf{map}_{Vtub \rightarrow Hal}(a\langle x\rangle) \rightarrow a!x \tag{4.3}$$

$$\mathbf{map}_{Vtub \rightarrow Hal}(P = Q) \rightarrow P(\mathbf{fn}(Q)) = \mathbf{map}_{Vtub \rightarrow Hal}(Q) \tag{4.4}$$

$$\mathbf{map}_{Vtub \rightarrow Hal}(\alpha.P) \rightarrow \mathbf{map}_{Vtub \rightarrow Hal}(\alpha).\mathbf{map}_{Vtub \rightarrow Hal}(P) \tag{4.5}$$

$$\mathbf{map}_{Vtub \rightarrow Hal}((\hat{}\, x)P) \rightarrow (x)\mathbf{map}_{Vtub \rightarrow Hal}(P) \tag{4.6}$$

$$\mathbf{map}_{Vtub \rightarrow Hal}(P_1|P_2) \rightarrow \mathbf{map}_{Vtub \rightarrow Hal}(P_1)|\mathbf{map}_{Vtub \rightarrow Hal}(P_2) \tag{4.7}$$

$$\mathbf{map}_{Vtub \rightarrow Hal}(()) \rightarrow \tag{4.8}$$

De maneira similar, definimos a função inversa:

Definição 21 *Definimos* $\mathbf{map}_{Hal \rightarrow Vtub}()$ *como a função que mapeia os processos da HAL para processos da VTUBAINA:*

$$\mathbf{map}_{Hal \rightarrow Vtub}() : P_{HAL} \rightarrow P_{VTUBAINA}$$

A função $\mathbf{map}_{Hal \rightarrow Vtub}()$ *é sintaticamente definida como:*

$$\mathbf{map}_{Hal \rightarrow Vtub}(tau) \rightarrow tau \tag{4.9}$$

$$\mathbf{map}_{Hal \rightarrow Vtub}(a?(x)) \rightarrow a(x) \tag{4.10}$$

$$\mathbf{map}_{Hal \rightarrow Vtub}(a!x) \rightarrow a\langle x\rangle \tag{4.11}$$

$$\mathbf{map}_{Hal \rightarrow Vtub}(P(\mathbf{fn}(Q)) = Q) \rightarrow P = \mathbf{map}_{Hal \rightarrow Vtub}(Q) \tag{4.12}$$

$$\mathbf{map}_{Hal \rightarrow Vtub}(\alpha.P) \rightarrow \mathbf{map}_{Hal \rightarrow Vtub}(\alpha).\mathbf{map}_{Hal \rightarrow Vtub}(P) \tag{4.13}$$

$$\mathbf{map}_{Hal \rightarrow Vtub}((x)P) \rightarrow (\hat{}\, x)\mathbf{map}_{Hal \rightarrow Vtub}(P) \tag{4.14}$$

$$\mathbf{map}_{Hal \rightarrow Vtub}(P_1|P_2) \rightarrow \mathbf{map}_{Hal \rightarrow Vtub}(P_1)|\mathbf{map}_{Hal \rightarrow Vtub}(P_2) \tag{4.15}$$

$$\mathbf{map}_{Hal \rightarrow Vtub}(()) \rightarrow \tag{4.16}$$

Esse mesmo tipo de mapeamento, apenas sintático, não pode ser aplicado nem para a replicação, presente na VTUBAINA e nem para a alternativa presente na HAL, visto que não fazem parte do fragmento de π-*calculus* comum a ambas as ferramentas.

Tomemos o exemplo do início da seção, descrito na linguagem da VTUBAINA e façamos o mapeamento para a linguagem da HAL:

```
map-vtub-hal(Q)=> map-vtub-hal(Q=a(x).b<y>.b<x>|(^z)(a<z>))
map-vtub-hal(Q)=> Q(a,b,y)=map-vtub-hal(a(x).b<y>.b<x>|(^z)(a<z>))
map-vtub-hal(Q)=> Q(a,b,y)=map-vtub-hal(a(x).b<y>.b<x>)|
                           map-vtub-hal((^z)(a<z>))
map-vtub-hal(Q)=> Q(a,b,y)=map-vtub-hal(a(x)).map-vtub-hal(b<y>).
                           map-vtub-hal(b<x>)|(z)(map-vtub-hal(a<z>))
map-vtub-hal(Q)=> Q(a,b,y)=a?(x).b!y.b!x|(z)(a!z)
```

Assim, o processo Q=a(x).b<y>.b<x>|(ẑ)(a<z>) descrito para a VTUBAINA pode ser sintaticamente transformado em Q(a,b,y)=a?(x).b!y.b!x|(z)(a!z) aceito pela HAL.

4.3.2 Mapeamentos semânticos

O processo $P \stackrel{def}{=} !a(x).P$, por exemplo, pode ser um processo de entrada para a VTUBAINA, mas não é um processo válido para a HAL, porque não faz parte do fragmento de π-*calculus* reconhecido para processamento pela HAL. Por isso, nem todos os processos podem ser reescritos sintaticamente entre os verificadores.

Assim, nem todos os processos π-*calculus* podem ser intercambiados entre as ferramentas dadas as restrições de operadores de cada uma delas. Uma das formas de forçar que os processos possam ser manipulados por ambas as ferramentas, seria simplesmente descartando todos os processos que possuam replicações ou alternativas. Mas esta é uma solução muito restritiva e limitaria o uso de cada ferramenta individualmente. Assim desenvolvemos um estudo do mapeamento semântico dos operadores de replicação e alternativa, como veremos a seguir.

Mapeamento semântico da replicação

Os processos que possuem replicação dividem-se em dois grupos: os que geram um autômato com um número **infinito** de estados e os que geram autômatos com um número **finito** de estados.

Ainda podemos subdividir os processos replicantes com número finito de estados em mais dois grupos: os que podem ser reescritos sem a replicação (através de congruências), e os que não podem.

Definição 22 *Um processo Q é não replicante (denotado por $\mathcal{NR}$) se sua forma normal pode ser escrita na seguinte sintaxe:*

$$P \stackrel{def}{=} \mathbf{O} \mid \alpha.P \mid (\nu x)\ P \mid (P_1 \mid P_2) \mid P_1 + P_2$$
$$\alpha \stackrel{def}{=} a(b) \mid \overline{a}b \mid \tau$$

Logo, $Q \in \mathcal{NR}$

Por exemplo, o processo $P = !(\nu a)\ a(x).P_1 + \overline{b}y$. Como a é um nome restrito ao processo P e não existe nenhuma ação de saída para este canal, esta ação não poderá ser executada, e este processo é equivalente a $P = \overline{b}y$.

Os processos que podem ser reescritos sem o uso da replicação, estão contidos na semântica e na sintaxe com a qual a HAL consegue lidar, uma vez que o operador replicação de π-*calculus* não é implementado pela HAL. Desta forma, processos que podem ser reescritos sem replicação podem ser mapeados e a comunicação entre os dois verificadores pode ocorrer.

Para efetuar a comunicação desses processos entre a VTUBAINA e a HAL, adicionaremos mais uma regra à função $\mathbf{map}_{\mathrm{Vtub}\rightarrow\mathrm{Hal}}()$ assim definida:

Definição 23 *Se $!\alpha.P \equiv Q$, e Q for um processo não replicante ($Q \in \mathcal{NR}$), então* $\mathbf{map}_{Vtub\rightarrow Hal}(\ !\alpha.P) \rightarrow \mathbf{map}_{Vtub\rightarrow Hal}(Q)$

Esta proposição é correta uma vez que processos estruturalmente congruentes têm as mesmas capacidades. Entretanto, resta saber quando é possível achar um processo não replicante congruente ao processo que possui replicação e como fazer isso.

Mapeamento sintático da replicação

Para descobrir um processo semanticamente congruente e que não possua replicação, faremos uso (quando possível) da reescritura sintática dos processos e suas formas normais. Como vimos, cada processo possui apenas uma única forma normal, sendo estruturalmente congruente a ela.

A normalização, segundo proposta por Hirshkoff em [7] é obtida através da aplicação reiterada das regras enunciadas a seguir. Também no mesmo trabalho, mostra-se que a aplicação das regras reiteradamente converge para a forma normal, sendo finito o número de vezes que podem ser aplicadas.

A forma normal de um processo em *π-calculus* pode ser calculada através da aplicação das regras do seguinte sistema de reescritura:

Definição 24 REGRA 1: $P \mid \mathbf{O} \longrightarrow P$

REGRA 2: $P \mid Q \equiv Q \mid P$

REGRA 3: $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

REGRA 4: $(\nu x)\ (\nu y)\ (P) \equiv (\nu y)\ (\nu x)\ (P)$

REGRA 5: $(x \notin \mathbf{fn}(P)) \Rightarrow (\nu x)\ P \equiv P$

REGRA 6: $(x \notin \mathbf{fn}(P)) \Rightarrow P \mid (\nu x)\ Q \equiv\ (\nu x)\ (P \mid Q)$

REGRA 7: $!\alpha.P \mid \alpha.P \equiv\ !\alpha.P$

REGRA 8: $!\alpha.P \mid !\alpha.P \equiv\ !\alpha.P$

REGRA 9: *se* x *ocorre em* α *na posição de sujeito* $\Rightarrow (\nu x)\ \alpha.P \equiv \mathbf{O}$

A regra 9 é responsável pela eliminação de parte dos processos. Quando ela detecta uma ação impossível de ocorrer, ela imediatamente substitui o processo pelo processo nulo, uma vez que ambos têm o mesmo comportamento. Ocasionalmente, a parte substituída pode justamente ser a parte do processo que continha a replicação.

Com isso, podemos fazer uso da forma normal do processo $!\alpha.P$, gerada pela VTUBAINA, e repassá-la à HAL para a verificação.

Para esta transformação não existe uma reversa, visto que os processos expressos na linguagem da HAL nunca conterão uma replicação pois este operador não faz parte da linguagem aceita pela ferramenta.

Exemplo de mapeamento da replicação

Tomemos o processo $P \stackrel{def}{=} !\,(\nu x)\; x(y).\overline{y}z$. Para mapear semanticamente o operador replicação, utilizaremos a forma normal apresentada acima.

Utilizando a regra 9, temos que $P \equiv !\,\mathrm{O}$, pois em $(\nu x)\; x(y).\overline{y}z$, o nome restrito x aparece pa posição de sujeito da ação $x(y)$. E como $!\,\mathrm{O} \equiv \mathrm{O}$, temos que $P \equiv \mathrm{O}$. Sendo este processo livre do operador replicação.

Mapeando semântico da alternativa

O operador de alternativa, presente na linguagem da HAL não pode ser traduzido para a VTUBAINA de maneira simples. De uma forma geral, a alternativa pode ser reescrita, conforme consta em [18], da seguinte maneira:

Definição 25 *Um processo da forma* $y_1(x).P_1 + ... + y_n(x).P_n$ *pode ser reescrito sem o operador alternativa da seguinte maneira:*

$y_1(x).P_1 + ... + y_n(x).P_n \rightarrow$
$(\nu a)\; (\overline{a}t \mid (y_1(x).a(z).(\overline{a}f \mid \textit{if } z = t \textit{ then } P_1 \textit{ else } \overline{y_1}x)|...|$
$y_n(x).a(z).(\overline{a}f \mid \textit{if } z = t \textit{ then } P_n \textit{ else } \overline{y_n}x)$

Mas a VTUBAINA também não possui o operador if, impossibilitando portanto um mapeamento.

Uma outra maneira de se mapear o operador de alternativa, é transformá-lo no operador paralelo quando possível. Isto é feito através do uso da lei de expansão assim definida [11]:

Definição 26 $a(x).P|\overline{a}b.Q \equiv a(x).P|Q + P\{x/\textit{entrada de } a\}|\overline{a}b.Q + \tau.P\{x/b\}|Q$

Como a transformação denota uma congruência, quando o processo apresenta o operador alternativa da maneira como descrita na definição, tal processo pode ser reescrito com o operador paralelo sem alteração semântica. Mas esta é uma forma restritiva de uso.

Entretanto, existem dois casos especiais onde o operador de alternativa pode ser reescrito com o operador paralelo. Um dos casos acontece quando os processos se comunicam e o outro quando não há comunicação entre os processos.

Definição 27 *Dois processos* $\alpha_1.P_1|\alpha_2.P_2$ *se comunicam se e somente se:*

1. $\alpha_1 \in \mathcal{I}$ *e* $\alpha_2 \in \mathcal{O}$ *e* $ch(\alpha_1) = ch(\alpha_2)$ *ou*

2. $\alpha_1 \in \mathcal{O}$ *e* $\alpha_2 \in \mathcal{I}$ *e* $ch(\alpha_1) = ch(\alpha_2)$

Tomando como referência um processo Q que possua o operador alternativa, apresentaremos a caracterização sintática onde dois processos P_1 e P_2, prefixados respectivamente por ações α_1 e α_2, podem ser reescrito com o operador paralelo.

Definição 28 *Sejam* $\alpha_1.P_1$ *e* $\alpha_2.P_2$ *dois processos que não se comunicam. Se* $Q \stackrel{def}{=} \alpha_1.(P_1|\alpha_2.P_2) + \alpha_2.(\alpha_1.P_1|P_2)$ *então* $Q = \alpha_1.P_1|\alpha_2.P_2$ *e* $\mathbf{map}_{Hal \rightarrow Vtub}(Q) = \mathbf{map}_{Hal \rightarrow Vtub}(\alpha_1.P_1|\alpha_2.P_2)$

Definição 29 *Sejam* $\alpha_1.P_1$ *e* $\alpha_2.P_2$ *dois processos que se comunicam. Então,*

Caso 1: Se $\alpha_1 \in \mathcal{I}$ *e* $\alpha_2 \in \mathcal{O}$ *tal que*

$Q \stackrel{def}{=} \alpha_1.(P_1|\alpha_2.P_2) + \alpha_2.(\alpha_1.P_1|P_2) + \tau.(P_1\{obj(\alpha_2)/obj(\alpha_1)\}|P_2)$ *e* τ *representar a comunicação entre* α_1 *e* α_2 *então*

$Q = \alpha_1.P_1|\alpha_2.P_2$ *e* $\mathbf{map}_{Hal \rightarrow Vtub}(Q) = \mathbf{map}_{Hal \rightarrow Vtub}(\alpha_1.P_1|\alpha_2.P_2)$

Caso 2: Se $\alpha_1 \in \mathcal{O}$ *e* $\alpha_2 \in \mathcal{I}$ *tal que*

$Q \stackrel{def}{=} \alpha_1.(P_1|\alpha_2.P_2) + \alpha_2.(\alpha_1.P_1|P_2) + \tau.(P_1|P_2\{obj(\alpha_1)/obj(\alpha_2)\})$ *e* τ *representar a comunicação entre* α_1 *e* α_2 *então*

$Q = \alpha_1.P_1|\alpha_2.P_2$ *e* $\mathbf{map}_{Hal \rightarrow Vtub}(Q) = \mathbf{map}_{Hal \rightarrow Vtub}(\alpha_1.P_1|\alpha_2.P_2)$

Uma última regra pode ainda ser derivada das leis de congruência estrutural entre processos. Em específico, da lei:

$$P \sim P + P$$

Logo, se temos um processo da forma $Q \stackrel{def}{=} P+P'$ e $P \sim P'$ então, $Q \stackrel{def}{=} P+P$ e logo $Q \equiv P$ pela regra de congruência estrutural, eliminando assim a alternativa.

Para isso é preciso verificar a equivalência entre os processos. Se P e P' forem processos que não contenham o operador alternativa, então eles podem ser mapeados sintaticamente para a linguagem da VTUBAINA, que pode verificar a equivalência de processos.

Definição 30 *Se* $Q \stackrel{def}{=} P + P'$ *e* P *for equivalente a* P' $(P \sim P')$ *então,* $\mathbf{map}_{Hal \rightarrow Vtub}(Q) \rightarrow \mathbf{map}_{Hal \rightarrow Vtub}(P)$ *ou equivalentemente,* $\mathbf{map}_{Hal \rightarrow Vtub}(Q) \rightarrow \mathbf{map}_{Hal \rightarrow Vtub}(P')$

Exemplo de mapeamento do operador alternativa

Considere o processo $Q \stackrel{def}{=} \overline{c}d.(\overline{z}x \mid \overline{a}b.\overline{k}w) + \overline{a}b.(\overline{c}d.\overline{z}x \mid \overline{k}w)$.

Podemos constatar que o processo Q é composto por um operador alternativa, entre dois processos $Q1 \stackrel{def}{=} \overline{c}d.(\overline{z}x \mid \overline{a}b.\overline{k}w)$ e $Q2 \stackrel{def}{=} \overline{a}b.(\overline{c}d.\overline{z}x \mid \overline{k}w)$.

Os processos $Q1$ e $Q2$ não se comunicam, pois $\overline{c}d \in \mathcal{O}$ e $\overline{a}b \in \mathcal{O}$. Mais que isso, se chamarmos de $\alpha_1 \stackrel{def}{=} \overline{c}d$ e de $\alpha_2 \stackrel{def}{=} \overline{a}b$, e chamarmos de $P_1 \stackrel{def}{=} \overline{z}x$ e de $P_2 \stackrel{def}{=} \overline{k}w$, recaímos no primeiro caso do mapeamento de processos que não se comunicam, pois $\alpha_1 \in \mathcal{O}$ e $\alpha_2 \in \mathcal{O}$ e $Q \stackrel{def}{=} \alpha_1.(P_1|\alpha_2.P_2) + \alpha_2.(\alpha_1.P_1|P_2)$ então temos que $Q \equiv \alpha_1.P_1|\alpha_2.P_2$, ou seja, $Q \equiv \overline{c}d.\overline{z}x \mid \overline{a}b.\overline{k}w$, sendo este processo livre do operador alternativa.

4.4 Considerações Finais

Uma vez estudadas as duas ferramentas, HAL e VTUBAINA, descobrimos que:

- É possível integrar as duas ferramentas através de mapeamentos.
- Infelizmente mapeamentos sintáticos não são suficientes, portanto nos utilizamos também de mapeamentos semânticos, o que torna a tarefa mais trabalhosa e dispendiosa.
- Mesmo assim, a tarefa é menos dispendiosa que reescrever os verificadores, e ao mesmo tempo, é capaz de ampliar, em certos casos, as capacidades de

verificação que as ferramentas teriam separadamente através da cooperação entre as mesmas.

Desta forma, iremos descrever no capítulo seguinte como implementar a solução de cooperação estudada, comprovando os resultados teóricos obtidos neste capítulo.

Capítulo 5

Hydra

5.1 Introdução

Para estudar de maneira prática a teoria dos mapeamentos apresentados no capítulo anterior, e com o intuito de integrar as duas ferramentas de verificação estudadas, desenvolvemos um protótipo executável capaz de aplicar os mapeamentos e integrar as duas ferramentas.

5.2 Requisitos

A Hydra tem como requisitos básicos as seguintes funcionalidades:

1. Ser capaz de reconhecer e interpretar as linguagens das duas ferramentas, apontando a existência de erros sintáticos nas especificações escritas.

2. Ser capaz de mapear sintaticamente as linguagens das ferramentas em questão quando a especificação de entrada contiver apenas operadores comuns às ferramentas.

3. Utilizar os mapeamentos semânticos propostos para traduzir os operadores não comuns às linguagens das ferramentas

4. Detectar quando for impossível efetuar a tradução e portanto a funcionalidade requerida pelo usuário.

5.3 Arquitetura

Para a arquitetura de nosso sistema, decidimos utilizar uma abordagem componentizada separando as funcionalidades em módulos que interligam-se através dos comandos especificados via linha de comando

Optou-se por esta abordagem porque alguns dos componentes desenvolvidos poderiam ser usados em outros projetos do grupo de pesquisa e porque a componentização diminui a curva de aprendizagem do uso de novas tecnologias (basta saber o que um componente faz e sua interface. Não é preciso entrar no detalhe do como a funcionalidade foi implementada)

Dentre outras vantagens da arquitetura componentizada podemos citar que a reutilização de componentes torna mais fácil a detecção de erros (porque mais pessoas usam e testam o componente). Além disso, existe o aumento da produtividade do desenvolvimento de aplicações com funcionalidades semelhantes.

Assim, a ferramenta foi dividida em 4 componentes fundamentais:

Parser: Responsável por transformar a entrada de dados quer na linguagem da VTUBAINA quer na linguagem da HAL em uma estrutura única de árvore sintática com a qual o programa irá lidar daqui pra frente para efetuar as traduções e modificações necessárias. O módulo de parser também é responsável pela detecção de erros sintáticos nos arquivos de especificação fornecidos como entrada para o programa.

Tradutores Sintáticos: Responsável por efetuar o mapeamento sintático entre as ferramentas quando os operadores são comuns a elas. Implementa o mapeamento sintático descrito no capítulo anterior.

Tradutores Semânticos: Responsáveis por efetuar os mapeamentos semânticos descritos no capítulo anterior. Faz uso das ferramentas para isso através do componente descrito a seguir.

Runner: Responsável por chamar a linha de comando de ferramentas externas e montar os arquivos com as especificações de entrada corretos para cada uma das ferramentas e obter delas a saída de cada funcionalidade requisitada.

Optou-se por desenvolver a Hydra como uma ferramenta que pudesse ser invocada via linha de comando sem uma interface gráfica e isso por algumas razões:

- Uma ferramenta puramente gráfica não poderia ser utilizada como parte de um processo automatizado. E no caso da Hydra, que serve justamente como uma ferramenta de integração que efetua traduções automatizadas, isso seria particularmente problemático.
- Uma vez que a Hydra possa ser invocada via linha de comando, não é difícil escrever uma interface gráfica que encapsule as chamadas à Hydra.
- Por constituir um módulo independente, a Hydra pode ser facilmente reutilizada na construção de outras ferramentas.
- Existe uma ferramenta de especificação gráfica de processos em π-*calculus* desenvolvida por nosso grupo de pesquisa denominada PiG[2] que poderia ser utilizada para a geração dos processos na linguagem da VTUBAINA.

Passaremos agora a detalhar cada um dos componentes com relação às tecnologias utilizadas, capacidade e limitações.

5.3.1 Parser

O parser é o módulo responsável pela geração das árvores sintáticas no protótipo. Uma vez que existem duas sintaxes diferentes para serem interpretadas, ao invés de se utilizar um gerador automático de analisadores sintáticos para cada uma delas, optamos por utilizar uma ferramenta livre chamada *chaperon* do projeto jakarta [1]

Este componente permite que a sintaxe das linguagens seja definida em XML, de maneira que ele gera em tempo de execução um analisador léxico para cada uma das linguagens.

A vantagem desta abordagem é que se houver qualquer alteração na sintaxe da linguagem das ferramentas, é fácil efetuar modificações em nosso protótipo, bastando alterar um XML, sendo desnecessário recompilar o protótipo novamente. A desvantagem é que a árvore sintática necessita ter uma forma muito bem definida.

[1] http://www.chaperon.sourceforge.net

Qualquer alteração na estrutura da árvore sintática necessitaria alteração em todo o protótipo.

5.3.2 Tradutores Sintáticos

Conforme descrito no capítulo 4, os módulos de Tradutores Sintáticos implementam a tradução entre a linguagem utilizada pela VTUBAINA e a linguagem utilizada pela HAL.

Cada módulo percorre a árvore sintática gerada pelo parser, convertendo os operadores correspondentes que são sintaticamente mapeados.

A vantagem de modularizar cada um dos tradutores é que isto permite que possamos alterar e implementar novas funcionalidade em cada um separadamente caso seja necessário. A desvantagem é que o desacoplamento entre tradutores sintáticos e semânticos aumenta a complexidade do sistema.

5.3.3 Tradutores Semânticos

Os tradutores semânticos foram implementados, conforme descrito no capítulo 4, para efetuar as conversões de operadores cujo mapeamento sintático não seja evidente.

Quando a conversão ocorre entre a VTUBAINA e a HAL, é necessário efetuar a transformação de certos processos para a forma normal. Para isto, este módulo utiliza a própria VTUBAINA, através do módulo de Runner, que será descrito abaixo.

Este módulo possui as mesmas limitações impostas pela teoria, ou seja, implementa apenas os mapeamentos estudados, não representando a totalidade de mapeamentos semânticos possíveis de serem efetuados.

Por este motivo, o fato de estarem implementados de maneira modular representa uma grande vantagem, caso haja necessidade de se implementar novos mapeamentos.

Outra limitação importante é que o nosso protótipo não faz as combinações entre todas as ordens possíveis para inferir a existência do padrão sintático que ele utiliza para efetuar os mapeamentos. Ele assume como premissa que os processos sejam

descritos exatamente como no estudo teórico. Exemplo:

$$Q \stackrel{def}{=} \alpha_1.(P_1|\alpha_2.P_2) + \alpha_2.(\alpha_1.P_1|P_2) + \tau.(P_1\{obj(\alpha_2)/obj(\alpha_1)\}|P_2)$$

Seria reconhecido e corretamente mapeado. Mas:

$$Q \stackrel{def}{=} \alpha_1.(P_1|\alpha_2.P_2) + \tau.(P_1\{obj(\alpha_2)/obj(\alpha_1)\}|P_2) + \alpha_2.(\alpha_1.P_1|P_2)$$

Não seria reconhecido, dada a ordem incorreta dos operandos.

5.3.4 Runner

Este módulo foi modelado para executar ferramentas externas. No caso, nosso protótipo utilizará este módulo para a execução da VTUBAINA, uma vez que precisamos utilizar a normalização implementada nesta ferramenta para a execução de uma tradução semântica.

Foi implementado completamente em *JAVA* utilizando a funcionalidade padrão da *API* do mesmo. Como o *JAVA* é multi-plataforma, este módulo é capaz de executar em qualquer sistema.

Nosso protótipo utiliza apenas a execução em ambiente *Windows* uma vez que a última versão da VTUBAINA está implementada, por hora, apenas neste sistema operacional.

5.4 Exemplos de uso

5.4.1 Convertendo da VTUBAINA para a HAL

Para demonstrar a utilização dos mapeamentos da VTUBAINA para a HAL, utilizaremos o seguinte processo:

$$P \stackrel{def}{=} (\nu k)\ !\ k(x) \mid (\nu a)\ a(x).\overline{b}z$$

Suponha que desejamos demonstrar que a ação $\overline{b}z$ nunca ocorre. Para isso, precisariamos verificar a seguinte propriedade $EF < \overline{b}z > true$. Ou seja, se existe

uma seqüência de ações finitas que me leve a executar $\overline{b}z$.

Mas, para verificar esta propriedade, temos de fazer uso da HAL, que efetua verificação de propriedades. No entanto este processo possui uma replicação, logo não poderia ser aceito pela HAL. No entanto, os dois processos compostos: $(\nu k)\ !\ k(x)$ e $(\nu a)\ a(x).\overline{b}z$ começam pela restrição de um nome que aparece como sujeito da primeira ação, ambos jamais serão executados. Assim, $P \equiv 0$

Para utilizar a Hydra, devemos representar o processo acima descrito na linguagem da VTUBAINA. Isto será feito em um arquivo de texto chamado p1.vtub:

```
P=(^k)!k(x)|(^a)a(x).b<z>
```

Utilizando a Hydra, podemos facilmente averiguar esta congruência através do resultado final obtido pelos mapeamentos. Para isso, utilizaremos o seguinte comando:

```
hydra vtub p1.vtub p1.hal
```

Obtendo como resultado o arquivo p1.hal contendo a especificação do mesmo processo convertido para a linguagem da HAL, com o seguinte conteúdo:

```
P(b,z)=
```

Como o processo P descrito no arquivo p1.hal está na linguagem da hal e não contém replicações, podemos utilizar agora a HAL para efetuar a verificação da propriedade.

A VTUBAINA sozinha não conseguia efetuar a verificação da propriedade. A HAL sozinha também não poderia efetuar a verificação uma vez que o processo possuia um operador não reconhecido pela mesma. Mas o uso conjunto das duas ferramentas mais os mapeamentos providos pela Hydra permitiram a ampliação da capacidade de verificação.

5.4.2 Convertendo da HAL para a VTUBAINA

Para demonstrar a conversão da HAL para a VTUBAINA, faremos uso do seguinte processo:

$$Q \stackrel{def}{=} \overline{c}d.(\overline{z}x \mid \overline{a}b.\overline{k}w) + \overline{a}b.(\overline{c}d.\overline{z}x \mid \overline{k}w)$$

Suponha que necessitamos simular passo a passo o comportamento deste processo, inclusive, podendo verificar congruências durante a simulação. A ferramenta que disponibiliza este recurso é a VTUBAINA.

Porém este processo possui o operador alternativa, e portanto não poderia ser aceito pela VTUBAINA. No entanto, ele recai em um dois casos onde a conversão semântica é possível segundo definido no capítulo anterior. Para que a Hydra possa convertê-lo, ele deve estar especificado na linguagem da HAL e em nosso exemplo, esta especificação encontra-se no arquivo q1.hal:

```
Q(a,b,c,d,z,x,k,w)=c!d.(z!x|a!b.k!w)+a!b.(c!d.z!x|k!w)
```

Utilizando a Hydra, com o seguinte comando, obtemos a conversão do processo Q especificado para a linguagem da VTUBAINA, descrito em um arquivo chamado q1.vtub:

```
hydra hal q1.hal q1.vtub
```

O arquivo q1.vtub tem o seguinte conteúdo:

```
Q=c<d>.z<x>|a<b>.k<w>
```

O processo Q descrito no arquivo q1.vtub não contém o operador alternativa, e está descrito na linguagem utilizada pela VTUBAINA, podendo ser utilizado como entrada na ferramenta para utilização de qualquer uma de suas capacidades, incluindo a simulação e a verificação de congruencias estruturais.

A VTUBAINA sozinha, não conseguiria simular o processo, uma vez que este estava escrito na linguagem da HAL e continha o operador alternativa. A HAL, sozinha, não conseguiria efetuar as simulações interativas que necesitávamos. Mas a utilização conjunta da VTUBAINA e dos mapeamentos disponibilizados pela Hydra permitiu ampliar a capacidade isolada de cada uma das ferramentas.

5.5 Considerações Finais

Neste capítulo descrevemos a arquitetura interna e detalhes de implementação do protótipo construído para demonstração da teoria exposta neste trabalho.

Mostramos dois exemplos de uso para o protótipo. Cada um deles utilizando os mapeamentos sintáticos e semânticos implementados.

Expusemos também as limitações do protótipo que poderão muito bem servir como trabalhos futuros de implementação para torná-lo completamente funcional.

Capítulo 6

Conclusão e trabalhos futuros

6.1 Conclusão

Muitos estudos estão sendo efetuados com o intuito de desenvolver novas ferramentas de verificação formal e considerando o número de ferramentas já existentes, a complexidade de se aprender todas as capacidades e linguagens das ferramentas torna-se cada vez maior. E, com o aumento dessa complexidade, torna-se cada vez mais propensa a erros a tarefa de conversão manual entre as linguagens de uma e outra ferramenta.

Para minimizar este problema, existem duas soluções possíveis:

1. A reprogramação das ferramentas visando a utilização de uma linguagem única. Esta solução exigiria um grande re-trabalho, e erros poderiam ser introduzidos durante a reprogramação. Mais do que isso, a criação de uma linguagem única, que atendesse a todas as capacidades oferecidas pelas diferentes ferramentas, exigiria um estudo abrangente.

2. A criação de tradutores automáticos capazes de converter os processos especificados na linguagem de uma ferramenta para a linguagem de outra.

Nosso estudo demonstra a realização da segunda solução, ou seja, a criação de uma ferramenta de tradução automatizada entre duas ferramentas de verificação formal, a VTUBAINA e a HAL. Estas ferramentas foram escolhidas por possuírem linguagens semelhantes, por trabalharem com a mesma teoria π-*calculus* e porque a HAL é uma ferramenta de referência na área de verificação formal.

Através deste trabalho, pudemos concluir que a montagem de uma ferramenta de conversão passa por algumas fases distintas:

Escolha das Ferramentas: Nesta primeira etapa, estuda-se as ferramentas existentes e as capacidades de cada uma. De acordo com a necessidade, escolhe-se as ferramentas que apresentam as capacidades requeridas e as maiores semelhanças teóricas e de linguagem.

Estudo das Teorias: Como cada ferramenta pode utilizar uma teoria diferente, é necessário estudar as teorias utilizadas por cada uma, bem como a linguagem utilizada pelas teorias, incluindo sintaxe e semântica.

Estudo das Capacidades: Cada ferramenta oferece capacidades distintas que podem ser utilizadas em conjunto. O levantamento dessas capacidades leva a compreensão das funcionalidades que conjuntamente as ferramentas oferecerão

Mapeamentos Sintáticos: Aqui estuda-se qual o sub-conjunto da teoria que cada ferramenta implementa e como mapear os operadores correlatos da linguagem de uma ferramenta para a linguagem da outra.

Mapeamentos Semânticos: Nem sempre os operadores possuem a mesma semântica, e nem sempre todos os operadores presentes no sub-conjunto da linguagem de uma das ferramentas está presente também no sub-conjunto da linguagem da ferramenta que se pretende integrar. Assim, é necessário estudar maneiras de estabelecer uma correlação semântica para esses operadores, quando possível.

Desenvolvimento: Fase de desenvolvimento da ferramenta de integração que deve levar em conta vários fatores, como por exemplo, plataforma sobre a qual cada ferramenta executa, portabilidade e escalabilidade.

Este método para o desenvolvimento de tradutores automatizados decorreu diretamente deste trabalho onde a realização de cada uma dessas etapas utilizando as ferramentas escolhidas foi efetuado.

A solução de estabelecer mapeamentos e efetuar a tradução automatizada não é simples,mas viável.

6.2 Trabalho Futuros

Como trabalhos futuros, podemos citar:

Expansão dos Mapeamentos: Certamente, nosso estudo não demonstra todos os mapeamentos semânticos possíveis entre as duas ferramentas. O estudo de mais mapeamentos permitiria ampliar o conjunto de processos possíveis de traduzir de maneira automática.

Estudo da expressividade dos mapeamentos: Visaria esclarecer com que freqüencia os mapeamentos apresentados ocorrem nas especificações formais.

Expansão do trabalho para novas ferramentas: Utilizando o mesmo método desenvolvido neste trabalho, outros conversores automáticos poderiam ser criados ampliando as capacidades isoladas das ferramentas existentes.

Utilização do método com ferramentas que utilizam teorias diferentes: Em nosso estudo, as duas ferramentas utilizadas tinham π-*calculus* como teoria. Porém, a utilização de ferramentas que utilizem teorias diferentes geraria a vantajosa capacidade de que processos expressos em teorias com menos ferramentas disponíveis pudessem fazer uso de ferramentas já consolidadas em outras teorias.

Aperfeiçoamento do protótipo: Para que ele possa reconhecer inversões na ordem dos operandos, quando houver necessidade de mapeamentos semânticos.

Interface Gráfica: Que pudesse reunir diversos conversores e oferecer ao usuário diversas capacidades de maneira transparente, executando os conversores e ferramentas necessárias para a tarefa sem que o usuário necessite especificar quais são os conversores e ferramentas requeridos.

Apêndice A

Ferramentas Analisadas

A.1 Ferramentas Analisadas

A.1.1 Linguagens de Especificação

Linguagens de especificação são qualquer tipo de programa capaz de receber como entrada uma especificação escrita em alguma linguagens específica e simular o comportamento do programa.

Assim, toda linguagem de especificação para as quais existam um compilador ou bibliotecas são capazes de simulação também. Mais abaixo existe uma seção específica para ferramentas de simulação.

Funnel

WebPage: *http://lampwww.epfl.ch/funnel/ (última visualização: 16/07/2005)*

Descrição: É uma linguagem de programação baseada em *Functional Nets*. Utiliza-se de uma máquina virtual para a execução do código.

Linguagem em que foi desenvolvida: Tanto o compilador quanto a máquina virtual foram desenvolvidos em Java.

Linguagem de entrada: Própria. Admite o uso de objetos, bem como de programação funcional e imperativa. Exemplo:

```
val c = newAsyncChannel
```

```
def producer = {
  var x := 1
  while (true) { val y := x ; x := x + 1 & c.write y }
}
def consumer = {
  while (true) { val y = c.read ; print y }
}
producer & consumer
```

Linguagem de saída: Tela.

Capacidades: Pode lidar com concorrência, canais assíncronos e orientação a objetos.

Fundamentação Teórica: *Functional Nets*

Aglets

WebPage: *http://www.trl.ibm.com/aglets/ (última visualização: 16/07/2005)*

Descrição: Bibliotecas para computação móvel, paralela e distribuída.

Linguagem em que foi desenvolvida: Java

Linguagem de entrada: Java. Sendo uma biblioteca, os códigos java fazem uso diretamente das funções implementadas pela mesma.

Linguagem de saída: Aplicação em si.

Capacidades: Aglets permitem a migração de estado e código de uma máquina a outra em um processo que lembra bastante join-calculus.

Fundamentação Teórica: Inexistente.

Join Calculus

WebPage: *http://join.inria.fr/ (última visualização: 16/07/2005)*

Descrição: Uma linguagem experimental baseada nas máquinas químicas abstratas reflexivas.

Linguagem em que foi desenvolvida: Objective-CAML

Linguagem de entrada: Própria, muito parecida com a linguagem funcional ML. Exemplo:

```
# let size = 1000
# let chunk = 200
#
# let join(name,there) =
#   loc mobile
#   init
#     let start(i,done) =
#       let loop(u,s) = if u<(i+1)*chunk then { loop(u+1,s+u) }
else { done(s) } in
#       loop(i*chunk,0) in
#     go(there);
#     worker(name,mobile,start)
#   end in
#   print_string(name^" joins the party\n");
#
# and job(i) | worker(name,there,start) =
#   print_string(name^","^ml.string_of_int(i*chunk)^"\n");
#   let once() | done(s)  = add(s) | worker(name,there,start)
#   and once() | failed() = print_string(name^" went down\n");
job(i) in
#   once() | start(i,done) | fail(there);failed()
#
# and result(n,s) | add(ds) =
#   let s' = s + ds in
#   if n > 0 then { result(n-1,s') }
#   else {
     print_string("The sum is "^ml.string_of_int(s')^"\n");
```

```
        }
#
# do ns.register("join",join)
#
# spawn
#   result(size/chunk-1,0)
# | let jobs(n) = job(n) |
        if n>0 then { jobs(n-1) } in jobs(size/chunk-1)
```

Linguagem de saída: Tela

Capacidades: Todas as englobadas pelo modelo de máquinas químicas abstratas, ou seja, concorrência e mobilidade.

Fundamentação Teórica: Máquinas químicas abstratas e reflexivas

Pict

WebPage: *http://www.cis.upenn.edu/ bcpierce/papers/pict/Html/Pict.html (última visualização: 16/07/2005)*

Descrição: Uma linguagem de especificação baseada em π-*calculus*. Também é capaz de simulação.

Linguagem em que foi desenvolvida: C

Linguagem de entrada: Pict. Exemplo:

```
run(x!y
   |x?z=z!u
   |y?w=print!''Got it'')
```

Linguagem de saída: Tela

Capacidades: Especificação e simulação de processos descritos em π-*calculus*. A linguagem de entrada se parece muito com ML. Usa um layer intermediário, utilizando-se de um compilador para transformar o código Pict em código C.

Fundamentação Teórica: *π-calculus.*

Klaim e Klava

WebPage: *http://music.dsi.unifi.it/klaim.html (última visualização: 16/07/2005)*

Descrição: Klaim é um formalismo que suporta que processos migrem de uma máquina para outra. É baseado em Linda. X-Klaim é uma linguagem de programação baseada em Klaim para desenvolvimento de aplicações distribuídas com módulos móveis orientados a objeto. X-Klaim compila uma especificação Klaim em código java, que faz uso da biblioteca Klava que implementa as rotinas de Klaim.

Linguagem em que foi desenvolvida: Java

Linguagem de entrada: Klaim

Linguagem de saída: Código java, diretamente compilável através do uso da biblioteca Klava.

Capacidades: Implementação de mobilidade em códigos java, seguindo uma especificação formal.

Fundamentação Teórica: Linda.

Piccola

WebPage: *http://www.iam.unibe.ch/scg/Research/Piccola/ (última visualização: 16/07/2005)*

Descrição: É uma pequena linguagem de composição. É capaz de compor aplicações a partir de componentes já implementados. Toda computação da aplicação é feita pelos componentes.

Linguagem em que foi desenvolvida: Existem duas implementações da linguagem Piccola. Uma desenvolvida em Java para integrar componentes Java e outra desenvolvida em Squeak [1]

[1] Um sistema de programação para a confecção de applets educacionais.

Linguagem de entrada: Piccola. Exemplo:

```
# File: duke.picl
# 1. load nawt services
root = (root, load("nawt")) # use event and AWT wrappers style
# 2. create AWT Components
duke = awtComponent("demos.duke.Duke")
waveButton = awtComponent("java.awt.Button").set(Label = "wave")
speedScrollbar = awtComponent("java.awt.Scrollbar").set
Minimum = 1
Maximum = 800
Value = duke.getSpeed()
# 3. do the event wiring
speedScrollbar ? Adjustment
do: (duke.set(Speed = speedScrollbar.getValue()))
waveButton ? Action(do: duke.wave(val = 1))
# 4. click on Duke
counter = load("counter").newCounter(0)
sleep() = javaClass("java.lang.Thread").sleep(val = 2000)
duke ? MouseClicked
do:
duke.set(Message = "ouch")
counter.inc()
sleep() # sleep 2 seconds
if (counter.dec() <= 0) # if this was the last click
then: duke.clearMessage()
# 5. arrange components in a panel
panel = newBorderPanel
center = newBorderPanel
north = Components + waveButton
center = duke
west = speedScrollbar
# 6. add panel into a frame and display it
exit() = javaClass("java.lang.System").exit(val = 0)
```

```
frame =
awtComponent("java.awt.Frame").set(Title = "This is duke")
frame.add(val = panel.java, type = "java.awt.Component")
frame ? WindowClosing(do: exit())
frame.pack()
frame.show()
```

Linguagem de saída: Aplicação. JPiccola converte os scripts em π-*calculus* e executa-os repassando as mensagens para cada componente através de uma máquina abstrata capaz de interpretar π-*calculus*.

Capacidades: Geração de aplicações através da composição de componentes.

Fundamentação Teórica: ?

Tyco

WebPáge: *http://www.ncc.up.pt/tyco/ (última visualização: 16/07/2005)*

Descrição: É uma linguagem orientada a objetos, tipada e concorrente baseada numa extensão de π-*calculus* assíncrono.

Linguagem em que foi desenvolvida: Java. Baseia-se em uma máquina virtual e um compilador.

Linguagem de entrada: Própria: Tyco. Exemplo:

```
new r1 Fib[n-1, r1] |
new r2 Fib[n-2, r2] |
r2?(v2)=r1?(v1)=r![v1+v2]
```

Linguagem de saída: Tela.

Capacidades: Desenvolvimento de programas concorrentes.

Fundamentação Teórica: π-*calculus*.

Ambicojb

WebPage: *http://www-sop.inria.fr/mimosa/ambicobjs/ (última visualização: 16/07/2005)*

Descrição: É uma implementação gráfica para o *ambiente calculus*. É a associação de um ícone gráfico a comportamentos (programas reativos). Ambient Calculus é um framework teórico desenvolvido por Luca Cardelli.

Linguagem em que foi desenvolvida: Java

Linguagem de entrada: Própria, composta dos seguintes operadores:

Definição de ambiente: nome[]

Composição: |

Criação de um nome novo: nu

Recursão: rec

Espera: sleep

Sequência: .

Ações: in, out, open, in_, out_, open_

Linguagem de Saída: Representação gráfica na tela.

Capacidades: Representação e simulação de agentes em ambientes móveis, seguros, robustos e controlados.

Fundamentação Teórica: Ambient Calculus

A.1.2 Ferramentas de Verificação

Jack

WebPage: *http://rep1.iei.pi.cnr.it/projects/JACK/ (última visualização: 16/07/2005)*

Descrição: Compõe-se de duas partes principais:

Jack: Conjunto de ferramentas para sistemas concorrentes. Será descrito com mais detalhes abaixo.

HAL: Conjunto de ferramentas para especificação, verificação e análise de sistemas concorrentes e distribuídos.

Linguagem em que foi desenvolvida: Várias. Cada componente foi desenvolvido individualmente, independentemente, não havendo homogeneidade de código fonte. Dentre as linguagens utilizadas podemos citar C, C++, ADA e Lisp.

Linguagem de entrada: Varia. No conjunto, o Jack é composto por vários módulos, cada um com sua função e linguagem de entrada próprias.

O Jack é capaz de ler e escrever objetos algébricos, através de dois componentes: *NSS(C) e MAUTO(Lisp)*, ambos capazes de trabalhar com CCS. Visualizar gráficos através do componente *ATG(C++)*, utilizando-se do formato FC2. É capaz de atuar como ModelChecker para formulas lógicas ACTL através do módulo *AMC(C)*. Outro módulo capaz de fazer a mesma checagem, mas de forma simbólica é o *BMC*. Outro componente integrante do Jack é o *FMC(ADA)* capaz de montar um modelo maior a partir de partes já verificadas. Existem outros módulos ainda como por exemplo *FC2IMPLICIT(C++) e FC2EXPLICIT(C++)* desenvolvidos para a manipulação simbólica ou não de autômatos.

HAL (HD-Automata Laboratory) consiste de 5 módulos: três módulos traduzem π-*calculus* para HD-Automato*(pi-to-hd)*, de HD-Automato para automato comum*(hd-to-aut)* e de π-*calculus* para lógica ACTL*(pl-to-actl)*. O quarto módulo contém rotinas para manipulação de HD-automatos *(hd reduce)*. O quinto módulo é o próprio Jack, o que torna o HAL uma extensão deste.

Linguagem de saída: Variadas. Arquivos com descrições algébricas, com descrições de autômatos, com especificação em lógica ACTL, tela, dentre outras. Possui interface gráfica para alguns módulos, sendo a maioria escrita em Java.

Capacidades: Verificação de modelos descritos em lógica ACTL, visualização de autômatos descritos na linguagem FC2, manipulação com autômatos, manipulação com Hd-automato, conversão de π-*calculus* para hd-automato, de hd-automato para automatos normais, conversão de π-*calculus* para lógica ACTL.

Fundamentação Teórica: CCS, ACTL, π-*calculus*.

MWB

WebPage: *404*

Descrição: O MWB *Mobile WorkBench* é uma ferramenta destinada à verificação de processos descritos em π-*calculus*, bem como a simulação dos mesmos e a verificação de propriedades descritas em lógica temporal.

Linguagem em que foi desenvolvida: SML

Linguagem de entrada: Própria.

Linguagem de saída: Tela

Capacidades: Verificação de *Open-Bissimulation*, bem como verificação de algumas propriedades descritas via lógica temporal. É capaz de checar *weak* e *strong open-bissimulations*

Fundamentação Teórica: π-*calculus*.

OBCW

WebPage: *http://www.cs.auc.dk/research/FS/ny/PR-pi/ (última visualização: 16/07/2005)*

Descrição: Ferramenta capaz de verificar *Open Bissimulation* entre dois processos descritos em π-*calculus*.

Linguagem em que foi desenvolvida: SML

Linguagem de entrada: Própria

Linguagem de saída: Tela

Capacidades: Verificação de *Open-Bissimulation*

Fundamentação Teórica: π-*calculus*.

VTubaina

WebPage:

Descrição: Verificador para π-*calculus*, bem como gerador de hd-automatos.

Linguagem em que foi desenvolvida: C++

Linguagem de entrada: Própria, baseada no π-*calculus* de Milner. Algumas funções, como por exemplo, a simulação apresentam entradas de dados manuais.

Linguagem de saída: Tela. Os autômatos podem ser salvos em formato FC2

Capacidades: Simulação de agentes móveis, Verificação de bissimilaridade de processos (por particionamento ou *on-the-fly*, equivalências (up to restrição, up to composição e up to congruência estrutural), congruências estruturais.

Fundamentação Teórica: π-*calculus*.

A.1.3 Verificadores de Modelos

Os verificadores de modelo descritos aqui são capazes de verificar propriedades expressas em alguma lógica temporal. Assim, estão restritos a programas com número finito de estados.

Além disso, todos são especificamente para trabalhar com autômatos. Desta forma, aplicam-se a modelos concorrentes ou mesmo móveis quando estes possam ser representados na forma de autômatos finitos.

Jack O Jack, citado anteriormente também se encaixa nesta categoria

EST

WebPage: *http://lms.uni-mb.si/EST/ (última visualização: 16/07/2005)*

Descrição: Verificador de especificações baseadas em CCS. Utiliza-se da lógica ACTL para a especificação ads propriedades

Linguagem em que foi desenvolvida: C

Linguagem de entrada: CCS e ACTL. Possui interface gráfica escrita em Tcl/Tk

Linguagem de saída: Tela, via interface Tcl/Tk se disponível.

Capacidades: Verificação de modelos temporais sem explosão de estados.

Fundamentação Teórica: CCS, ACTL.

TLA e TLC

WebPage: *http://research.microsoft.com/research/sv/TLA_Tools/ (última visualização: 16/07/2005)*

Descrição: Verificador de especificações baseadas em uma linguagem própria chamada TLA (amplamente descrita).

Linguagem em que foi desenvolvida: Java

Linguagem de entrada: TLA. Exemplo:

```
---------------- MODULE Peano -------------
PeanoAxioms(N, Z, Sc) ==
  /\ Z \in N
  /\ Sc \in [N -> N]
  /\ \A n \in N : (\E m \in N : n = Sc[m]) <=> (n # Z)
  /\ \A S \in SUBSET N : (Z \in S)
/\ (\A n \in S : Sc[n] \in S) => (S = N)

ASSUME \E N, Z, Sc : PeanoAxioms(N, Z, Sc)

Succ == CHOOSE Sc : \E N, Z : PeanoAxioms(N, Z, Sc)
Nat  == DOMAIN Succ
Zero == CHOOSE Z : PeanoAxioms(Nat, Z, Succ)
=========================================
```

Linguagem de saída: Tela.

Capacidades: Verificação de modelos temporais sem explosão de estados. Algumas limitações extras podem ser encontradas: nem todo o conjunto da linguagem TLA é verificado pelo verificador TLC.

Fundamentação Teórica: ?

Spin

WebPage: *http://spinroot.com/spin/whatispin.html (última visualização: 16/07/2005)*

Descrição: Um verificador de programas e um verificador de modelos bastante desenvolvido.

Linguagem em que foi desenvolvida: C

Linguagem de entrada: Promela. Exemplo:

```
proctype P () {
do
:: wait(sem);
   critical++;
   printf("MSC: Process %d in CS\n",_pid);
   if :: _pid == 1 -> P1inCS = true :: else fi;
   assert (critical <= 1);
   critical--;
   signal(sem);
od
}

init {
initSem(sem, 1);
atomic {
for (i,1,NPROCS)
run P()
rof (i)
}
```

}

Linguagem de saída: Tela

Capacidades: Verificação de programas escritos em Promela bem como verificação de modelos. Spin também possui problemas com explosão de estados.

Fundamentação Teórica: Promela

A.1.4 Miscelânia

Cryptyc

WebPage: *http://cryptyc.cs.depaul.edu/ (última visualização: 16/07/2005)*

Descrição: É um verificador de tipos para protocolos. Mas além de erros normais de tipos, ele é capaz de checar violação de segurança como erros de autenticidade.

Linguagem em que foi desenvolvida: Java

Linguagem de entrada: Própria, baseada no Spi Calculus de Abadi e Gordon, que por sua vez é baseado no π-*calculus*.

Clientes: *client* - Capazes de enviar mensagens para os servidores através de sockets.

Servidores: *server* - Capazes de receber mensagens de clientes

Socket: *socket* - Canais de comunicação entre clientes e servidores.

Permissões: *Public, Private* - Tipos das mensagens. Uma mensagem pública pode ser visualizada por qualquer pessoa, incluindo um intruso. Mensagens privadas não devem ser vistas por intrusos, necessitando portanto serem seguras.

Entradas e Saídas: *input, output* - Entrada e saída de mensagens via sockets.

Criação de mensagens: *new* - Cria mensagens.

Proteção: *begin, end* - Cada end deve ser atingido unicamente através do seu begin. Do contrário o trecho especificado não é seguro.

Dentre outras palavras reservadas com funções mais específicas relativas à segurança, como podemos ver no exemplo a seguir:

```
client Sender at Alice is {
  establish Receiver at Bob is (socket : Socket);
  input socket is (nonce : Challenge);
  new (msg : Payload);
  begin (Sender sent msg);
  cast nonce is (nonce' : MyNonce (msg));
  output socket is ({ msg, nonce' }SKey);
}

server Receiver at Bob is (socket : Socket) {
  new (nonce : Challenge);
  output socket is (nonce);
  input socket is
   ({ msg : Payload, nonce' : MyNonce (msg) }
      SKey);
  check nonce is nonce';
  end (Sender sent msg);
}
```

Linguagem de saída: Tela, via texto.

Capacidades: Verificação de tipos em um protocolo com agentes (mensagens) móveis, bem como a segurança do tráfego desses agentes contra ataques de falsificação de indentidade (Ataque de Dolev Yao) [2].

Fundamentação Teórica: Spi-Calculus.

Trust

WebPage: *http://www.cmi.univ-mrs.fr/vvanacke/trust/ (última visualização: 16/07/2005)*

[2] Ataque de Dolev Yao é o nome que se dá a possibilidade de alguém interceptar uma comunicação e forjar uma mensagem, sem necessariamente conhecer as chaves criptográficas adequadas.

Descrição: Analisador simbólico para protocolos.

Linguagem em que foi desenvolvida: OCAML

Linguagem de entrada: Própria

Linguagem de saída: Tela

Capacidades: Verificar a segurança de protocolos contra ataques do tipo Dolev Yao.

Fundamentação Teórica:

STA

WebPage: *404*

Descrição: STA(Symbolic Trace Analyser) é uma ferramenta protótipo para a análise de protocolos de segurança.

Linguagem em que foi desenvolvida: MOSML

Linguagem de entrada: MOSML (funciona como uma API). Exemplo:

```
(* ===== NEEDHAM-SCHROEDER WITH CONVENTIONAL KEYS =========*)

(*
0. - --> E: A,B,0,1,2,kOld,{A,B,kOld}kBS
1. A --> S: A, B, Na1
2. S --> A: {A, B, Na1, kAB, {A,B,kAB}kBS }kAS
3. A --> B: {A, B, kAB}kBS
4. B --> A: {Nb1, 0}kAB
5. A --> B: {Bb1, 1}kAB
6. B --> A: {d,2}kAB
*)

DeclLabel $a1, a2, a3, a4, a5, a6, acceptAB, b1, b2, b3
, b4, s1, s2, disclose, guard$;
DeclVar $x, xk, xf, w, xInN, xReN, yk, t $;
```

```
DeclName $A, B, kAS, kAB, kBS, Nb1, Na1, kOld, d , one, zero , two $;

(* INITIATOR A*)

val prA = a1!(A , B , Na1) >> a2?( {A , B , Na1 , xk , xf }kAS ) >>
a3!xf >> a4?{xReN , zero}xk>>
a5! {xReN , one}xk >> a6?{w , two}xk >> acceptAB!{w , two}xk >> stop;

(* SERVER S*)

val prS = s1?(A , B , xInN) >>
s2!( {A , B , xInN , k , {A , B , k}kBS }kAS )>>stop;

(* RESPONDER B*)

val prB = b1?{A , B , yk }kBS >> b2! {Nb1 , zero}yk >>
b3?{Nb1 , one}yk >> b4!{d , two}yk >> stop;

(* the whole SYSTEM: it also includes a 'guardian' that can detect if
the environments learns some piece of sensible information, like d *)

val NS = prA || prS || prB || guard?x >>stop;

(* the INITIAL CONFIGURATION: the environment initially knows:
an old session key kOld, the corresponding certificate
{A , B , kOld}kBS, numerals and agents identifiers *)

val Conf2 = ( [disclose!(A , B, zero , one , two , kOld ,
{A , B , kOld}kBS)] @ NS);

val Secrecy = (Absurd <-- guard?d);
(* the property: guard?d is never executed, i.e.
the environments never learns d *)
(* try: CHECK Conf2 Secrecy; *)
```

```
(* The protocol enjoys the following authentication property: if A accept
something, it must come from B. Since the whole symbolic state-space is
explored, checking this protocol takes a bit longer -- a few seconds. *)

val Auth1 = (b4!t <-- acceptAB!t);

(* try: CHECK Conf2 Auth1; *)
```

Linguagem de saída: Tela

Capacidades:

Fundamentação Teórica:

HD-Reducer

WebPage: *http://jordie.di.unipi.it:8080/mihda (última visualização: 16/07/2005)*

Descrição: Framework para minimização de HD-Autômatos através de um refinamento do algoritmo de particionamento.

Linguagem em que foi desenvolvida: OCAML

Linguagem de entrada: *π-calculus*. Exemplo:

```
define
  Car(talk,switch,out) =
    talk?(msg).out!msg.Car(talk,switch,out) +
    switch?(t).switch?(s).Car(t,s,out)

define
  Base(talkcentre,talkcar,give,switch,alert) =
    talkcentre?(msg).talkcar!msg.
             Base(talkcentre,talkcar,give,switch,alert)
    +
```

```
    give?(t).give?(s).switch!t.switch!s.give!give.
              IdleBase(talkcentre,talkcar,give,switch,alert)

define
  IdleBase(talkcentre,talkcar,give,switch,alert) =
    alert?(empty).Base(talkcentre,talkcar,give,switch,alert)

define
  Centre(in,tca,ta,ga,sa,aa,tcp,tp,gp,sp,ap) =
    in?(msg).tca!msg.Centre(in,tca,ta,ga,sa,aa,tcp,tp,gp,sp,ap)
    +
    tau.ga!tp.ga!sp.ga?(empty).ap!ap.
         Centre(in,tcp,tp,gp,sp,ap,tca,ta,ga,sa,aa)

define
  System(in,out) =
    (tca)(ta)(ga)(sa)(aa)(tcp)(tp)(gp)(sp)(ap)
    | (Car(ta,sa,out),
       Base(tca,ta,ga,sa,aa),
       IdleBase(tcp,tp,gp,sp,ap),
       Centre(in,tca,ta,ga,sa,aa,tcp,tp,gp,sp,ap))

write_hd System
```

Linguagem de saída: Tela

Capacidades: Minimização de autômatos ou HD-autômatos com número de estados finitos.

Fundamentação Teórica: π-*calculus*, HD-Autômatos

A.2 Conclusão

Observando as capacidades e linguagens envolvidas entre os verificadores disponíveis, resolvemos começar o trabalho pela integração do Jack com a VTubaina, através de

um estudo detalhado dos mesmos, conforme mostrarei a seguir.[3]

[3]Todas as citações de páginas web feitas nesse capítulo tem anotadas as respectivas datas quando foram visualizadas pela última vez. Não há nenhuma garantia de que as páginas persistam nos mesmos endereços ou sequer continuem existindo após esta data.

Referências Bibliográficas

[1] Marcelo M. Amorim. Uma técnica de verificação para pi-calculus baseada em bissimulação up-to e algoritmos de particionamento, 2005.

[2] André Gustavo Andrade, Ana C.V. de Melo, and Marcelo M. Amorim. Da especificação à verificação de agentes móveis - um ambiente gráfico. In Mauricio Solar, David Fernández-Baca, and Ernesto Cuadros-Vargas, editors, *30ma Conferencia Latinoamericana de Informática (CLEI2004)*, pages 236–245. Sociedad Peruana de Computación, September 2004. ISBN 9972-9876-2-0.

[3] J-C. Fernandez and L. Mounier. "on the fly" verification of behavioural equivalence and preorders. In *Proc. 3rd International Computer Aided Verification Conference*, pages 181–191, 1991.

[4] G. Ferrari, S. Gnesi, U. Montanari, M. Pistore, and G. Ristori. Verifying mobile processes in the HAL environment. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of CAV '98*, volume 1427 of *LNCS*. Springer, 1998. Tool Poster.

[5] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385. ACM, January 1996.

[6] S. Gnesi. A formal verification environment for concurrent systems design. In *Proceedings Workshop on Automated Formal Methods*, volume 5 of *ENTCS*. University of Oxford, 1996.

[7] D. Hirschkoff. Automatically proving up to bisimulation. In Petr Jancar and Mojmir Kretinsky, editors, *Proceedings of MFCS '98 Workshop on Concurrency*, volume 18 of *ENTCS*. Elsevier Science Publishers, 1998.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[9] E. Koutsofios. Editing graphs with *dotty*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1994. This report, and the program, is included in the `graphviz` package, available for non-commercial use at `http://www.research.att.com/sw/tools/graphviz/`.

[10] E. Koutsofios and S. North. Drawing graphs with *dot*. Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, USA, September 1991.

[11] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[12] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.

[14] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.

[15] Ugo Montanari and Marco Pistore. History-dependent automata. Technical Report TR-98-11, Dipartimento di Informatica, October 5 1998. Wed, 09 Dec 1998 11:19:14 GMT.

[16] F. Orava and J. Parrow. An algebraic verification of a mobile network. *Journal of Formal Aspects of Computing*, 4:497–543, 1992.

[17] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

[18] J. Parrow. An introduction to the π-calculus. In Bergstra, Ponse, and Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.

[19] R. De Nicola and F. W. Vaandrager. Action versus state based logic for transition systems. In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume LNCS 469. Springer Verlag, 1990.

[20] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, LFCS, University of Edinburgh, 1993. CST-99-93 (also published as ECS-LFCS-93-266).

[21] D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. Earlier version published as Report ECS-LFCS-93-270, University of Edinburgh. An extended abstract appeared in the *Proceedings of CONCUR '93*, LNCS 715.

[22] B. Victor. *A Verification Tool for the Polyadic π-Calculus.* Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.

[23] D. Walker. Objects in the π-calculus. *Journal of Information and Computation*, 116(2):253–271, 1995.