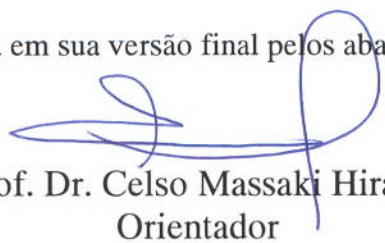
Tese apresentada à Pró-Reitoria de Pós-Graduação e Pesquisa do Instituto Tecnológico de Aeronáutica, como parte dos requisitos para obtenção do título de Mestre no Curso de Pós-Graduação em Engenharia Eletrônica e Computação, Área de Informática.

Juliana de Melo Bezerra

MAPEAMENTO UML-RT PARA π-CALCULUS

Tese aprovada em sua versão final pelos abaixo assinados:



Prof. Dr. Celso Massaki Hirata
Orientador

Prof. Dr. Homero Santiago Maciel
Pró-Reitor de Pós-Graduação e Pesquisa

Campo Montenegro
São José dos Campos, SP – Brasil
2006

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão Biblioteca Central do ITA/CTA

Bezerra, Juliana de Melo
Mapeamento UML-RT para π-calculus / Juliana de Melo Bezerra.
São José dos Campos, 2006.
129f.

Tese de Mestrado – Curso de Engenharia Eletrônica e Computação. Área de Informática –
Instituto Tecnológico de Aeronáutica, 2006. Orientador: Prof. Dr. Celso Massaki Hirata.

1. UML-RT. 2. π-calculus. 3. Semântica formal. I. Centro Técnico Aeroespacial. Instituto Tecnológico de Aeronáutica. Divisão de Ciência da Computação. II.Título.

REFERÊNCIA BIBLIOGRÁFICA

BEZERRA, Juliana de Melo. **Mapeamento UML-RT para π-calculus**. 2006. 129f. Tese de Mestrado – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: Juliana de Melo Bezerra
TÍTULO DO TRABALHO: Mapeamento UML-RT para π-calculus
TIPO DO TRABALHO/ANO: Tese / 2006

_____

Juliana de Melo Bezerra
Rua José Francisco Alves, 24, apt 104, Vila Ema.
CEP 12.243-060 – São José dos Campos - SP

MAPEAMENTO UML-RT PARA π-CALCULUS

Juliana de Melo Bezerra

Composição da Banca Examinadora:

Prof. Dr.	José Maria Parente de Oliveira	Presidente	- ITA
Prof. Dr.	Celso Massaki Hirata	Orientador	- ITA
Prof. Dr.	Ana Cristina Vieira de Melo	Membro Externo	- IME/USP
Prof. Dr.	Edgar Toshiro Yano	Membro	- ITA
Prof. Dr.	Clóvis Torres Fernandes	Membro	- ITA

ITA

A vovó Mera (*in memoriam*), pelo exemplo de mulher determinada.

Agradecimentos

Agradeço a Deus pela minha vida e por esta oportunidade de crescer moral e intelectualmente. Agradeço a todos os amigos protetores que me animaram e incentivaram nesta conquista.

Agradeço a minha mãe Janete e ao meu pai Sidraque que batalharam muito para me proporcionar uma ótima educação durante a infância, e que sempre me deram muito amor e força na minha caminhada.

Agradeço a meu irmão Júnior, a meu irmão Tales e a minha cunhada Luciana, pelo carinho e compreensão pela minha ausência. Agradeço a minha sobrinha e afilhada Giovanna que está enchendo meu coração de felicidade.

Ao meu namorado Gustavo, um agradecimento especial pela amizade, carinho e paciência.

As minhas amigas e companheiras Renata e Daniela, que dividiram comigo esses anos de trabalho e estudo, a minha eterna gratidão. Agradeço a minha amiga Carine que mesmo de longe sempre me deu muito incentivo.

Agradeço ao meu orientador Prof. Hirata pela dedicação e presteza. Ao Mauro Assano da IBM por viabilizar a instalação do RoseRT e ao colega Baggio por compartilhar seus conhecimentos.

Agradeço a Embraer por me conceder um espaço para atividades acadêmicas. Agradeço também aos meus colegas de trabalho por me ajudarem com as tarefas da Embraer enquanto eu estava no ITA.

Por fim, agradeço a todos familiares e amigos de Recife, Brasília e São José dos Campos que partilharam mais esta etapa comigo.

Resumo

A UML (*Unified Modeling Language*) é uma linguagem de modelagem para especificar, construir e documentar artefatos de sistemas de software. A UML-RT, usada pela ferramenta Rational Rose RealTime (RoseRT), é uma extensão da UML que permite a modelagem de sistemas de tempo real distribuídos e guiados por evento. A UML-RT não possui semântica formal, logo não é possível realizar verificação formal do modelo. O presente trabalho propõe o mapeamento dos elementos de comunicação da UML-RT para a álgebra de processos π-calculus, a fim de prover semântica formal à UML-RT. Com objetivo de automatizar o mapeamento, foi desenvolvido um protótipo de tradutor que captura o modelo UML-RT especificado na ferramenta RoseRT e determina suas definições π-calculus. As definições π-calculus geradas utilizam a sintaxe da gramática do HAL-JACK, que é uma ferramenta integrada para verificação e análise de sistemas expressos em π-calculus, assim as definições π-calculus podem ser submetidas ao HAL-JACK para verificação formal de propriedades. Este trabalho detalha o mapeamento UML-RT para π-calculus, descreve o protótipo desenvolvido e apresenta alguns exemplos do mapeamento do modelo UML-RT para definições π-calculus.

Abstract

The UML (Unified Modeling Language) is a language for designing, constructing, and documenting the artifacts of software systems. The UML-RT, supported by Rational Rose RealTime (RoseRT) tool, is an UML extension that allows to model event-driven and distributed real-time systems. The UML-RT does not have formal semantics; therefore it is not possible to do formal verification. This work proposes a mapping of UML-RT communicating elements to π-calculus process algebra, aiming to provide formal semantics to UML-RT. In order to automatize the proposed mapping, a prototype of a translator was developed. The prototype captures information of UML-RT model from RoseRT tool and generates π-calculus definitions. The generated π-calculus definitions use the syntax of HAL-JACK, that is an integrated tool for verification and analysis of systems expressed by π-calculus formalism, so that the π-calculus definitions could be used in HAL-JACK for the formal verification of properties. In this article, we detail the mapping of UML-RT to π-calculus, we describe the prototype, and we present some examples of UML-RT model translation to π-calculus definitions.

Sumário

Capítulo 1

Introdução

Neste capítulo é apresentada a motivação do trabalho. Em seguida, descreve-se o objetivo do trabalho considerando o problema identificado e a solução proposta. Por fim, são descritos os capítulos que compõem o trabalho.

1.1 Motivação

Segundo a OMG [1], um bom modelo de sistema deve enfatizar apenas aspectos importantes, ser entendido prontamente pelos participantes do projeto, representar fielmente o sistema, e ainda ser mais barato de construir e estudar que o sistema modelado.

O modelo é usado como base para discussão e validação de requisitos entre desenvolvedores, clientes e demais envolvidos no projeto, e como guia de desenvolvimento, sendo o sistema final, produto de sucessivos refinamentos do modelo. Além disso, a modelagem ajuda a entender problemas complexos, o que ocorre através de análise e experimentação do modelo, e de investigação de alternativas de solução.

No processo de modelagem não apenas um modelo é desenvolvido para o sistema, mas sim sucessivos modelos, sendo um modelo o refinamento do anterior. Porém, é interessante elucidar que a modelagem não torna o projeto infalível, pois o sistema construído respeitando

o modelo pode ainda apresentar discrepâncias em relação ao sistema desejado, devido a erros no entendimento do sistema durante o processo de modelagem e erros na própria construção.

A linguagem de modelagem mais conhecida e utilizada nos dias de hoje é a UML (*Unified Modeling Language*). A UML foi definida pela OMG como uma linguagem de modelagem para especificar, construir e documentar artefatos de sistemas de software, assim como para modelar processos e outros sistemas não baseados em software [1].

Com o intuito de adequar a UML à modelagem de sistemas de tempo real, foram propostas várias extensões conhecidas como perfis (*profiles*) [7]. A empresa Rational Software (hoje IBM) definiu a extensão UML-RT, que permite a modelagem de sistemas distribuídos e guiados por evento (*event-driven*) [8] e é usada pela ferramenta Rational Rose RealTime (RoseRT) [9]. A UML-RT estende a UML com quatro componentes: cápsula, porta, conector e protocolo. A cápsula representa um componente do sistema e é definida pelos diagramas de estados e de estrutura, podendo se utilizar também de um diagrama de classe.

Pressman [4] aponta que na indústria entre 50% e 70% de todo esforço gasto num sistema serão despendidos depois que ele estiver pronto. Ele também ilustra o impacto no custo da detecção de erros feita antecipadamente: "*supondo que um erro descoberto durante a fase de projeto custe 1 unidade monetária para ser corrigido; o mesmo erro descoberto logo antes do início dos testes custará 6,5 unidades; durante os testes, 15 unidades; e, após o lançamento, entre 60 e 100 unidades*".

Dada a importância de antecipar os testes para fases iniciais do desenvolvimento do sistema, em especial para a fase de modelagem, surge a necessidade da verificação formal do modelo do sistema. A verificação formal engloba a modelagem formal do sistema, a especificação formal de requisitos ou propriedades, e o estabelecimento de regras de inferência para provar que o modelo satisfaz as propriedades [5]. A especificação formal é um

modelo matemático do sistema e pode ser realizada através de métodos formais ou álgebras de processos [6].

A verificação formal adquire maior relevância quando se trata de requisitos não funcionais, que estão cada vez mais presentes nos sistemas complexos e críticos desenvolvidos atualmente. Cada requisito não funcional, como tolerância a falhas, segurança lógica (*security*), segurança física (*safety*) e *liveness* [2], possui associada uma classe de propriedades que se deseja provar a respeito do sistema. Por exemplo, as propriedades de *safety* podem garantir que o sistema nunca entrará em um estado indesejado que cause prejuízo a pessoas e equipamentos; já propriedades de *liveness* podem assegurar que um estado desejado pode ser atingido em algum ponto da execução do sistema [3].

Como a UML-RT não possui semântica formal [7], não é possível realizar verificação formal de propriedades nos modelos especificados em UML-RT. Algumas abordagens, por exemplo, [10] e [11], sugerem modelar o sistema usando diretamente um método formal, a fim de viabilizar a verificação formal do modelo. Outras abordagens propõem a transformação do modelo UML-RT para uma álgebra de processos, de modo que a verificação formal seja feita no modelo previamente especificado em UML-RT. Terriza et al [13] descrevem o mapeamento de UML-RT para CSP+T [14]. Fischer et al [15] e Engels et al [17] propõem a conversão da UML-RT em CSP [16]. Mota et al [18] apresentam o mapeamento da UML-RT para o método formal Circus [19], que combina conceitos de CSP e Z [20]. Assano [21] propõe a tradução da UML-RT para CCS [22].

1.2 Objetivo

Dada a ausência de semântica formal da UML-RT que impossibilita a verificação formal de propriedades nos modelos especificados em UML-RT, o objetivo do presente trabalho é prover semântica formal à UML-RT. Para isso é apresentado o mapeamento da UML-RT para álgebra de processos π-calculus [23], considerando os aspectos relacionados à comunicação entre os componentes do sistema.

O π-calculus é uma álgebra de processos, proposta por Milner [23], para modelar sistemas concorrentes. A vantagem de π-calculus em relação a seus precursores CSP [16] e CCS [22] é o conceito de mobilidade, que é proporcionada pela reconfiguração dinâmica de nomes. A característica de mobilidade permite duas grandes contribuições deste trabalho em relação aos trabalhos correlatos: o reaproveitamento da definição das cápsulas e o tratamento de portas não conectadas.

O mapeamento proposto prevê a geração de definições π-calculus usando a sintaxe da ferramenta HAL-JACK [24]. O HAL-JACK (*HD-Automata Laboratory*) é uma ferramenta integrada para especificação, verificação e análise de sistemas expressos em π-calculus, e utiliza a lógica temporal pi-logic para definir propriedades a serem verificadas. Assim, o HAL-JACK pode ser empregado para verificar formalmente propriedades do modelo dado pelas definições π-calculus oriundas do mapeamento.

Outra grande contribuição deste trabalho é o desenvolvimento de um protótipo que captura as informações do modelo UML-RT especificado na ferramenta RoseRT e provê as definições π-calculus de acordo as regras de mapeamento estabelecidas. O protótipo simplifica o trabalho desenvolvedor que utiliza o RoseRT para modelar sistemas de tempo real, uma vez que gera automaticamente as definições π-calculus do modelo, permitindo que o desenvolvedor dedique mais tempo em análises e verificações formais do modelo.

1.3 Próximos capítulos

O próximo capítulo fornece a fundamentação teórica do trabalho. A fundamentação inclui conceitos gerais de UML-RT, explicação sobre o que são métodos formais e álgebras de processo, apresentação do π-calculus e particularidades da gramática HAL-JACK para escrita de definições π-calculus. Além disso, são descritos os trabalhos correlatos e uma comparação entre eles.

As regras de mapeamento de UML-RT para π-calculus são apresentadas no Capítulo 3. O Capítulo 4 explica a arquitetura básica do protótipo desenvolvido para gerar automaticamente as definições π-calculus a partir do modelo em UML-RT com base nas regras de mapeamento propostas. O Capítulo 5 descreve alguns exemplos do mapeamento e da utilização do protótipo. O Capítulo 6 resume as considerações tomadas na definição do mapeamento e na construção do protótipo, além de prover uma análise do trabalho em relação aos trabalhos correlatos. Por fim, o Capítulo 7 relata as conclusões e comentários finais do trabalho.

Capítulo 2

Fundamentação Teórica

Este capítulo expõe os fundamentos teóricos necessários à compreensão do trabalho. Inicialmente comenta-se o que são sistemas de tempo real a fim de indicar o desafio de modelá-los e construí-los. Em seguida, é apresentada UML-RT, uma extensão da UML para modelagem de sistemas de tempo real. Explicam-se os elementos básicos da UML-RT (cápsula, porta, conector e protocolo) e descreve-se um exemplo de sistema que possui um componente que envia mensagem e outro que recebe tal mensagem.

Os conceitos de métodos formais e álgebra de processos também são apresentados, enfatizando pontos fortes, pontos fracos e alguns mitos acerca do uso de métodos formais no desenvolvimento de sistemas. Posteriormente, a álgebra de processos π-calculus é descrita. Exemplifica-se a principal característica do π-calculus, a mobilidade, e elencam-se algumas frentes de utilização dessa álgebra atualmente. Em seguida, explica-se a gramática do HAL-JACK, usada neste trabalho como sintaxe para as definições π-calculus com objetivo de permitir que as definições π-calculus possam ser submetidas à ferramenta HAL-JACK para verificações formais. Por fim, é provida uma análise comparativa entre os trabalhos correlatos.

2.1 Sistemas de Tempo Real

A principal característica de um sistema de tempo real é responder corretamente a entradas em intervalos de tempo válidos. Dessa forma, a exatidão do sistema não depende somente dos resultados que ele provê, mas também do tempo em que tais resultados estão disponíveis [8]. Geralmente esses sistemas encontram-se embarcados em sistemas maiores. Além disso, eles podem ser distribuídos e concorrentes a fim de minimizar o tempo de resposta de processamento e otimizar a utilização dos recursos [25].

Os sistemas de tempo real, em sua maioria, são também reativos, gerando certa ação em resposta a eventos ou mudanças do ambiente. Contudo os conceitos de reativo e de tempo real são distintos, como exemplificados por Carlson [26] com um freio de emergência: um botão quando apertado aciona o freio de emergência de um dado componente, logo o comportamento é reativo; caso o componente deva ser parado após dois segundos do acionamento do botão, o freio de emergência será definido também como sistema de tempo real.

Dentre as classificações para sistemas de tempo real, têm-se *hard*, *soft*, *crítico*, *não-crítico*, *guiado por tempo* ou *por evento*. Num sistema *hard*, considera-se incorreto o não cumprimento da restrição de tempo; enquanto um sistema *soft* pode fornecer seu resultado fora do tempo especificado. O sistema é *crítico* se a violação da restrição de tempo traz conseqüências desastrosas, por exemplo, prejuízos a equipamentos e pessoas. Categoriza-se um sistema como *guiado por tempo* quando suas atividades são disparadas em certos instantes de tempo. Já num sistema *guiado por evento*, as atividades são acionadas por ocorrências não determinísticas de eventos internos e externos [26].

As características apresentadas indicam a complexidade de sistemas de tempo real e, conseqüentemente, o grande desafio de especificá-los e construí-los. Existem projetos como

Ptolemy [27] e Quack [28] focados em metodologias de desenvolvimento de sistemas embarcados e de tempo real, bem como na construção de ferramentas que auxiliem seu desenvolvimento.

2.2 UML-RT

Nesta seção são apresentados os elementos que compõem a UML-RT e, em seguida, é descrito um exemplo de modelagem UML-RT que permite compreender como os elementos da UML-RT são usados para compor um modelo de sistema.

2.2.1 Elementos da UML-RT

A UML-RT estendeu o padrão UML com elementos para facilitar a especificação de sistemas de tempo real. Esses elementos têm origem na linguagem de modelagem ROOM (*Real Time Object Oriented Modeling*) [8]. Três dos elementos, *cápsula*, *porta* e *conector*, são usados para modelar estrutura do sistema, e o quarto elemento, *protocolo*, modela a comunicação dentro do sistema. A ferramenta case RoseRT [9] usa a UML-RT e permite a geração automática de código a partir do modelo.

2.2.1.1 Cápsula

A *cápsula* (classe ativa) modela componentes de software complexos que podem ser concorrentes e estar fisicamente distribuídos. Seu comportamento é descrito com diagrama de estados proveniente do padrão UML. Sua estrutura interna, dada pelo diagrama de estrutura, é composta por subcápsulas (*capsule roles*) e conexões entre elas. Pode-se definir também um diagrama de classes para a cápsula.

A cápsula pode ter uma cardinalidade associada e ser classificada em fixa, opcional e plug-in. A cápsula fixa é instanciada, automaticamente em tempo de execução, em cada cápsula-mãe que a possue. A cápsula opcional deve ser instanciada explicitamente pela cápsula-mãe através de códigos no diagrama de estados da cápsula-mãe. A cápsula plug-in nunca é instanciada diretamente, sua cápsula-mãe requisita através de códigos que deseja importar tal cápsula (que está disponível após a decomposição de outra cápsula-mãe).

Somente a cápsula fixa será tratada neste trabalho por ser este o tipo padrão. A Figura 1 mostra o diagrama de estrutura da cápsula *A* composta por três subcápsulas: *capB* (instância da cápsula *B*), *capC* (instância da cápsula *C*) e *capD* (instância da cápsula *D*). A cápsula *A* pode ser chamada de cápsula-mãe e as subcápsulas de cápsulas-filha.

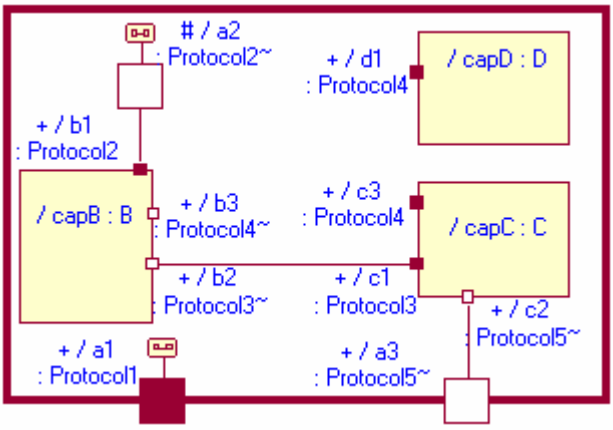


Figura 1: Diagrama de estrutura da cápsula A.

2.2.1.2 Protocolo

O *protocolo* define o número de participantes da comunicação e quais sinais são recebidos e enviados por cada participante. Ele pode também especificar, através de um diagrama de estados, a seqüência válida da troca de sinais.

2.2.1.3 Conector

Os *conectores* agem como canais de comunicação entre duas ou mais portas, que devem desempenhar diferentes papéis (*protocol roles*) de um mesmo protocolo. O conector é representado por uma linha contínua ligando duas portas.

2.2.1.4 Porta

A *porta* permite troca de mensagem entre cápsulas e é definida através do protocolo. Na Figura 1, as portas *b2* e *c1* pertencem ao protocolo chamado *Protocol3* e estão ligadas através de um conector. Vale notar que essas portas desempenham papéis complementares, a ver pela representação de *b2* como quadrado vazio, e de *c1* como quadrado preenchido.

A porta pode ter cardinalidade associada e ainda ser classificada quanto à visibilidade (*pública* ou *protegida*), terminação (*end* ou *relay*) e conexão (*conectadas* ou *não conectadas*). A porta *pública* localiza-se na fronteira da cápsula e pode se comunicar com portas de outras cápsulas. A porta *protegida* não é visível fora da cápsula, ela provê a comunicação da cápsula com suas subcápsulas. Na Figura 1, a cápsula *A* possui portas *a1* e *a3* públicas (indicadas pelo símbolo +) e *a2* protegida (indicada pelo símbolo #).

Quanto à terminação, a porta *end* provê a conexão entre o comportamento da cápsula (dado pelo diagrama de estados) e outras cápsulas. Na Figura 1, *a1* e *a2* são portas *end*. A porta *relay* é usada para exportar a interface das subcápsulas. Na Figura 1, *a3* é porta *relay*. Quando a porta *relay* da cápsula-mãe recebe uma mensagem, a mensagem é passada automaticamente à porta da subcápsula a qual está conectada. Quando a subcápsula envia uma mensagem pela sua porta, a mensagem é transmitida para a porta *relay* da cápsula-mãe através da conexão entre as portas, em seguida, a cápsula-mãe passa a mensagem para o meio externo. A porta do tipo *relay* por definição é pública e conectada.

A porta *conectada* (*b2* e *c1* na Figura 1) possui um conector unindo-a com outra porta para troca de mensagens. A porta *não conectada* não possui um conector com outras portas, mas permite comunicação dinâmica em tempo de execução através do registro de seu nome. Na Figura 1, *b3*, *c3* e *d1* são exemplos de portas não conectadas. Por exemplo, a porta *c3* pode ser registrada com o nome de assinante *alertaC* durante um dado momento na execução do diagrama de estados da cápsula *capC*. Da mesma forma, a porta *d1* pode ter o nome de registro de assinante *alertaD*. Se a porta *b3* for configurada com o nome de provedora *alertaC*, o sinal enviado por *capB* através de *b3* será recebido por *capC*. Analogamente, se o nome de *b3* for *alertaD*, seu sinal será recebido por *capD*. Na ferramenta RoseRT, a porta assinante é chamada de SAP (*Service Access Point*) e a porta provedora é chamada de SPP (*Service Provisioning Point*).

2.2.1.5 Diagrama de estrutura da cápsula

O diagrama de estrutura representa a estrutura interna da cápsula, como pode ser visto na Figura 1, que é composta por subcápsulas (*capsule roles*), portas e conectores.

2.2.1.6 Diagrama de estados da cápsula

O diagrama de estados descreve o comportamento da cápsula. A Figura 2 representa o diagrama de estados da cápsula *A*. O *ponto inicial* (círculo preto preenchido) é um ponto especial que indica o início do diagrama de estados. A *transição inicial* (*Initial* na Figura 2) conecta o ponto inicial ao estado inicial e não possui gatilho (*trigger*) associado. O *estado inicial* (*S1* na Figura 2) é o primeiro objeto ativado dentre os objetos que compõem o diagrama de estados.

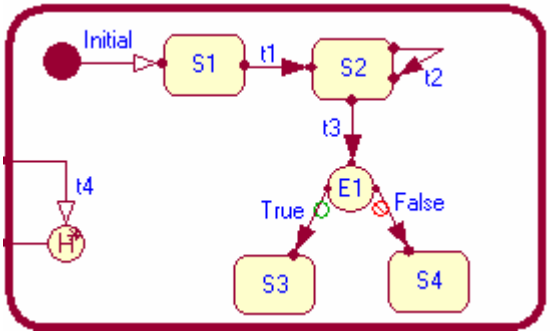


Figura 2: Diagrama de estados da cápsula A.

O *estado* é a condição durante o tempo de vida do componente (cápsula) no qual ele está preparado para processar eventos. Na Figura 2, há os estados *S1*, *S2*, *S3* e *S4*. O estado pode possuir ações que são executadas ao entrar nele (*entry actions*) e ao sair dele (*exit action*). Com o propósito de modelar um estado complexo através da abstração de seu comportamento em diferentes níveis, o estado pode ser composto por outros estados, chamados de sub-estados. O estado que possui sub-estados é conhecido como estado composto ou hierárquico. Os demais estados são chamados de simples.

O estado pode ser classificado como estado de escolha ou *ponto de escolha*, por exemplo, *E1* na Figura 2. O ponto de escolha possui duas transições de saída, podendo cada transição ser ligada a um estado diferente. A decisão sobre qual transição percorrer é tomada após a execução da transição original e da verificação da condição contida no ponto de escolha.

A *transição* é a relação entre dois estados: um estado origem e um estado destino. Ela especifica quando um objeto no estado de origem passa ao estado destino ao receber um dado evento e ao cumprir certa condição. Na Figura 2, *t1* e *t2* são exemplos de transição. Três conceitos em torno da transição de estados necessitam ser elucidados, são eles: *gatilho* (*trigger*), *guarda* (*guard condition*) e *ação* (*action*). O *gatilho* indica quais sinais de quais portas podem acionar a transição, mas a transição só irá realmente ocorrer se também a condição estabelecida na *guarda* for satisfeita. Uma vez acionada a transição, sua *ação* é

executada. A ação pode ser uma chamada de operação, uma manipulação de variável ou um envio de sinal a outras cápsulas.

O diagrama de estados pode também conter *pontos de junção*. Existe o ponto de junção do tipo história (*history*) que sempre guarda o último estado ativo no diagrama de estados da cápsula. Na Figura 2, a transição *t4* leva a um estado do tipo história. Há ainda o ponto de junção localizado na fronteira dos estados, representando a origem ou o destino de uma transição, por exemplo, os pontos extremos na transição *t1* da Figura 2.

2.2.2 Exemplo de modelagem UML-RT

Para entender melhor os elementos da UML-RT, segue um exemplo de envio e recebimento de mensagem que foi adaptado da documentação do RoseRT. O sistema é representado pela cápsula *TxRxSytem* da Figura 3 e é composto por uma subcápsula *sender* do tipo *Sender* e uma subcápsula *receiver* do tipo *Receiver*. A porta *a* de *sender* está ligada à port *b* de *receiver* através de um conector. Para definir os sinais trocados na comunicação das subcápsulas foi especificado o protocolo *TxRxProtocol*. Este protocolo possui apenas um sinal de saída chamado *stringSig* (Figura 4), cujo tipo é uma classe interna do RoseRT para representar string.

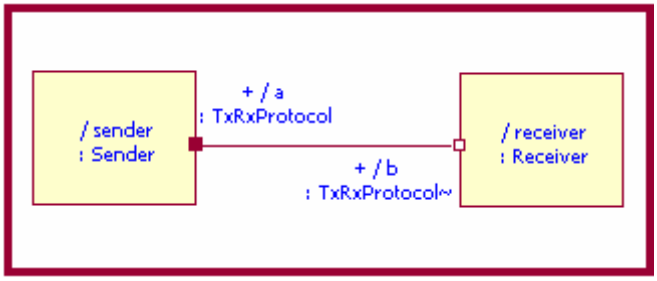


Figura 3: Diagrama de estrutura de *TxRxSystem*.

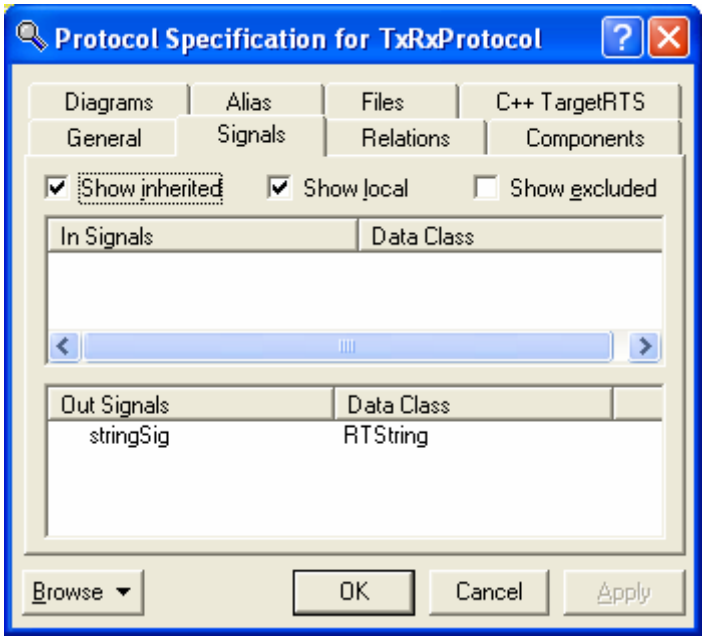


Figura 4: Tela de especificação do protocolo *TxRxProtocol.*

A cápsula *Sender* possui duas portas como mostra seu diagrama de estrutura na Figura 5. A porta *a* implementa o protocolo *TxRxProtocol*. A porta *timer* implementa o protocolo *Timing*, provido pelo RoseRT, que permite que a cápsula receba um sinal em intervalos de tempo regulares ou após um intervalo de tempo estabelecido.

A Figura 6 mostra o diagrama de estados da cápsula *Sender*. As informações das transições foram colocadas na Tabela 1 auxiliar, pois são configuradas em telas próprias do RoseRT, não sendo visíveis diretamente pelo diagrama de estados. Por exemplo, para transição *sendMessage* da cápsula *Sender*, a Figura 7 mostra onde a informação do gatilho é cadastrada, e a Figura 8 mostra onde a ação da transição é inserida no RoseRT.

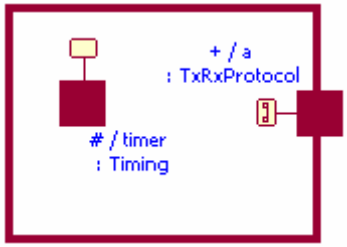


Figura 5: Diagrama de estrutura de *Sender*.

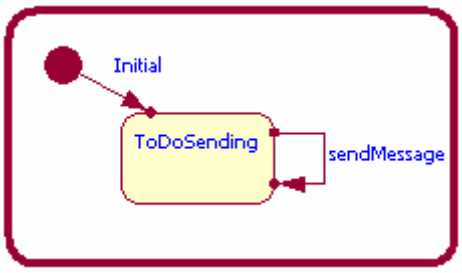


Figura 6: Diagrama de estados de *Sender*.

Tabela 1: Transições do diagrama de estados de *Sender*.

Transição	Porta	Sinal	Ação
Initial	-	-	*timer.informEvery(RTTimespec(10,0));*
sendMessage	*Timer*	*timeout*	*RTString xAtt = "teste";* *a.stringSig(xAtt).send();*

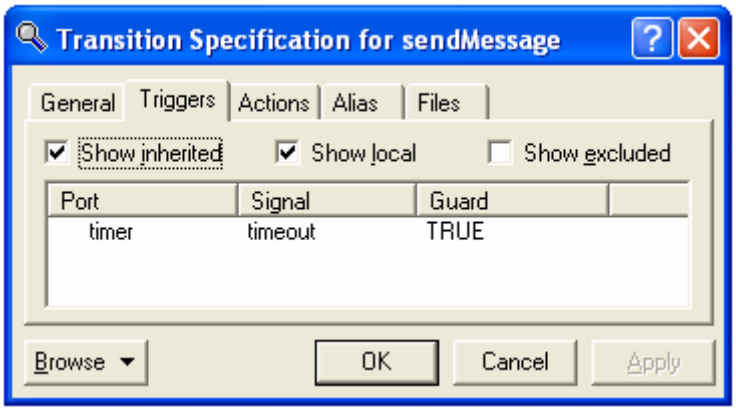


Figura 7: Tela no RoseRT com gatilho da transição.

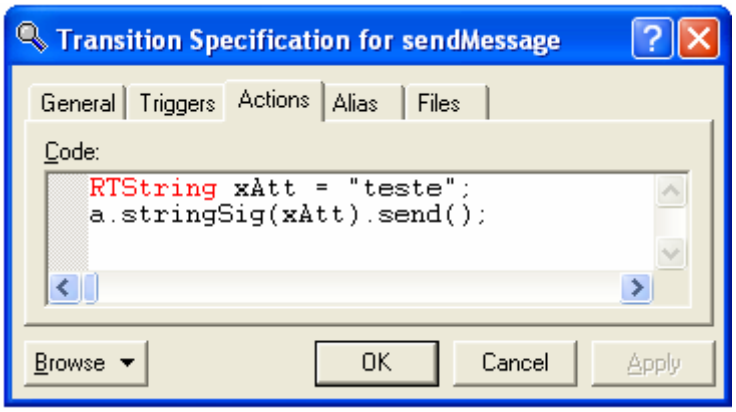


Figura 8: Tela no RoseRT com ação da transição.

Quando o diagrama de estado de *Sender* (Figura 6) se inicia, a transição *Initial* é ativada e sua ação executada. Esta ação configura a porta *timer* para receber sinais de *timeout* a cada 10 segundos, conforme o código disposto na Tabela 1. No estado *ToDoSending*, a cápsula *Sender* pode receber um sinal de *timeout*, como indicado pelos campos Porta e Sinal na Tabela 1 para a transição *sendMessage*. Uma vez acionada a transição *sendMessage*, sua ação é executada. O código dessa ação é definir uma variável *xAtt* do tipo RTString com o valor "teste", e em seguida enviá-la pelo sinal *stringSig* através da porta *a*. Após a execução da transição *sendMessage*, a cápsula Sender volta ao estado *ToDoSending*.

A cápsula *Receiver* possui apenas a porta *b* em seu diagrama de estrutura, mostrado na Figura 9. Quando seu diagrama de estados, ilustrado na Figura 10, inicia, a transição *Initial* é ativada e a cápsula atinge o estado *Ready*. A transição *receiveMessage* é acionada quando há o recebimento de um sinal *stringSig* pela porta *b* como mostra a Tabela 2. A ação dessa transição é capturar a mensagem recebida no sinal através do código **rtdata*, passando o valor para variável *msgAtt*. A variável *msgAtt* poderia ter sido definida no próprio código, mas neste exemplo ela foi definida como atributo do diagrama de estados. Após executada a transição *receiveMessage*, a cápsula volta ao estado *Ready* estando apta a receber novos sinais.

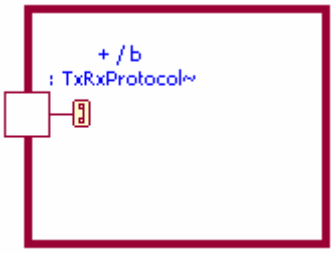


Figura 9: Diagrama de estrutura de *Receiver*.

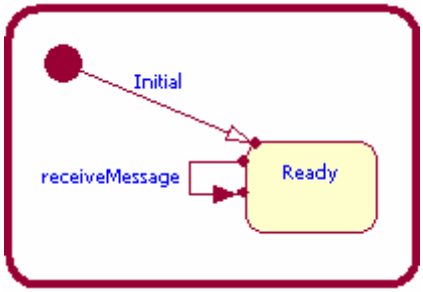


Figura 10: Diagrama de estados de *Receiver*.

Tabela 2: Transições do diagrama de estados de *Receiver*.

Transição	Porta	Sinal	Ação
Initial	-	-	-
receiveMessage	*B*	*stringSig*	*msgAtt = *rtdata;*

2.3 Métodos Formais e Álgebra de Processos

Método formal significa o uso de notações e técnicas matemáticas, em particular que auxiliam o desenvolvimento (especificação, análise, desenho, construção e verificação) e manutenção de sistemas computacionais.

A álgebra (ou cálculo) de processos é uma teoria de processos concorrentes. Para entender melhor este conceito, é importante definir álgebra como uma estrutura matemática composta por um conjunto de valores e operações sobre esses valores. Numa álgebra de processos, processos são valores e existem operadores, por exemplo, composição paralela, escolha e seqüência, que permitem cálculos com processos [29].

O termo processo refere-se ao comportamento de um sistema. Comportamento é o conjunto de eventos ou ações que um sistema pode realizar, como a execução de um sistema de software e até ações de seres humanos [30]. Ao se construir com a álgebra de processos as definições que representam o comportamento de um sistema, comumente essas definições são denominadas de modelo (ou modelo formal) do sistema.

Baeten [30] descreve a história da álgebra de processos e a evolução da teoria de automata à teoria de sistemas concorrentes. A álgebra de processos aborda a teoria de sistemas concorrentes, onde existe interação por troca de mensagens entre sistemas executados em paralelo e que ainda podem estar distribuídos no meio. Isso explica a importância da composição paralela como operador da álgebra de processos [29].

A principal vantagem da álgebra de processos é a possibilidade de verificação formal do sistema, com objetivo de analisar se dois sistemas são equivalentes (*equivalence checking*) ou confrontar o sistema com certas propriedades (*model checking*) [31].

A verificação formal pode ser realizada pela manipulação manual das fórmulas que descrevem o sistema, ou com auxílio de uma ferramenta que analise exaustivamente o modelo sem a interação do usuário. Este tipo de ferramenta gera uma árvore de possibilidades de execução do sistema a partir do modelo fornecido, e faz uma varredura nesta árvore a fim de detectar se uma dada propriedade é cumprida ou se este modelo é equivalente a um outro. Contudo, a verificação formal do modelo enfrenta um sério problema devido ao grande número de estados que a árvore mencionada pode possuir. Tal problema está sendo estudado por vários grupos atualmente, por exemplo, o projeto Behave [32], com o objetivo de construir algoritmos eficientes para minimização de estados. Uma outra dificuldade da verificação formal é expressar formalmente as propriedades, as quais em geral são escritas usando lógicas temporais ou modais.

Bowen e Hinchey [33] explicam os muitos mitos acerca da utilização de métodos formais no processo de desenvolvimento de sistemas. Entre os mitos está a crença em que métodos formais garantem inteiramente a correção do sistema construído, e que substituem métodos tradicionais de engenharia de software. Abaixo estão descritos pontos fortes e fracos de álgebra de processos reunidos também dos trabalhos de Thomas [34] e Calder [35].

Pontos fortes de álgebra de processos:

- A álgebra de processos permite uma descrição rigorosa e não ambígua do sistema.
- Modelos mais abstratos permitem definições genéricas das interações do sistema.

- A definição de um modelo formal força serem explícitos contextos e suposições sobre o sistema. Assim, erros podem ser detectados antes da construção definitiva do sistema, reduzindo o custo com manutenções.
- A descrição formal de um sistema facilita a análise quanto à viabilidade de integração com outro sistema.
- Não há necessidade de empregar álgebra de processo para especificar todo o sistema; pode-se aplicá-la somente em alguns componentes mais críticos.
- Modelos fiéis ao comportamento do sistema provêem plataformas para experimentação e investigação de novos comportamentos.
- A análise do modelo formal pode ser realizada através de manipulação algébrica e de simulação com ferramentas próprias.

Pontos fracos de álgebra de processos:

- Uma resistência cultural existe para adoção de álgebra de processo, entretanto esta resistência é inerente a qualquer mudança proposta.
- Dificuldade de capturar o nível adequado de abstração. Quer dizer, ao se modelar um sistema faz-se uma abstração de seu comportamento para representá-lo através de notações formais, porém uma abstração excessiva pode ocasionar divergências entre o modelo e o sistema original.
- As notações formais são complexas, isso restringe seu uso uma vez que há a necessidade de um público mais especializado para adotar tais notações. Porém vale ressaltar que este público especializado não precisa ser de matemáticos altamente treinados, e sim de pessoas com conhecimento do método formal empregado.

- As novas características do método formal são providas por extensões do método já existente ou são criadas novas álgebras ao invés de enriquecer uma única. A variedade de álgebras existentes dificulta a adoção de uma única e pode levar a existência de álgebras incompatíveis na especificação de componentes de um mesmo sistema.

2.4 π-calculus

O π-calculus [23] ou pi-calculus, desenvolvido por Milner, é uma álgebra de processos para sistemas que se comunicam concorrentemente, ou seja, sistemas compostos por processos que rodam em paralelo e que interagem entre si através de canais. O canal pode ser definido como uma abstração do *link* de comunicação entre dois processos, permitindo que processos interajam entre si enviando e recebendo mensagens através de canais [36][37][38].

A principal diferença entre π-calculus e seus antecessores, CCS e CSP, está na possibilidade de passar nomes de canais como dado através de outros canais. Esta característica permite ao π-calculus expressar mobilidade. Mobilidade não significa a mobilidade da posição do processo, mas sim das conexões entre os processos. Assim, π-calculus pode ser entendido como um modelo de mudança de conectividade entre processos interativos.

Nesta seção é usada a notação proposta por Milner [23] para escrever as definições π-calculus. O prefixo π do π-calculus representa o envio de mensagem, ou recebimento de mensagem ou ação não observável.

A sintaxe é:

$\pi ::= x(y)$: recebe a mensagem *y* pelo canal *x*

$\bar{x}\langle y \rangle$: envia a mensagem *y* pelo canal *x*

τ : ação não observável

É comum se referir a *x* em *x(y)* como porta de entrada, pois recebe a mensagem *y*, e se referir a *x* em $\bar{x}\langle y \rangle$ como porta de saída, já que envia a mensagem *y*. A expressão que define um processo *P* em π-calculus é:

$$P ::= \sum_{i \in I} \pi_i.P_i \quad | \quad P_1 \mid P_2 \quad | \quad new\, a\, P \quad | \quad !P$$

onde *I* é um conjunto finito de índices.

O termo $\sum_{i \in I} \pi_i.P_i$ indica escolha não determinística (*summation*). O ponto é o operador de seqüência, onde a ação dada por π_i deve ocorrer antes de P_i se tornar ativo. Se *i* for zero, tem-se soma nula ou processo parado, que é representado por *0*. Em geral, omite-se o processo parado após uma ação, por exemplo, escreve-se *x(y)* ao invés de *x(y).0*.

A composição paralela (*composition*), denotada por $P_1 \mid P_2$, significa que os processos P_1 e P_2 executam concorrentemente. O termo $new\, a\, P$ representa a restrição do nome *a* ao contexto do processo *P*. O operador replicação (*replication*), dado por !*P*, é a composição paralela de um número infinito de cópias de *P*.

No envio de mensagem $\bar{x}\langle y \rangle$, o nome *y* é livre. Entretanto, na restrição $new\, y\, P$ e no recebimento *x(y)*, *y* é um nome restrito. Mudar um nome restrito por um nome livre é chamado de *alpha-conversion*, por exemplo, pode-se alterar o nome *b* em $(new\, b)a.b$ para *b'*, gerando $(new\, b')a.b'$. Se $P = (new\, b)a.b$ e deseja-se obter $\{b/a\}P$, ou seja, deseja-se que o

nome *a* em *P* seja substituído por *b*, é preciso utilizar *alpha-conversion* a fim de alterar *b* para b' antes de alterar *a* para *b*; logo $\{b/a\}P = (new\, b')b.b'$.

Dois processos *P* e *Q* em π-calculus são estruturalmente congruentes, escrito como $P \equiv Q$, se é possível transformar um no outro usando as seguintes regras:

- Mudar nomes restritos (*alpha-conversion*)
- Reordenar termos numa escolha
- $P \mid 0 \equiv P, \quad P \mid Q \equiv Q \mid P, \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
- $new\, x(P \mid Q) \equiv P \mid new\, x\, Q$, se *x* não pertencer ao conjunto de nomes livres de *P*

  $new\, x\, 0 \equiv 0, \quad new\, xy\, P \equiv new\, yx\, P$
- $!P \equiv P \mid !P$

As regras de transição no π-calculus estão dispostas abaixo:

- TAU: $\tau.P + M \rightarrow P$
- REACT: $(x(y).P + M) \mid (\overline{x}\langle z\rangle.Q + N) \rightarrow \{z/y\}P \mid Q$

  onde $\{z/y\}P$ significa que todos os nomes *y* em *P* devem ser substituídos por *z*.
- PAR: $\dfrac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$
- RES: $\dfrac{P \rightarrow P'}{new\, x\, P \rightarrow new\, x\, P'}$
- STRUCT: $\dfrac{P \rightarrow P'}{Q \rightarrow Q'}$ se $P \equiv Q$ e $P' \equiv Q'$

A fim de exemplificar a aplicação das regras de transição no π-calculus, seja a seguinte definição do processo *P*:

$$P = Q \mid R \mid S \mid T$$

onde:

$Q = \bar{z}\langle x \rangle + \bar{z}\langle y \rangle$

$R = z(a).\bar{a}\langle b \rangle$

$S = x(c)$

$T = y(d)$

Logo:

$P = (\bar{z}\langle x \rangle + \bar{z}\langle y \rangle) \mid z(a).\bar{a}\langle b \rangle \mid x(c) \mid y(d)$

A Figura 11 representa o processo *P* composto por *Q*, *R*, *S* e *T*. As ações $\bar{z}\langle x \rangle$ de *Q* e *z(a)* de *R* são complementares e podem interagir entre si. Esta possível interação está ilustrada pela conexão *z* entre *Q* e *R*. Caso haja sincronização entre os termos, *P* se transforma em *P1* (Figura 12):

$$P1 = 0 \mid \bar{x}\langle b \rangle \mid x(c) \mid y(d)$$

Após a reação $P \rightarrow P1$, todas as ocorrências de *a* em *R* são substituídas pelo nome recebido *x*. Na Figura 12, assume-se que $U = \bar{x}\langle b \rangle$ e não se representa *Q* por ele ser *0* (processo parado).

Há outra interação possível em *P* entre $\bar{z}\langle y \rangle$ de *Q* e *z(a)* de *R*. Esta interação transforma *P* em *P2* (Figura 13):

$$P2 = 0 \mid \bar{y}\langle b \rangle \mid x(c) \mid y(d)$$

Após a reação $P \rightarrow P2$, todas as ocorrências de *a* em *R* são agora substituídas pelo nome *y* recebido através do canal *z*. Na Figura 13, assume-se que $V = \bar{y}\langle b \rangle$ e não se representa *Q* por ele ser *0* (processo parado).

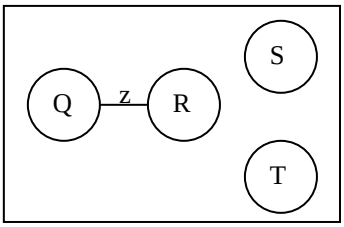


Figura 11: Processo *P*.

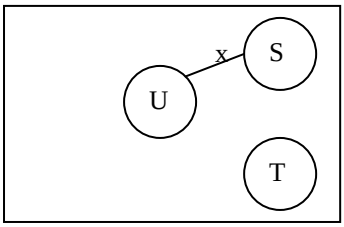


Figura 12: Processo *P1*.

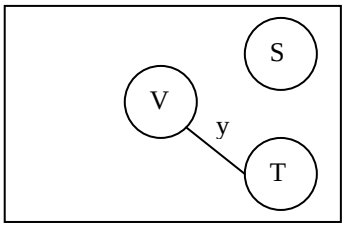


Figura 13: Processo *P2*.

O conceito de mobilidade do π-calculus pode ser percebido pelo exemplo acima, pois os canais *x* (na reação $P \rightarrow P1$) e *y* (na reação $P \rightarrow P2$) são passados como mensagem através do canal *z* que liga os processos *Q* e *R*. A mobilidade está na possibilidade de *U* se conectar a *S* após a reação $P \rightarrow P1$, ou de *V* se conectar a *T* após a reação $P \rightarrow P2$.

Em *P1*, se $\overline{x}\langle b \rangle$ interagir com *x(c)*, tem-se a reação $P1 \rightarrow P3$, onde:

$$P3 = 0 \mid 0 \mid 0 \mid y(d) = T$$

Em *P2*, se $\overline{y}\langle b \rangle$ interagir com *y(d)*, tem-se a reação $P2 \rightarrow P4$, onde:

$$P4 = 0 \mid 0 \mid x(c) \mid 0 = S$$

A versão do π-calculus apresentada foi o *monadic* π-calculus, que consiste na troca de mensagens compostas por apenas um nome. O *polyadic* π-calculus permite o envio de mensagens compostas por mais de um nome. Segundo Milner [23], é possível representar o *polyadic* em termos do *monadic*.

Embora π-calculus não seja uma linguagem de programação, ele é visto como um dos melhores frameworks formais para prover semântica às linguagens de programação concorrentes e distribuídas. Por exemplo, Pierce e Turner [39] propõem uma linguagem de programação chamada Pict com base em π-calculus. Como formalismo, π-calculus pode ajudar a responder algumas questões fundamentais, como no estudo entre comunicação síncrona e assíncrona realizado por Palamidessi [40].

O π-calculus pode ser usado também como linguagem de modelagem. Por exemplo, Power e Sinclair [41] descrevem com π-calculus o modelo formal do compilador Forth, usado

para sistemas embarcados, e Regev et al [42] representam com π-calculus alguns processos bioquímicos. O π-calculus tem sido também usado como base para desenvolvimento de vários métodos formais, dentre eles estão Spi calculus de Abadi e Gordon [43] e Ambient calculus de Cardelli e Gordon [44]. Outros calculi originados de π-calculus são: Join-calculus [45], Fusion calculus [46], calculus para cometimento atômico [47] e calculus para longas transações [48].

2.5 HAL-JACK

HAL-JACK [24] é uma ferramenta integrada para especificação, verificação e análise de sistemas expressos em π-calculus. Esta ferramenta utiliza a versão *monadic* π-calculus e emprega a lógica temporal pi-logic para definir propriedades a serem verificadas. A gramática usada pelo HAL-JACK para representar processos em π-calculus possui algumas regras especiais em relação à sintaxe usada por Milner [23], como mostra a Tabela 3.

A primeira diferença está na definição do processo, que deve conter inicialmente a palavra *define* e sempre possuir uma lista de entradas mesmo que seja vazia. Assim, a definição do processo *P* for "*P = termo*" deve ser escrita como "*define P()= termo*", e a definição "*P(x) = termo*" deve ser escrita como "*define P(x) = termo*", onde *termo* representa o segundo membro da definição. Além disso, após todas as definições deve-se escrever "*build principal*", onde *principal* é o nome do processo principal.

Conforme a Tabela 3, o processo nulo no HAL-JACK é representado por *nil* e a ação invisível por *tau*. Em relação aos operadores seqüência, escolha e paralelismo do π-calculus, suas representações são as mesmas usadas por Milner. Não existe o operador replicação no

HAL-JACK. E os nomes restritos ao processo são colocados entre parênteses no início do segundo membro da definição.

Tabela 3: Comparação entre a sintaxe usada por Milner e sintaxe usada pela ferramenta HAL-JACK.

	π-calculus por Milner	**π-calculus no HAL-JACK**
Definição de um processo sem entradas	*P = termo*	*define P() = termo*
Definição de um processo com entrada x	*P(x) = termo*	*define P(x) = termo*
Processo nulo	*0*	*nil*
Ação invisível	τ	*tau*
Operador seqüência	.	.
Operador escolha	+	+
Operador composição paralela	|	|
Operador restrição	*new a P*	*(a)P*
Operador replicação	!*P*	Não há
Envio de mensagem	$\bar{x}\langle y\rangle$	*x!y*
Envio de mensagem (apenas sincronização)	$\bar{x}$	*x!x*
Recebimento de mensagem	*x(y)*	*x?(y)*
Recebimento de mensagem (apenas sincronização)	*x*	*x?(empty)*
Definição de um processo que chama outro processo	*P(x) =* $Q\langle x\rangle$ Onde: a) *Q(y) = termo* é a definição de *Q* e *x* é o valor passado por *P* para *y*. b) Usam-se os símbolos < e > para passar o novo valor *x*.	*define P(x) = Q(x)* Onde: a) *define Q(y) = termo* é a definição de *Q* e *x* é o valor passado por *P* para *y*. b) Usam-se parênteses para passar o novo valor *x*.

Todo recebimento ou envio de mensagem na sintaxe do HAL-JACK deve ser seguido do processo nulo ou da chamada a outro processo. O recebimento de uma mensagem é representado por *x?(y)*, ou seja, o canal *x* seguido do sinal de interrogação e da mensagem recebida *y* entre parênteses. Como a mensagem recebida é uma variável restrita, ela deve ser um nome novo na definição. Para a definição ser válida para o HAL-JACK, o canal *x* deve ser definido por algum dos seguintes modos:

- Entrada da definição: *define P(x) = x?(y).nil*
- Variável restrita no início da definição: *define P() = (x)x?(y).nil*
- Mensagem que acaba de ser recebida: *define P(z) = z?(x).x?(y).nil*

Caso haja uma sincronização entre canais, mas nenhuma mensagem seja recebida, é necessário um artifício para representar tal comportamento no HAL-JACK: uma variável chamada de *empty* é usada pontualmente para indicar o recebimento, por exemplo, *define P(x) = x?(empty).nil*

Denota-se o envio de uma mensagem por *x!y*, quer dizer, o canal *x* seguido do sinal de exclamação e da mensagem enviada *y*. Conforme colocado anteriormente, para a definição ser válida para o HAL-JACK, o canal *x* deve constar na definição. A mensagem enviada *y* também deve fazer parte da definição do processo por algum dos seguintes modos:

- Entrada da definição: *define A(x,y) = x!y.nil*
- Variável restrita no início da definição: *define A(x) = (y)x!y.nil*
- Mensagem que acaba de ser recebida: *define A(x,z) = x?(y).z!y.nil*

Nos exemplos acima de envio de mensagem, o canal *x* foi definido como entrada da definição, mas poderia ser uma variável restrita ou uma mensgaem recém recebida. Na ocorrência de uma sincronização entre canais onde nenhuma mensagem seja enviada, o recurso para representar este caso no HAL-JACK é passar como mensagem enviada o mesmo nome do canal, por exemplo, *define A(x) = x!x.nil*.

A última diferença entre a representação do HAL-JACK e a de Milner, segundo a Tabela 3, é a chamada de um processo *Q* por outro processo *P*. No HAL-JACK as entradas passadas por *P* para *Q* devem ser colocadas entre parênteses, já a notação de Milner usa os sinais < e >.

2.6 Trabalhos correlatos

O uso de métodos formais em conjunto em conjunto com engenharia de software é apresentado por Venkatesh e Liu [12]. Usando as práticas de métodos formais e engenharia de software, Venkatesh e Liu [12] propõem a especificação de uma ferramenta para desenvolver sistemas com alta produtividade e corretos por construção. Em se tratando de sistemas de tempo real, Terriza et al [13] descrevem o mapeamento de UML-RT para CSP+T, com o propósito de suprir a ausência de restrições temporais na UML-RT através da transformação do diagrama de estados e do diagrama de classes da cápsula em definições CSP+T.

Fischer et al [15] propõem a conversão do diagrama de estrutura da UML-RT em CSP [16]. Engels et al [17] descrevem a tradução de diagrama de estados e de estrutura da UML-RT para CSP. Mota et al [18] apresentam o mapeamento de elementos do diagrama de estado, de estrutura e de classe da UML-RT para o método formal Circus, que possui conceitos de CSP e Z. Assano [21] propõe a tradução dos diagramas de estado e de estrutura da UML-RT para CCS.

A Tabela 4 apresenta a comparação entre os trabalhos correlatos. A primeira linha da tabela menciona a álgebra de processos empregada no mapeamento, já a linha “Comunicação síncrona” indica que todos os trabalhos consideram a comunicação entre cápsulas como síncrona.

As linhas “Diagrama de estados”, “Diagrama de estrutura” e “Diagrama de classes” mostram se o trabalho descreve o mapeamento para estes diagramas. Vale ressaltar que, embora em [17] hajam transformações do diagrama de estrutura da UML-RT para CSP, este trabalho sugere uma definição para o conector que é relevante apenas no escopo da validação do conjunto cápsula-conector-cápsula proposto.

Tabela 4: Comparação entre trabalhos correlatos

	Terriza et al [13]	**Fischer et al [15]**	**Engels et al [17]**	**Mota et al [18]**	**Assano [21]**
Álgebra de processos	CSP+T	CSP	CSP	Circus	CCS
Comunicação síncrona	Sim	Sim	Sim	Sim	Sim
Diagrama de estados	Sim	Não	Sim	Sim	Sim
Diagrama de estrutura	Não	Sim	Sim	Sim	Sim
Diagrama de classe	Sim	Não	Não	Sim	Não
Diagrama de estados do protocolo	Não	Não	Sim	Sim	Não
Reaproveitamento da definição da cápsula	Não	Não	Não	Sim	Não
Tipo de cápsula (fixa, opcional ou plug-in)	Fixa	Fixa	Fixa	Fixa	Fixa
Porta conectada	Sim	Sim	Sim	Sim	Sim
Porta não conectada	Não	Não	Não	Não	Não
Cardinalidade de porta e cápsula	Não	Sim	Não	Sim	Não
Condição de guarda	Sim	Não	Não	Sim	Não
Ação de entrada e saída do estado	Não	Não	Não	Sim	Sim
Estado composto ou hierarquizado	Sim	Não	Não	Sim	Não
Ponto de junção histórico	Não	Não	Não	Não	Não
Protótipo	Não	Não	Não	Não	Sim

A linha “Diagrama de estados do protocolo” revela que somente [17] e [18] tratam este tipo de diagrama. Entretanto, [17] menciona a tradução do diagrama de estados do protocolo apenas num exemplo, não apresentando regras gerais de mapeamento.

Supondo que a cápsula-mãe *A* possui duas cápsulas-filhas do tipo *B*. Em CSP ou CCS é necessário escrever duas definições para *B*, sendo uma definição para cada configuração de *B*. Isso significa que não há reaproveitamento da definição da cápsula. Um exemplo disso em

CSP pode ser visto no sistema de sinal de trânsito proposto por Engels et al [17]. A linha "Reaproveitamento da definição da cápsula" da Tabela 4 indica se o reaproveitamento é possível.

A Tabela 4 compara ainda os trabalhos correlatos quanto ao mapeamento de diversos elementos, a saber: portas conectadas e não conectadas, cardinalidade de portas e cápsulas, condição de guarda, ação de entrada e saída do estado, estado composto ou hierarquizado, e ponto de junção histórico.

A última linha da Tabela 4 indica se o trabalho correlato desenvolveu ou não um protótipo para automatizar o mapeamento de UML-RT para a álgebra de processos em questão. Como é possível observar, somente Assano [21] implementou um protótipo. Este protótipo é um plug-in do RoseRT, porém permite pouca reutilização de seus componentes. Além disso, este protótipo não captura informações do gatilho e da ação das transições; ele assume que estas informações estão dispostas no nome das transições do modelo.

2.7 Considerações finais

Neste capítulo foram apresentadas as características de um sistema de tempo real, bem como os elementos da UML-RT usados para modelar tais sistemas. Em seguida, definiram-se métodos formais e álgebras de processo. Destacou-se a álgebra de processos π-calculus criada por Milner para modelar sistemas concorrentes. Foram detalhadas as principais diferenças entre a sintaxe usada por Milner e a sintaxe da gramática do HAL-JACK para escrita de definições π-calculus. Além disso, foram descritos os trabalhos correlatos e uma análise comparativa entre eles.

O próximo capítulo descreve as regras de mapeamento usadas para gerar as definições π-calculus a partir de um modelo de sistema especificado em UML-RT. Vale ressaltar que nos demais capítulos todas as definições π-calculus são escritas segundo a sintaxe do HAL-JACK. O intuito de usar a sintaxe do HAL-JACK nas definições π-calculus geradas a partir do mapeamento proposto é possibilitar que as definições π-calculus sejam submetidas à ferramenta HAL-JACK para verificação formal de propriedades.

Capítulo 3

Mapeamento UML-RT para π-calculus

Este capítulo apresenta as regras de mapeamento para conversão do modelo do sistema especificado em UML-RT para π-calculus. As definições π-calculus geradas utilizam a gramática do HAL-JACK como sintaxe. São descritas as hipóteses assumidas e as restrições adotadas no mapeamento, depois são descritas as regras de mapeamento da cápsula, do diagrama de estados e do diagrama de estrutura.

No mapeamento do diagrama de estados são detalhados estados sem transições de saída, recebimento e envio de mensagem, múltiplas transições de saída, ponto de escolha, reconfiguração de nome e configuração inicial de portas não conectadas. Onde os dois últimos casos são contribuições desse trabalho baseadas no conceito de mobilidade do π-calculus. No mapeamento do diagrama de estrutura é explicada a conexão entre portas de subcápsulas e também a conexão entre portas da cápsula-mãe e da subcápsula. Ainda é abordado como a configuração inicial de portas não conectadas reflete no mapeamento do diagrama de estrutura.

3.1 Hipóteses e restrições

O mapeamento proposto de UML-RT para π-calculus considera as restrições do π-calculus para modelar os elementos da UML-RT bem como as restrições da própria UML-RT. O mapeamento se beneficia do fato da UML-RT assumir alguns contextos; por exemplo, é possível especificar uma cápsula composta por outras cápsulas previamente especificadas. Então o reuso da definição da cápsula pode ser representado por um processo em π-calculus que referencia outras definições de processos.

As principais hipóteses e restrições adotadas para mapeamento UML-RT para π-calculus são:

- As definições π-calculus resultantes do mapeamento seguem a sintaxe proposta pela ferramenta HAL-JACK, pois se deseja que as definições geradas possam ser submetidas ao HAL-JACK para verificação formal de propriedades.
- Uso do *monadic* π-calculus, uma vez que esta é a versão do π-calculus usada pelo HAL-JACK. Assim, o mapeamento proposto expressa somente envio e recebimento de mensagens composta por um nome, o que implica que todos os sinais trocados pelas cápsulas devem conter apenas uma única mensagem.
- Assume-se que somente a comunicação síncrona é permitida. Esta restrição respeita o modelo de comunicação do π-calculus. Contudo, é possível modelar tanto a comunicação assíncrona como a síncrona como explica Palamidessi [40].
- O operador replicação do π-calculus não é contemplado por não fazer parte da gramática do HAL-JACK.
- Somente cápsulas fixas são consideradas no mapeamento, por serem o tipo padrão de cápsulas e não necessitarem de códigos especiais para serem instanciadas pela cápsula-mãe.

- No mapeamento são usados os diagramas de estrutura e de estados da cápsula por englobarem, respectivamente, a estrutura interna e o comportamento da cápsula.
- O diagrama de classe não é considerado, uma vez que este diagrama é usado em caso de definições complexas de tipo de dado, o que está fora escopo do mapeamento. Este diagrama pode ter também informações sobre relacionamentos de cápsula, mas essas informações também estão disponíveis no diagrama de estrutura.
- A cardinalidade das cápsulas não é abordada. Caso o modelo necessite de, por exemplo, duas cápsulas do mesmo tipo, sugere-se usar duas instâncias dessa cápsula ao invés de uma instância com cardinalidade dois.
- O diagrama de estados do protocolo não é tratado, porque se assume que os protocolos usados possuem apenas um sinal de entrada ou um de saída com o mesmo nome. Esta restrição sobre protocolo visa simplificar o mapeamento para π-calculus, cujos elementos básicos são porta e mensagem. Dessa forma, a porta da UML-RT é representada pela porta do π-calculus, enquanto a mensagem contida no sinal do protocolo da UML-RT é representada pela mensagem do π-calculus.
- Não são contemplados estados compostos (ou hierarquizados), embora não seja complexa a representação de estados compostos com estados simples.
- Ações de entrada (*entry action*) e saída (*exit action*) do estado não são mapeadas, porque as principais ações constam nas transições entre estados.
- Condição de guarda do gatilho não é mapeada, uma vez que π-calculus modela comunicação entre componentes e não possui uma forma simplificada para tratamento de dados.
- Pontos de junção históricos não são tratados. Pontos de junção na fronteira dos estados não precisam ser considerados, pois o importante é o mapeamento da transição que os possui.

- São tratadas portas conectadas e não conectadas.
- Não se contempla a cardinalidade de portas.
- Os nomes de portas dentro de uma cápsula devem ser distintos para evitar conflito de nomes nas definições geradas em π-calculus. Como esta restrição é imposta pela própria ferramenta RoseRT usada no desenho do modelo UML-RT, o desenvolvedor não necessita ater-se a este detalhe durante o processo de especificação do modelo.

3.2 Mapeamento da cápsula

Considerando que *A* é um processo em π-calculus e *L* é uma lista de nomes, o termo *A(L)* representa a definição do processo *A* com suas entradas (elementos de *L*). Por exemplo, se *L = {x, y}*, tem-se a definição *A(x,y)*. No mapeamento da UML-RT, as entradas podem ser portas ou nomes de variáveis. As variáveis podem ser mensagens recebidas ou enviadas.

Podem-se definir dois operadores para as listas de entradas: complemento (símbolo –) e união (símbolo ∪). O resultado de *L1 – L2* é a operação complemento, ou seja, *L1* sem os elementos contidos em *L2*. O termo *L1 ∪ L2* representa a concatenação de *L1* com os elementos de *L2* que não estão em *L1*. Todas as operações com listas respeitam a ordem dos elementos. Por exemplo, se *L1 = {a, b, c}* e *L2 = {b, d}*, tem-se *L1 – L2 = {a, c}* e *L1 ∪ L2 = {a, b, c, d}*.

O resultado do mapeamento da UML-RT é a representação em π-calculus da principal cápsula do modelo UML-RT, que deve ser identificada pelo estereótipo *Top Level*.

Na UML-RT, cada cápsula possui seu próprio diagrama de estados e pode conter outras cápsulas no seu diagrama de estrutura. Assim, separa-se a definição da cápsula em duas partes: a definição do seu diagrama de estados e a definição do seu diagrama de estrutura.

Essas duas definições são relacionadas com operador composição paralela do π-calculus, pois o comportamento da cápsula e as suas subcápsulas executam concorrentemente. Vale ressaltar que as subcápsulas de primeiro nível são referenciadas na definição do diagrama de estrutura da cápsula.

Def. 1 (Definição da cápsula) Dada a cápsula *P*, sua definição π-calculus é a composição paralela da definição do seu diagrama de estados ($E_{Estados\_P}$) e do seu diagrama de estrutura ($E_{Estrutura\_P}$):

$$define\ P(LC[P]) = (LR[P])\ (E_{Estados\_P}(LF[P]) \mid E_{Estrutura\_P})$$

São apresentadas abaixo as definições dos termos empregados na definição da cápsula:

- ***LF[P]***: lista de entradas de $E_{Estados\_P}$ no final do mapeamento do diagrama de estados da cápsula *P*.
- ***LC[P]***: lista de entradas da cápsula *P*. Esta lista é dada por:

$$LC[P] = LF[P] - LPTP[P] \cup LRP[P]$$

- ***LPTP[P]***: lista de portas protegidas no diagrama de estrutura da cápsula *P*. Para o exemplo da Figura 14, tem-se *LPTP[A] = {a2}*. Não é possível definir *LPTP[C]* porque não foi provido o diagrama de estrutura de *C*.
- ***LRP[P]***: lista de portas *relay* no diagrama de estrutura da cápsula *P*. Para o exemplo da Figura 14, tem-se *LRP[A] = {a3}*.
- ***LR[P]***: lista de mensagens restritas ao contexto da cápsula *P*. Seja *N* o número de subcápsulas Q_i de *P*:

$$LR[P] = \bigcup_{i}^{N} LCR[P, Q_i] - LC[P]$$

- ***LCR[P, Q]***: lista de entradas usada pela cápsula *P* para referenciar a definição da sua subcápsula *Q*. Para o exemplo da Figura 14, *C* é subcápsula da cápsula *A*, então a definição de *A* usa a definição de *C*. Assumindo que *LC[C] = {x, y}*, tem-se que a definição da subcápsula *C* é *C(x, y)*. Se também for assumido que a cápsula *A* passa o valor *m* para a entrada *x* e o valor *n* para a entrada *y*, tem-se *LCR[A, C] = {m, n}*.

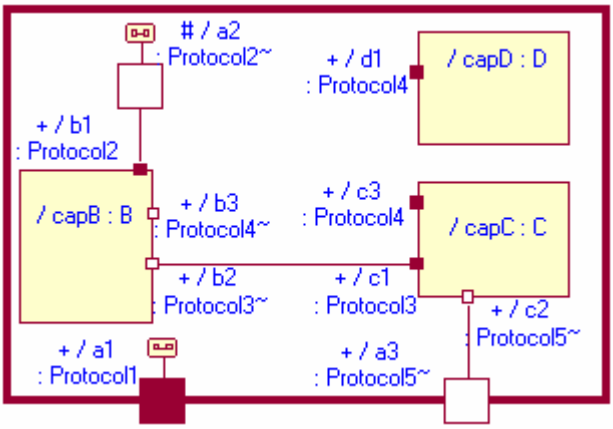


Figura 14: Diagrama de estrutura da cápsula *A*.

A seguir explica-se como obter a definição do diagrama de estados e de estrutura de uma cápsula a fim de compor a definição da cápsula dada pela Def. 1.

3.3 Mapeamento do diagrama de estados

A definição do diagrama de estados da cápsula *P*, $E_{Estados\_P}(LF[P])$, é a definição do seu estado inicial. O estado inicial do diagrama de estados é o estado destino da transição inicial. A definição de cada estado é a composição de suas transições e dos estados destinos dessas transições. Assim, para obter a definição do diagrama de estados, necessita-se analisar todo o diagrama a fim de se determinar as definições dos estados.

O nome do estado na definição π-calculus é dado pela concatenação do nome da cápsula com o nome do próprio estado. Tal concatenação visa permitir que estados de mesmo nome em cápsulas distintas possuam nomes distintos nas definições π-calculus.

O objetivo é computar $E_{Estados\_P}(LF[P])$. No início da análise do diagrama de estados para determinar as definições dos estados, a lista de entrada de $E_{Estados\_P}$ é ***LI[P]***. Durante a análise do diagrama de estados, podem surgir novos nomes nas operações de envio de mensagem. Os novos nomes são adicionados à lista ***LE[P]***. No final da análise do diagrama de estados, é necessário atualizar todas as listas de entradas das definições com os elementos de *LE[P]*.

Abaixo são apresentados os termos que compõem a definição do diagrama de estados:

- ***LF[P]***: lista de entradas de $E_{Estados\_P}$ no **final** do mapeamento do diagrama de estados da cápsula *P*. Esta lista foi mencionada na Def. 1 e é composta pela lista inicial do diagrama de estados mais a lista de novos nomes:

$$LF[P] = LI[P] \cup LE[P]$$

- ***LI[P]***: lista de entradas de $E_{Estados\_P}$ no **início** do mapeamento do diagrama de estados da cápsula *P*. Esta lista é composta por portas conectadas que não são *relay*:

$$LI[P] = LWP[P] - LRP[P]$$

- ***LWP[P]***: lista de portas conectadas no diagrama de estrutura da cápsula *P*. Para o exemplo da Figura 14, tem-se $\{c1, c2\} \subset LWP[C]$.
- ***LRP[P]***: já definida como a lista de portas *relay* no diagrama de estrutura da cápsula *P*.
- ***LE[P]***: lista de novos nomes. Os novos nomes pertencem a operações de envio de mensagem e são acrescentados a lista *LE[P]* durante a análise do diagrama de estados.

- ***LS[P,S]***: lista de entradas da definição do estado *S* pertencente à cápsula *P*. As entradas podem ser portas ou nomes de variáveis. As variáveis podem ser mensagens recebidas ou enviadas.
- ***LSR[P,S1,S2]***: lista de entradas usada pelo estado *S1* para referenciar a definição do estado *S2*, onde *S1* e *S2* são estados no diagrama de estados da cápsula *P*. Assumindo que *LS[P,S2] = {x, y}*, *S1* pode passar o valor *m* para *x* e *n* para *y*. Neste caso, *LSR[P, S1, S2] = {m, n}*.

A Def. 2 explica como obter a definição final do diagrama de estados, que consiste apenas em acrescentar a lista de novos nomes *LE[P]* a todas as listas de entradas presentes na definição inicial do diagrama de estados explicada na Def. 3. Usa-se o termo "*definição do estado*" para indicar que a análise do estado ainda necessita ser realizada de acordo com as regras de mapeamento.

Def. 2 (Definição final do diagrama de estados) Dada uma cápsula *P*. Assumindo que *S1* é o estado inicial. A definição final do diagrama de estados é a definição do primeiro estado considerando a lista de entradas final do diagrama de estados *LF[P]*:

$E_{Estados\_P}(LF[P]) = P\_S1(LF[P])$

$P\_S1(LF[P])$ = *definição do estado*

onde todas as listas de entradas de definições refenciadas a partir do primeiro estado em "*definição do estado*" devem conter também a lista de novos nomes *LE[P]*, ou seja:

$LS[P,S_i] = LS[P,S_i] \cup LE[P]$

$LSR(P,S_i,S_j) = LSR(P,S_i,S_j) \cup LE[P]$, sendo S_i e S_j são estados da cápsula *P* e existe uma transição entre S_i e S_j.

Def. 3 (Definição inicial do diagrama de estados) Dada uma cápsula *P*. Assumindo que *S1* é o estado inicial. A definição inicial do diagrama de estados é a definição do primeiro estado considerando a lista de entradas inicial do diagrama de estados *LI[P]*:

$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$

P_S1(LI[P]) = definição do estado

Um caso especial da definição do diagrama de estados ocorre quando ele não possui estados. Então sua definição π-calculus é o processo *nil*.

Def. 4 (Diagrama de estados sem estados) Dada uma cápsula P. Assumindo que o diagrama de estados de *P* não possui estados, a definição do diagrama de estados é: $E_{Estados\_P}(LI[P]) = nil$.

Nas próximas seções sobre diagrama de estados, o foco é a determinação da definição inicial do diagrama de estados através das definições dos estados. Para obter a definição de um estado é preciso mapear suas transições.

O recebimento de sinal é o que aciona uma transição entre estados, assim pode-se escrever "*p1 receive e1(x1)*", que significa que a porta *p1* recebe o sinal *e1* com a mensagem *x1*. A transição, após acionada, pode executar uma ação "*p2 send e2(x2)*" ou "nome1 = nome2". O termo "*p2 send e2(x2)*" significa o envio pela porta *p2* do sinal *e2* com a mensagem *x2*. O termo "*nome1 = nome2*" representa a reconfiguração de nomes e é usado para indicar qualquer atribuição de valor de uma variável a outra. Para separar o recebimento do sinal na transição da ação executada, usa-se o símbolo /, por exemplo, pode-se escrever "*p1 receive e1(x1) / p2 send e2(x2)*".

Os sinais *e1* e *e2* não são mapeados, conforme mencionado em hipóteses e restrições do mapeamento (Seção 3.1). Os tipos das mensagens *x1* e *x2* também não são tratados, pois π-calculus foca na comunicação entre processos e não possui um modo simples de indicar tipos de dados (string, inteiro, etc.). Assim, as mensagens *x1* e *x2* são nomes de variáveis e não um valor propriamente dito, por exemplo, para enviar a string "teste" pela porta *p2*, as ações seriam: definir o sinal *e2* como portador de mensagem do tipo string, definir a variável string *x2* igual a "teste", e só então enviar o sinal *e2* com a mensagem *x2* pela porta *p2*.

Os termos sugeridos "*p1 receive e1 (x1)*", "*p2 send e2 (x2)*" e "*nome1 = nome2*" visam facilitar o entendimento das regras de mapeamento. Na ferramenta RoseRT existe um modo próprio de configurar informações de transições ao invés de escrever explicitamente esses termos.

Com o objetivo de explicar o mapeamento de estados e transições, as próximas seções discutem os seguintes casos: estado sem transições de saída, recebimento de mensagem, envio de mensagens, múltiplas transições de saída e ponto de escolha. Posteriormente, acrescentam-se casos especiais no mapeamento do diagrama de estados, como a reconfiguração de nomes (seja de atributos simples ou de portas não conectadas) e a configuração inicial de portas não conectadas.

Torna-se necessário lembrar que não são consideradas ações de entrada (*entry action*) e saída (*exit action*) de um estado; o mapeamento trata apenas as ações executadas durante as transições de estados. Além disso, estados compostos (ou hierarquizados) não são considerados no mapeamento.

3.3.1 Estado sem transições de saída

Def. 5 (Estado sem transições de saída) Dado o estado *S* de uma cápsula *P*. Se o estado não possui transições de saída, sua definição é a seguinte: *P_S(LS[P,S]) = nil*.

A Figura 15 representa o diagrama de estados da cápsula *P*. O diagrama de estados possui apenas o estado *S1*, e *S1* não possui transições de saída. Usando Def. 3 e Def. 5, a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = nil$$

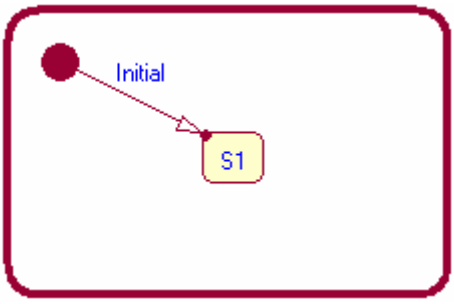


Figura 15: Estado sem transições de saída.

3.3.2 Recebimento de mensagem

Def. 6 (Recebimento de mensagem – caso geral) Dada uma transição do estado *S1* para o estado *S2* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Assumindo o recebimento de mensagem "*p1 receive e1 (x1)*" no gatilho da transição. Assumindo que não há ação na transição. Então a definição de *S1* é a seguinte:

P_S1(LS[P,S1]) = p1?(x1) . P_S2(LSR[P,S1,S2])

P_S2(LS[P,S2]) = definição do estado

onde:

$LSR[P,S1,S2] = LS[P,S1] \cup \{x1\}$

$LS[P,S2] = LS[P,S1] \cup \{x1\}$

A Figura 16 mostra o recebimento da mensagem *x1* pela porta *p1* na transição entre os estados *S1* e *S2*. Logo a definição de *S1* recebe uma mensagem e chama a definição de *S2* passando a nova mensagem recebida. Neste exemplo *S1* é o estado inicial e *S2* não possui transições de saída. Usando Def. 3, Def. 5 e Def. 6, a definição completa do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(x1).\ P\_S2(LI[P] \cup \{x1\})$$

$$P\_S2(LI[P] \cup \{x1\}) = nil$$

Na definição acima, $LS[P,S2] = LI[P] \cup \{x1\}$ é a lista de entradas especificada na definição de *S2*, e $LSR[P,S1,S2] = LI[P] \cup \{x1\}$ é a lista de entradas usada por *S1* para referenciar a definição de *S2*. No caso, $LSR[P,S1,S2] = LS[P,S2]$.

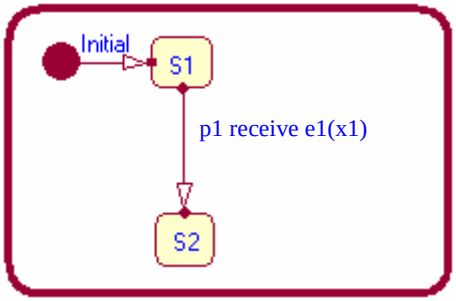


Figura 16: Recebimento de mensagem.

Existem três casos especiais de recebimento de mensagem. O primeiro trata-se de *e1* como sinal de *timeout*. Este caso é detectado pelo tipo da porta *p1* que implementa o

protocolo *Timing*. O protocolo *Timing*, provido pela ferramenta RoseRT, permite que o sistema receba um sinal de timeout em intervalos de tempo regulares ou após um intervalo estabelecido. Este tipo de interação é interno ao sistema e é representado pela ação invisível *tau*. A porta que implementa o protocolo *Timing* não é considerada nas definições π-calculus.

Def. 7 (Recebimento de mensagem – timeout) Dada uma transição de um estado qualquer *S1* para outro estado qualquer *S2* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Assumindo o recebimento de mensagem "*p1 receive e1 (x1)*" no gatilho da transição. Assumindo que *e1* é sinal de *timeout*. Assumindo que não há ação na transição. Então a definição de *S1* é a seguinte:

P_S1(LS[P,S1]) = tau . P_S2(LSR[P,S1,S2])

P_S2(LS[P,S2]) = definição do estado

onde:

LSR[P,S1,S2] = LS[P,S1]

LS[P,S2] = LS[P,S1]

Na Figura 16, assumindo que *e1* é um sinal de *timeout*. Usando Def. 7, a definição do diagrama de estados é escrita como:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = tau\ .\ P\_S2(LI[P])$$

$$P\_S2(LI[P]) = nil$$

O segundo caso especial de recebimento de mensagem é quando há apenas sincronização entre as portas, ou seja, nenhuma mensagem é trocada. Pode-se dizer que o sinal recebido é do tipo void. A gramática do HAL-JACK exige que se tenha sempre uma

mensagem recebida. Como esta gramática foi empregada como sintaxe das definições π-calculus, é usado um artifício sugerido pela equipe desenvolvedora do HAL-JACK [24]. O artifício visa representar a mensagem recebida com a palavra reservada *empty* que não deve ser transmitida aos demais estados.

Def. 8 (Recebimento de mensagem – apenas sincronização) Dada uma transição de um estado qualquer *S1* para outro estado qualquer *S2* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Assumindo o recebimento de mensagem "*p1 receive e1 ()*" no gatilho da transição, onde não há mensagem trocada. Assumindo que não há ação na transição.

Então a definição de *S1* é a seguinte:

P_S1(LS[P,S1]) = p1?(empty) . P_S2(LSR[P,S1,S2])

P_S2(LS[P,S2]) = definição do estado

onde:

LSR[P,S1,S2] = LS[P,S1]

LS[P,S2] = LS[P,S1]

Na Figura 16, assumindo que só há sincronização na porta *p1*. Usando Def. 8, a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(empty) . P\_S2(LI[P])$$

$$P\_S2(LI[P]) = nil$$

O último caso especial de recebimento de mensagem ocorre quando mais de um sinal pode acionar o gatilho da transição. O mapeamento compõe os diferentes termos de

recebimento de mensagem com o operador escolha do π-calculus. Todas as possíveis mensagens recebidas devem ser transmitidas à definição do próximo estado.

Def. 9 (Recebimento de mensagem – múltiplos eventos) Dada uma transição de um estado qualquer *S1* para outro estado qualquer *S2* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Seja *N* o número finito de termos "p_i *receive* e_i *(x)*", $i \in N$, no gatilho da transição. Assumindo que não há ação na transição. Então a definição de *S1* é a seguinte:

$$P\_S1(LS[P,S1]) = \left(\sum_i^N p_i ? x \right) . P\_S2(LSR[P,S1,S2])$$

$$P\_S2(LS[P,S2]) = definição\ do\ estado$$

onde:

$$LSR[P,S1,S2] = LS[P,S1] \cup \{x\}$$

$$LS[P,S2] = LS[P,S1] \cup \{x\}$$

A Figura 16 mostra somente um sinal a ser recebido na transição entre *S1* e *S2*, porém o gatilho pode ser configurado para receber mais de um sinal, por exemplo, "*p1 receive e1(x)*" e "*p2 receive e2(x)*". Vale lembrar que dois sinais podem ser recebidos, mas a mensagem recebida é única, no caso a mensagem é *x*. Usando Def. 9, a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = (p1?(x) + p2?(x)) . P\_S2(LI[P] \cup \{x\})$$

$$P\_S2(LI[P] \cup \{x\}) = nil$$

3.3.3 Envio de mensagem

Def. 10 (Envio de mensagem – caso geral) Dada uma transição do estado *S1* para o estado *S2* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Assumindo o caso geral de recebimento de mensagem "*p1 receive e1 (x1)*" no gatilho da transição. Assumindo o envio de mensagem "*p2 send e2(x2)*" na ação da transição, onde *x2* obedece a uma das seguintes regras:

x2 = x1: a mensagem enviada *x2* é igual a mensagem recebida *x1*; ou

$x2 \in LS[P,S1]$: a mensagem enviada *x2* pertence à lista de entradas de *S1*.

Então a definição de *S1* é a seguinte:

$P\_S1(LS[P,S1]) = p1?(x1) \,.\, p2!x2 \,.\, P\_S2(LSR[P,S1,S2])$

$P\_S2(LS[P,S2]) =$ *definição do estado*

onde:

$LSR[P,S1,S2] = LS[P,S1] \cup \{x1\}$

$LS[P,S2] = LS[P,S1] \cup \{x1\}$

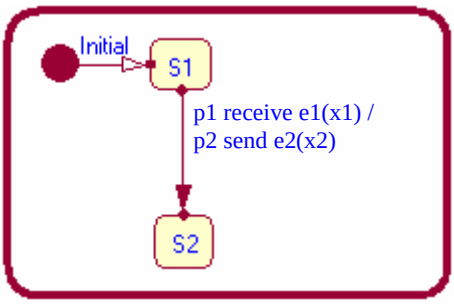


Figura 17: Envio de mensagem.

A Figura 17 apresenta o caso geral de envio de mensagem. O estado *S1* recebe a mensagem *x1* pela porta *p1*, envia a mensagem *x2* pela porta *p2* e usa a definição de *S2*. Neste

exemplo, *S1* é o estado inicial e *S2* não possui transições de saída. Usando Def. 3, Def. 5 e Def. 10, a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(x1) \,.\, p2!x2 \,.\, P\_S2(LI[P] \cup \{x1\})$$

$$P\_S2(LI[P] \cup \{x1\}) = nil$$

Onde: $x2 = x1$ ou $x2 \in LS[P,S1] = LI[P]$

O caso geral de envio de mensagem considera o gatilho da transição como o caso geral de recebimento de mensagem, porém os casos especiais de recebimento de mensagem podem ser aplicados à Def. 10, bastando sempre usar o operador seqüência do π-calculus para compor recebimento e envio de mensagens.

Considerando apenas o envio de mensagem, também existem três casos especiais. O primeiro ocorre quando há apenas sincronização entre as portas, ou seja, não há mensagem enviada. Para se adequar à gramática do HAL-JACK, usa-se o artifício de repetir o nome da porta no local da mensagem enviada. Este artifício é sugerido pela equipe que desenvolveu o HAL-JACK [24].

Def. 11 (Envio de mensagem – apenas sincronização) Dada uma transição do estado *S1* para o estado *S2* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Assumindo o caso geral de recebimento de mensagem "*p1 receive e1 (x1)*" no gatilho da transição. Assumindo o envio de mensagem "*p2 send e2()*" na ação da transição, onde não há mensagem enviada. Então a definição de *S1* é a seguinte:

P_S1(LS[P,S1]) = p1?(x1) . p2! p2 . P_S2(LSR[P,S1,S2])

P_S2(LS[P,S2]) = definição do estado

onde:

$LSR[P,S1,S2] = LS[P,S1] \cup \{x1\}$

$LS[P,S2] = LS[P,S1] \cup \{x1\}$

Na Figura 17, assumindo que há apenas sincronização na porta *p2*, a definição do diagrama de estados usando Def. 11 é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(x1) \,.\, p2!\, p2 \,.\, P\_S2(LI[P] \cup \{x1\})$$

$$P\_S2(LI[P] \cup \{x1\}) = nil$$

O segundo caso especial é quando a mensagem enviada *x2* não está presente na definição de seu estado, ou seja, não obedece às regras do caso geral expresso na Def. 10. Nesta situação, *x2* será acrescentada a uma lista *LE[P]* de novos nomes estabelecidos durante a análise do diagrama de estados da cápsula. No final da análise do diagrama de estados, os elementos de *LE[P]* são adicionados a todas as listas de entradas que fazem parte da definição do diagrama de estados, conforme explicado no início da Seção 3.3. A motivação para o uso da lista *LE[P]* é simplificar a definição de *x2* nas definições, visto que o diagrama de estados pode conter ciclos que dificultam a atualização das lista de entrada dos estados dinamicamente.

Def. 12 (Envio de mensagem – novos nomes durante a análise) Dada uma transição do estado *S1* para o estado *S2* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Assumindo o caso geral de recebimento de mensagem "*p1 receive e1 (x1)*" no gatilho da transição. Assumindo o envio de mensagem "*p2 send e2(x2)*" na ação da transição, onde *x2* não obedece às regras da Def. 10.

Então a definição de *S1* é a seguinte:

P_S1(LS[P,S1]) = p1?(x1) . p2!x2 . P_S2(LSR[P,S1,S2])

P_S2(LS[P,S2]) = definição do estado

onde:

LSR[P,S1,S2] = LS[P,S1] ∪ {x1}

LS[P,S2] = LS[P,S1] ∪ {x1}

LE[P] = x2

Na Figura 17, assumindo que o envio de mensagem obedece à Def. 12, a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(x1) . p2!x2 . P\_S2(LI[P] \cup \{x1\})$$

$$P\_S2(LI[P] \cup \{x1\}) = nil$$

Onde: $LE[P] = \{x2\}$

Para o exemplo acima, a definição final do diagrama de estados considerando os elementos da lista *LE[P]* é:

$$E_{Estados\_P}(LI[P] \cup \{x2\}) = P\_S1(LI[P] \cup \{x2\})$$

$$P\_S1(LI[P] \cup \{x2\}) = p1?(x1) . p2!x2 . P\_S2(LI[P] \cup \{x1,x2\})$$

$$P\_S2(LI[P] \cup \{x1,x2\}) = nil$$

O terceiro caso especial de envio de mensagem ocorre quando há mais de um envio de mensagem na ação executada pela transição. É necessário relacionar os envios sucessivos com o operador seqüência do π-calculus.

Def. 13 (Envio de mensagem – múltiplas operações) Dada uma transição do estado *S1* para o estado *S2* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Assumindo o caso geral de recebimento de mensagem "*p1 receive e1 (x1)*" no gatilho da transição. Assumindo que a ação da transição é composta por um número finito *N* de termos de envio de mensagem "p_i *send* $e_i(x_i)$", $i \in N$, e cada termo segue o caso geral de envio de mensagem. Então a definição de *S1* é a seguinte:

$P\_S1(LS[P,S1]) = p1?(x1) . p_i!x_i . \ldots . p_N!x_N . P\_S2(LSR[P,S1,S2])$

$P\_S2(LS[P,S2]) =$ *definição do estado*

onde:

$LSR[P,S1,S2] = LS[P,S1] \cup \{x1\}$

$LS[P,S2] = LS[P,S1] \cup \{x1\}$

Supondo que haja duas operações de envio de mensagem na transição entre *S1* e *S2* da Figura 17 e que cada operação obedece ao caso geral de envio de mensagem, a definição do diagrama de estados usando Def. 13 é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(x1) . p2!x2 . p3!x3 . P\_S2(LI[P] \cup \{x1\})$$

$$P\_S2(LI[P] \cup \{x1\}) = nil$$

3.3.4 Múltiplas transições de saída

Quando um estado possui mais de uma transição de saída, a definição do estado relaciona as definições das transições com o operador escolha do π-calculus.

Def. 14 (Múltiplas transições de saída) Dado o estado *S1* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Dado um número finito *N* de transições de saída, onde o estado origem é *S1* e o estado destino é S_i, tal que $S_i \quad S1$, $i \in N$. Assumindo que a cada transição seja determinada pelo caso geral de recebimento e de envio de mensagem "p_i *receive* e_i (x_i) / p_j *send* $e_j(x_j)$", onde $j \in M$ e *M* é um número finito. Então a definição de *S1* é a seguinte:

$$P\_S1(LS[P,S1]) = \sum_{i}^{N} (p_i?(x_i) \,.\, p_j!x_j \,.\, P\_S_i(LSR[P,S1,S_i]))$$

$$P\_S_i(LS[P,S_i]) = \text{definição do estado}$$

onde:

$$LSR[P,S1,S_i] = LS[P,S1] \cup \{x_i\}$$

$$LS[P,S_i] = LS[P,S1] \cup \{x_i\}$$

A Figura 18 mostra transições do estado *S1* para estados distintos *S2* e *S3*. Considerando que as transições seguem os casos gerais de recebimento e envio de mensagem. Usando Def. 3, Def. 5 e Def. 14, a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p2?(x2) \,.\, p4!x4 \,.\, P\_S2(LI[P] \cup \{x2\}) + p3?(x3) \,.\, p5!x5 \,.\, P\_S3(LI[P] \cup \{x3\})$$

$$P\_S2(LI[P] \cup \{x2\}) = nil$$

$$P\_S3(LI[P] \cup \{x3\}) = nil$$

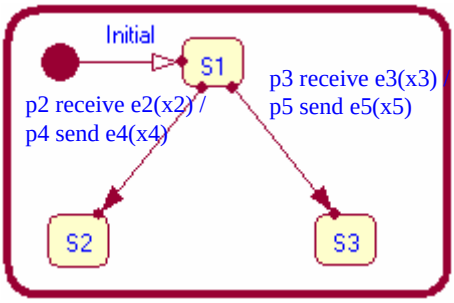


Figura 18: Múltiplas transições de saída.

3.3.5 Ponto de escolha

O mapeamento do ponto de escolha para π-calculus utiliza o operador escolha para relacionar as definições das transições que partem do ponto de escolha, abstraindo a lógica embutida no ponto de escolha que indica qual transição deve ser acionada. Embora haja uma limitação em considerar a escolha determinística do ponto de escolha como a escolha não determinística do π-calculus, este direcionamento foi dado porque que π-calculus modela apenas a comunicação de processos e não o processamento de dados.

Def. 15 (Ponto de escolha) Dados os estados *S1*, *S2* e *S3* e o ponto de escolha *CP* no diagrama de estados da cápsula *P*. Dado *LS[P,S1]*. Dada uma transição do estado *S1* ao ponto de escolha *CP*. Dadas as seguintes transições de saída a partir de *CP*:

a) Transição de *CP* ao estado *S2* acionada quando a condição de CP é satisfeita;

b) Transição de *CP* ao estado *S3* acionada quando a condição de CP não é satisfeita.

Assumindo que a transição de *S1* a *CP* é determinada pelo caso geral de recebimento e envio de mensagem "p_a *receive* e_a (x_a) / p_b *send* $e_b(x_b)$". Assumindo que as transições de *CP* a *S2* e de *CP* a *S3* são definidas pelo caso geral de envio de mensagem, respectivamente "p_c *send* $e_c(x_c)$" e "p_d *send* $e_d(x_d)$", onde *a*, *b*, *c* e *d* são números naturais definidos.

Então a definição de *S1* é a seguinte:

$P\_S1(LS[P,S1]) = p_a?(x_a) \,.\, p_b!x_b \,.\, P\_CP(LSR[P,S1,CP]))$

$P\_CP(LS[P,CP]) = p_c!x_c \,.\, P\_S2(LSR[P,CP,S2])) + p_d!x_d \,.\, P\_S3(LSR[P,CP,S3]))$

$P\_S2(LS[P,S2]) =$ *definição do estado*

$P\_S3(LS[P,S3]) =$ *definição do estado*

onde:

$LSR[P,S1,CP] = LS[P,S1] \cup \{x_a\}$

$LS[P,CP] = LS[P,S1] \cup \{x_a\}$

$LSR[P,CP,S2] = LS[P,CP]$ e $LS[P,S2] = LS[P,CP]$

$LSR[P,CP,S3] = LS[P,CP]$ e $LS[P,S3] = LS[P,CP]$

A Figura 19 apresenta o estado *S1* que possui uma transição para o ponto de escolha *CP*. A partir de *CP* existem duas transições sem ação especificada, dessa forma a definição de *CP* usa diretamente as definições de *S2* e *S3*. Usando Def. 3, Def. 5 e Def. 15 , a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(x1) \,.\, p2!x2 \,.\, P\_CP(LI[P] \cup \{x1\})$$

$$P\_CP(LI[P] \cup \{x1\}) = P\_S2(LI[P] \cup \{x1\}) + P\_S3(LI[P] \cup \{x1\})$$

$$P\_S2(LI[P] \cup \{x1\}) = nil$$

$$P\_S3(LI[P] \cup \{x1\}) = nil$$

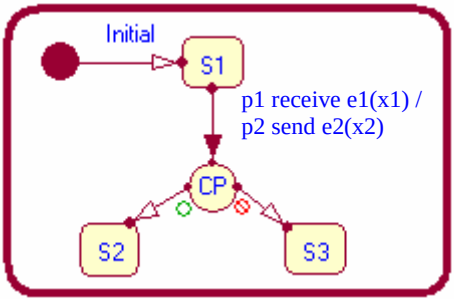


Figura 19: Ponto de escolha.

As transições de saída de *CP* podem ter ação de envio de mensagem. Na Figura 19 a transição de *CP* para *S2* pode ser "*p3 send e3(x3)*", e a transição de *CP* para *S3* pode ser "*p4 send e4(x4)*". Neste caso, a definição do diagrama de estados segundo Def. 15 é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(x1) \,.\, p2!x2 \,.\, P\_CP(LI[P] \cup \{x1\})$$

$$P\_CP(LI[P] \cup \{x1\}) = p3!x3 \,.\, P\_S2\,(LI[P] \cup \{x1\}) + p4!x4 \,.\, P\_S3\,(LI[P] \cup \{x1\})$$

$$P\_S2(LI[P] \cup \{x1\}) = nil$$

$$P\_S3(LI[P] \cup \{x1\}) = nil$$

3.3.6 Reconfiguração de nome

No RoseRT, conforme comentado na fundamentação teórica, uma porta não conectada pode ser definida como SAP (*Service Access Point*) ou SPP (*Service Provisioning Point*) e receber um nome pelo qual será identificada. Chama-se este caso de reconfiguração de porta, ou mais genericamente, de reconfiguração de nomes. O termo reconfiguração de nome no presente trabalho é usado para indicar qualquer atribuição de valor de uma variável a outra, e é indicado no modelo UML-RT pelo código "*nome1 = nome2*" na ação da transição entre estados. O código de reconfiguração de nomes deve ser escrito como ação de uma transição.

Def. 16 (Reconfiguração de nome) Dada uma transição de *S1* e *S2*, onde *S1* e *S2* são estados da cápsula *P*. Dado *LS[P,S1]* e um número fixo *k*. Assumindo que o gatilho da transição é determinado pelo caso geral de recebimento de mensagem "p_k *receive* $e_k(x_k)$. Assumindo que a ação da transição é a reconfiguração de nome "$y = x_k$". Então a definição de *S1* é a seguinte:

$P\_S1(LS[P,S1]) = p_k?(x_k) \,.\, P\_S2(LSR[P,S1,S2])$

$P\_S2(LS[P,S2]) =$ *definição do estado*

onde:

$LSR[P,S1,S2] = LS[P,S1] \cup \{x_k\}$

$LS[P,S2] = LS[P,S1] \cup \{y\}$

A Figura 20 mostra um caso simples de reconfiguração de nomes. A porta *p1* recebe a mensagem *x1*, e o atributo *x2* recebe o valor de *x1*. O atributo *x2* pode denotar tanto uma porta como uma variável qualquer do modelo. O nome *x2* é o que será usado para montar a definição do próximo estado *S2*, mas o nome *x1* é o nome usado por *S1* na chamada da definição de *S2*. Usando Def. 3, Def. 5 e Def. 16, a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(x1) \,.\, P\_S2(LI[P] \cup \{x1\})$$

$$P\_S2(LI[P] \cup \{x2\}) = nil$$

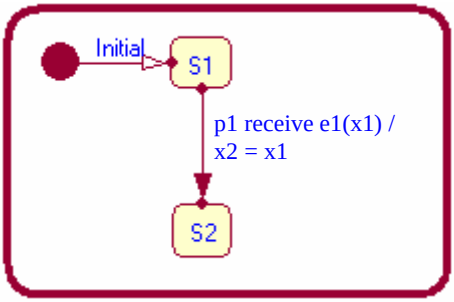


Figura 20: Reconfiguração de nome.

A fim de simplificar a manipulação das entradas dos estados, o presente trabalho considera que um estado possui uma lista única de entradas, não importando a rota (seqüência de transições) usada para chegar a tal estado. É necessário então o uso de reconfiguração de nome para garantir a lista única de entradas do estado. A idéia é que transições, em diferentes rotas, passem o valor da mensagem recebida a um terceiro nome; e este novo nome seja a entrada usada pelo estado destino das rotas.

Na Figura 21, *q* é o nome usado pelo estado *S3*. A transição entre *S1* e *S3* recebe a mensagem *x3* pela porta *p3* e reconfigura *q* com o valor de *x3*. Já transição entre *S2* e *S3* recebe a mensagem *x2* pela porta *p2* e reconfigura *q* com o valor de *x2*. A transição entre *S1* e

S2 é acionada quando há sincronização na porta *p1*. Usando Def. 3, Def. 5, Def. 8 e Def. 16, a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S1(LI[P])$$

$$P\_S1(LI[P]) = p1?(empty) \,.\, S2(LI[P]) + \; p3?(x3) \,.\, S3(LI[P] \cup \{x3\})$$

$$S2(LI[P]) = p2?(x2) \,.\, S3(LI[P] \cup \{x2\})$$

$$S3(LI[P] \cup \{q\}) = nil$$

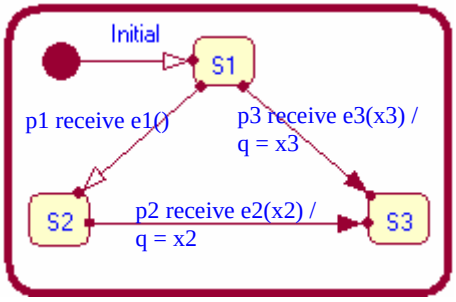


Figura 21: Múltiplas rotas.

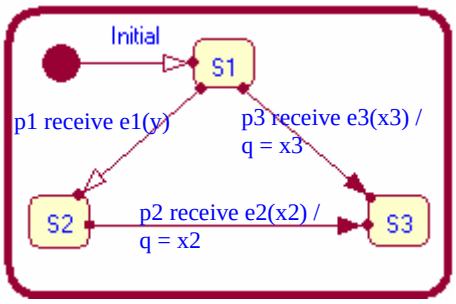


Figura 22: Múltiplas rotas fora do escopo do trabalho.

É importante explicar melhor porque na Figura 21 a transição entre *S1* e *S2* não recebe mensagem. Caso isso ocorresse, por exemplo, se a transição entre *S1* e *S2* recebesse a mensagem *y* como na Figura 22, *S3* teria diferentes listas de entradas dependendo da rota entre o estado inicial e ele mesmo. Segundo a rota *S1*→ *S3*, a lista de entradas de *S3* seria formada apenas por *q*. Entretanto, segundo a rota *S1*→ *S2*→ *S3*, a lista de entradas de *S3* seria composta por *y* e *q*. Como explicado, o presente trabalho considera que um estado possui um

conjunto único de entradas, independente da rota que leva a este estado, assim o exemplo da Figura 22 não faz parte o escopo o presente trabalho.

3.3.7 Configuração inicial de portas não conectadas

A reconfiguração de nomes permite a alteração dinâmica de nomes (portas e atributos) no decorrer do diagrama de estados. Torna-se necessário uma forma de ter portas não conectadas na lista de entradas da definição da cápsula, com o intuito da cápsula-mãe poder iniciar sua cápsula-filha passando diretamente os valores desejados para as portas não conectadas da filha.

A fim de identificar as mensagens de iniciação de portas não conectadas, foi definido que toda configuração desse tipo deve ser feita através de uma mensagem enviada por uma porta que implementa o protocolo *ConfigProtocol*. Este protocolo é simples e possui apenas um sinal de entrada chamado *signal*.

Para realizar a configuração inicial de uma porta não conectada, a cápsula-mãe possui uma porta protegida que implementa o protocolo *ConfigProtocol*. Esta porta é conectada com uma porta pública da cápsula-filha que também implementa *ConfigProtocol*. O código de envio de mensagem para iniciação de portas não conectadas da cápsula-filha está contido na transição inicial do diagrama de estados da cápsula-mãe. O valor enviado pela cápsula-mãe é usado posteriormente na definição do seu diagrama de estrutura.

As primeiras transições do diagrama de estados da cápsula-filha são reservadas para receber as mensagens de configuração de portas não conectadas. Como se trata de uma reconfiguração de porta, essas transições devem conter o código de reconfiguração de nomes "*nome1 = nome2*". O atributo *nome1* é a porta que recebe a configuração inicial, ela pertence ao conjunto ***LUWPC[P]***. O *nome2* é o valor recebido da cápsula-mãe que inicia *nome1*.

Esta abordagem gera duas mudanças no mapeamento do diagrama de estados da cápsula-filha. A primeira mudança é a definição do estado inicial, que passa ser o primeiro estado após o recebimento de todas as mensagens de iniciação de portas não conectadas. A segunda mudança está na lista de entradas *LI[P]* (expressa na Def. 3) do diagrama de estados, que passa a considerar também as portas não conectadas pertencentes a *LUWPC[P]*.

Def. 17 (Configuração inicial de portas não conectadas) Dada uma cápsula-mãe *A*, que contém a cápsula-filha *P* em seu diagrama de estrutura. Dados os estados S_i de *P*, $i \in N$, onde *N* é o número de estados de *P*. Dadas as transições $S_1 \rightarrow S_{i+1} \rightarrow \ldots \rightarrow S_{k-1} \rightarrow S_k$, onde $k \in N$. Assumindo que cada transição até S_k é determinada pelo caso geral de recebimento e pela reconfiguração de nome "p_j *receive* $e_j(x_j)$ / $p_n = x_j$", onde:

a) *j* e *n* são números naturais definidos;

b) A porta p_j é a porta pública de *P* que implementa o protocolo *ConfigProtocol*;

c) A porta p_n é a porta não conectada de *P* que recebe a configuração inicial através da mensagem x_j.

Assumindo que a ação da transição inicial de *A* é determinada pela seqüência de casos gerais de envio de mensagem "q_j *send* $e_j(y_j)$", onde:

a) A porta q_j é a porta protegida de *A*. Ela implementa o protocolo *ConfigProtocol* e está conectada a p_j.

b) A mensagem y_j é o valor enviado por *A* para configurar cada porta não conectada p_n de *P*.

Logo:

a) A lista de portas não conectadas da cápsula-filha *P* que recebem configuração inicial é: $LUWPC[P] = \sum\{p_n\}$

b) O estado inicial da cápsula-filha *P* é S_k.

c) A lista de entradas do diagrama de estados da cápsula-filha *P* é:

$$LI[P] = LWP[P] - LRP[P] \cup LUWPC[P]$$

Onde: *LWP[P]* é a lista de portas conectadas e *LRP[P]* é a lista de portas *relay* como explicado na Def. 3.

d) O diagrama de estrutura da cápsula-mãe *A* usa $\sum\{y_j\}$ para referenciar a definição da cápsula-filha *P*.

e) As portas que implementam o protocolo *ConfigProtocol* são auxiliares, então $p_i \notin LC[P]$ e $q_i \notin LC[A]$.

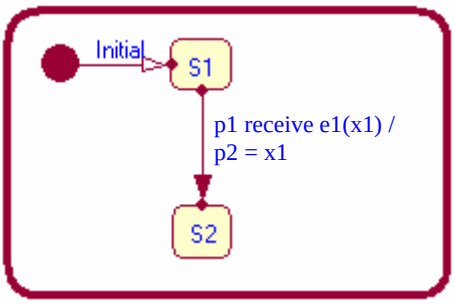


Figura 23: Configuração inicial de porta não conectada.

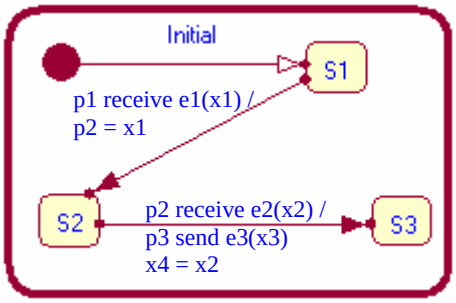


Figura 24: Configuração inicial de porta não conectada seguida por outros códigos.

Na Figura 23, supondo que *p1* implementa o protocolo *ConfigProtocol*, o estado inicial é *S2*. A porta *p2* é um elemento de *LUWPC[P]*. Usando Def. 3, Def. 5 e Def. 17, a definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S2(LI[P])$$

$$P\_S2(LI[P]) = nil$$

Onde: $LUWPC[P] = \{p2\}$

A Figura 24 une os conceitos de configuração inicial de porta não conectada, com recebimento e envio de mensagem, e ainda com reconfiguração simples de nome. Supondo que a porta *p1* implementa *ConfigProtocol* e que *p2* é a porta não conectada a ser iniciada, logo o estado inicial é *S2*. A transição entre *S2* e *S3* recebe a mensagem *x2*, envia a mensagem *x3* e renomeia *x4* para *x2*. A definição do diagrama de estados é:

$$E_{Estados\_P}(LI[P]) = P\_S2(LI[P])$$

$$P\_S2(LI[P]) = p2?(x2) \,.\, p3!x3 \,.\, P\_S3(LI[P] \cup \{x2\})$$

$$P\_S3(LI[P] \cup \{x4\}) = nil$$

Onde: $LUWPC[P] = \{p2\}$

3.4 Mapeamento do diagrama de estruturas

A definição do diagrama de estrutura não possui entradas e é a composição paralela entre as definições de suas subcápsulas. Se não há subcápsulas no diagrama de estrutura de uma dada cápsula, a definição do diagrama de estrutura é o processo nulo.

Vale lembrar que a definição da subcápsula é a sua definição final (conforme Def. 1), considerando tanto a análise do seu diagrama de estados quanto do seu diagrama de estrutura. Assim a lista de entradas da subcápsula P_i é $\boldsymbol{LC[P_i]}$. Quando a cápsula-mãe *A* referencia a definição da cápsula-filha P_i, ela usa a lista de entradas renomeadas ***LCR[A,P]*** que é determinada de acordo com a conexão entre as portas.

Usa-se o termo "*definição da cápsula*" para indicar que a análise da cápsula ainda necessita ser realizada de acordo com as regras de mapeamento.

Def. 18 (Definição do diagrama de estrutura) Dada uma cápsula-mãe *A* e suas subcápsulas P_i, $i \in N$, onde *N* é o número de subcápsulas no diagrama de estrutura de *A*. Dada cada lista de entradas das subcápsulas $LC[P_i]$, e cada lista de entradas $LCR[A,P_i]$ usadas por *A* para referenciar as definições das subcápsulas. Assumindo que as subcápsulas P_i podem conter outras subcápsulas. A definição do diagrama de estrutura de *A* é:

$$E_{Estrutura\_A}() = P_1(LCR[A,P_1]) \mid \ldots \mid P_N(LCR[A,P_N])$$

onde:

$$P_i(LC[P_i]) = \textit{definição da cápsula } P_i$$

Def. 19 (Diagrama de estrutura sem subcápsulas) Dada uma cápsula *A*. Assumindo que o diagrama de estrutura de *A* não possui subcápsulas, então: $E_{Estrutura\_A}() = nil$

Ao considerar portas conectadas, têm-se três tipos de conexão: conexão entre uma porta protegida da cápsula-mãe e uma porta pública da subcápsula; conexão entre uma porta do tipo *relay* da cápsula-mãe e uma porta pública da subcápsula; e conexão entre portas das subcápsulas. Para cada tipo de conexão, há um modo de definir a lista de entradas *LCR[A,P]* que a cápsula-mãe *A* usa ao referenciar a definição da subcápsula *P*.

Def. 20 (Conexão protegida-pública) Dada uma cápsula-mãe *A* e sua subcápsula *P*. Assumindo que a porta protegida *a* de *A* está conectada à porta pública *p* de *P*. Então *p* deve ser renomeado para *a*: $p \in LC[P]$ e $a \in LCR[A,P]$.

Def. 21 (Conexão relay-pública) Dada uma cápsula-mãe *A* e sua subcápsula *P*. Assumindo que a porta *relay a* de *A* está conectada à porta pública *p* de *P*. Então *p* deve ser renomeado para *a*: $p \in LC[P]$ e $a \in LCR[A,P]$.

Def. 22 (Conexão entre subcápsulas) Dada uma cápsula-mãe *A* e suas subcápsulas *P* e *Q*. Assumindo que a porta pública *p* de P está conectada à porta pública *q* de Q. Então um novo nome *x* é gerado dinamicamente para renomear as portas *p* e *q*: $p \in LC[P]$, $q \in LC[Q]$, $x \in LCR[A,P]$ e $x \in LCR[A,Q]$

A Figura 25 mostra o diagrama de estrutura da cápsula *A*, onde *a* é porta *relay* e *b* é porta protegida. A subcápsula *Q* possui as portas públicas *c* e *d*, e a subcápsula *R* possui as portas públicas *e* e *f*. De acordo com Def. 20, *e* é renomeada para *b*. Usando Def. 21, *c* é renomeado para *a*. Segundo Def. 22, um nome é gerado dinamicamente, no caso *p1*, para renomear as portas *d* e *f*. Usando Def. 18, a definição do diagrama de estrutura de *A* é:

$$E_{Estrutura\_A}() = Q(a, p1) \mid R(b, p1)$$

Onde:

$$Q(c, d) = definição\ da\ cápsula$$

$$R(e, f) = definição\ da\ cápsula$$

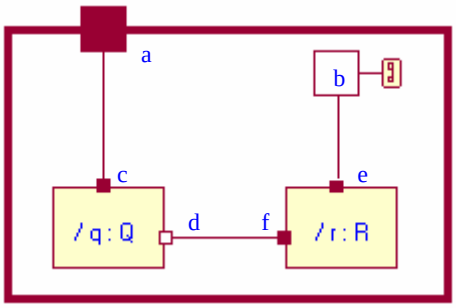


Figura 25: Diagrama de estruturas de *A* com portas conectadas.

Em relação às portas não conectadas, somente as portas não conectadas com configuração inicial *LUWPC[P]* fazem parte da lista de entradas da subcápsula, pois *LUWPC[P]* $\subset$ *LC[P]*. Logo, somente as portas pertencentes à *LUWPC[P]* são renomeadas pela sua cápsula-mãe *A* em *LCR[A,P]*. É importante salientar que as portas que implementam o protocolo *ConfigProtocol* são auxiliares e não aparecem nas definições π-calculus.

Def. 23 (Iniciação de porta não conectada) Considerando todas as informações contidas em Def. 17 sobre configuração inicial de portas não conectadas, então cada p_n é renomeado para seu respectivo y_i: $\sum\{p_n\} \subset$ *LC[P]* e $\sum\{y_j\} \subset$ *LCR[A,P]*

A Figura 26 apresenta o diagrama de estruturas de *A*. A porta protegida *a* implementa o protocolo *ConfigProtocol* e está conectada com a porta *b* da subcápsula *T*. As portas *c* e *d* são não conectadas e pertencem à subcápsula T. Supondo que *A* envia pela porta *a* a mensagem *x*, que é usada na iniciação da porta *c*, e ainda que a porta *d* não recebe iniciação. Usando Def. 23, a definição do diagrama de estrutura de *A* é:

$$E_{Estrutura\_A}() = T(x)$$

Onde: *T(c) = definição da cápsula*

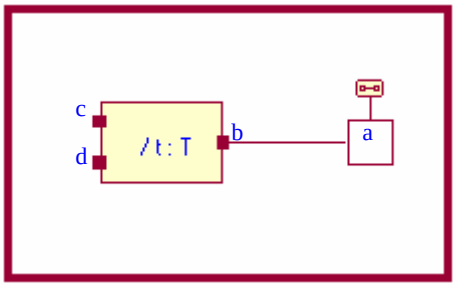


Figura 26: Diagrama de estruturas de *A* considerando portas conectadas e não conectadas.

3.5 Considerações finais

Neste capítulo foram apresentadas as regras de mapeamento dos elementos de comunicação da UML-RT para π-calculus. Destacou-se o mapeamento da cápsula, do diagrama de estados e do diagrama de estrutura. Vale ressaltar que as definições π-calculus utilizam a sintaxe do HAL-JACK.

O próximo capítulo descreve o protótipo desenvolvido para gerar definição π-calculus diretamente do modelo do sistema em UML-RT especificado na ferramenta RoseRT.

Capítulo 4

Protótipo de sistema de mapeamento

Este capítulo apresenta o protótipo de sistema desenvolvido para capturar as informações do modelo UML-RT a partir da ferramenta RoseRT e gerar as definições π-calculus para este modelo. O protótipo segue as regras de mapeamento anteriormente propostas para a tradução de UML-RT para π-calculus. A arquitetura básica do protótipo é descrita e, em seguida, são detalhados os componentes do protótipo.

4.1 Arquitetura do protótipo

O protótipo desenvolvido tem por objetivo gerar as definições π-calculus diretamente do modelo UML-RT especificado na ferramenta RoseRT, utilizando as regras de mapeamento de UML-RT para π-calculus anteriormente apresentadas. Dois módulos compõem o protótipo: um componente VB (Visual Basic) [49] e um componente Java [50].

No primeiro componente foi utilizado VB, porque VB é a linguagem sugerida pela ferramenta RoseRT para desenvolver seus plug-ins. O componente VB (primeira engrenagem da Figura 27) extrai informações dos diagramas de estados e de estrutura das cápsulas que compõem o modelo em UML-RT no RoseRT e as disponibiliza num arquivo em formato

XML.. Em seguida, o componente VB executa o componente Java (segunda engrenagem da Figura 27), que processa os dados do XML e gera as definições π-calculus do modelo.

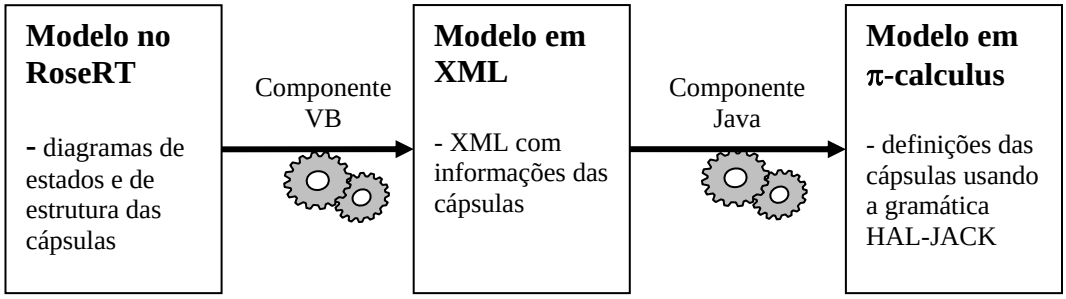


Figura 27: Arquitetura do protótipo.

O XML usado na arquitetura do protótipo visa diminuir o acoplamento entre os componentes em VB e em Java, provendo informações do modelo em um único ponto. O schema (Anexo A) que define a estrutura do XML foi elaborado no presente trabalho e representa um subconjunto de elementos da UML-RT. Gopinath [51] propõe um schema mais completo para representar todos os elementos da UML-RT, que não foi usado no presente trabalho por ter sido estabelecido após o desenvolvimento do protótipo.

4.2 Componente VB

A ferramenta RoseRT disponibiliza instruções sobre como construir plug-ins em VB para ela no tópico *Extensibility Interface Reference* (RRTEI) de seu tutorial. Com base nessas instruções, foi desenvolvido o componente VB. A Figura 28 mostra o acesso ao componente VB através do menu *Tools > Pi-Calculus Add-in* do RoseRT.

O componente VB foi construído no ambiente de desenvolvimento *Microsoft Visual Basic 6.0* utilizando as bibliotecas *Microsoft XML v6.0* e *Microsoft VBScript Regular*

Expressions 5.5. A primeira biblioteca é usada na montagem do XML e a segunda para manipular, com auxílio de expressões regulares, os códigos pertencentes às ações das transições nos diagramas de estado.

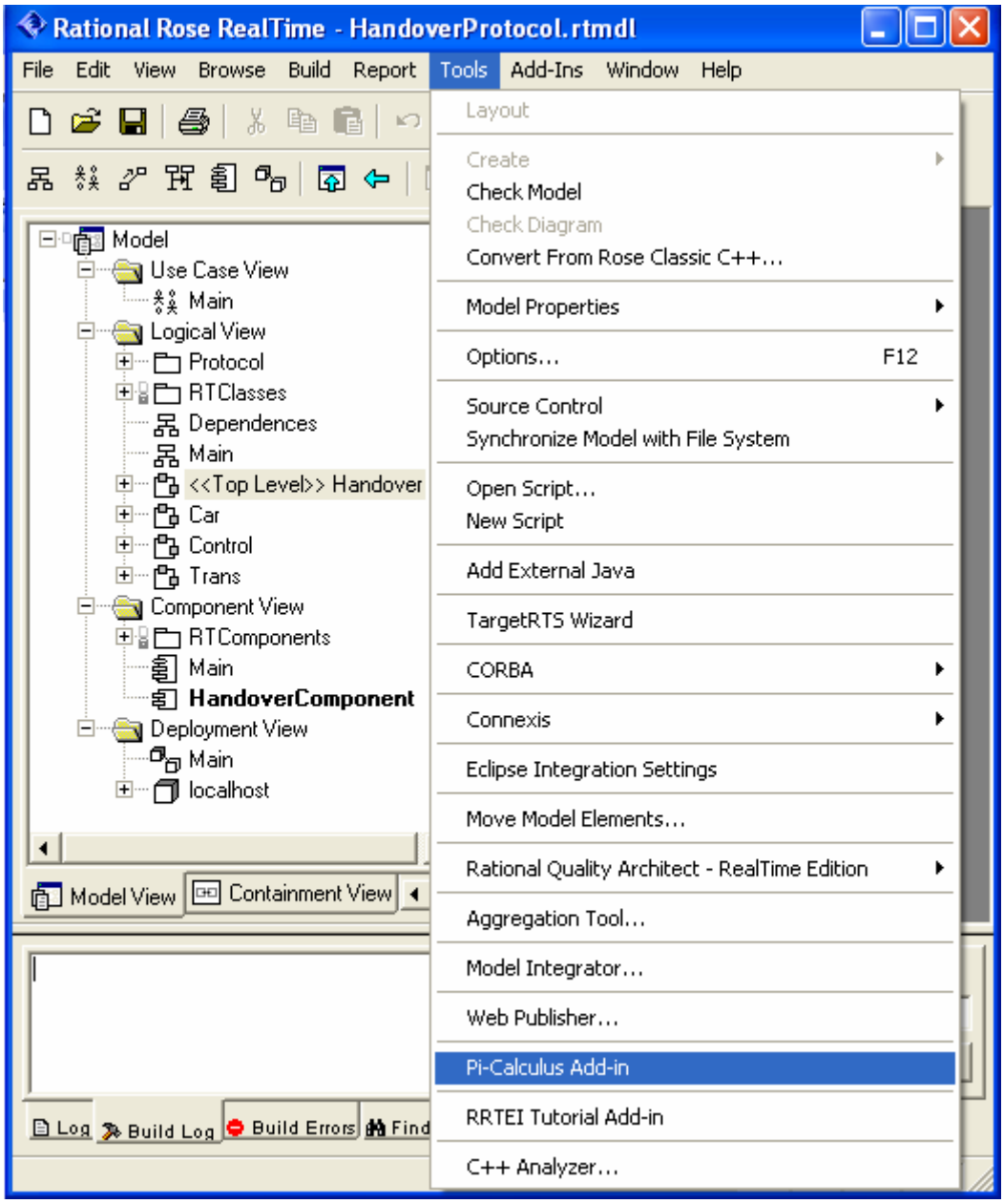


Figura 28: Menu no RoseRT para acessar o componente VB.

Nos exemplos apresentados até este ponto do trabalho, todas as transições continham informações em seu rótulo sobre seu gatilho (recebimento de mensagem) e ação executada (envio de mensagem ou reconfiguração de nomes). Na ferramenta RoseRT, somente o nome

da transição é apresentado em seu rótulo; enquanto as demais informações são configuradas e mantidas em telas especiais. O componente VB está apto a extrair essas informações estabelecidas na transição.

No RoseRT, para auxiliar a definição de recebimento de mensagem, envio de mensagem e reconfiguração de nomes, usam-se códigos na ação das transições do diagrama de estado. Para o componente VB é fundamental que os códigos de recebimento e envio de mensagem utilizem atributos como mensagem e não o valor real da mensagem. O sufixo *Att* deve constar em todo atributo para que o componente VB processe corretamente o código das transições. Este sufixo foi estabelecido por este trabalho para o desenvolvimento do protótipo, porque no RoseRT um atributo do diagrama de estados da cápsula não pode ter o mesmo nome de uma porta da cápsula. O sufixo *Att* não é transmitido ao modelo em XML nem às definições π-calculus.

Em relação ao termo de recebimento de mensagem "*p1 receive e1(x1)*", no RoseRT as informações do gatilho (porta *p1* e sinal *e1*) são configuradas internamente. Enquanto a mensagem *x1* é escrita com o código "*x1Att = *rtdata;*" na ação da transição, onde *x1Att* é um atributo do diagrama de estados e **rtdata* é a forma de recuperar a mensagem recebida. A Figura 29 exibe uma parte do modelo em XML com dados da transição representando o recebimento de mensagem.

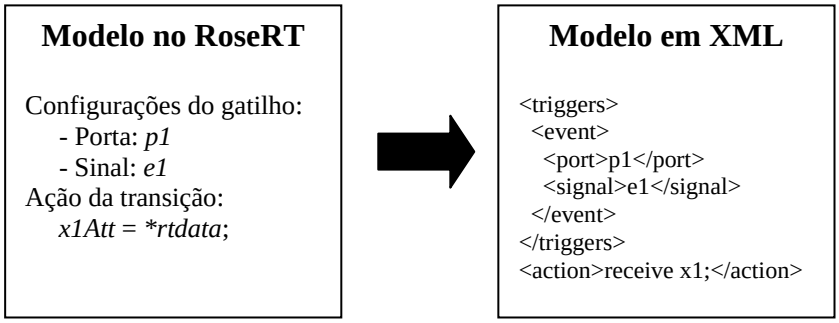


Figura 29: Representação de recebimento de mensagem no RoseRT e no XML.

O termo de envio de mensagem "*p2 send e2(x2)*" é escrito no RoseRT com o código "*p2.e2(x2Att).send();*" na ação da transição, onde *p2* é a porta que envia o sinal *e2* com a mensagem *x2Att*. A Figura 30 mostra um trecho do modelo em XML com ação da transição representando o envio de mensagem.

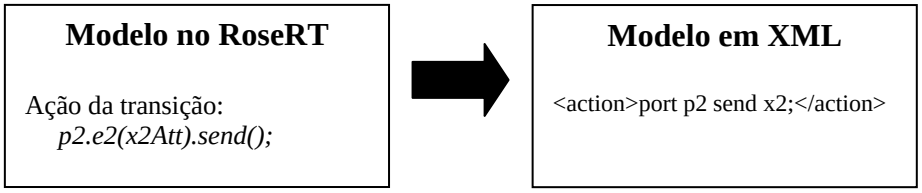


Figura 30: Representação de envio de mensagem no RoseRT e no XML

Considerando o uso do sufixo *Att*, o termo de reconfiguração de nome "*nome1 = nome2*" deve ser escrito no RoseRT como "*nome1Att = nome2Att;*". O código no RoseRT para configurar porta não conectada provedora é "*p.registerSPP(nomeAtt);*" e para porta assinante é "*p.registerSAP(nomeAtt);*", onde *p* é a porta e *nomeAtt* é o atributo com nome de registro da porta. Entretanto é preciso indicar claramente para o protótipo que está sendo feita a reconfiguração de nome. Para isso, tanto para porta SAP quanto para porta SPP, basta usar o código "*pAtt = nomeAtt;*" que é um caso particular de "*nome1Att = nome2Att;*". O protótipo não lê os códigos de configuração de portas SAP e SPP, ele lê somente o código de reconfiguração de nome.

4.3 Componente Java

A versão do Java usada para a construção do componente foi a 5.0. O ambiente de desenvolvimento foi Eclipse SDK 3.1.2 [52] com o plug-in *Omondo EclipseUML* [53] para geração dos diagramas de classe e seqüência.

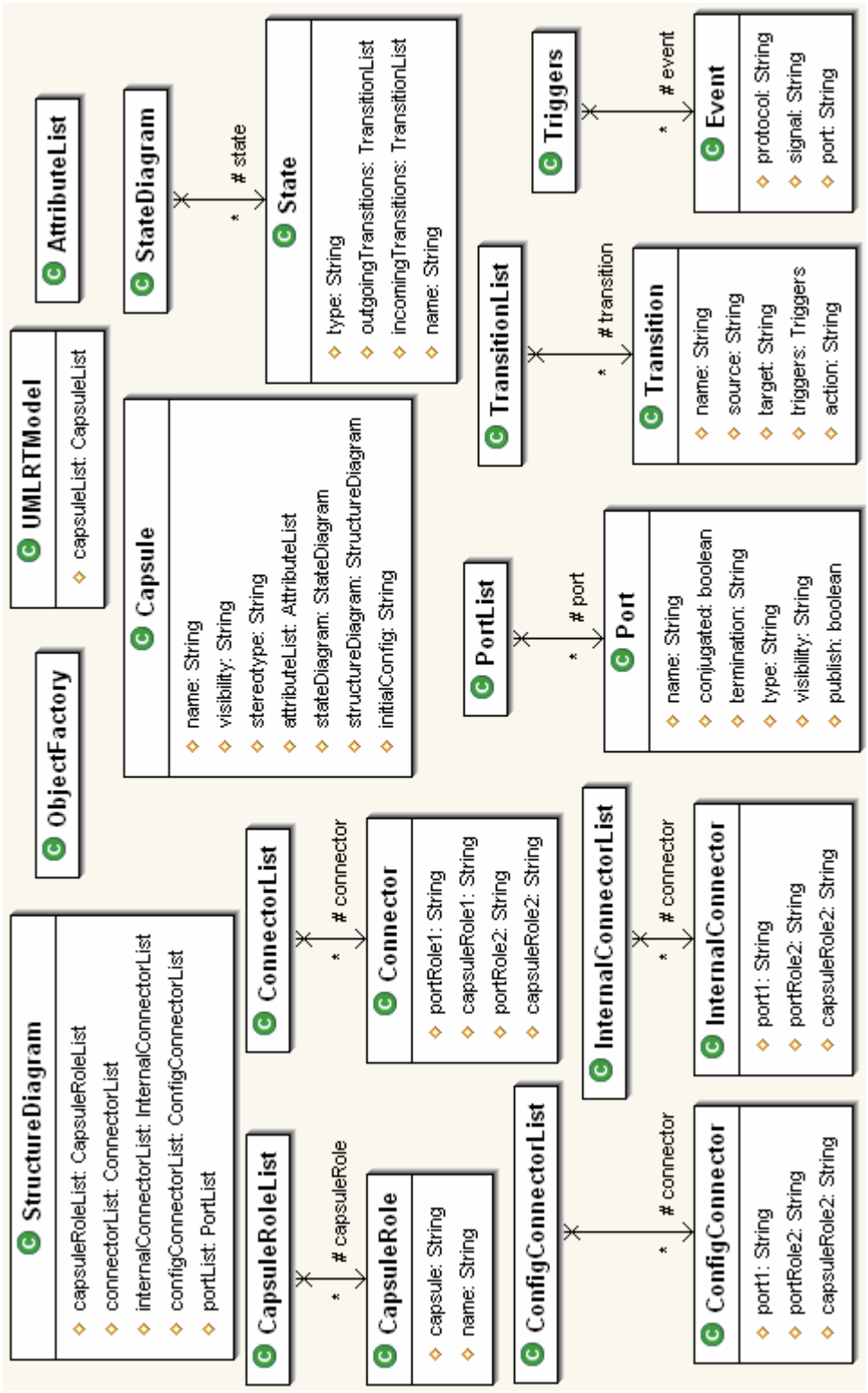


Figura 31: Diagrama das classes geradas pelo JAXB.

Para facilitar a manipulação do XML no componente Java, optou-se pelo uso de JAXB (*Java Architecture for XML Binding*) pertencente ao pacote *Java Web Services Developer Pack 2.0* [54] . O JAXB é uma tecnologia que gera automaticamente as classes Java a partir do schema de um XML, e possui métodos de *unmarshalling* e *marshalling*. O primeiro método captura o conteúdo do XML e provê uma hierarquia de objetos Java com tal conteúdo. O segundo método monta o XML a partir de objetos Java. No componente Java do protótipo foi usado o método de *unmarshalling* do JAXB. Para o schema definido desse trabalho, as classes Java geradas pelo JAXB são apresentadas na Figura 31.

No diagrama de classes da Figura 31, a classe *ObjectFactory* possui métodos para criação de instâncias das demais classes do diagrama. Analisando os elementos do schema, é possível identificar a classe ou atributo correspondente. Por exemplo, a classe *Estados* representa o elemento *Estados* definido no schema, possuindo nome (atributo *name*), tipo (atributo *type*), lista de transições de entrada (atributo *incomingTransitions*) e de saída (atributo *outgoingTransitions*).

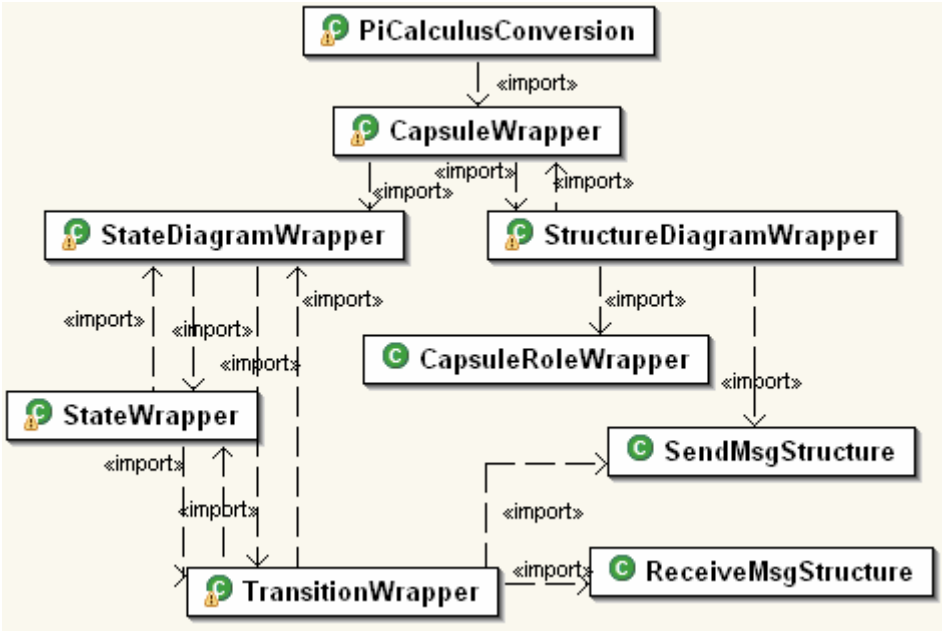


Figura 32: Diagrama das classes com a lógica de processamento.

O diagrama de classes da Figura 32 mostra as classes que possuem a lógica de processamento do componente Java. A classe *PiCalculusConversion* é a principal; ela chama o JAXB para montar os objetos (instâncias das classes da Figura 31) a partir dos dados contidos no XML. Em seguida, esta classe solicita a construção das definições π-calculus da cápsula *Top Level*. As classes *SendMsgEstrutura* e *ReceiveMsgEstrutura* são usadas para guardar informações de envio e recebimento de mensagens, respectivamente. As demais classes da Figura 32 estão diretamente associadas às classes geradas pelo JAXB, por exemplo, a classe *TransitionWrapper* captura informações da classe *Transition* e aplica as regras de mapeamento de UML-RT para π-calculus a fim de compor a definição de um dado estado.

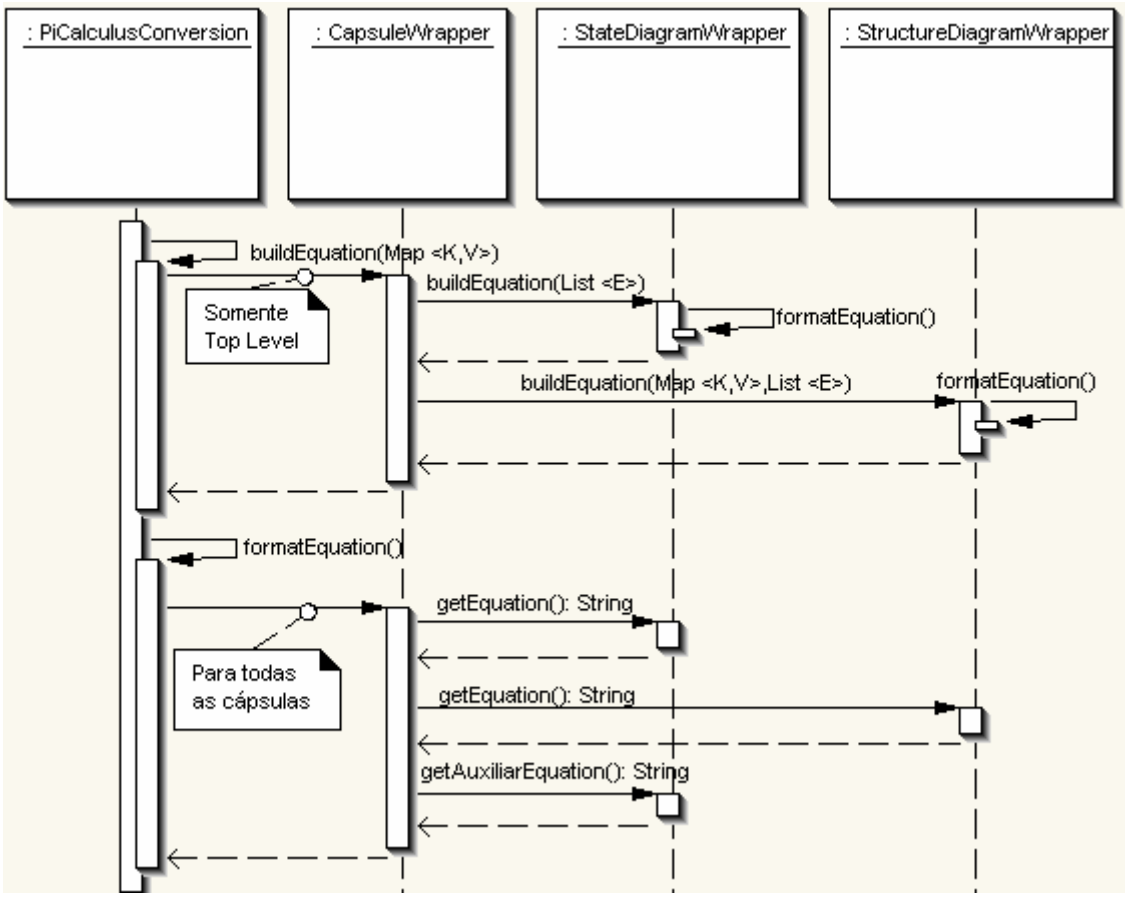


Figura 33: Diagrama de seqüência para obter as definições das cápsulas.

A Figura 33 ilustra o diagrama de seqüência para obter das definições das cápsulas. Este processo engloba a chamada da construção da cápsula principal (*Top Level*) do modelo e depois a formatação de todas as definições das cápsulas que compõem o modelo. Por construção (métodos *buildEquation*) entende-se que as informações do XML serão processadas de acordo com as regras de mapeamento e armazenadas em atributos das classes. Já formatação (métodos *formatEquation*) é a utilização desses atributos para compor a definição segundo a sintaxe da gramática do HAL-JACK.

Na Figura 33, o método *buildEquation* da cápsula *CapsuleWrapper* requisita a construção das definições do diagrama de estados e do diagrama de estrutura. O método *buildEquation* da classe *EstadosDiagramWrapper* analisa todos os estados e as transições da cápsula numa busca em profundidade marcando a cada etapa os estados visitados. Posteriormente o método *buildEquation* de *EstadosDiagramWrapper* chama o método privado *formatEquation* para formatar e disponibilizar a definição do diagrama de estados. A definição do diagrama de estados, que é definição do primeiro estado do diagrama, pode ser recuperada através do método *getEquation*. O complemento da definição do diagrama de estados são as definições de todos os estados que o compõem, e este complemento pode ser recuperado através do método *getAuxiliarEquation*.

O método *buildEquation* da classe *EstruturaDiagramWrapper* analisa todas as subcápsulas. Como cada subcápsula é uma instância de uma cápsula, o método *buildEquation* solicita a construção dessas cápsulas marcando-as como visitadas. Em seguida, este método define as entradas que vai usar ao referenciar as definições das cápsulas e formata a definição do diagrama de estrutura, que em linhas gerais é a composição paralela das definições das subcápsulas. A definição do diagrama de estrutura pode ser acessada pelo método *getEquation*.

Como as definições π-calculus geradas pelo protótipo seguem a gramática do HAL-JACK, elas podem ser submetidas à ferramenta HAL-JACK para verificação formal de uma propriedade escrita em pi-logic. O retorno da verificação formal é verdadeiro ou falso, como mostra a Figura 34, indicando respectivamente se o modelo satisfaz a propriedade ou não.

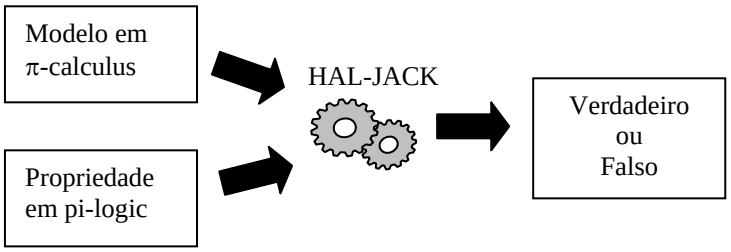


Figura 34: Verificação formal usando HAL-JACK.

4.4 Considerações finais

Neste capítulo foi apresentada a arquitetura do protótipo desenvolvido para gerar as definições π-calculus a partir do modelo UML-RT. Em linhas gerais, o modelo em UML-RT especificado na ferramenta RoseRT é capturado e disponibilizado em XML. Posteriormente, este XML é processado segundo as regras de mapeamento gerando as definições π-calculus.

O próximo capítulo destaca alguns exemplos de sistemas clássicos especificados em UML-RT na ferramenta RoseRT e convertidos para π-calculus usando o protótipo desenvolvido.

Capítulo 5

Exemplos de mapeamento

Neste capítulo são apresentados quatro exemplos de sistemas modelados em UML-RT no RoseRT. Os modelos são mapeados para definições π-calculus utilizando o protótipo desenvolvido. O primeiro sistema é um buffer de capacidade dois descrito em [23], um caso simples com apenas portas conectadas no modelo UML-RT. O segundo sistema é uma adaptação do exemplo de semáforos de trânsito provido no tutorial do RoseRT com também somente portas conectadas.

O terceiro sistema é um roteador que pode se conectar a dois componentes diferentes dependendo da configuração de sua porta não conectada. O quarto sistema é apresentado em [23] e é conhecido por Protocolo Handover pela equipe que desenvolveu o HAL-JACK [24]; trata-se de um exemplo de processos móveis e envolve portas não conectadas.

Para todos os exemplos são dispostos comentários sobre as definições π-calculus geradas. E para os sistemas buffer e Handover são verificadas algumas propriedades sugeridas pelo HAL-JACK [24] usando sua versão online disponível em http://fmt.isti.cnr.it:8080/hal/bin/HALOnLine.

5.1 Buffer de capacidade dois

Milner [23] apresenta e define em π-calculus exemplos de buffer de capacidade dois e superior. O buffer de capacidade dois ilustrado na Figura 35 é composto pela junção de duas cápsulas *Cell*, chamadas de *cell1* e *cell2*. O protocolo implementado pelas portas é o *CommunicationProtocol* composto pelo sinal de entrada *signal*.

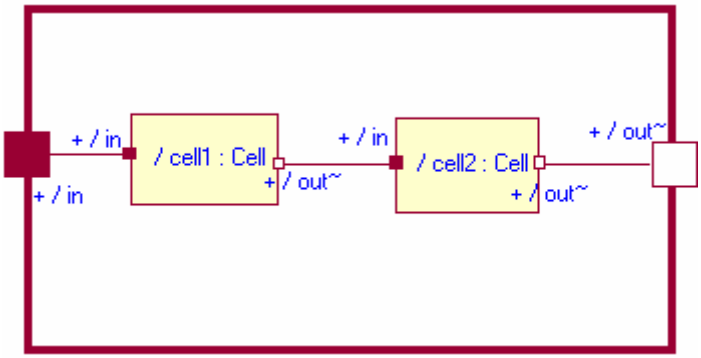


Figura 35: Diagrama de estrutura de Buffer.

Conforme a Figura 36, a cápsula *Cell* possui três portas: timer (para receber sinais de *timeout*), *in* (para receber mensagens) e *out* (para enviar mensagens). A Figura 37 apresenta o diagrama de estados de *Cell* e a Tabela 5 contém informações sobre as transições. O primeiro estado é *S1*. A transição entre *S1* e *S2* é disparada quando *Cell* recebe uma mensagem pela porta *in* e salva a mensagem recebida no atributo *msgAtt*. Em *S2*, após o recebimento do sinal de *timeout*, *Cell* envia *msgAtt* pela porta *out*.

O estado *S2* possui um símbolo dentro dele que indica a existência de ação de entrada do estado (*entry action*). Este tipo de ação não faz parte do escopo do mapeamento proposto neste trabalho, porém o modelo pode conter códigos que não são relevantes para o mapeamento, como a configuração de sinais de *timeout*, mas são fundamentais para executar o modelo no RoseRT. No caso do estado *S2*, o código é "*timer.informIn(RTTimespec(5,0));*".

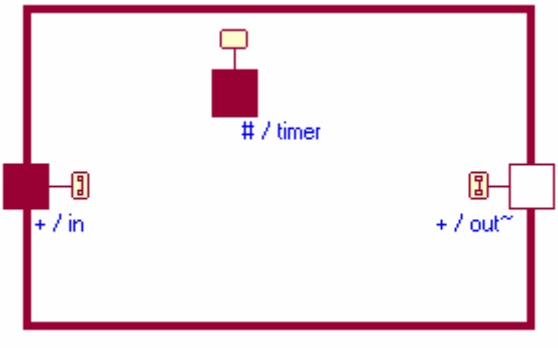


Figura 36: Diagrama de estrutura de Cell.

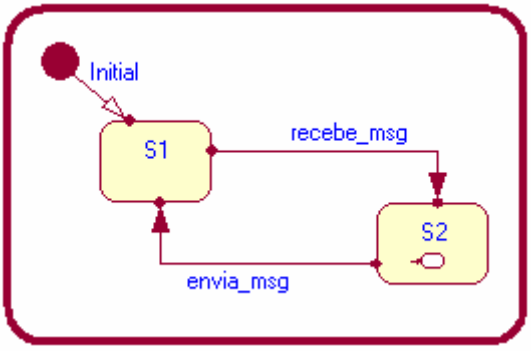


Figura 37: Diagrama de estado de Cell.

Tabela 5: Transições do diagrama de estados de *Cell*.

Transição	Porta	Sinal	Ação
Initial	-	-	-
recebe_msg	*in*	*signal*	*msgAtt = *rtdata;*
envia_msg	*timer*	*timeout*	*out.signal(msgAtt).send();*

Uma mensagem quando entra no buffer, obedece ao seguinte fluxo:

- *Buffer* recebe uma mensagem pela sua porta *in* do tipo *relay*. Esta mensagem passa diretamente para a porta *in* de *cell1*;
- A subcápsula *cell1* envia a mensagem pela porta *out*, que está conectada à porta *in* de *cell2*.
- A subcápsula *cell2* recebe a mensagem em sua porta *in*;

- A subcápsula *cell2* envia a mensagem pela porta *out*, que está conectada à porta *out* de *Buffer*. A mensagem passa diretamente ao meio externo.

Ao executar o protótipo de conversão UML-RT para π-calculus, as definições geradas foram:

define Buffer(out, in) = (p1) (Cell(in, p1) | Cell(p1, out))

define Cell(in, out) = (Cell_S1(in, out))

define Cell_S1(in, out) = in?(msg).Cell_S2(in, out, msg)

define Cell_S2(in, out, msg) = tau.out!msg.Cell_S1(in, out)

build Buffer

Através das definições acima, pode-se perceber que existe somente uma definição para cápsula *Cell*. Como a cápsula *Buffer* possui duas instâncias de *Cell*, ela passa diferentes entradas para a definição de *Cell* de acordo com as a conexão entre as portas: a instância *cell1* é representada por *Cell(in, p1)* e a instância *cell2* é representada por *Cell(p1, out)*. O nome *p1* foi gerado automaticamente pelo protótipo para renomear as portas *out* de *cell1* e *in* de *cell2* que estão conectadas.

Na definição de *Cell*, tem-se que a definição da cápsula é igual à definição do primeiro estado *S1* (indicado por *Cell_S1*), pois *Cell* não possui definição para seu diagrama de estrutura. Na definição de *S1*, há o recebimento da mensagem *msg* que é adicionada à lista de entradas do próximo estado *S2*. O estado *S2* envia *msg* através da porta *out* e volta ao estado *S1*.

Um ponto interessante de notar é que no modelo UML-RT escreve-se *msgAtt* e nas definições π-calculus escreve-se *msg*. Conforme anteriormente explicado, o sufixo *Att* é usado para facilitar o tratamento, realizado pelo protótipo, dos códigos escritos no modelo UML-RT.

Foi realizada verificação formal do modelo π-calculus de *Buffer* com as propriedades *Memory* e *NoDeadlock* indicadas pelo HAL-JACK [24]:

define Memory = AG([in?m]EF(<out!m>true))

define NoDeadlock = AG(<in?>true | <out!*>true)*

A propriedade *Memory* afirma que sempre que a porta *in* receber a mensagem *m*, esta mensagem será enviada pela porta *out*. A propriedade *NoDeadlock* indica que o sistema sempre está recebendo ou enviando mensagens. O resultado fornecido pelo HAL-JACK para ambas a propriedades foi verdadeiro, o que indica que o modelo do sistema satisfaz as propriedades.

5.2 Semáforos de Trânsito

O sistema de semáforos de trânsito foi adaptado do exemplo provido pela ferramenta RoseRT. A cápsula principal *Intersection* representa a interseção entre dois semáforos. A cápsula *Intersection* não possui diagrama de estados. Conforme o diagrama de estrutura na Figura 38, a cápsula *Intersection* é composta por dois semáforos de trânsito *trafficLightA* e *trafficLightB* que são instâncias da cápsula *TrafficLight*, e pelo controlador *control* que é instância da cápsula *TrafficControl*.

Em relação aos protocolos da Figura 38, o protocolo *StartGreen* e *StartRed* possuem apenas um sinal de entrada chamado *signal*. O protocolo *BetweenLight* possui um sinal de entrada e um de saída com o mesmo nome de *turningRed*. Todos os sinais desses protocolos são do tipo *void*, logo na comunicação entre as portas há sincronização de portas, mas nenhuma mensagem é efetivamente trocada.

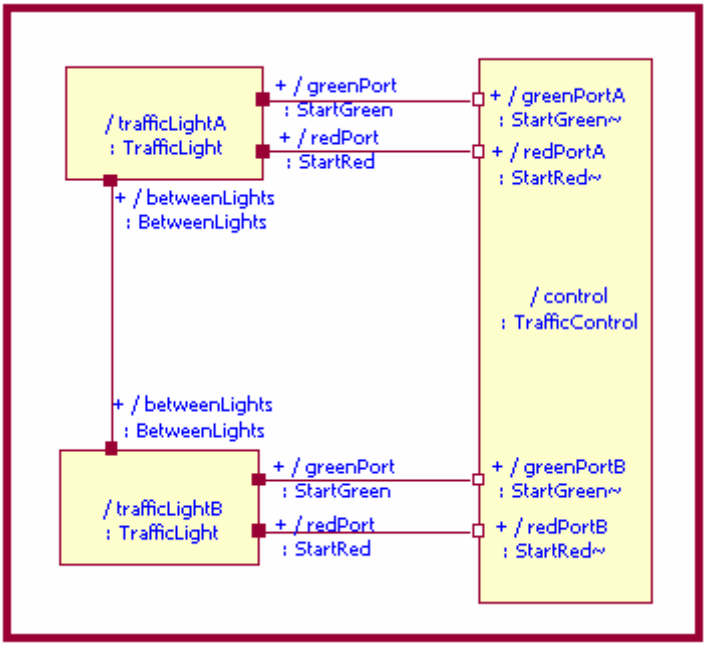


Figura 38: Diagrama de estrutura de *Intersection*.

O diagrama de estrutura da cápsula *Control* é mostrado na Figura 39. O controlador possui a porta *timer* que implementa o protocolo *Timing* a fim de lhe prover sinais de *timeout*, e possui outras quatro portas, a saber:

- *greenPortA* para indicar ao semáforo *trafficLightA* que ele deve iniciar como verde;
- *redPortA* para indicar ao semáforo *trafficLightA* que ele deve iniciar como vermelho;
- *greenPortB* para indicar ao semáforo *trafficLightB* que ele deve iniciar como verde;
- *redPortB* para indicar ao semáforo *trafficLightB* que ele deve iniciar como vermelho.

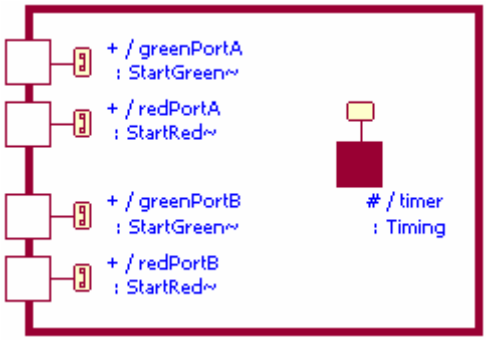


Figura 39: Diagrama de estrutura de *TrafficControl*.

O diagrama de estados de *TrafficControl* consta na Figura 40 e suas transições estão descritas na Tabela 6. Através da transição *enable_A_green*, *TrafficControl* envia um sinal à *trafficLightA* para que ele inicie como verde. Em seguida, através da transição *enable_B_red*, *TrafficControl* envia um sinal à *trafficLightB* para que ele comece como vermelho. As ações de entrada de *S1* e *S2*, embora não relevantes ao mapeamento, possuem o código "*timer.informIn(RTTimespec(1,0));*" que é fundamental para configuração dos sinais de *timeout* no modelo UML-RT.

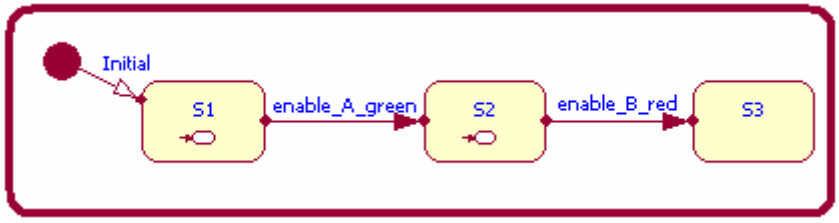


Figura 40: Diagrama de estados de *TrafficControl*.

Tabela 6: Transições do diagrama de estados de *TrafficControl*.

Transição	Porta	Sinal	Ação
Initial	-	-	-
enable_A_green	*timer*	*timeout*	*greenPortA.signal().send();*
enable_B_red	*timer*	*timeout*	*redPortB.signal().send();*

O diagrama de estrutura do semáforo *TrafficLight* é apresentado na Figura 41. O diagrama de estados consta na Figura 42 e as transições são detalhadas na Tabela 7.

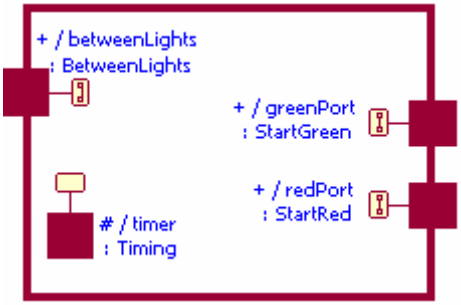


Figura 41: Diagrama de estrutura de *TrafficLight*.

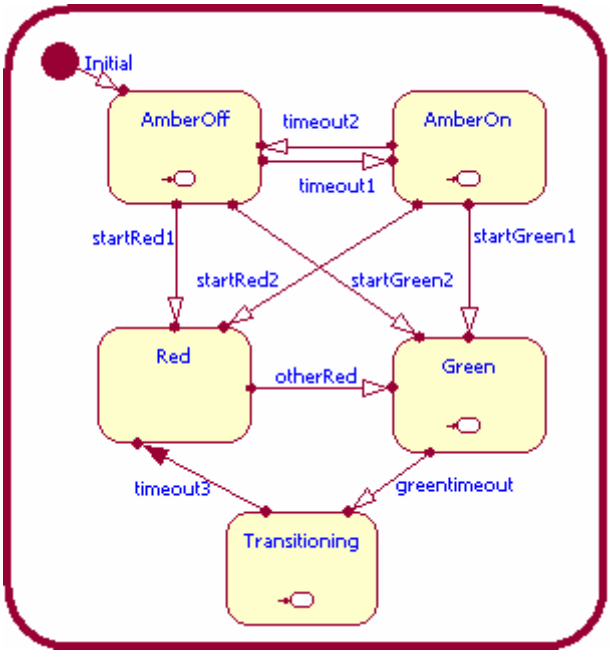


Figura 42: Diagrama de estados de *TrafficLight*.

Tabela 7: Transições do diagrama de estados de *TrafficLight*.

Transição	Porta	Sinal	Ação
Initial	-	-	-
timeout1	*timer*	*timeout*	-
timeout2	*timer*	*timeout*	-
startRed1	*redPort*	*signal*	-
startRed2	*redPort*	*signal*	-
startGreen1	*greenPort*	*signal*	-
startGreen2	*greenPort*	*signal*	-
greentimeout	*timer*	*timeout*	-
timeout3	*timer*	*timeout*	*betweenLights.turningRed().send();*
otherRed	*betweenLights*	*turningRed*	-

O estado inicial de *TrafficLight* é *AmberOff* que indica que a luz amarela do semáforo está apagada. Um sinal de *timeout* (transição *timeout1*) leva o semáforo ao estado *AmberOn*

simbolizando que a luz amarela está ligada. O semáforo poderá transitar entre os estados *AmberOn* e *AmberOff* enquanto estiver inoperante até receber algum sinal de *TrafficControl*.

A partir dos estados *AmberOn* e *AmberOff* da Figura 42, o semáforo pode receber um sinal pela porta *redPort* (transições *startRed1* ou *startRed2*), passando ao estado *Red* (semáforo vermelho). O semáforo pode também receber um sinal pela porta *greenPort* (transições *startGreen1* ou *startGreen2*) e ser levado ao estado *Green* (semáforo verde). O semáforo no estado *Green* aguarda um tempo predeterminado e vai através da transição *greentimeout* ao estado *Transitioning*, que representa a luz amarela do semáforo intermediária entre a verde e a vermelha. Após receber um sinal de *timeout* (transição *timeout3*), o semáforo passa do estado *Transitioning* para o estado *Red*. O semáforo só sai do estado *Red* para o estado *Green* se receber do outro semáforo, que está no mesmo cruzamento, um sinal autorizando sua mudança para verde através da transição *otherRed*.

Todas as ações de entrada dos estados *AmberOff*, *AmberOn*, *Green* e *Trasitioning* foram usadas para configurar os sinais de *timeout* com o código "*timer.informIn(RTTimespec(10,0));*".

Ao executar o protótipo de conversão UML-RT para π-calculus, as definições geradas foram:

define Intersection() = (p1)(p5)(p3)(p2)(p4) (TrafficLight(p1, p5, p3) | TrafficLight(p1, p2, p4) | TrafficControl(p2, p3, p4, p5))

define TrafficControl(greenPortA, redPortB, redPortA, greenPortB) = (TrafficControl_S1(greenPortA, redPortB, redPortA, greenPortB))

```
define TrafficControl_S1(greenPortA, redPortB, redPortA, greenPortB) =
tau.greenPortA!greenPortA.TrafficControl_S2(greenPortA, redPortB, redPortA, greenPortB)
define TrafficControl_S2(greenPortA, redPortB, redPortA, greenPortB) =
tau.redPortB!redPortB.TrafficControl_S3(greenPortA, redPortB, redPortA, greenPortB)
define TrafficControl_S3(greenPortA, redPortB, redPortA, greenPortB) = nil

define TrafficLight(betweenLights, greenPort, redPort) =  (
TrafficLight_AmberOff(betweenLights, greenPort, redPort) )
define TrafficLight_AmberOff(betweenLights, greenPort, redPort) =
tau.TrafficLight_AmberOn(betweenLights, greenPort, redPort) +
redPort?(empty).TrafficLight_Red(betweenLights, greenPort, redPort) +
greenPort?(empty).TrafficLight_Green(betweenLights, greenPort, redPort)
define TrafficLight_Transitioning(betweenLights, greenPort, redPort) =
tau.betweenLights!betweenLights.TrafficLight_Red(betweenLights, greenPort, redPort)
define TrafficLight_Red(betweenLights, greenPort, redPort) =
betweenLights?(empty).TrafficLight_Green(betweenLights, greenPort, redPort)
define TrafficLight_Green(betweenLights, greenPort, redPort) =
tau.TrafficLight_Transitioning(betweenLights, greenPort, redPort)
define TrafficLight_AmberOn(betweenLights, greenPort, redPort) =
tau.TrafficLight_AmberOff(betweenLights, greenPort, redPort) +
greenPort?(empty).TrafficLight_Green(betweenLights, greenPort, redPort) +
redPort?(empty).TrafficLight_Red(betweenLights, greenPort, redPort)

build Intersection
```

Analisando as definições acima, podem-se ver cinco nomes gerados automaticamente pelo protótipo (*p1*, *p2*, *p3*, *p4* e *p5*) que são usados para renomear as portas conectadas no diagrama de estrutura de *Intersection* (Figura 38).

Como os protocolos usados possuem sinal do tipo *void*, não há mensagem trocada, apenas sincronização entre as portas. Isso pode ser visto, por exemplo, no trecho "*redPortB!redPortB*" da definição de *TrafficControl_S2* e no trecho "*redPort?(empty)*" da definição de *TrafficLight_AmberOff*.

5.3 Roteador

O exemplo do roteador visa demostrar o mapeamento de portas não conectadas. A cápsula principal é *P*, cujo diagrama de estrutura é mostrado na Figura 43. A cápsula *P* é composta pelas subcápsulas: *source* (instância da cápsula *Q*), *router* (instância da cápsula *R*), *target1* (instância da cápsula *S*) e *target2* (instância da cápsula *T*).

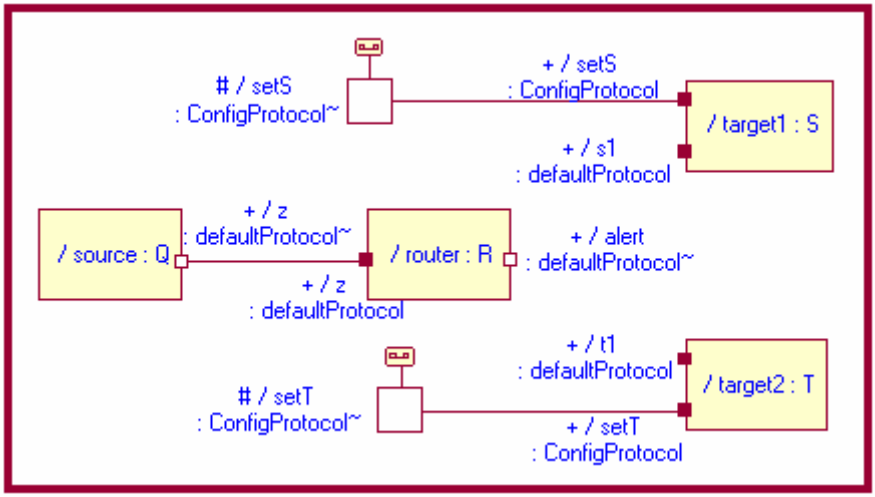


Figura 43: Diagrama de estrutura de *P*.

Conforme a Figura 43, a porta *z* de *Q* está conectada à porta *z* de *R*. A porta *alert* de *R* é não conectada, mas durante a execução do diagrama de estados de *R*, *alert* pode ser renomeada de forma a se conectar dinamicamente com a porta *s1* de *S* ou *t1* de *T*. As portas *setS* e *setT* pertencentes à cápsula *P* implementam o protocolo *ConfigProtocol*, logo são usadas para configuração inicial de portas não conectadas. No caso, a porta *setS* de *P* envia a configuração inicial para *s1* de *S*, e a porta *setT* de *P* envia a configuração inicial para *t1* de *T*. O código usado por *P* para envio da configuração inicial das portas não conectadas é escrito da transição inicial de seu diagrama de estados ilustrado na Figura 44. O código propriamente dito é apresentado na Tabela 8.

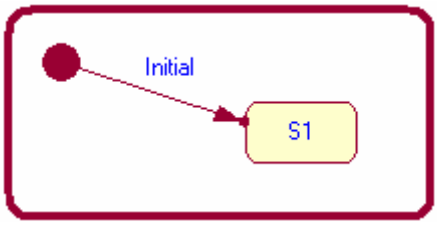


Figura 44: Diagrama de estados de *P*.

Tabela 8: Transições do diagrama de estados de *P*.

Transição	Porta	Sinal	Ação
Initial	-	-	*xAtt = "x";* *yAtt = "y";* *setS.signal(xAtt).send();* *setT.signal(yAtt).send();*

A Figura 45 apresenta o diagrama de estrutura da cápsula *Q*, composto pelas portas *timer* e *z*. A porta *timer* é usada para definir os sinais de *timeout*. De acordo com o diagrama de estados de *Q*, ilustrado na Figura 46, seu primeiro estado é *S1*. Após um sinal de *timeout*, a cápsula *Q* atinge o ponto de escolha *CP1* onde, de acordo com uma lógica interna, pode passar ao estado *S2* ou *S3*. A Tabela 9 descreve as informações das transições do diagrama de

estados de *Q*. Se acionada a transição *send_x*, *Q* envia a mensagem *xAtt* através da porta *z*. Entretanto, se acionada a transição *send_y*, *Q* envia a mensagem *yAtt* através da porta *z*. Esta mensagem é recebida pela porta *z* de *R*.

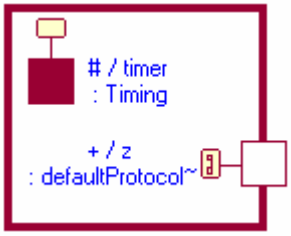


Figura 45: Diagrama de estrutura de *Q*.

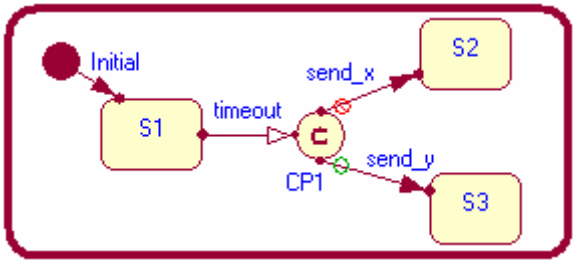


Figura 46: Diagrama de estados de *Q*.

Tabela 9: Transições do diagrama de estados de *Q*.

Transição	Porta	Sinal	Ação
Initial	-	-	*timer.informIn(RTTimespec(2,0));* *xAtt = "x";* *yAtt = "y";*
timeout	*timer*	*timeout*	-
send_x	-	-	*z.signal(xAtt).send();*
send_y	-	-	*z.signal(yAtt).send();*

A cápsula *R* possui as portas *timer*, *z* e *alert* de acordo com seu diagrama de estrutura, ilustrado na Figura 47. O diagrama de estados de *R* é ilustrado na Figura 48 e suas transições encontram-se detalhadas na Tabela 10.

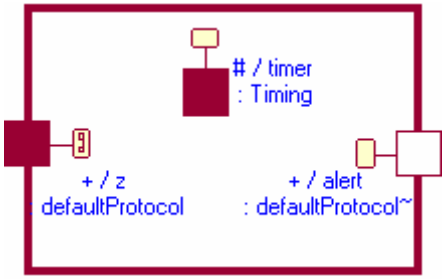


Figura 47: Diagrama de estrutura de *R*.

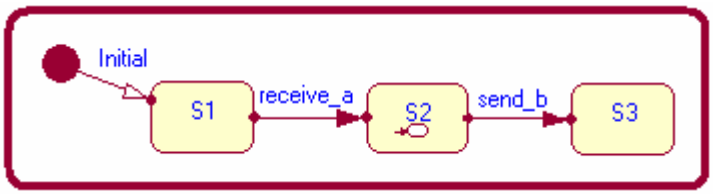


Figura 48: Diagrama de estados de *R*.

Tabela 10: Transições do diagrama de estados de *R*.

Transição	Porta	Sinal	Ação
Initial	-	-	-
receive_a	*z*	*signal*	*aAtt = *rtdata;* *alert.registerSPP(aAtt);* *alertAtt = aAtt;*
send_b	*timer*	*timeout*	*alert.signal(bAtt).send();*

O estado inicial de *R* é *S1*. A transição *receive_a* é acionada quando *R* recebe um sinal através da porta *z*. A mensagem recebida *aAtt* é usada para configurar a porta não conectada *alert* (conforme o código "*alert.registerSPP(aAtt);*"). O código "*alertAtt = aAtt;*" é usado para indicar ao protótipo que houve uma reconfiguração de nome, no caso da porta *alert*.

O estado *S2* de *R* na Figura 48 possui código de entrada "*timer.informIn(RTTimespec(2,0));*" que define a ocorrência de *timeout* após 2 segundos. Quando ocorre o *timeout*, a transição *send_b* é acionada e executa a ação de enviar a mensagem *bAtt* através da porta *alert*. Se a porta *alert* estiver configurada com *xAtt*, a

mensagem *bAtt* vai para cápsula *S*. Porém, se a porta *alert* estiver configurada com *yAtt*, a mensagem *bAtt* vai para cápsula *T*.

A Figura 49 apresenta o diagrama de estrutura de *S* e a Figura 50 mostra o diagrama de estados de *S*, cujas transições constam na Tabela 11. A transição *config_s1* é acionada quando *S* recebe um sinal pela porta *setS* para fazer a configuração inicial da porta não conectada *s1*. No estado *S2*, *S* está apto a receber através da porta *s1* uma mensagem que é capturada usando o atributo *cAtt*.

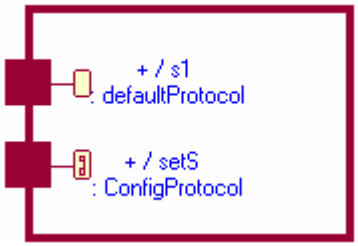


Figura 49: Diagrama de estrutura de *S*.

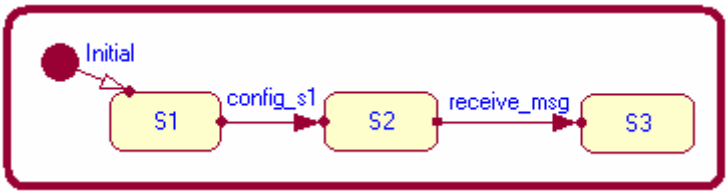


Figura 50: Diagrama de estados de *S*.

Tabela 11: Transições do diagrama de estados de *S*.

Transição	Porta	Sinal	Ação
Initial	-	-	-
config_s1	*setS*	*signal*	*s1Att = *rtdata;* *s1.registerSAP(s1Att);*
receive_msg	*s1*	*signal*	*cAtt = *rtdata;*

O diagrama de estrutura de *T* é apresentado na Figura 51 e o diagrama de estados, na Figura 52. A Tabela 12 contém os dados das transições de *T*. O comportamento de *T* é similar

ao da cápsula *S*: após a configuração inicial de sua porta não conectada *t1*, *T* pode receber uma mensagem através da porta *t1*.

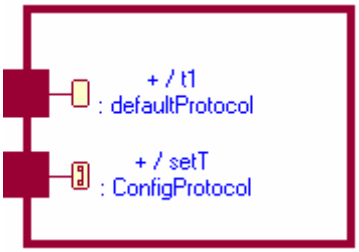


Figura 51: Diagrama de estrutura de *T*.

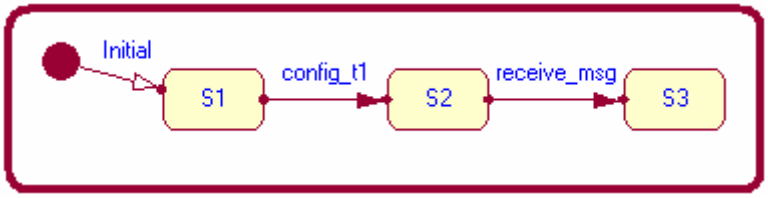


Figura 52: Diagrama de estados de *T*.

Tabela 12: Transições do diagrama de estados de *T*.

Transição	Porta	Sinal	Ação
Initial	-	-	-
config_t1	*setT*	*signal*	*t1Att = *rtdata;* *t1.registerSAP(t1Att);*
receive_msg	*t1*	*signal*	*dAtt = *rtdata;*

Ao executar o protótipo de conversão UML-RT para π-calculus, as definições geradas foram:

define P() = (p1)(y)(x)(b) (P_S1() | Q(p1, y, x) | R(p1, b) | S(x) | T(y))

define P_S1() = nil

define Q(z, y, x) = (Q_S1(z, y, x))

define Q_S1(z, y, x) = tau.Q_CP1(z, y, x)

```
define Q_S2(z, y, x) = nil

define Q_S3(z, y, x) = nil

define Q_CP1(z, y, x) = tau.z!y.Q_S3(z, y, x) + tau.z!x.Q_S2(z, y, x)


define R(z, b) =  ( R_S1(z, b) )

define R_S1(z, b) = z?(a).R_S2(z, a, b)

define R_S2(z, alert, b) = tau.alert!b.R_S3(z, alert, b)

define R_S3(z, alert, b) = nil

define S(s1) =  ( S_S2(s1) )

define S_S2(s1) = s1?(c).S_S3(s1, c)

define S_S3(s1, c) = nil


define T(t1) =  ( T_S2(t1) )

define T_S3(t1, d) = nil

define T_S2(t1) = t1?(d).T_S3(t1, d)

build P
```

Na definição da cápsula *P* é interessante notar o termo *P_S1()* que representa a definição do diagrama de estados de *P*. Como o diagrama de estados tem apenas o primeiro estado e este estado não possui transições de saída, sua definição é *nil* (processo nulo). O termo *p1* foi gerado automaticamente pelo protótipo para representar a conexão entre as portas *z* de *Q* e *z* de *R*. O nome *x* em *S(x)* é a configuração inicial da porta não conectada *s1* de *S*. O nome *y* em *T(y)* é a configuração inicial da porta não conectada *t1* de *T*.

Na definição de *S* percebe-se que o primeiro estado considerado é *S2*. Isso ocorre porque a transição entre *S1* e *S2* é usada para configuração inicial de porta não conectada. Caso semelhante pode ser visto na definição de *T*.

Analisando a definição da cápsula *Q*, vê-se o mapeamento de ponto de escolha *CP1*, que é feito com uso do operador escolha do π-calculus abstraindo a lógica contida no ponto de escolha. As mensagens *x* e *y* enviadas na definição de *Q_CP1(z, y, x)* são novos nomes que não foram determinados anteriormente como entradas de estados ou mensagens recebidas. As mensagens *x* e *y* fazem parte da lista *LE[Q]*, assim foram adicionadas pelo protótipo a todas as listas de entradas do diagrama de estados de *Q* no final da análise desse diagrama conforme Def. 2.

Na definição da cápsula *R*, o nome *b* pertence à lista *LE[R]* de novos nomes estabelecidos no decorrer do diagrama de estados de *R*, por isso *b* foi adicionado a todas as listas de entradas do diagrama de estados de *R*. Além disso, vale ressaltar que *alert* é a porta não conectada de *R* configurada a partir da mensagem *a* recebida pela porta *z*. Isso pode ser visto através da definição de *R_S2(z, alert, b)*, que é chamada pelo estado S1 como *R_S2(z, a, b)*.

5.4 Protocolo Handover

Milner [23] apresenta o Protocolo Handover como o sistema clássico para entendimento da mobilidade das conexões entre componentes. Porém o nome Protocolo Handover aqui utilizado foi emprestado do HAL-JACK [24].

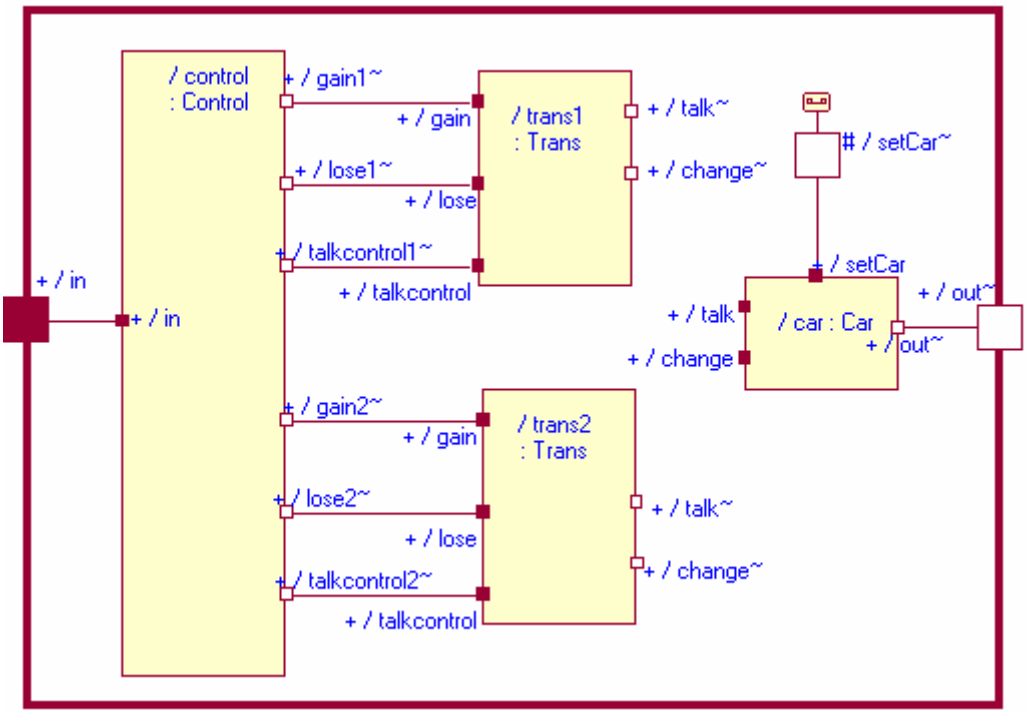


Figura 53: Diagrama de estrutura de *Handover*.

O sistema *Handover* mostrado na Figura 53 é composto por uma torre de controle (cápsula *Control*), duas torres de transmissão (cápsulas *Trans*) e um carro (cápsula *Car*). Para melhor visualização da Figura 53, o tipo de protocolo de cada porta foi omitido. Porém no diagrama de estrutura de cada cápsula há indicação dos protocolos, todos eles compostos por um sinal de entrada chamado *signal*.

A torre de controle possui conexões fixas com as torres de transmissão, o que é expresso pelas portas conectadas entre a cápsula *Control* e as duas cápsulas *Trans*. O carro se comunica com uma torre de transmissão, mas um dado evento pode fazer o carro se comunicar com a outra torre de transmissão; daí a importância das portas não conectadas da cápsula *Car*. Milner ressalta que neste sistema é importante não confundir o movimento físico do carro, com o movimento dos *links* de comunicação entre o carro e as torres de transmissão. O principal objetivo do exemplo é entender o movimento virtual dos *links*.

Pode-se fazer uma analogia do *Handover* com um sistema que processa a mensagem em duas etapas. A primeira etapa é o processamento realizado por uma cápsula *Trans* e a segunda

etapa é o processamento feito pela *cápsula* Car. A torre do controle pode ser entendida como um processo auxiliar que gerencia a demanda de requisições entre duas cápsulas *Trans*.

O diagrama de estados da cápsula-mãe *Handover*, ilustrado na Figura 54, é composto apenas pela transição inicial. Seu código disposto na

Tabela 13 inicia os atributos *talk1Att* e *change1Att*, e os envia como mensagem pela porta *setCar*. A porta *setCar* implementa o protocolo *ConfigProtocol* e é usada para configuração inicial das portas não conectadas da cápsula *Car*.

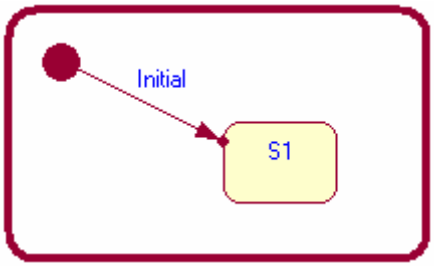


Figura 54: Diagrama de estados de *Handover*.

Tabela 13: Transições do diagrama de estados de *Handover*.

Transição	Porta	Sinal	Ação
Initial	-	-	*talk1Att = "talk1";* *change1Att = "change1";* *setCar.signal(talk1Att).send();* *setCar.signal(change1Att).send();*

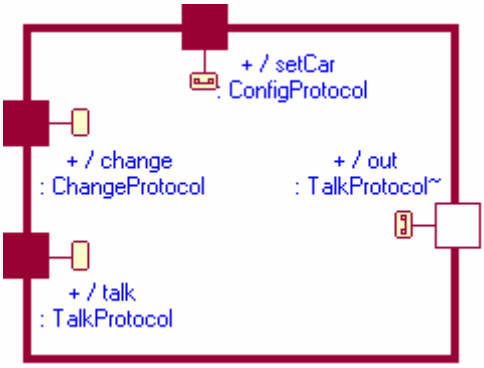


Figura 55: Diagrama de estrutura de *Car*.

A Figura 55 mostra o diagrama de estrutura da cápsula *Car*. O diagrama de estado de *Car* consta na Figura 56 e suas transições estão disponibilizadas na Tabela 14.

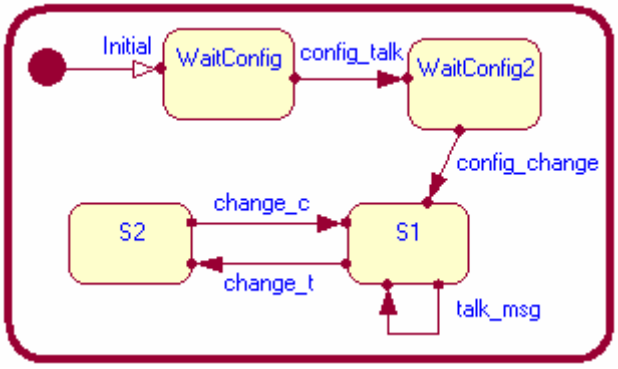


Figura 56: Diagrama de estados de *Car*.

Tabela 14: Transições do diagrama de estados de *Car*.

Transição	Porta	Sinal	Ação
config_talk	*setCar*	*signal*	*talkAtt = *rtdata;* *talk.registerSAP(talkAtt);*
config_change	*setCar*	*signal*	*changeAtt = *rtdata;* *change.registerSAP(changeAtt);*
talk_msg	*talk*	*signal*	*msgAtt = *rtdata;* *out.signal(msgAtt).send();*
change_t	*change*	*signal*	*tAtt = *rtdata;* *talkAtt = tAtt;* *talk.registerSAP(talkAtt);*
change_c	*change*	*signal*	*cAtt = *rtdata;* *changeAtt = cAtt;* *change.registerSAP(changeAtt);*

A primeira etapa do diagrama de estados de *Car* é receber as configurações iniciais das portas não conectadas *talk* e *change* através das transições *config_talk* e *config_change*. Em seguida, o carro no estado *S1* está apto a se comunicar com a torre *trans1*. Entretanto o carro também pode receber uma mensagem de *trans1* pela porta *change*, indicando um novo nome para a porta *talk* (transição *change_t*) e depois indicando um novo nome para porta *change* (transição *change_c*). Esta é a etapa que o carro reconfigura suas portas para começar a operar com a torre *trans2*.

O diagrama de estrutura da torre *Trans* é apresentado na Figura 57. A torre *Trans* não recebe configuração inicial para suas portas não conectadas *talk* e *change*, uma vez que não possui nenhuma porta que implemente o protocolo *ConfigProtocol* para este fim.

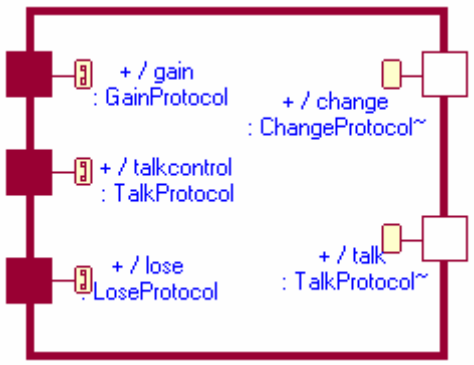


Figura 57: Diagrama de estrutura de *Trans*.

De acordo com o diagrama de estados na Figura 58 e com as informações das transições na Tabela 15, os nomes que irão configurar as portas *talk* e *change* são enviados por *Control* e recebidos nas transições *gain_t* e *gain_c*. No estado *ActiveTrans*, a torre pode receber uma mensagem de *Control* e enviá-la para o carro (transição *talkcontrol_msg*). No estado *ActiveTrans*, a torre também pode receber uma mensagem de *Control* com um novo nome *t* (transição *lose_t*) e depois um novo nome *c* (transição *lose_c*). Esses nomes são passados para o carro e indicam como as portas não conectadas do carro deverão ser reconfiguradas.

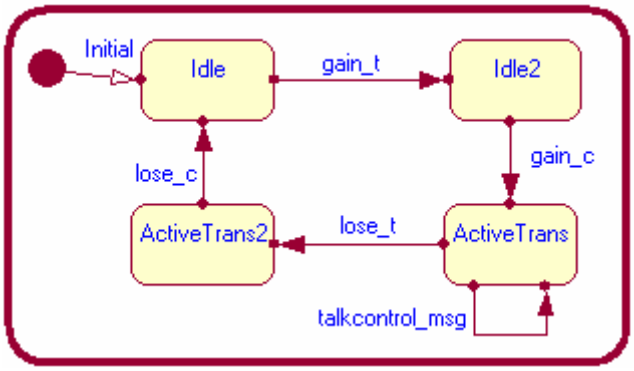


Figura 58: Diagrama de estados de *Trans*.

Tabela 15: Transições do diagrama de estados de *Trans*.

Transição	Porta	Sinal	Ação
gain_t	*gain*	*signal*	*tAtt = *rtdata;* *talkAtt = tAtt;* *talk.registerSPP(talkAtt);*
gain_c	*gain*	*signal*	*cAtt = *rtdata;* *changeAtt = cAtt;* *change.registerSPP(changeAtt);*
talkcontrol_msg	*talkcontrol*	*signal*	*msgAtt = *rtdata;* *talk.signal(msgAtt).send();*
lose_t	*lose*	*signal*	*tAtt = *rtdata;*
lose_c	*lose*	*signal*	*cAtt = *rtdata;* *change.signal(tAtt).send();* *change.signal(cAtt).send();*

A Figura 59 mostra o diagrama de estrutura da torre de controle *Control*. O diagrama de estados de *Control* da Figura 60 inicia enviando informações para torre *trans1* configurar suas portas não conectadas (transição *config_trans1*). As informações das transições de *Control* constam na Tabela 16. No estado *Control1*, *Control* pode receber uma mensagem do meio

externo (transição *in_msg1*). Mas pode também enviar os nomes *talk2* e *change2* para *trans1* notificar ao carro os novos nomes das suas portas não conectadas, e em seguida enviar os mesmos nomes para *trans2* configurar suas portas não conectadas. Dessa forma, a transição de *Control1* para *Control2* indica que *trans2* é que está apta a conversar com o carro. Enquanto a transição de *Control2* para *Control1*, habilita novamente *trans1*.

Quanto às ações de entrada dos estados *S1*, *Control1* e *Control2*, embora não mapeadas, elas são necessárias no modelo UML-RT para configuração de sinais de *timeout*. Seus códigos são respectivamente: "*timer.informIn(RTTimespec(5,0));*", "*timer.informIn(RTTimespec(10,0));*" e "*timer.informIn(RTTimespec(20,0));*".

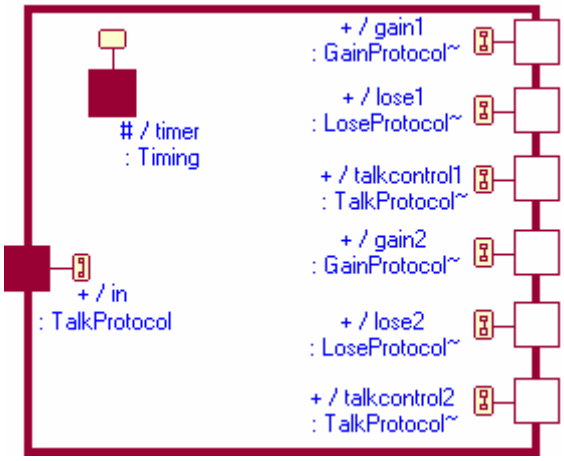


Figura 59: Diagrama de estrutura de *Control*.

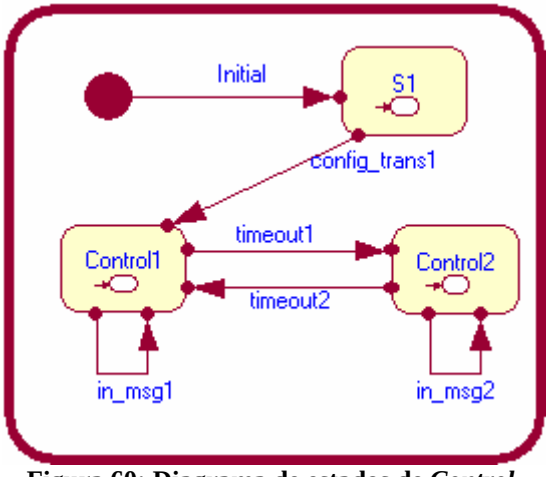


Figura 60: Diagrama de estados de *Control*.

Tabela 16: Transições do diagrama de estados de *Control*.

Transição	Porta	Sinal	Ação
config_trans1	*timer*	*timeout*	*gain1.signal(talk1Att).send();* *gain1.signal(change1Att).send();*
in_msg1	*in*	*signal*	*msgAtt = *rtdata;* *talkcontrol1.signal(msgAtt).send();*
timeout1	*timer*	*timeout*	*lose1.signal(talk2Att).send();* *lose1.signal(change2Att).send();* *gain2.signal(talk2Att).send();* *gain2.signal(change2Att).send();*
in_msg2	*in*	*signal*	*msgAtt = *rtdata;* *talkcontrol2.signal(msgAtt).send();*
timeout2	*timer*	*timeout*	*lose2.signal(talk1Att).send();* *lose2.signal(change1Att).send();* *gain1.signal(talk1Att).send();* *gain1.signal(change1Att).send();*

As definições π-calculus geradas pelo protótipo de conversão UML-RT para π-calculus foram:

define Handover(in, out) =

(p1)(p2)(p3)(p4)(p5)(p6)(talk1)(change1)(talk2)(change2)

(Handover_S1(in, out) | Control(p1, p2, p3, p4, p5, p6, in, talk1, change1, talk2, change2) |

Trans(p1, p2, p6) | Car(talk1, change1, out) | Trans(p3, p4, p5))

define Handover_S1(in, out) = nil

define Car(talk, change, out) = (Car_S1(talk, change, out))

define Car_S1(talk, change, out) = talk?(msg).out!msg.Car_S1(talk, change, out) +

change?(t).Car_S2(t, change, out)

define Car_S2(talk, change, out) = change?(c).Car_S1(talk, c, out)

define Trans(gain, lose, talkcontrol) =

(Trans_Idle(gain, lose, talkcontrol))

define Trans_Idle(gain, lose, talkcontrol) = gain?(t).Trans_Idle2(gain, lose, talkcontrol, t)

define Trans_Idle2(gain, lose, talkcontrol, talk) = gain?(c).Trans_ActiveTrans(gain, lose,

talkcontrol, talk, c)

define Trans_ActiveTrans(gain, lose, talkcontrol, talk, change) =

lose?(t).Trans_ActiveTrans2(gain, lose, talkcontrol, talk, change, t) +

talkcontrol?(msg).talk!msg.Trans_ActiveTrans(gain, lose, talkcontrol, talk, change)

define Trans_ActiveTrans2(gain, lose, talkcontrol, talk, change, t) =

lose?(c).change!t.change!c.Trans_Idle(gain, lose, talkcontrol)

define Control(gain1, lose1, gain2, lose2, talkcontrol2, talkcontrol1, in, talk1, change1, talk2,

change2) = (Control_S1(gain1, lose1, gain2, lose2, talkcontrol2, talkcontrol1, in, talk1,

change1, talk2, change2))

define Control_S1(gain1, lose1, gain2, lose2, talkcontrol2, talkcontrol1, in, talk1, change1,

talk2, change2) = tau.gain1!talk1.gain1!change1.Control_Control1(gain1, lose1, gain2,

lose2, talkcontrol2, talkcontrol1, in, talk1, change1, talk2, change2)

define Control_Control1(gain1, lose1, gain2, lose2, talkcontrol2, talkcontrol1, in, talk1,

change1, talk2, change2) =

tau.lose1!talk2.lose1!change2.gain2!talk2.gain2!change2.Control_Control2(gain1, lose1, gain2, lose2, talkcontrol2, talkcontrol1, in, talk1, change1, talk2, change2) + in?(msg).talkcontrol1!msg.Control_Control1(gain1, lose1, gain2, lose2, talkcontrol2, talkcontrol1, in, talk1, change1, talk2, change2)

define Control_Control2(gain1, lose1, gain2, lose2, talkcontrol2, talkcontrol1, in, talk1, change1, talk2, change2) =

tau.lose2!talk1.lose2!change1.gain1!talk1.gain1!change1.Control_Control1(gain1, lose1, gain2, lose2, talkcontrol2, talkcontrol1, in, talk1, change1, talk2, change2) + in?(msg).talkcontrol2!msg.Control_Control2(gain1, lose1, gain2, lose2, talkcontrol2, talkcontrol1, in, talk1, change1, talk2, change2)

Duas propriedades do artigo do HAL-JACK [24] foram usadas na verificação formal da definição de *Handover* gerada pelo protótipo. A primeira propriedade *Reliable1* afirma que sempre que ocorrer uma entrada pela porta *in*, vai existir uma saída pela porta *out* para a mesma mensagem. A segunda propriedade *NoWait* alega que sempre existe a possibilidade de receber uma mensagem pela porta *in*.

define Reliable1 = AG([in?n]EF<out!n>true)

define NoWait = AG(<in?>true)*

Executando a ferramenta HAL-JACK para o sistema *Handover* e a propriedade *Reliable1*, o resultado foi verdadeiro. Já para a propriedade *NoWait*, o resultado foi falso, o que é coerente, pois o sistema executa atividades como mudança de uso de torres de transmissão e não está sempre disponível a receber novas mensagens do meio externo.

5.5 Considerações finais

Neste capítulo foram apresentados exemplos do mapeamento de UML-RT para π-calculus. O modelo UML-RT foi especificado no RoseRT e a conversão conforme as regras de mapeamento foi provida pelo protótipo desenvolvido. O próximo capítulo comenta as considerações adotadas sobre o mapeamento e o protótipo, e provê uma análise do trabalho.

Capítulo 6

Considerações e Análise

Durante os capítulos anteriores foram citadas algumas considerações e restrições a respeito do escopo do mapeamento UML-RT para π-calculus. Além disso, foram expostas considerações sobre como especificar o modelo na ferramenta RoseRT de forma que o protótipo consiga capturar informações do modelo usadas no mapeamento. Neste capítulo provê-se um resumo das considerações adotadas, além de uma análise geral do trabalho.

6.1 Considerações sobre o mapeamento

Quanto ao escopo do mapeamento UML-RT para π-calculus, a principal consideração foi o mapeamento somente de elementos de comunicação entre componentes do sistema, ou seja, a troca de mensagens entre eles. As hipóteses e restrições do mapeamento foram explicadas anteriormente na Seção 3.1 e abaixo segue um resumo delas:

- As definições π-calculus resultantes do mapeamento seguem a sintaxe proposta pelo HAL-JACK.
- Adoção do *monadic* π-calculus, que é a versão usada pelo HAL-JACK.
- A comunicação é síncrona.
- Somente cápsulas fixas são consideradas.
- Os diagramas de estrutura e de estados da cápsula são considerados.

- O diagrama de estados do protocolo não é tratado.
- Limitou-se o sinal do protocolo a ter uma mensagem para o uso de *monadic* π-calculus.
- A cardinalidade de cápsulas e de portas não são abordadas.
- Portas conectadas e não conectadas são consideradas.
- Não são contemplados os seguintes elementos: estados compostos (ou hierarquizados), ações de entrada (*entry action*) e saída (*exit action*) do estado, condição de guarda do gatilho, e pontos de junção históricos.

Dentre as hipótese e restrições colocadas acima, uma que merece destaque é o uso da sintaxe do HAL-JACK nas definições π-calculus geradas a partir do mapeamento. Segundo a sintaxe do HAL-JACK, todos os canais e as mensagens devem estar presentes nas definições π-calculus, seja na lista de entrada ou na lista de nomes restritos. Isso originou a complexidade para estabelecer as listas de entradas da cápsula e dos estados durante as regras de mapeamento apresentadas.

Outras considerações, mencionadas durante o texto, merecem destaque. São elas:

- A ação da transição inicial é considerada apenas quando possui dados de configuração inicial de portas não conectadas.
- Dentre as ações executadas numa transição entre estados, tem-se o envio de mensagem e a reconfiguração de nomes.
- A escolha determinística do ponto de escolha da UML-RT é representada como escolha não determinística do π-calculus.
- Considera-se que um dado estado sempre possui um conjunto único de entradas, não importando a rota usada para chegar a tal estado.

Uma consideração crítica mencionada acima é representar a escolha determinística do ponto de escolha da UML-RT como escolha não determinística do π-calculus, pois o mapeamento não considera a lógica contida no ponto de escolha que decide qual transição de saída será acionada.

6.2 Considerações sobre o protótipo

O desenvolvedor que usa o RoseRT, além de atentar ao escopo definido para o mapeamento, deve seguir algumas regras para especificar o modelo no RoseRT a fim de que este modelo possa ser traduzido corretamente pelo protótipo. Assim, o desenvolvedor deve fazer o seguinte:

- Construir o modelo com o framework RTC++ que permite o desenvolvimento de sistemas usando a linguagem C++.
- Indicar qual é a cápsula principal do sistema, configurando-a com o estereótipo *Top Level*.
- Definir atributos e outras variáveis com o sufixo *Att*, pois a ferramenta não permite que um atributo tenha nome igual ao de uma porta.
- Trabalhar sempre com atributos (ou outras variáveis) para receber e enviar mensagens. Lembrando que o código de recebimento de mensagem deve ser "*nomeAtt = *rtdata;*", onde *nomeAtt* é um atributo e **rtdata* é a forma de recuperar a mensagem recebida. O código de envio de mensagem deve ser "*p.s(nomeAtt).send();*", onde *p* é a porta por onde será enviado o sinal *s* com a mensagem *nomeAtt*.
- Representar a reconfiguração de nomes com o código "*nome1Att = nome2Att;*", onde *nome1Att* é o atributo que recebe o valor do outro atributo *nome2Att*. É importante

observar que, após a reconfiguração de nomes, *nome1Att* só pode ser usado na definição do próximo estado.

- Definir o sinal como tipo string (usando a classe proprietária *RTString*), quando a mensagem enviada é usada para configurar uma porta não conectada. Isso é necessário porque na ferramenta os métodos para realizar tal configuração requerem uma string como entrada. Embora o desenvolvedor use os códigos proprietários do RoseRT para configurar as portas SAP e SSP, é preciso indicar para o protótipo que está sendo feita uma reconfiguração de nomes. Esta indicação é realizada através do código "*p = nomeAtt;*", onde *p* é a porta renomeada para *nomeAtt*.
- Saber que os protocolos proprietários *Exception*, *External*, *Frame* e *Log* não foram tratados. Considerou-se somente o protocolo proprietário *Timing* que possibilita eventos de *timeout* representados com a ação invisível *tau* em π-calculus.
- Realizar a iniciação de portas não conectadas de subcápsulas através de mensagens enviadas por portas que implementem o protocolo *ConfigProtocol* definido neste trabalho. A configuração de porta não conectada é necessária quando o diagrama de estados não prevê um ponto de configuração da porta com base numa mensagem recebida. Neste caso, é preciso acrescentar a configuração inicial da porta no começo do diagrama de estados.

Dentre as recomendações acima, a mais crítica para o desenvolvedor é escrever os códigos nas transições da UML-RT de acordo com o requerido pelo protótipo, ou seja, sempre usando atributos ou outras variáveis. Outro ponto crítico para o desenvolvedor é entender quando e como empregar a configuração inicial de portas não conectadas.

6.3 Análise

A Tabela 17 mostra uma comparação entre o presente trabalho e os trabalhos correlatos. Esta tabela havia sido apresentada na Seção 2.6, porém sem as informações do presente trabalho.

Tabela 17: Comparação do presente trabalho com trabalhos correlatos

	Terriza et al [13]	**Fischer et al [15]**	**Engels et al [17]**	**Mota et al [18]**	**Assano [21]**	**Presente trabalho**
Álgebra de processos	CSP+T	CSP	CSP	Circus	CCS	π-calculus
Comunicação síncrona	Sim	Sim	Sim	Sim	Sim	Sim
Diagrama de estados	Sim	Não	Sim	Sim	Sim	Sim
Diagrama de estrutura	Não	Sim	Sim	Sim	Sim	Sim
Diagrama de classe	Sim	Não	Não	Sim	Não	Não
Diagrama de estados do protocolo	Não	Não	Sim	Sim	Não	Não
Reaproveitamento da definição da cápsula	Não	Não	Não	Sim	Não	Sim
Tipo de cápsula (fixa, opcional ou plug-in)	Fixa	Fixa	Fixa	Fixa	Fixa	Fixa
Porta conectada	Sim	Sim	Sim	Sim	Sim	Sim
Porta não conectada	Não	Não	Não	Não	Não	Sim
Cardinalidade de porta e cápsula	Não	Sim	Não	Sim	Não	Não
Condição de guarda	Sim	Não	Não	Sim	Não	Não
Ação de entrada e saída do estado	Não	Não	Não	Sim	Sim	Não
Estado composto ou hierarquizado	Sim	Não	Não	Sim	Não	Não
Ponto de junção histórico	Não	Não	Não	Não	Não	Não
Protótipo	Não	Não	Não	Não	Sim	Sim

As principais características de comparação na Tabela 17 são detalhadas abaixo:

- **Álgebra de processos**

A maioria das abordagens utiliza a álgebra de processos CSP, pois ela possui uma grande variedade de ferramentas para verificação formal. Vale ressaltar que a álgebra Circus é uma combinação de elementos de CSP e de Z.

O presente trabalho usa o π-calculus, uma vez que o π-calculus permite a reconfiguração dinâmica de nomes. A reconfiguração de nomes ou mobilidade é a principal característica do π-calculus e é uma grande vantagem em relação aos seus precursores CCS e CSP. A mobilidade foi usada para expressar dois pontos principais: reaproveitamento da definição da cápsula e porta não conectada. Estes pontos são mais bem explicados nos próximos tópicos.

- **Comunicação síncrona**

Todos os trabalhos mencionados na Tabela 17 consideram a comunicação como síncrona, pois as álgebras de processos se utilizam do modelo síncrono.

- **Diagrama de estados**

Esta característica indica se os trabalhos abordam o diagrama de estados de forma parcial ou completa. De acordo com a Tabela 17, somente Fischer et al [15] não tratam diagrama de estados, pois o escopo do seu trabalho foi o diagrama de estrutura. O mapeamento do diagrama de estados do presente trabalho possui diretrizes expostas por Engels et al [17] para escrita das definições dos estados de acordo com as informações de suas transições.

- **Diagrama de estrutura**

Esta característica indica se os trabalhos abordam o diagrama de estrutura de forma parcial ou completa. Terriza et al [13] não realizam mapeamento do diagrama de estrutura, mas obtêm as relações entre as cápsulas, dadas pelos conectores, a partir do diagrama de classe.

O mapeamento do diagrama de estrutura no presente trabalho é semelhante ao apresentado por Fischer et al [15] para portas conectadas. Fischer et al [15] renomeiam portas conectadas com o nome do conector. Entretanto, como o nome do conector não é provido pelo desenvolvedor no modelo UML-RT especificado no RoseRT, o presente trabalho optou por renomear uma porta com nome da outra porta a qual está conectada, ou gerar um novo nome dinamicamente para renomear as duas portas conectadas.

- **Diagrama de classe**

O diagrama de classes da cápsula não foi abordado no presente trabalho como indicado na Tabela 17. Terriza et al [13] adotam o diagrama de classe em alto nível somente para capturar as relações entre cápsulas conectadas, já que não utilizam diagrama de estrutura. O diagrama de classes é considerado por Mota et al [18] cujo escopo inclui não só a representação de elementos de comunicação, mas também a manipulação de atributos e chamadas a métodos.

- **Diagrama de estados do protocolo**

Assim como Fischer et al [15], o presente trabalho não tratou os protocolos explicitamente. O presente trabalho assume que os protocolos usados possuem apenas um sinal de entrada ou um de saída com o mesmo nome. Isso simplifica o mapeamento para π-calculus, à medida que a porta da UML-RT é representada pela porta do π-calculus; e a

mensagem contida no sinal do protocolo da UML-RT é representada pela mensagem do π-calculus.

Engels et al [17] mencionam o mapeamento do diagrama de estados do protocolo apenas num exemplo, mas não estabelecem regras gerais de mapeamento para este diagrama. Já Mota et al [18] mostram a representação do protocolo em termos da álgebra de processos Circus.

- **Reaproveitamento da definição da cápsula**

O reaproveitamento da definição da cápsula é possível com π-calculus devido a sua característica de mobilidade, provida pela reconfiguração dinâmica de nomes. Supondo que a cápsula-mãe *A* possui duas cápsulas-filhas do tipo *B*, a definição de *A* em π-calculus referencia duas vezes a definição de *B*, mas para cada uma delas passa variáveis de entrada diferentes. Em CSP ou CCS é necessário escrever duas definições para *B*, sendo uma para cada configuração de *B*.

Um exemplo disso em CSP pode ser visto no sistema de semáforos de trânsito proposto por Engels et al [17], que apresentam uma definição para cada semáforo. O presente trabalho descreve na Seção 5.2 um sistema semelhante com semáforos, onde há apenas uma definição π-calculus para o semáforo *trafficLight* e esta definição é referenciada duas vezes pela definição da cápsula-mãe *Intersection*.

- **Tipo de cápsula (fixa, opcional ou plug-in)**

De acordo com a Tabela 17, todos os trabalhos utilizam cápsula fixa, pois cápsula fixa é tipo padrão de cápsula e não necessita de códigos especiais para ser instanciada pela cápsula-mãe.

- **Porta não conectada**

Conforme colocado na Tabela 17, apenas o presente trabalho trata as portas não conectadas no mapeamento. A abordagem do presente trabalho foi entender que no RoseRT uma porta não conectada pode receber um nome durante a execução do diagrama de estados da cápsula, e passar a ser conhecida e acessada por este nome. A representação a configuração de portas não conectadas é possível devido à característica de mobilidade do π-calculus. Além disso, o presente trabalho propõe uma forma, chamada configuração inicial de porta não conectada, para a cápsula-mãe passar diretamente à subcápsula um nome para configurar uma porta não conectada da subcápsula.

- **Protótipo**

A Tabela 17 indica ainda se os trabalhos desenvolveram um protótipo para aplicar as regras de mapeamento e gerar automaticamente as definições das cápsulas na álgebra de processos escolhida.

Assano [21] desenvolveu um plug-in para RoseRT a fim de capturar as informações das cápsulas e gerar as definições CCS. Uma restrição deste protótipo é que há um alto acoplamento entre o componente que busca informações do RoseRT e o componente que gera as definições CCS, tornando o código mais difícil de ser entendido e reutilizado. Outra restrição deste protótipo é que não são capturadas informações do gatilho e da ação (código propriamente dito) das transições; Assano [21] parte do princípio que o desenvolvedor deve escrever, de uma forma preestabelecida, o que a transição representa no nome da própria transição.

A fim de diminuir o acoplamento entre os componentes do protótipo, o presente trabalho estabeleceu que o primeiro componente captura as informações do RoseRT e as disponibiliza em formato XML de acordo com um schema predefinido. Em seguida, o

segundo componente é acionado e trabalha com as informações do XML para gerar as definições π-calculus. Além disso, no presente trabalho as informações do RoseRT sobre gatilho e ação das transições são retiradas do RoseRT pelo protótipo, sem a necessidade do desenvolvedor ter de atentar para o nome da transição especificada.

Como o mapeamento proposto no presente trabalho prevê a utilização da sintaxe da ferramenta HAL-JACK nas definições π-calculus, as definições π-calculus geradas a partir do protótipo podem ser submetidas ao HAL-JACK para verificação formal de propriedades. É importante mencionar que, devido a restrições de tempo para estudo da lógica temporal pi-logic, usada para definir propriedades no HAL-JACK, restringiu-se o número de verificações formais realizadas nos exemplos propostos no presente trabalho.

6.4 Considerações finais

Este capítulo comentou as considerações tomadas neste trabalho tanto sobre o mapeamento quanto sobre o desenvolvimento do protótipo. Foi exposta também uma análise geral do trabalho, fazendo comparações com os trabalhos correlatos. O próximo capítulo descreve conclusões e apresenta os comentários finais.

Capítulo 7

Conclusões e Comentários finais

O trabalho apresentou uma semântica formal para UML-RT através do mapeamento dos elementos de comunicação da UML-RT para π-calculus. Trata-se de uma abordagem prática do π-calculus, o que ajuda a desmistificar a utilização de álgebras de processos no desenvolvimento de sistemas.

A UML-RT é uma extensão da UML amplamente empregada para modelagem de sistemas de tempo real. O mapeamento UML-RT para π-calculus, proposto pelo presente trabalho, permite que o desenvolvedor obtenha as definições π-calculus do modelo para realizar verificações formais. Assim, o desenvolvedor não necessita estabelecer um novo modelo diretamente com a álgebra de processos; ele pode usar o modelo UML-RT previamente especificado e mapeá-lo para π-calculus. Uma vantagem do mapeamento proposto é que as definições π-calculus utilizam a sintaxe da ferramenta HAL-JACK. Desse modo, de posse das definições π-calculus, o desenvolvedor pode usar o HAL-JACK para verificar formalmente propriedades do modelo.

A álgebra de processos π-calculus foi escolhida por ser o formalismo base para modelagem de sistemas concorrentes e por incluir o conceito de mobilidade provido pela reconfiguração dinâmica de nomes. Duas contribuições do presente trabalho são oriundas da mobilidade do π-calculus. A primeira contribuição foi o reaproveitamento da definição da

cápsula, que permite estabelecer apenas uma definição para a cápsula e referenciá-la quando necessário passando diferentes entradas. A segunda contribuição da mobilidade do π-calculus foi a representação de portas não conectadas, possibilitando que uma porta não conectada receba um novo nome e passe a ser conhecida e acessada por seu novo nome.

Outra contribuição importante do presente trabalho foi o desenvolvimento do protótipo de tradução de UML-RT para π-calculus com base no mapeamento apresentado. O protótipo automatiza o trabalho do desenvolvedor que especifica o modelo na ferramenta de mercado RoseRT e deseja obter as definições π-calculus do modelo. Além disso, o XML presente na arquitetura do protótipo diminui o acomplamento entre o componente que captura as informações do modelo no RoseRT e o componente que gera as definições π-calculus a partir dessas informações. O baixo acoplamento entre os componentes provê maior reusabilidade do componente diretamente ligado ao RoseRT, facilitando o desenvolvimento de novas funcionalidades para o RoseRT.

Como trabalho futuro em relação ao mapeamento, deseja-se estudar a representação dos elementos que não fizeram parte do escopo desse trabalho, por exemplo, os estados compostos e a cardinalidade de portas e cápsulas. Há ainda o intuito de estudar o método formal TLA (*Temporal Logic of Actions*) [55] a fim de propor o mapeamento de UML-RT para TLA que incorpore não só elementos de comunicação, mas também o tratamento de dados. Em relação a trabalhos futuros sobre o protótipo, há interesse em aprimorá-lo para ser independente de sintaxe do π-calculus; a idéia é poder escolher a sintaxe desejada para as definições π-calculus, que neste trabalho foi a da ferramenta HAL-JACK.

Referências

[1] OMG (Object Management Group). **OMG Unified Modeling Language Specification**. OMG document formal/03-03-01, 2003.

[2] BAETEN, J; BERGSTRA, J. A. Six Issues Concerning Future Directions in Concurrency Research. In: ACM COMPUTING SURVEYS (CSUR), 1996. New York: ACM Press, vol. 28, issue 4, 1996.

[3] BRYSON, K. Concurrency and Threads. Disponível em: <http://www.cs.ucl.ac.uk/staff/K.Bryson/teaching/course2007_05.html>. Acesso em: 01/10/2006.

[4] PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Pearson Makron Books, 2005, p. 633-670. Título original: Software Engineering: A Practitioner's Approach.

[5] NIST (National Institute of Standards and Technology). **Dictionary of Algorithms and Data Estruturas**. Disponível em: <http://www.nist.gov/dads/>. Acesso em: 01/10/2006.

[6] KRÖNING, D. Formal Verification Special Course, 2005. Dept. Computer Science, ETH Zürich Computer Systems Institute. Disponível em: <http://www.inf.ethz.ch/~daniekro>. Acesso em: 05/10/2006.

[7] GHERBI, A; KHENDEK, F. UML Profiles for Real-Time Systems and their Applications. Journal of Object Technology, vol. 5, n^{o} 3, 2006.

[8] SELIC, B; RUMBAUGH, J. Using UML for Modeling RealTime Systems. Rational Software Corporation, 1998.

[9] IBM. Rational Rose RealTime. Version 6.5. CD-ROM.

[10] FERNANDES, J. M.; PINTO, L. F; RIBEIRO, O. R. Model Checking Embedded System with PROMELA. In: ENGINEERING OF COMPUTER-BASED SYSTEMS (ECBS), 2005, Greenbelt, MD, USA.

[11] CORTÉS, L. A; ELES, P; PENG, Z. Verification of Embedded Systems using a Petri Net based Representation. In International Symposium on System Level Synthesis (ISSS), 13., Madrid, Spain, 2000.

[12] VENKATESH, R; LIU, Z. Methods and Tools for Formal Software Engineering. UNU_IIST Annual Report, 2004.

[13] TERRIZA, J. A. H; AKHLAKI, K. B; TUÑÓN, M. I. C. Combining the Description Features of UML-RT and CSP+T Specifications Applied to a Complete Design of Real-Time Systems. International Journal of Information Technology, vol. 2, nº 3, 2005.

[14] ZIC, J. J. Timed constrained buffer specifications in CSP+T and timed CSP. In: ACM TRANSACTION ON PROGRAMMING LANGUAGES AND SYSTEMS (TOPLAS), 1994. New York: ACM Press, vol. 6, 1994.

[15] FISCHER, C; OLDEROG, E. –R; WEHRHEIM, H. A CSP View on UML-RT Estrutura Diagrams". In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING (FASE), 4., 2001.

[16] HOARE, C. A. R. **Communicating Sequential Processes**. Prentice Hall, 1985.

[17] ENGELS, G; KÜSTER, J. M.; HECKEL, R; GROENEWEGEN, L. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE (ESEC), 8., ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE-9), 9., Vienna, Áustria, 2001.

[18] MOTA, A; SAMPAIO, A; RAMOS, R. A Semantics for UML-RT Active Classes via Mapping into Circus. In: IFIP WG 6.1 INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, 7., 2005.

[19] CAVALCANTI, A; SAMPAIO, A; WOODCOCK, J. Refinement in Circus. In: INTERNATIONAL SYMPOSIUM OF FORMAL METHODS EUROPE, 2002. Springer, 2002, vol. 2391 LNCS.

[20] SPIVEY, M. **The Z Notation: A Reference Manual**. Prentice Hall, 1992.

[21] ASSANO, M. E. V; HIRATA, C. M. Translation of UML-RT models to CCS applied to Static Model Verification. 2006. Não publicado.

[22] MILNER, R. **Communication and Concurrency.** Prentice Hall, 1989.

[23] MILNER. R. **Communicating and Mobile Systems: The π-calculus**. Cambridge University Press, 1999.

[24] FERRARI, G; PISTORE, M; GNESI, S; MONTANARI, U. A Model Checking Verification Environment for Mobile Processes. In: ACM TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY, 2004. New York: ACM Press, vol. 12, issue 4, pp. 440-473, 2004. Disponível em: <http://fmt.isti.cnr.it:8080/hal>.

[25] LEE, E. A. Embedded Software. Em Academic in Computers, vol. 56, M. Zelkowitz, Ed. Academic Press, 2002.

[26] CARLSON, J. Languages and methods for specifying real-time systems. MRTC report, Mälardalen Real-Time Research Centre, Mälardalen University, August, 2002.

[27] PTOLEMY Project. Disponível em: <http://ptolemy.eecs.berkeley.edu/>.

[28] QUACK Project: A Platform for the Quality of New Generation Integrated Embedded Systems. Disponível em: <http://www.lta.disco.unimib.it/quack/>.

[29] WING, J. M. FAQ on π-calculus. Microsoft Research, 2002.

[30] BAETEN, J. C. M. A Brief History of Process Algebra. In: Workshop on Process Algebra: Open Problems and Future Directions, 2005. Forli, Itália. **Anais…** Forli, Itália: vol. 335, n^{o} 2-3, p. 131-146.

[31] TARKOMA, S. Specification Languages and Their Use (Case: AsmL). In: SEMINAR ON GENERATIVE PROGRAMMING. University of Helsinki, Department of Computer Science, 2003. Disponível em: <http://www.cs.helsinki.fi/u/vihavain/k03/sem/spec_lang.pdf>.

[32] MICROSOFT. Behave! project. Disponível em: <http://research.microsoft.com/behave>.

[33] BOWEN, J. P; HINCHEY, M. G. Seven more myths of formal methods. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE (ESEC), 5., Barcelona,

1995. Disponível em: <http://www.cs.uidaho.edu/~jimaf/FMSeminar/seven%20more%20myths.pdf>.

[34] THOMAS, M. **Formal methods and their role in developing safe systems**. High Integrity Systems Journal. Oxford University Press, vol. 1, nº 5, p. 447-451, 1996.

[35] CALDER, M. What Use are Formal Design and Analysis Methods to Telecommunications Services?. In: INTERNATIONAL WORKSHOP ON FEATURE INTERACTIONS IN TELECOMMUNICATIONS AND SOFTWARE SYSTEMS, Lund, Suécia, 1998. **Anais…** Suécia: IOS Press, vol. 5, p. 10-31. Disponível em: <http://www.dcs.gla.ac.uk/~muffy/papers/mtfiw98.ps>.

[36] PARROW, J. **An Introduction to the pi-Calculus**. No livro BERGSTRA; PONSE; SMOLKA. Handbook of Process Algebra. Ed. Elsevier, p. 479-543, 2001.

[37] WISCHIK, L. New directions in implementing the pi-calculus. In: The CaberNet Radicals Workshop, 2002.

[38] ZILIO, S. D. Mobile Processes: a commented bibliography. Modeling and verification of parallel processes. New York: Springer-Verlag, p. 206-222, 2001.

[39] PIERCE, B. C; TURNER, D. N. Pict: A Programming Language Based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.

[40] PALAMIDESSI, C. **Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus**. Mathematical Estruturas in Computer Science. 13(5), p. 685-719, 2003.

[41] POWER, J; SINCLAIR, D. A Formal Model of Forth Control Words in the Pi-Calculus. In: International Workshop on Formal Methods for Industrial Critical Systems, 6., Paris, França, 2001.

[42] REGEV, A; SILVERMAN, W; SHARIPO, E. Representation and Simulation of Biochemical Processes Using the π-Calculus Process Algebra. In: Pacific Symposium on Biocomputing, p. 459-470, 2001.

[43] ABADI, B; GORDON, D. G. A Calculus for Cryptographic Protocols The Spi Calculus. In: ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 4., 1997.

[44] CARDELLI, L. GORDON, D. G. Mobile Ambients. Lecture Notes in Computer Science (LNCS). Springer Berlin / Heidelberg, vol. 1378, 1998.

[45] FOURNET, C; GONTHIER, G. The Join Calculus: a Language for Distributed Mobile Programming. Applied Semantics. International Summer School (APPSEM), Caminha, Portugal, 2000. Portugal: Springer-Verlag, vol. 2395,p. 268-332, 2000.

[46] VICTOR, B. **The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes**. PhD Thesis. Uppsala University, 1998.

[47] BOCCHI, L; WISCHIK, L. A Process Calculus of Atomic Commit. In: International Workshop on Web Services and Formal Methods (ENTCS), 2004.

[48] BOCCHI, L; LANEVE, C; ZAVATTARO, G. A Calculus for Long Running Transactions. In: International Conference on Formal Methods for Open Object-based Distributed Systems (IFIP), 6. LNCS 2884, p. 124-138, 2003.

[49] MICROSOFT. Visual Basic. Disponível em: < http://support.microsoft.com/default.aspx?pr=vbb >.

[50] SUN. Java EE 5 SDK. Disponível em: <http://java.sun.com/>.

[51] GOPINATH, S. Real-Time UML To XMI Conversion. Master of Science Thesis, KTH Computer Science and Communication, Stockholm, Sweden, 2006.

[52] ECLIPSE. Eclipse SDK. Version 3.1.2. Disponível em: <http://www.eclipse.org>.

[53] OMONDO. Omondo EclipseUML. Disponível em: <http://www.omondo.com>

[54] SUN. Java Architecture for XML Binding (JAXB). Disponível em: <http://java.sun.com/webservices/jaxb/>.

[55] LAMPORT, L. Introduction to TLA. SRC Technical Note, 1994. Disponível em: <http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1994-001.html>.

Anexos

Anexo A: Schema do XML para elementos UML-RT

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2005 sp1 U (http://www.xmlspy.com) by Juliana (Ru-Board) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:element name="UMLRTModel">
<xs:annotation>
<xs:documentation>Comment describing your root element</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element name="capsuleList">
<xs:complexType>
<xs:sequence maxOccurs="unbounded">
<xs:element name="capsule" type="Capsule"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:complexType name="AttributeList">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="attribute" type="xs:string">
<xs:annotation>
<xs:documentation>Attribute name</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="Event">
<xs:sequence>
<xs:element name="port" type="xs:string">
<xs:annotation>
<xs:documentation>Port that can receive the signal</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="protocol" type="xs:string">
<xs:annotation>
<xs:documentation>Protocol of the port that can receive the signal</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="signal" type="xs:string">
<xs:annotation>
<xs:documentation>Signal to be received</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="Triggers">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="event" type="Event">
<xs:annotation>
<xs:documentation>Each event that can trigger an transition</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="Transition">
<xs:sequence>
<xs:element name="name" type="xs:string">
<xs:annotation>
<xs:documentation>Transition name</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="source" type="xs:string">
<xs:annotation>
<xs:documentation>Name of source estados</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="target" type="xs:string">
<xs:annotation>
<xs:documentation>Name of target estados</xs:documentation>
</xs:annotation>
</xs:element>
<xs:sequence minOccurs="0">
<xs:element name="triggers" type="Triggers">
<xs:annotation>
<xs:documentation>List of events</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
<xs:element name="action" type="xs:string" minOccurs="0">
<xs:annotation>
<xs:documentation>Transition action</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="TransitionList">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="transition" type="Transition">
<xs:annotation>
<xs:documentation>Each transition in estados diagram</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="Estados">
<xs:sequence>
<xs:element name="name" type="xs:string">
<xs:annotation>
<xs:documentation>Estados name</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="type" type="xs:string">
<xs:annotation>
<xs:documentation>Estados type. Eg: CompositedEstados and ChoicePoint</xs:documentation>
</xs:annotation>
```

```
</xs:element>
<xs:element name="outgoingTransitions" type="TransitionList"/>
<xs:element name="incomingTransitions" type="TransitionList"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="EstadosDiagram">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="estados" type="Estados">
<xs:annotation>
<xs:documentation>Each estados in estados diagram</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="CapsuleRole">
<xs:sequence>
<xs:element name="name" type="xs:string">
<xs:annotation>
<xs:documentation>Sub-capsule name for the parent capsule</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="capsule" type="xs:string">
<xs:annotation>
<xs:documentation>Capsule type of the sub-capsule</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="CapsuleRoleList">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="capsuleRole" type="CapsuleRole">
<xs:annotation>
<xs:documentation>Each sub-capsule in estrutura diagram</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="Connector">
<xs:sequence>
<xs:element name="portRole1" type="xs:string">
<xs:annotation>
<xs:documentation>Port name of the first connected sub-capsule</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="capsuleRole1" type="xs:string">
<xs:annotation>
<xs:documentation>Capsule name of the first connected sub-capsule</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="portRole2" type="xs:string">
<xs:annotation>
<xs:documentation>Port name of the second  connected sub-capsule </xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="capsuleRole2" type="xs:string">
<xs:annotation>
<xs:documentation>Capsule name of the second  connected sub-capsule</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
```

```
</xs:complexType>
<xs:complexType name="ConnectorList">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="connector" type="Connector">
<xs:annotation>
<xs:documentation>Each connector between ports of sub-capsules </xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="Port">
<xs:sequence>
<xs:element name="name" type="xs:string">
<xs:annotation>
<xs:documentation>Port name</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="conjugated" type="xs:boolean">
<xs:annotation>
<xs:documentation>It can be true or false</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="termination" type="xs:string">
<xs:annotation>
<xs:documentation>It can be End or Relay</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="type" type="xs:string">
<xs:annotation>
<xs:documentation>It can be Wired or Unwired </xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="visibility" type="xs:string">
<xs:annotation>
<xs:documentation>It can be Protected or Public</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="publish" type="xs:boolean">
<xs:annotation>
<xs:documentation>It can be true or false</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="PortList">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="port" type="Port">
<xs:annotation>
<xs:documentation>Each capsule port. Do not consider port that implements RoseRT default protocols (eg
Timing and Log) and ConfigProtocol</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="EstruturaDiagram">
<xs:sequence>
<xs:element name="capsuleRoleList" type="CapsuleRoleList">
<xs:annotation>
<xs:documentation>List of sub-capsules</xs:documentation>
</xs:annotation>
```

```
</xs:element>
<xs:element name="connectorList" type="ConnectorList">
<xs:annotation>
<xs:documentation>List of connectors between sub-capsules</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="internalConnectorList" type="InternalConnectorList">
<xs:annotation>
<xs:documentation>List of internal connectors</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="configConnectorList" type="ConfigConnectorList">
<xs:annotation>
<xs:documentation>List of configuration connectors</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="portList" type="PortList">
<xs:annotation>
<xs:documentation>List of ports</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="Capsule">
<xs:sequence>
<xs:element name="name" type="xs:string">
<xs:annotation>
<xs:documentation>Capsule name</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="visibility" type="xs:string">
<xs:annotation>
<xs:documentation>Capsule visibility: Public, Protected, Private or Implementation</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="stereotype" type="xs:string">
<xs:annotation>
<xs:documentation>Capsule steriotype. For the main capsule, it is Top Level</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="attributeList" type="AttributeList">
<xs:annotation>
<xs:documentation>List of capsule attributes</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="estadosDiagram" type="EstadosDiagram"/>
<xs:element name="estruturaDiagram" type="EstruturaDiagram"/>
<xs:element name="initialConfig" type="xs:string" minOccurs="0">
<xs:annotation>
<xs:documentation>Action in Initial transition regarding initial configuration of unwired
ports</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="ConfigConnector">
<xs:sequence>
<xs:element name="port1" type="xs:string">
<xs:annotation>
<xs:documentation>Name of the parent capsule port</xs:documentation>
```

```
</xs:annotation>
</xs:element>
<xs:element name="portRole2" type="xs:string">
<xs:annotation>
<xs:documentation>Name of the sub-capsule port</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="capsuleRole2" type="xs:string">
<xs:annotation>
<xs:documentation>Capsule name of the sub-capsule</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="ConfigConnectorList">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="connector" type="ConfigConnector">
<xs:annotation>
<xs:documentation>Each connector between a port of the parent capsule and a port of a sub-capsule. These ports
are used to do the initial configuration of unwired ports </xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="InternalConnector">
<xs:sequence>
<xs:element name="port1" type="xs:string">
<xs:annotation>
<xs:documentation>Name of the parent capsule port</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="portRole2" type="xs:string">
<xs:annotation>
<xs:documentation>Name of the sub-capsule port</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="capsuleRole2" type="xs:string">
<xs:annotation>
<xs:documentation>Capsule name of the sub-capsule</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="InternalConnectorList">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="connector" type="InternalConnector">
<xs:annotation>
<xs:documentation>Each connector between a port of the parent capsule and a port of a sub-
capsule</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:schema>
```

Anexo B: Manual de instalação do protótipo

Para instalar o protótipo de conversão UML-RT para π-calculus no sistema operacional Windows, primeiramente é necessário instalar o *Rational Rose RealTime* (RoseRT), cuja versão usada no presente trabalho foi a 2003.06.15.734.000. Para compilar e rodar modelos com código C++, precisa-se instalar também o *Microsoft Visual C++ 6.0.* Sugere-se que se tenha um domínio básico da ferramenta RoseRT para criação de modelos (*Logical View*), criação de componente para compilação do modelo (*Component View*) e criação do deploy (*Deployment view*) para o componente desejado. Com isso é possível criar os modelos em UML-RT e rodá-los para verificar sua execução.

Os próximos passos para instalação do protótipo são:

- Instalar o Java 1.5;
- Criar o diretório local “C:\JUJUBA\Mestrado” designado por <diretório>;
- Descompactar o arquivo “Prototype.rar” em <diretório>;
- Clicar duas vezes para abrir o arquivo “<diretório>\Prototype\VB\rosert_plugin.reg”. Este arquivo faz o registro da *dll* em VB no Windows. Além disso, este arquivo indica, através do arquivo “rosert_plugin.mnu”, o nome do plug-in "Pi-Calculus Add-in” e a partir de qual menu (no caso, o menu *Tools*) o plug-in será acessado. Vale lembrar que o plug-in é o componente VB.

Para testar o protótipo, basta executar os passos:

- Descompactar o arquivo “RoseRTExamples.rar” em <diretório>. Este arquivo contém os exemplos descritos neste trabalho: recebimento e envio de mensagem (pasta *SendReceiveData*), Buffer com capacidade dois (pasta *Buffer*), semáforos de trânsito (pasta *TrafficLight*s) e *Handover* (pasta *HandoverProtocol*);
- Iniciar o RoseRT e cliar em File > Open;
- Escolher o exemplo desejado. No caso do exemplo básico de recebimento e envio de mensagem, escolher “<diretório>\SendReceive\SendReceiveData.rtmdl”;
- Clicar em Tools > Pi-Calculus Add-in;
- Clicar em Start na tela Pi-Calculus Add-in que aparecer, como mostra a Figura 61. O componente VB executado, salva o XML em “C:\JUJUBA\Mestrado\Prototype\UMLRTModel.xml” e chama automaticamente o componente Java para gerar as definições π-calculus conforme Figura 62.

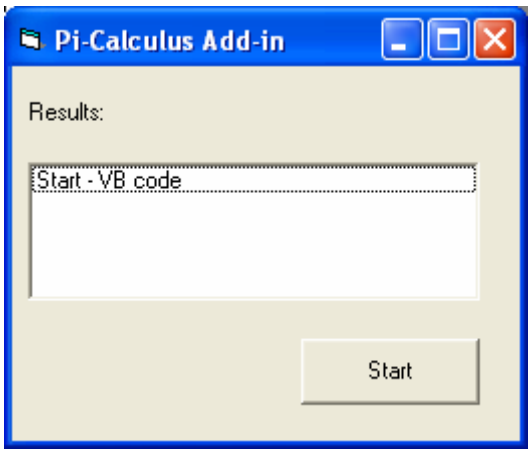


Figura 61: Tela inicial do protótipo.

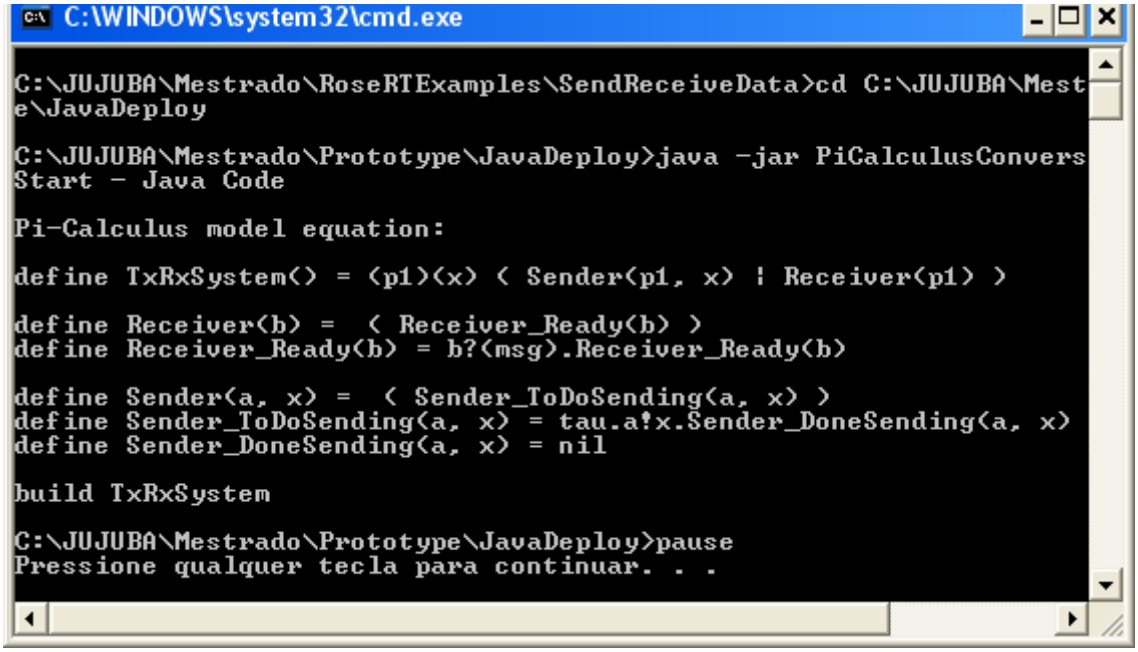


Figura 62: Tela com o resultado gerado pelo protótipo.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO N°	4. N° DE PÁGINAS
TM	27 de Dezembro de 2006	CTA/ITA-IEC/TM-010/2006	129

5. TÍTULO E SUBTÍTULO:

Mapeamento UML-RT para π-calculus

6. AUTOR(ES):

Juliana de Melo Bezerra

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica. Divisão de Ciência da Computação – ITA/IEC

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

1. UML-RT; 2. π-calculus; 3. HAL-JACK; 4. Álgebra de processos; 5. Semântica formal; 6. Verificação formal

9.PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

UML; Pi-cálculo; Álgebra de processos; Verificação formal; Semântica de linguagem de programação; Modelagem (processo); Operação em tempo real; Programação orientada para objetos; Desenvolvimento de software; Computação

10. APRESENTAÇÃO: **X Nacional** **Internacional**

ITA, São José dos Campos, 2006, 129 páginas

11. RESUMO:

A UML (*Unified Modeling Language*) é uma linguagem de modelagem para especificar, construir e documentar artefatos de sistemas de software. A UML-RT, usada pela ferramenta Rational Rose RealTime (RoseRT), é uma extensão da UML que permite a modelagem de sistemas de tempo real distribuídos e guiados por evento. A UML-RT não possui semântica formal, logo não é possível realizar verificação formal do modelo. O presente trabalho propõe o mapeamento dos elementos de comunicação da UML-RT para a álgebra de processos π-calculus, a fim de prover semântica formal à UML-RT. Com objetivo de automatizar o mapeamento, foi desenvolvido um protótipo de tradutor que captura o modelo UML-RT especificado na ferramenta RoseRT e determina suas definições π-calculus. As definições π-calculus geradas utilizam a sintaxe da gramática do HAL-JACK, que é uma ferramenta integrada para verificação e análise de sistemas expressos em π-calculus, assim as definições π-calculus podem ser submetidas ao HAL-JACK para verificação formal de propriedades. Este trabalho detalha o mapeamento UML-RT para π-calculus, descreve o protótipo desenvolvido e apresenta alguns exemplos do mapeamento do modelo UML-RT para definições π-calculus.

12. GRAU DE SIGILO:

(X) OSTENSIVO **( ) RESERVADO** **( ) CONFIDENCIAL** **( ) SECRETO**

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo