RODRIGO TACLA SAAD

Elementos para a Construção de uma Cadeia de Verificação para o Projeto TOPCASED

FLORIANÓPOLIS
2008

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO
EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Elementos para a Construção de uma Cadeia de Verificação para o Projeto TOPCASED

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia de Automação e Sistemas.

RODRIGO TACLA SAAD

Florianópolis, 02 de Abril de 2008.

Elementos para a Construção de uma Cadeia de Verificação para o Projeto TOPCASED

Rodrigo Tacla Saad

'Esta dissertação foi julgada adequada para a obtenção do título de **Mestre em Engenharia de Automação e Sistemas**, área de concentração **Controle, Automação e Sistemas**, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia de Automação e Sistemas (PPGEAS) da Universidade Federal de Santa Catarina.'

Florianópolis, 02 de abril de 2008

Jean-Marie Farines, Prof. Dr.
Orientador

François Vernadat, Prof. Dr.
Co-orientador

Eugênio de Bona Castelan Neto, Prof. Dr.
Coordenador do Programa PPGEAS

Banca Examinadora:

Jean-Marie Farines, Prof. Dr., Orientador

Aline Maria Santos Andrade, Prof. Dr

Leandro Buss Becker, Prof. Dr.

Olinto José Varela Furtado, Prof. Dr.

Rômulo Silva de Oliveira, Prof. Dr.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia de Automação e Sistemas.

Elementos para a Construção de uma Cadeia de Verificação para o Projeto TOPCASED

Rodrigo Tacla Saad

Abril/2008

Orientador: Jean-Marie Farines, Prof. Dr.
Área de Concentração: Controle, Automação e Sistemas
Palavras-chave: Verificação Formal, Métodos Formais, Projeto TOPCASED, SDL, UML
Número de Páginas: 160

Analisando a história dos sistemas embarcados, podemos dividí-la em dois momentos. Em um primeiro momento, a origem dos problemas destes sistemas provinha, na grande maioria dos casos, da parte física chamada hardware. A partir dos anos 60, graças à chegada dos circuitos integrados, desenvolvidos para o Programa Espacial Americano, a parte física dos sistemas se tornou mais confiável. Nós últimos 20 anos, devido à complexidade inerente ao desenvolvimento dos softwares para sistemas embarcados, estes se tornaram a origem da maior parte dos erros. Uma das grandes dificuldades no desenvolvimento destes softwares é assegurar um funcionamento correto (de acordo com as especificações). A fim de reduzir a incidência de erros, a indústria passou a estudar o uso de métodos formais para auxiliar o desenvolvimento destes sistemas complexos. Estas técnicas auxiliam o processo decisório porque permitem afirmar antes de implementar o protótipo se uma dada especificação será cumprida ou não pelo sistema.

Entretanto, a verificação formal ainda não é largamente empregada no ambiente industrial devido à dificuldade no intercâmbio de informações entre as linguagens de modelagem de alto nível (UML, AADL, SDL, etc) e as ferramentas de verificação formal. Esta dificuldade é decorrente da falta de uma semântica formal para estas linguagens de modelagem largamente utilizadas pela indústria. Além disto, cada ferramenta de verificação trabalha com formalismos matemáticos diferentes, não havendo uma fácil integração entre elas. Outro fator importante é que não podemos afirmar que existe um formalismo único capaz de atender a todas as necessidades de um sistema complexo. Isto implica que sistemas futuros vão cada vez mais requerer uma combinação de métodos baseados em modelos, tais como sistema de transição, álgebra de processos, lógica temporal, entre outros. Estas restrições impõem a indústria a necessidade de desenvolver uma ferramenta de tradução de modelos para cada par linguagem-formalismo empregado.

A fim de facilitar este intercâmbio de informações entre as diferentes linguagens de modelagem e as ferramentas de verificação formais existentes - tais como TINA (Time Petri Net Analyser), CADP (Construction and Analysis of Distributed Processes), entre outras - o projeto TOPCASED (Toolkit in Open-Source for Critical Application & SystEms Development) desenvolveu uma arquitetura de verificação original, que promove a transformação de

modelos entre os diferentes níveis. Esta transformação é simplificada pelo advento de uma linguagem intermediária formal chamada FIACRE (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués).

Dentro deste contexto, as atividades desenvolvidas neste trabalho fazem parte da especificação e operacionalização da linguagem FIACRE do projeto TOPCASED. A primeira atividade apresentada neste trabalho é o estudo preliminar da tradução entre SDL e FIACRE para auxiliar na especificação da linguagem FIACRE. A segunda atividade consiste inicialmente na proposição de um esquema conceitual para a tradução de FIACRE para o formalismo matemático TTS (Sistema de Transições Temporizados), e posteriormente na sua implementação na forma de um compilador (*front-end*) para a ferramenta TINA. Por último, um exemplo de verificação de sistema é apresentado com o intuito de demonstrar as vantagens das ferramentas que fazem parte do projeto TOPCASED.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Esta dissertação faz parte dos esforços do projeto TOPCASED [1] para fornecer uma plataforma completa de desenvolvimento, que acompanhe os projetistas desde a especificação do sistema até a sua concepção.

Este capítulo apresenta o contexto no qual este trabalho está inserido. A primeira seção apresenta uma visão geral do contexto do trabalho. A segunda seção apresenta a motivação principal desta dissertação, que é a verificação formal de sistemas. O projeto TOPCASED é exposto na terceira seção. Os objetivos e a organização desta dissertação são apresentados nas seções 1.4 e 1.5.

1.1 Introdução

No mundo moderno é cada vez mais comum a presença de sistemas embarcados na vida cotidiana. Porém, uma parte destes sistemas é dito críticos quando falhas ou funcionamentos inadequados podem resultar em ferimentos ou até mesmo morte para as pessoas envolvidas, danos irreversíveis ao equipamento ou catástrofes ambientais. Alguns exemplos de sistemas críticos que falharam tragicamente levando a grande perda (humanas ou de recursos):

- Therac26 (1985-1987): Entre junho de 1985 e janeiro de 1987, uma máquina de radioterapia controlada por computador, chamada de Therac-25, aplicou doses além do indicado em seis pacientes devido a um problema de codificação do software. [6].

[1] Toolkit In OPen source for Critical Applications & SystEms Development

- Ariane 5 (1996): Em 4 de junho de 1996, o lançamento inaugural do foguete europeu Ariane 5 terminou numa *falha de grande importância*. Esta falha foi causada por uma completa perda de informação sobre a guidagem e altitude. Esta perda de informação foi devido a erros de especificação e projeto no software do sistema de referência inercial [7].

- Nasa Mars Pathfinder (1997): O sistema foi reiniciado devido a um problema de inversão de prioridade e acarretou em atrasos de retransmissão de dados, perdendo tempo precioso de uma missão [8].

Em [9] podemos ver uma lista de incidentes que é atualizada anualmente.

Analisando a história dos sistemas embarcados, podemos dividi-la em dois momentos. Em um primeiro momento, a origem dos problemas destes sistemas provinha, na grande maioria dos casos, da parte física chamada hardware. A partir dos anos 60, graças à chegada dos circuitos integrados, desenvolvidos para o Programa Espacial Americano, a parte física dos sistemas se tornou mais confiável. Nós últimos 20 anos, devido à complexidade inerente ao desenvolvimento dos softwares para sistemas embarcados, estes se tornaram a origem da maior parte dos erros [10]. Uma das grandes dificuldades no desenvolvimento destes softwares é assegurar um funcionamento correto (de acordo com as especificações). A fim de reduzir a incidência de erros, a indústria passou a estudar o uso de métodos formais para auxiliar o desenvolvimento destes sistemas complexos. Estas técnicas auxiliam o processo decisório porque permitem afirmar antes de implementar o protótipo se uma dada especificação será cumprida ou não pelo sistema.

Entretanto, a verificação formal ainda não é largamente empregada no ambiente industrial devido à dificuldade no intercâmbio de informações entre as linguagens de modelagem de alto nível (UML, AADL, SDL, etc) e as ferramentas de verificação formal. Esta dificuldade é decorrente da falta de uma semântica formal para estas linguagens de modelagem largamente utilizadas pela indústria. Além disto, cada ferramenta de verificação trabalha com formalismos matemáticos diferentes, não havendo uma fácil integração entre elas. Outro fator importante é que não podemos afirmar que existe um formalismo único capaz de atender a todas as necessidades de um sistema complexo. Isto implica que sistemas futuros vão cada vez mais requerer uma combinação de métodos baseados em modelos, tais como sistema de transição, álgebra de processos, lógica temporal, entre outros [11]. Estas restrições impõem a necessidade de desenvolver uma ferramenta de tradução de modelos para cada par de linguagem-formalismo empregado (ver Figura 1.1).

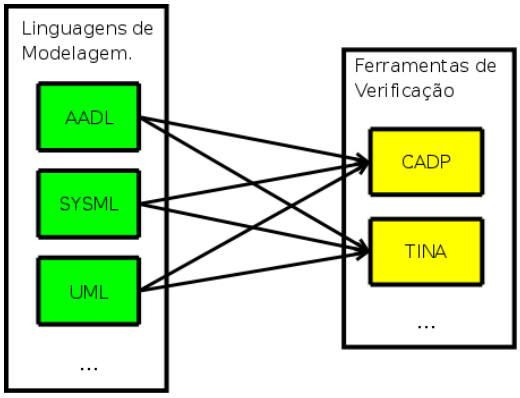


Figura 1.1: *Acoplamento Linguagem-Formalismo.*

A fim de facilitar este intercâmbio de informações entre as diferentes linguagens de modelagem e as ferramentas de verificação formais existentes - tais como TINA[2] [4], CADP[3] [12], entre outras - o projeto TOPCASED desenvolveu uma arquitetura de verificação original, que promove a transformação de modelos entre os diferentes níveis. Esta transformação é simplificada pelo advento de uma linguagem intermediária formal chamada FIACRE[4] (ver Figura 1.2). A partir desta abordagem, este intercâmbio pode ser substancialmente simplificado por duas razões:

- somente uma ferramenta de acoplamento entre uma linguagem de modelagem e FIACRE permite utilizar todas as ferramentas de verificação formal aceitas pela plataforma, e vice-versa;
- o acoplamento de novas ferramentas de verificação é mais simples porque FIACRE, sendo uma linguagem intermediária, não possui construções tão complexas em comparação com as linguagens de modelagem de alto nível.

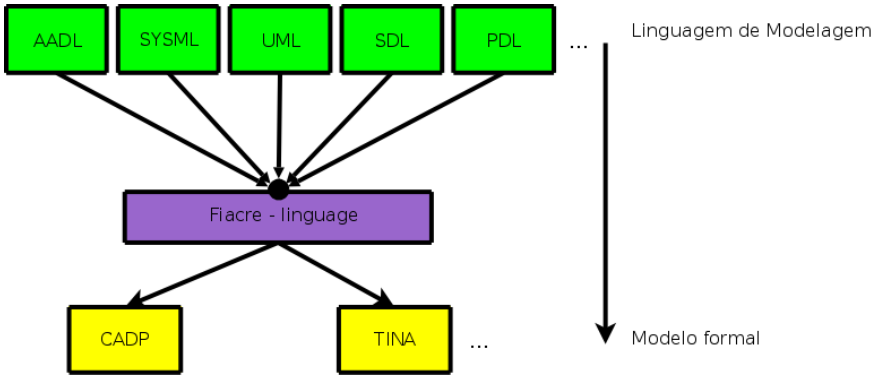


Figura 1.2: *Linguagem Intermediária* FIACRE *[1].*

[2]Time Petri Net Analyser
[3]Construction and Analysis of Distributed Processes
[4]Format Intermédiaire pour les Architectures de Composants Répartis Embarqués

Dentro deste contexto, as atividades desenvolvidas neste trabalho fazem parte da especificação e operacionalização da linguagem FIACRE do projeto TOPCASED. Este trabalho foi financiado pelo programa de cooperação entre França e Brasil (CAPES-COFECUB), tendo suas atividades desenvolvidas em parte no Laboratoire d'Analise et d'Architecture des Systèmes do CNRS(LAAS-CNRS), e em parte no Departamento de Automação e Sistemas da UFSC (DAS-UFSC). As seções seguintes fornecem maiores esclarecimentos em relação ao contexto do trabalho, do projeto do qual faz parte e de seus objetivos.

1.2 Verificação Formal

Nos últimos anos, a indústria começou a estudar a possibilidade de empregar métodos de verificação formal para reduzir o número de erros no desenvolvimento de sistemas. Por métodos formais, entendemos neste trabalho "toda técnica que permite confrontar um sistema (sua descrição operacional) a suas especificações (propriedades esperadas)" cf. Figura 1.3 [2].

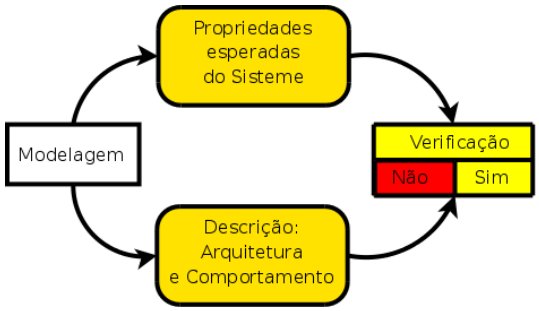


Figura 1.3: *Processo de Verificação Formal[2].*

Este processo de verificação é formado por três componentes:

- uma descrição operacional do sistema.
- uma linguagem de especificação que permita expressar as propriedades do sistema que desejamos verificar;
- um procedimento de decisão que permite verificar que o sistema satisfaz efetivamente as propriedades esperadas;

A linguagem de especificação e o procedimento de decisão são fortemente acoplados. Existem duas abordagens clássicas:

- Comportamental: Nesta abordagem, tanto a descrição operacional do sistema (grafo de comportamento associado) como as propriedades especificadas são ambas expressas na forma de comportamento. O procedimento de decisão consiste em provar a equivalência entre os dois grafos.

- Lógica: nesta abordagem, as propriedades são expressas em linguagens específicas, por exemplo, a lógica temporal. Neste caso, a descrição operacional do sistema é considerada como um modelo para a verificação através desta lógica. O procedimento de decisão consiste em efetuar um controle do modelo ("*Model Checking*") associando a cada formula lógica (propriedade) um conjunto de estados do modelo da descrição operacional que as satisfazem.

A automação destes procedimentos consiste em verificar através de algoritmos se um dado modelo formal satisfaz as propriedades atendidas. Quando uma propriedade não é verificada, a ferramenta de verificação fornece um contra exemplo para ajudar o usuário a encontrar o erro e corrigi-lo.

1.3 Projeto TOPCASED

O processo de desenvolvimento de sistemas embarcados integra diferentes tipos de modelos e ferramentas. Levando em conta as características distintas destes modelos e também o longo tempo de vida dos produtos (no caso de sistemas aeronáuticos podem chegar a dezenas de anos), é muito difícil garantir que estes modelos poderão ser mantidos por longos períodos. Além do mais, a gestão das ferramentas deve levar em conta outros pontos, tais como: a formação das equipes de desenvolvimento; a administração das ferramentas; a troca de informação entre as atividades de desenvolvimento; a manutenção do conhecimento ou a certificação das ferramentas. O projeto TOPCASED foi concebido para atender a estas necessidades colocando a disposição da comunidade Open Source um conjunto de ferramentas de engenharia de sistemas suficientemente madura para o desenvolvimento de sistemas críticos de tempo real. Os principais objetivos do projeto TOPCASED são:

- perpetuar métodos e ferramentas para o desenvolvimento de sistemas e softwares;
- reduzir o custo de desenvolvimento com licenças de softwares;
- garantir a independência de plataformas de desenvolvimento;
- integrar rapidamente as inovações acadêmicas para o mundo industrial;
- ser capaz de adaptar as ferramentas aos processos e não o contrário.

O ambiente TOPCASED oferece não apenas um conjunto de ferramentas capaz de integrar diferentes tipos de modelos (modelos de especificação, comunicação, comportamento, arquitetura em tempo real ou arquitetura estática), mas também todas as fases que fazem parte do processo de desenvolvimento (Figura 1.4). Além disso, o ambiente TOPCASED suporta o desenvolvimento do sistema desde a sua especificação até sua implementação tanto de software como de hardware, passando pela definição da arquitetura do sistema. Logo, esta abordagem implica a utilização de diversas linguagens de modelagem, cada orientada a uma atividade do processo de desenvolvimento, uma característica funcional ou arquitetura do sistema. O conjunto de ferramentas do ambiente TOPCASED não apenas permite manipular todos esses formalismos, mas também facilita a comunicação de dados entre eles. Por esta razão, o ambiente é orientado ao uso de meta-modelos, o que permite ao projeto ser capaz de gerar os editores gráficos e também de especificar as transformações de modelos de forma eficaz e compacta, simplificando o processo de integração da cadeia de desenvolvimento através do intercâmbio de informação das diferentes fases. Este trabalho se encaixa neste contexto.

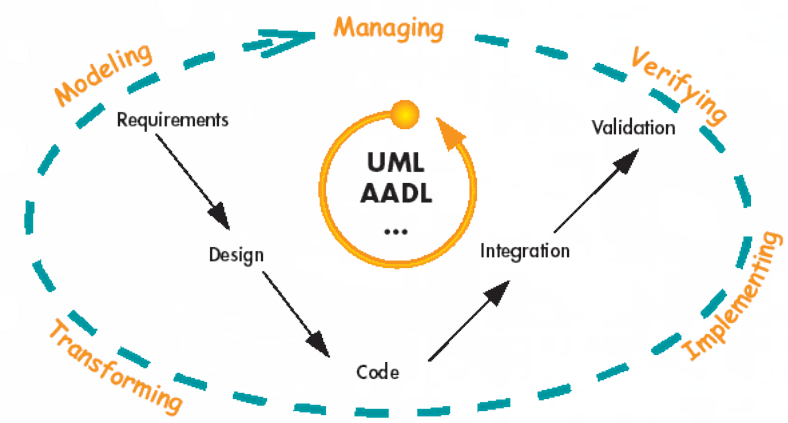


Figura 1.4: *Visão Geral do projeto* TOPCASED*[3].*

Devido ao caráter multidisciplinar deste projeto, ele é dividido em diversas partes combinando estudo e desenvolvimento a fim de determinar:

- os tipos de processos que as ferramentas devem suportar;
- os meios de modelagem;
- as técnicas de verificação;
- os meios de geração de código;
- técnicas de transformação de modelos;

- a interoperabilidade das ferramentas e a gestão de dados durante o processo de desenvolvimento.

O trabalho desenvolvido nesta dissertação se enquadra no contexto da transformação de modelos, visando o melhor aproveitamento das ferramentas de verificação. O projeto TOPCASED é integralmente apresentado em [3].

1.4 Objetivos

As atividades desenvolvidas nesta dissertação consistem em estudar a arquitetura de verificação do ambiente TOPCASED e operacionalizar a verificação formal da linguagem intermediária FIACRE utilizando o ambiente de verificação TINA. Os principais objetivos destas atividades são:

- propor um esquema conceitual de tradução de FIACRE para o formalismo Sistemas de Transição Temporizados (TTS), utilizado pela ferramenta TINA;
- desenvolver um compilador (*front-end*) de FIACRE para TINA (TTS-TINA) a fim de tornar operacional a verificação de sistemas descritos em FIACRE;
- fazer um estudo preliminar da tradução de SDL para FIACRE;
- exemplificar e testar a arquitetura de verificação utilizando modelos descritos em diagramas UML.

1.5 Organização do Documento

Este trabalho é organizado em cinco capítulos. O capítulo 2 introduz a arquitetura de verificação do ambiente TOPCASED em conjunto com a linguagem intermediária formal FIACRE. O esquema conceitual de tradução de FIACRE para o formalismo TTS e o compilador(*front-end*) para TINA são apresentados no capítulo 3. O capítulo 4 aborda um estudo da transformação entre as linguagens SDL e FIACRE que contribuiu para a definição da linguagem FIACRE. Uma cadeia de verificação que utiliza diagramas UML é ilustrada no capítulo 5. Finalmente, a conclusão e as perspectivas futuras deste trabalho são apresentadas no capítulo 6.

Capítulo 2

Arquitetura de Verificação do Ambiente TOPCASED

A verificação formal de sistemas no contexto do projeto TOPCASED é baseada em uma arquitetura que visa à independência de domínio de aplicação e do tipo de propriedades a serem verificadas. Em outras palavras, esta arquitetura visa tanto à independência em relação à linguagem de modelagem quanto ao formalismo selecionado. Para tal, esta arquitetura utiliza uma linguagem intermediária formal chamada FIACRE para operacionalizar a verificação formal de sistemas embarcados críticos.

Este capítulo apresenta a arquitetura de verificação do ambiente TOPCASED e a sintaxe da linguagem FIACRE. A motivação e uma visão geral desta arquitetura são apresentadas nas seções 2.1 e 2.2, respectivamente. A sintaxe da linguagem formal FIACRE é apresentada na seção 2.4. Um exemplo é ilustrado para introduzir as características de FIACRE na seção 2.4.10. Uma conclusão referente à arquitetura do ambiente TOPCASED é apresentada ao fim do capítulo.

2.1 Motivação

A metodologia *Model Driven Engineering* (MDE) é uma proposta promissora para desenvolvimento de sistemas complexos porque combina as seguintes tecnologias [13]:

- Linguagem de Modelagem para Domínios Específicos (DSML - *Domain-Specific Modeling Language*): DSML formaliza sistemas, que pertencem a domínios específicos, a partir da estrutura da aplicação, do seu comportamento e de seus requisitos. Sistemas modelados em DSML

são descritos utilizando-se meta-modelos, que permitem definir precisamente as semânticas chaves e as restrições associadas ao domínio do sistema.

- Motores de Transformação e Geradores (*Transformation engines and generators*): ferramentas capazes de analisar certos aspectos dos modelos e sintetizá-los em diversos objetos, tais como modelos alternativos, descrições para fins específicos, código fonte, entre outras. Esta habilidade para sintetizar objetos permite aumentar a consistência entre a implementação e as informações, que são associadas aos requisitos funcionais e de qualidade, capturadas do modelo. Este processo de transformação automática é frequentemente referenciado como "correto por construção" (*correct-by-construction*), em oposição às metodologias convencionais que podem ser classificadas como "construção por correção" (*construct-by-correction*).

Contudo, grande parte dos modelos ou linguagens (DSML) empregadas pela metodologia MDE não são adequados para a verificação formal. Por conseguinte, a verificação dos requisitos dos sistemas é avaliada basicamente através de testes dinâmicos: execução do modelo no caso de simulação ou a execução do produto final para a realização de testes de validação. Processos como estes não podem garantir a busca exaustiva de todas as situações possíveis para uma dada propriedade. Por exemplo, mesmo uma campanha longa de testes não pode permitir afirmar que uma dada propriedade "não deve ocorrer nunca".

Esta questão relativa à verificação formal ou testes dinâmicos precisa ser considerada cuidadosamente para sistemas críticos. No contexto destes sistemas, a atividade de verificação é considerada tão importante quanto às demais. Porém, para realizar a verificação formal neste caso é necessário utilizar pelos menos duas visões de modelos: uma voltada para a produtividade e outra para a verificação.

Dentro deste contexto, o ambiente TOPCASED implementa uma arquitetura de verificação original que permite combinar os métodos formais com a metodologia MDE sem prejudicar sua produtividade. Esta arquitetura é baseada na transformação dos modelos utilizados pela metodologia MDE em modelos formais aceitos pelas ferramentas de verificação. Com isto, não é mais necessário manter duas visões, evitando assim problemas de consistência entre ambas. Outra razão é o conforto de utilizar apenas uma linguagem de modelagem, assim, não é necessário aprender ferramentas e linguagens diferentes. E por último, a identificação de erros é facilitada no uso de apenas uma linguagem, pois um erro detectado em uma atividade de verificação é facilmente mapeado no modelo original.

Todavia, as dificuldades desta abordagem estão diretamente ligadas à capacidade da linguagem de modelagem selecionada lidar com a verificação. Como o primeiro objetivo desta linguagem é a representação do sistema, os aspectos inerentes à verificação nem sempre foram considerados. Por exemplo, podemos citar as linguagens não formais ou semi-formais que podem resultar

em interpretações ambíguas, sem mencionar que poucas linguagens incorporam em sua sintaxe expressões para definir as propriedades a serem verificadas. Por conseguinte, devido a estas dificuldades, esta abordagem só é possível através de uma transformação que traduz o modelo inicial em um modelo apto para a verificação formal. Contudo, esta abordagem ainda levanta algumas questões relacionadas à independência de domínio de aplicação, técnicas de transformação e linguagens alvo para verificação. Estes aspectos fazem parte dos objetivos do projeto TOPCASED, o qual consiste em dar o maior grau de liberdade para o usuário final.

2.2 Arquitetura para a Verificação Formal de Sistemas

Uma das principais fases durante o desenvolvimento de sistemas é a de verificação do sistema. Esta consiste em confrontar um modelo do sistema às suas especificações a fim de analisar se o sistema está operando conforme o esperado. Devido à complexidade do uso de formalismos matemáticos, o projeto TOPCASED objetivou escondê-los dos usuários para simplificar o processo de verificação. Entretanto, é necessário obter meios de traduzir os modelos largamente utilizados pela indústria em modelos formais, usualmente referenciados como sendo de uso acadêmico, para poder utilizar as ferramentas de verificação.

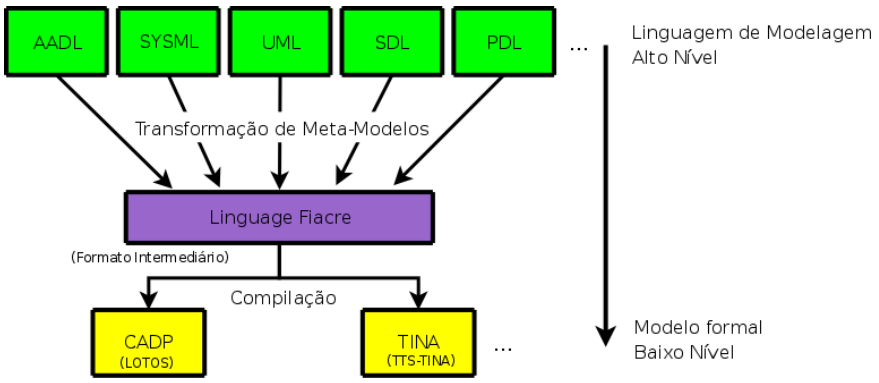


Figura 2.1: *Arquitetura de Verificação* FIACRE *[1].*

A Figura 2.1 apresenta a arquitetura utilizada pelo projeto para operacionalizar a verificação de sistemas de forma transparente para o usuário. Como podemos observar, ela é baseada em uma linguagem formal intermediária altamente expressiva denominada FIACRE[1], capaz de ser o resultado da tradução de vários modelos provenientes de diferentes linguagens de modelagem largamente utilizadas pela indústria. Como resultado, esta arquitetura separa a tradução em duas etapas: transformação de meta-modelos e compilação. Esta abordagem simplifica a adição de novas

[1] Format Intermédiaire pour les Architectures de Composants Répartis Embarqués

ferramentas de verificação, visto que toda linguagem de modelagem que possuir um acoplamento funcional (transformador de meta-modelos) com FIACRE poderá usufruir de todas as ferramentas de verificação suportadas pela arquitetura.

2.3 Cadeia de Verificação do Ambiente TOPCASED

Para o usuário, a arquitetura proposta inova porque não impõe nenhuma restrição quanto à escolha da linguagem de modelagem ou de verificação. Esta liberdade permite diversas combinações, ou seja, para cada par - formado por uma linguagem de modelagem e uma ferramenta de verificação formal - existe uma cadeia de verificação associada no ambiente TOPCASED. No contexto deste projeto, denomina-se por cadeia de verificação, a seqüência de eventos necessários para realizar o processo de verificação formal de um sistema descrito em uma linguagem de modelagem (UML, SYSML, AADL, SDL, etc). A Figura 2.2 descreve as etapas de uma cadeia de verificação genérica.

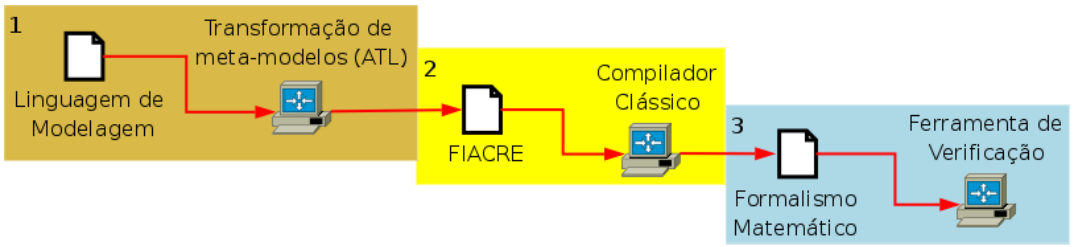


Figura 2.2: *Cadeia de Verificação Genérica.*

Como podemos observar na Figura 2.2, uma cadeia de verificação é formada por duas transformações de modelo e uma ferramenta de verificação. A primeira transformação consiste na tradução do sistema descrito em alguma linguagem de modelagem (podendo ser em alguns casos uma descrição informal ou semi-formal) para FIACRE. A segunda é a tradução de FIACRE para o formalismo matemático selecionado de acordo com a ferramenta de verificação selecionada.

2.3.1 Tradução da Linguagem de Entrada para FIACRE

Como foi exposto anteriormente, a cadeia de verificação inicia-se a partir de uma linguagem de modelagem (alto nível). Por conseguinte, a primeira etapa é a tradução desta descrição em FIACRE. Entretanto, esta etapa pode oferecer alguns obstáculos porque algumas destas linguagens são informais ou semi-formais. A falta de uma semântica concreta aumenta a complexidade de uma tradução automática porque pode resultar em interpretações formais distintas. Outro problema é que

diversas linguagens não possuem integrada à sua sintaxe formas para expressar propriedades a serem verificadas.

Quanto à transformação em si, esta etapa envolve a transformação entre linguagens que são fáceis de serem mapeadas entre si (Linguagem de modelagem e FIACRE) por possuírem características em comum. Logo, o projeto disponibiliza ferramentas baseadas na linguagem ATL[2] [14], a qual é uma linguagem para tradução de modelos que especifica tanto o meta-modelo como também a sintaxe textual concreta da linguagem. Um programa de tradução ATL é composto por: um conjunto de mapeamentos de padrões entre o meta-modelo de partida e o alvo; e um conjunto de regras que definem como a tradução deve ser realizada. Maiores informações sobre a linguagem ATL podem ser obtida em [15].

2.3.2 Tradução de FIACRE para um Formalismo Matemático

A segunda etapa da cadeia de verificação consiste em traduzir o modelo FIACRE em um formalismo matemático aceito pela ferramenta de verificação selecionada.

Esta transformação, diferentemente da anterior, é realizada preferencialmente utilizando-se compiladores concebidos com o auxílio de ferramentas clássicas como Lex[3] e Yacc [4] [16]. Por exemplo, a principal atividade desenvolvida neste trabalho foi à implementação de um compilador usando técnicas clássicas para traduzir modelos escritos em FIACRE para o formalismo matemático TTS[5] [17], empregado pelo ambiente TINA(TTS-TINA). Esta escolha se deve em parte pela necessidade de ferramentas otimizadas. Outra razão para utilizar compiladores clássicos é devido ao fato de os formalismos matemáticos serem enquadrados como linguagens de baixo nível, o que aumenta a complexidade do mapeamento entre as linguagens fonte e alvo.

2.3.3 Verificação Formal do Sistema

A terceira e última etapa é o uso de uma ferramenta de verificação formal. Até o presente momento, são duas as ferramentas suportadas: CADP [12] e TINA [4]. Para cada uma delas existe um formalismo específico de entrada, são eles: LOTOS[6] para CADP e TTS para TINA. As versões

[2] ATLAS Transformation Language
[3] Lexical Analyzer
[4] Yet Another Compiler-Compiler
[5] Time Transition System
[6] Linguagem para Especificação de Ordenamento Temporal

finais dos tradutores de FIACRE para estas ferramentas estão sendo desenvolvidos pelos laboratórios INRIA[7] e LAAS [8], respectivamente.

2.4 Linguagem Intermediária FIACRE

A linguagem FIACRE foi concebida para ser uma linguagem intermediária formal entre as linguagens de modelagem de alto nível (SYSML, SDL, etc) e as ferramentas de verificação que trabalham com linguagens baseadas em formalismos matemáticos (Autômatos, Redes de Petri, Sistemas de Transição Temporizados, etc), ver [1]. Esta linguagem formal foi concebida a partir da experiência adquirida no projeto V-Cotre[9] [18] e também foi baseada nas linguagens NTIF[10] [19] e BIP[11] [20].

FIACRE é uma linguagem orientada a processos e possui as seguintes características:

- os processos são construídos por um conjunto de estados, uma lista de transições entre estes estados formadas por construções clássicas (atribuições de variáveis, construções if-elsif-else, while, composições seqüenciais, etc), não determinísticas e para comunicação através de portas;
- os componentes descrevem a composição de processos. Um componente é construído como uma composição paralela de outros componentes ou processos que podem se comunicar através de portas ou variáveis compartilhadas. As prioridades e as restrições temporais são associadas à comunicação.

As seções seguintes apresentam a descrição da sintaxe de FIACRE através de uma variante da notação EBNF [12]. Esta notação descreve um conjunto de regras na forma "*simb*=*exp*", o qual significa que o símbolo *simb* não terminal representa qualquer coisa que possa ser gerado pela expressão EBNF *exp*.

2.4.1 Tipos

Os tipos de dados aceitos por FIACRE são divididos em dois grupos: *tipos de base* e *tipos construídos*. FIACRE aceita também a definição de *alias*, ou seja, permite renomear os tipos existen-

[7] Institut National de Recherche en Informatique et Automatique
[8] Laboratoire d'Architecture et d'Analyse des Systèmes
[9] Cotre Verification Language
[10] New Technology Intermediate Form
[11] Behavior, Interactin, Priority Language
[12] Extended Backus-Naur Form

tes.

```
tipe_id::=IDENT
enum::=IDENT
field::=IDENT
size::=NATURAL
integer::=['+'|'-']NATURAL
```

Tipos de base

Existem três tipos de base nativos : inteiro (int), natural (nat) e booleano (bool).

```
basic_type::=int|nat|bool
```

Tipos construídos

FIACRE permite ao usuário a criação de tipos específicos utilizando como base os tipos nativos:

- intervalos de inteiros: **interval** 2..7;
- enumerações: **enum** red, green, blue **end**;
- estruturas: **record** width, height:nat **end**;
- matrizes de tamanho fixo: **array of** 10 **nat**;
- pilhas de tamanho limitado: **queue** of 5 **bool**;
- uniões: **union** width, height:**nat end**.

```
build_type::= interval integer '..' integer
        |enum enum*, end
        |array of size type
        |record(field*;':'type)*, end
        |union(field*;':'type)*, end
```

Alias

FIACRE permite a criação de *aliases*, ou seja, permite renomear os tipos de dados. Por exemplo: **type** new_int **is** int.

```
type::= basic_type | build_type | type_id
type_decl::= type type_id is type
```

2.4.2 Variáveis e Expressões

As variáveis e expressões aceitas por FIACRE são descritas pela sintaxe apresentada abaixo:

```
var::=IDENT
literal::=NATURAL|true|false|enum|new NATURAL type
initalize::=literal
        | ('+'|'-')NATURAL
        | '['initializer*,']'
        | ''(field'='initializer)*,''
access::=var
        |access'['exp']'
        |access'.'field
unop::='-'|'S'|not|empty|dequeue|first
infixop::=or
        |and
        |'='|'<>'
        |'<'|'>'|'<='|'>='
        |'+'|'-'
        |'*'|'/'|'%'
binop::=enqueue
```

```
exp::=literal
        |access
        |unop exp
        |exp infixop exp
        |binop '(' exp ',' exp ')'
        |binop '('exp ',' exp')'
        |'('exp')'
        |exp'.?'field
```

As operações unárias **full**, **empty**, **dequeue** e **first**, assim como a operação binária **enqueue** são exclusivamente utilizadas pelas filas (**queue**). A expressão "*exp '?' field*"é permitida somente para variáveis do tipo **union**.

2.4.3 Portas e Canais de comunicação

As portas (**port**) fazem parte da interface de um processo FIACRE. Elas são responsáveis pela comunicação e podem ser utilizadas para a troca de dados. Os canais (**channel**) são usados para definir um conjunto de tipos de dados aceitos por uma porta.

```
port::=IDENT
channel_id::=IDENT
profile::=none|type*
channel::=channel_id|profile|channel'|'channel
channel_id::=channel IDENT is channel
```

Um perfil do tipo **none** sinaliza que a comunicação em questão é uma sincronização pura sem nenhuma troca de valor.

2.4.4 Processos

Um processo FIACRE pode ser visto como uma quíntupla

$$Pc = \langle Pt, Pm, S, V, T \rangle$$

onde:

- $Pt = pt_1 ... pt_n$ é um conjunto finito de portas. Estas portas são utilizadas para efetuar a sincronização de comportamento com outros elementos do sistema (processos ou componentes);
- *Pm* é um conjunto finito de parâmetros (formais e variáveis compartilhadas);
- *S* é um conjunto finito de estados para o controle interno dos processos;
- *V* é um conjunto finito de variáveis locais;
- *T* é um conjunto de transições atômicas.

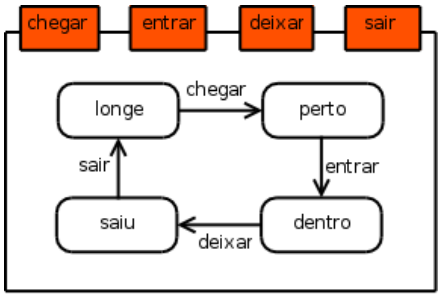


Figura 2.3: *Processo Trem (trem_p).*

A Figura 2.3 ilustra informalmente um processo chamado Trem (*trem_p*) formado por quatro portas (*chegar*, *entrar*, *deixar* e *sair*) e quatro estados de controle (*longe*, *perto*, *dentro* e *saiu*). Este processo possui quatro transições etiquetadas com os nomes das portas. A execução (disparo) destas transições depende de uma interação externa (evento) através das portas do processo. A sintaxe para um processo FIACRE é apresentada abaixo:

```
state::=IDENT
tag::=IDENT
name::=IDENT
time_interval::=('['|']')DECIMAL',(DECIMAL('['|']')|inf'['
port_dec::=([in][out]port)*,':'channel
arg_dec::=([read][write]var)*,':'type
var_dec::=var*,':'type[':='initializar]
process_dec::=process name ['['port_dec*,']'][('arg_dec*,')'] is
        states pstates*, init state [var
        [var_dec*,]
        transition*
```

A descrição FIACRE para o processo *trem_p* da Figura 2.3 é apresentada abaixo:

```
process trem_p [chegar, entrar, deixar, sair: none] is
states longe, perto, dentro, saiu init longe
        ..............transições..............
```

2.4.5 Comportamento

O comportamento dos processos FIACRE é descrito de forma estruturada a partir de uma lista de transições (conjunto T). Estas transições possuem um estado de partida e outro de chegada, e seus corpos são formados por uma rica estrutura de controle. Esta estrutura de controle permite:

- atribuição de variáveis deterministas: X**'**:='expressão;
- atribuição de variáveis não deterministas: X':='**any where** expressão;
- uso de estruturas de repetição condicional: while;
- uso de estruturas determinísticas: **if** expressão **then** ação **elsif** expressão **then** ação **else** ação;
- uso de estruturas não determinísticas: **select** ..... **end**;
- definição de estruturas seqüenciais: ....';'....;
- permite uma comunicação através de uma porta: '**?**'(emissão), '**!**'(recepção) ou sincronização;
- evolução para o estado final: **to** estado.

A sintaxe de uma transição FIACRE é apresentada abaixo:

```
transition::=from state statement
statement::= null
        |access+,':='exp+,
        |access+,':='any [where exp]
        |communication
        |if exp then statement (elsif exp then statement)*[else statement] end
        |select statement+[] end
        |to state
        |statement ';' statement
```

Seguem algumas considerações sobre uma transição FIACRE:

- a execução de uma transição é atômica;
- o uso das estruturas de controle determinísticas (**if-elsif-else**) e não determinísticas (**select**) permite a definição de macro transições (ramificações). Cada caminho possível pode fazer apenas uma comunicação e termina com a construção **to**.

2.4.6 Comunicação

FIACRE permite a comunicação síncrona entre processos e/ou componentes através das portas de comunicação. Estas portas permitem a sincronização pura ou a passagem de um ou diversos valores simultaneamente. Uma comunicação pode ser restringida pela opção **where**, a qual impede a sincronização se a expressão **exp** não for verdadeira. Segue abaixo a sintaxe BNF:

```
communication::= port[':'profile]
        |port [':' profile]'?'var+,[where exp]
        |port [':' profile]'!' exp+,
```

Os operadores '**?**' e '**!**' determinam o sentido da comunicação, ou seja, se a transição envia ou recebe um dado. No caso de uma comunicação através de uma porta sobrecarregada (mais de um perfil), um campo opcional (**profile**) é utilizado para precisar o perfil desejado.

2.4.7 Componente

Um componente FIACRE compreende a composição paralela de processos ou de outros componentes. São construídos a partir da instanciação de outros elementos do sistema (processos ou componentes), especificando as suas interconexões (canais de comunicação) e fornecendo uma relação de prioridades entre elas. Um componente pode ser visto como uma quíntupla

$$Cmp = \langle Pt, Pm, V, C, Pr, Cm \rangle$$

onde:

- *Pt* é um conjunto finito de portas que fazem parte da interface do componente. Estas portas são utilizadas para interligar as instâncias dos processos que compõem o componente e são chamadas de portas externas porque promovem a comunicação com o ambiente externo;
- *Pm* é um conjunto finito de parâmetros (formais e variáveis compartilhadas);
- *V* é um conjunto finito de variáveis locais;
- *C* é um conjunto finito de portas locais (**port**) associadas a restrições temporais (**is**). Estas portas locais também são utilizadas para interligar as instâncias que compõem o componente e são chamadas de canais de comunicação porque permitem a comunicação interna no componente;
- *Pr* é um conjunto finito de prioridades. Cada prioridade $pr \in Pr$ é uma relação em *C*, em outras palavras, ela define para cada subconjunto $\rho \subseteq C$ um outro subconjunto $pr(\rho) \subseteq C$. Esta relação adota a notação "$>$" e pode ser denotado como segue : $a > b$ onde $\{a,b \subseteq C | a \bigcap b = \emptyset\}$;
- *Cm* é uma composição paralela de instâncias de processos. A composição descreve a interação entre as instâncias que compõem o componente e será detalhada na seção Composição deste trabalho.

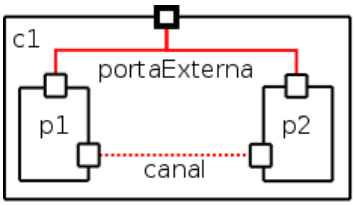


Figura 2.4: *Exemplo de um Componente.*

A Figura 2.4 apresenta um componente chamado c_1, que é formado pela composição paralela dos processos, p_1 e p_2, interligados por um canal de comunicação (*canal*) e por uma porta externa (*portaExterna*). A diferença entre os canais de comunicação e as portas externas será detalhada na seção Comunicação deste capítulo.

A sintaxe de um componente, em BNF, é apresentada abaixo:

```
component_decl::= component name ['['port_dec+,']']['('arg_dec+,')'] is
        [var var_dec+,]
        [port (port_dec [in time_interval])+,]
        [priority (port+| >port+|)+,]
        composition
```

A descrição FIACRE para o componente ilustrado na Figura 2.4 é apresentado abaixo:

```
component c1 [portaExterna:none] is
        port in out canal: none,
        sync
                p1[portaExterna, canal]
                p2[portaExterna, canal]
        end
```

O operador **sync** realiza a composição síncrona entre as instâncias dos processos p_1 e p_2. As características deste operador são apresentadas na seção Composição (2.4.8).

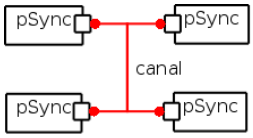
2.4.8 Composição

A comunicação síncrona presente em FIACRE é resultado da composição paralela de um conjunto de instâncias. A topologia da comunicação que esta composição promove pode ser divida em quatro grupos, são elas:

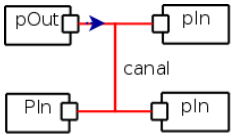
- sincronização sem passagem de valor (ver Figure 2.5a);
- sincronização entre $1 \longrightarrow N$: um emissor e N receptores (ver Figure 2.5b);
- sincronização entre $M \longrightarrow N$: este tipo de sincronização envolve M emissores e N receptores. Entretanto, para esta comunicação ocorrer, os M emissores precisam enviar os mesmos dados e os N receptores precisam estar aptos a receberem estes dados (ver Figure 2.5c);
- sincronização entre $M \longrightarrow 1$: M emissores e um receptor. Esta sincronização também só ocorre se todos os M emissores enviarem os mesmos dados. Como esta comunicação envolve apenas uma entidade receptora, não existe nenhuma condição imposta para o receptor (ver Figure 2.5d).

No entanto, esta topologia é descrita através dos operadores de composição paralela, são eles:

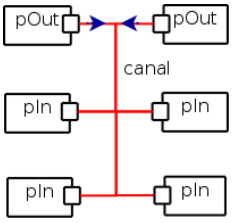
- **par** (Composição Paralela): este operador realiza a execução simultânea das ações observáveis entre processos e/ou componentes. As ações são ditas observáveis quando são relacionadas às portas declaradas pelo operador **par**.



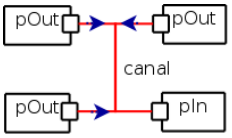
a)Sincronização sem passagem de valor.



b)Sincronização $1 \longrightarrow N$



c)Sincronização $M \longrightarrow N$



d)Sincronização $M \longrightarrow 1$

Figura 2.5: *Topologias para comunicação.*

- **sync** (Sincronização): este operador é uma simplificação do operador de composição paralela quando todas as ações são observáveis e representa a sincronização total entre os processos ou componentes implicados. Neste caso, o operador **sync** é uma simplificação do operador **par** quando todas as portas que fazem parte da interface dos processos ou componentes são locais.
- **Shuffle** (Paralelismo Puro) : este operador também é uma simplificação do operador **par** quando não existe nenhuma ação observável, ou seja, representa uma operação de composição paralela **par** com um conjunto vazio de portas.

Estes operadores composicionais manipulam instâncias de elementos do sistema. Uma instância pode ser vista como uma dupla

$$I = \langle Pti, Pmi \rangle$$

onde:

- Pti é um conjunto de canais de comunicação;
- Pmi é um conjunto de parâmetros formais.

A seguir, apresenta-se um pequeno exemplo para descrever em FIACRE as topologias de sincronização apresentadas na Figura 2.5. Sendo os processos *pSync*, *pOut* e *pIn*:

a)Processo *pSync*:

```
process pSync[portSync:none] is
states s1, s2 init s1
        from s1 portSync; to s2
```

b)Processo *pIn*:

```
process pIn[out portOut:bool] is
states s1, s2 init s1
        from s1 portOut! true; to s2
```

c)Processo *POut*:

```
process pOut[in portIn:bool] is
states s1, s2 init s1
var x:bool
        from s1 portIn? x; kwto s2
```

As descrições FIACRE para as topologias ilustradas na Figura 2.5 são apresentadas baixo:

a)Exemplo de Sincronização Pura

```
component main
port canal:none
        sync
                pSync[canal]
                pSync[canal]
                pSync[canal]
                pSync[canal]
        end
```

b)Exemplo de Sincronização $1 \longrightarrow N$

```
component main
port in out canal:bool
        sync
                pOut[canal]
                pIn[canal]
                pIn[canal]
                pIn[canal]
        end
```

c)Exemplo de Sincronização $M \longrightarrow N$

```
component main
port in out canal:bool
        sync
                pOut[canal]
                pOut[canal]
                pIn[canal]
                pIn[canal]
                pIn[canal]
                pIn[canal]
        end
```

d)Exemplo de Sincronização $M \longrightarrow 1$

```
component main
port in out canal:bool
        sync
                pOut[canal]
                pOut[canal]
                pOut[canal]
                pIn[canal]
        end
```

As notações EBNF dos operadores composicionais e da instância são apresentados abaixo:

```
instance::=name ['['port+,']']['('exp+,')']
composition::=shuffle composition+end
        |sync composition+end
        |par (port*, '->' composition)+end
        |instance
```

2.4.9 Um programa FIACRE

Um programa FIACRE completo compreende:

- um conjunto de tipos: **type** nome **is** int ;
- um conjunto de canais de comunicação: **channel** nome **is** int ;
- um conjunto de processos : **process** .....;
- um conjunto de componentes: **component** ..... ;

A notação EBNF para um programa completo FIACRE é apresentada abaixo:

```
declaration::= type_decl
        |channel_decl
        |process_decl
        |component_decl
program::= declaration+name
```

2.4.10 Exemplo: Trem em passagem de nível

O exemplo clássico de dois trens atravessando uma passagem de nível será utilizado para ilustrar as características da linguagem intermediária formal FIACRE.

Seja uma passagem de nível no qual uma estrada de ferro cruza uma estrada comum e uma cancela (ou uma sinaleira vermelho-verde) está controlando este cruzamento. O trem indica a sua entrada numa zona de proximidade do cruzamento e solicita a descida da cancela. Ao deixar a região de cruzamento, o trem sinaliza a sua saída. O intervalo de tempo entre a chegada do trem na zona de proximidade e a passagem de nível é entre 30 e 50 segundos. A cancela pode estar na posição levantada, fechada ou em situações intermediárias (descendo ou subindo). Inicialmente, a cancela

está na posição levantada ; quando o trem se aproxima, ela entra na situação intermediária descendo . A cancela leva de 10 a 15 segundos para descer ou subir completamente. Quando o trem deixa a região de passagem, a cancela entra na situação intermediária subindo. O intervalo que o trem leva para deixar a passagem de nível varia entre 30 e 50 segundos.

A Figura 2.6 mostra uma situação em que dois trens estão atravessando a zona de cruzamento. Podemos observar que o trem emite dois sinais, um quando ingressa na Zona de Proximidade (≪ *chegada* ≫) e outro quando sai da Zona de Cruzamento (≪ *saida* ≫) . Os intervalos de tempo "tempo de aproximação" e "tempo de passagem" são os intervalos que o trem leva para entrar e sair da zona de cruzamento, respectivamente.

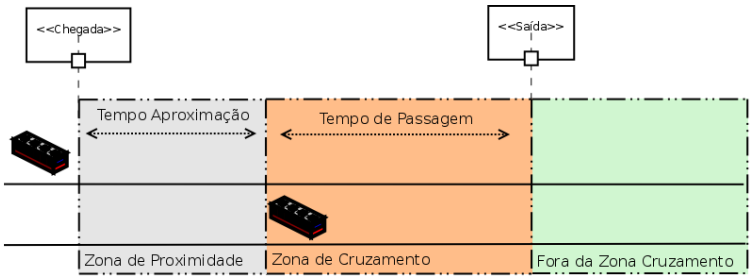


Figura 2.6: *Exemplo de dois trens atravessando uma zona de cruzamento.*

A partir da definição do problema acima, é possível identificar 4 elementos distintos que compõem o sistema, são eles:

- cancela: Este elemento representa a cancela responsável por fechar a passagem de nível. Este elemento possui quatro estados (*levantada*, *descendo*, *fechada* e *subindo*) e aguarda os comandos **subir** ou **descer**. A descrição FIACRE para este elemento do sistema é apresentada abaixo:

```
process cancela_p [descer, tempoDescida, subir, tempoSubida : none] is
        states levantada, descendo, fechada, subindo init levantada
        from levantada descer; to descendo
        from descendo tempoDescida; to fechada
        from fechada subir; to subindo
        from subindo
                %Durante a subida, a cancela pode receber
                %um comando (descer) para descer novamente.
                select
                        tempoSubida; to levantada
                []      descer; to descendo
```

```
        end

component cancela [comandoDescer, comandoSubir: none] is
        %Canais de comunicação para associar as restrições temporais
        port tDescida, tSubida : none in [5,10]
        cancela_p [comandoDescer, tDescida, comandoSubir, tSubida]
```

O comportamento deste elemento é descrito pelo processo *cancela_p* apresentado acima. Este processo é formado por quatro transições que aguardam eventos externos, são eles: *descer*, *tempoDescida*, *subir* e *tempoSubida*. Como FIACRE associa as restrições temporais às interações do sistema, é preciso encapsular este processo em um componente (*cancela*) para associar as restrições temporais de subida(*tempoSubida*) e descida (*tempoDescida*) da cancela. É importante observar que somente as portas associadas aos comandos de controle da cancela (**subir** e **descer**) são visíveis externamente.

- controle: o *controle* é responsável por enviar comandos à cancela e também por atender às solicitações dos trens. Este elemento possui os estados *aguardar*, *emitirDescer* e *emitirSubir*. A descrição FIACRE do *controle* na forma de um processo FIACRE é apresentado abaixo:

```
process controle [chegadaTrem, saidaTrem, fechaCancela, abreCancela: none] is
        states aguardar, emitirDescer, emitirSubir init aguardar
        var x : int := 0
        from aguardar
                %Aguarda a solicitação do trem para fechar a cancela ou
                %a sinalização de que o trem já deixou a região de passagem
                select
                        chegadaTrem; x:=x+1; if x=1 then to emitirDescer
                                                else to aguardar end
                []      saidaTrem; x:=x-1; if x=0 then to emitirSubir
                                           else to aguardar end
                end
        from emitirDescer fechaCancela; to aguardar
        from emitirSubir  abreCancela; to aguardar
```

Como podemos observar, o processo *controle* do sistema é descrito como tendo três transições que aguardam eventos externos, são eles: *chegadaTrem*, *saidaTrem*, *fechaCancela* e *abreCancela*. A segunda transição é o que caracterizamos anteriormente como uma macro transição

não determinística (**select**). Por último, através de uma variável inteira (x), o processo *controle* pode saber se existe ou não algum trem na região de passagem.

- trem: o elemento trem precisa sinalizar o controle sobre a sua posição (longe, perto, etc). Por exemplo, quando o trem entra na Zona de Proximidade (ver Figura 2.6), ele deve sinalizar a sua chegada ao controle da cancela (*sinalChegada*). Outro sinal deve ser enviado pelo trem quando este deixa a Zona de Cruzamento (*sinalSaida*). A descrição FIACRE para o *trem* é apresentada abaixo:

```
process trem_p [chegar, entrar, deixar, sair : none] is
        states longe, perto, dentro, saiu init far
        from longe      chegar; to perto
        from perto      entrar; to dentro
        from dentro     deixar; to saiu
        from saiu       sair;   to longe

component trem [sinalChegada, sinalSaida: none] is
        port entrando: none in [15,20], saindo: none in [30,50]
        train_p [sinalChegada, entrando, saindo, sinalSaida]
```

  O comportamento deste elemento é descrito pelo processo *trem_p*. Este processo é formado por quatro transições que aguardam eventos externos, são eles: *chegar*, *entrar*, *deixar* e *sair*. Este elemento também possui restrições temporais, logo, ele será encapsulado em um componente (*trem*) para associar as restrições temporais referentes às zonas de proximidade (*entrando*) e cruzamento (*saindo*). É importante observar que somente as portas associadas aos comandos de controle da cancela (**sinalChegada**) e (**sinalSaida**) são visíveis externamente.

A Figura 2.7 apresenta o componente principal (*main*) deste sistema, o qual é formado por dois trens (*trem*), uma cancela(*cancela*) e uma unidade de controle (*controle*). Estes componentes são interligados internamente pelos canais de comunicação *chegadaTrem*, *saidaTrem*, *fecharCancela* e *abrirCancela*. As portas em amarelo são canais de comunicação internos dos sub-componentes, os quais foram utilizados para associar restrições temporais e não são observáveis externamente.

```
component main is
        port chegada, saida : none,
             fechar, abrir : none in [0,0]
```

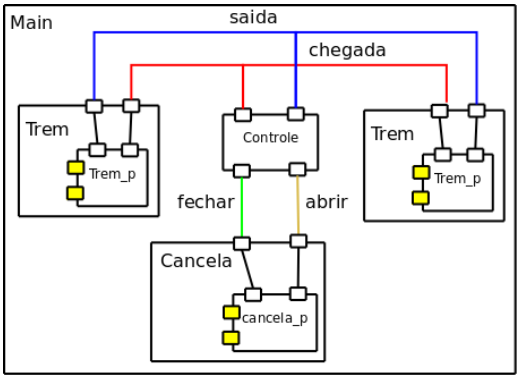


Figura 2.7: *Exemplo de um componente.*

```
        %Promove a sincronização entre todos os elementos.
        sync
                cancela [fecharCancela, abrirCancela]
                controle [chegadaTrem,saidaTrem, fecharCancela, abrirCancela]
                %Não permite a sincronização dos trens entre si.
                shuffle
                        trem [chegadaTrem,saidaTrem]
                        trem [chegadaTrem,saidaTrem]
                end
        end
sistema
```

Este exemplo mostra o alto poder de expressão dos operadores de composição. Por exemplo, os dois trens não estão sincronizados entre si (*shuflle*) mas estão sincronizados independentemente com a cancela e o controle.

2.5 Conclusão

Este capítulo apresentou a arquitetura de verificação do ambiente TOPCASED, em conjunto com a linguagem intermediária formal FIACRE. Esta arquitetura foi desenvolvida com o objetivo de realizar a verificação formal de forma transparente para o usuário, dando liberdade quanto à escolha da linguagem de modelagem e da ferramenta para verificação formal. Para atingir este objetivo, optou-se por utilizar uma linguagem intermediária formal, chamada FIACRE, para promover o intercâmbio de informação entre a modelagem e a verificação do sistema.

Como este trabalho faz parte dos esforços para operacionalizar FIACRE, nos próximos capítulos serão abordados temas relacionados à integração de FIACRE no ambiente TOPCASED. No capítulo 3 será detalhado um esquema conceitual de tradução da linguagem FIACRE para a ferramenta de verificação formal chamada TINA, mais especificamente para o formalismo suportado por esta ferramenta. Como parte também desta integração, o capítulo 4 apresenta um estudo sobre a transformação de SDL para FIACRE cujo objetivo foi avaliar a capacidade expressiva de FIACRE. Por último, oferecemos no capítulo 5 um exemplo completo de verificação para apresentar as vantagens que esta arquitetura promove.

Capítulo 3

Um compilador para a ferramenta TINA: Tradução de FIACRE para TTS.

O ambiente TOPCASED prevê a integração de diversas ferramentas de verificação formal. Uma das ferramentas suportadas pelo projeto é o ambiente para verificação e edição de Redes de Petri (PN) e Sistemas de Transição Temporizados (TTS) chamado TINA (Time Petri Net Analyser).

Para integrar TINA ao ambiente TOPCASED, é preciso traduzir os modelos FIACRE para TTS, um dos formalismos aceitos por TINA. Devido à complexidade desta tradução, ela foi dividida em duas etapas, chamadas de expansão e geração, interligadas por um formato intermediário chamado aqui de SUBFIACRE.

Neste capítulo vamos apresentar o modelo conceitual e a implementação de um compilador (*front-end* para o ambiente TINA) para traduzir modelos FIACRE para o formalismo TTS-TINA. Inicialmente serão abordados os conceitos fundamentais da fase de expansão de FIACRE para SUBFIACRE. Em seguida será detalhada a fase de geração de código, mostrando a conversão do formato intermediário interno SUBFIACRE para TTS. Por último, será mostrada a implementação de um compilador de FIACRE para TTS utilizando técnicas clássicas (tradução dirigida pela sintaxe, gerenciamento de nomes únicos, etc).

3.1 Ambiente TINA

TINA (Time Petri Net Analyser) é uma das ferramentas para verificação formal que foi integrada ao projeto e consiste em um ambiente para editar e analisar Redes de Petri Temporais e outras

extensões como Sistemas de Transição Temporizados.

TINA também oferece, além das facilidades usuais de edição e análise, diversas construções de espaço de estados abstratos que preservam classes específicas de propriedades, tais como ausência de *deadlocks*, propriedades temporais lineares, ou bi-similaridade. Estas construções são divididas em 3 grupos de ferramentas. O primeiro implementa as construções e métodos clássicos (grafo de marcações, grafo de marcações acessíveis e análise estrutural - invariantes). O segundo grupo possibilita o uso das técnicas de redução de ordem parcial a fim de evitar, se possível, a explosão combinatória. O terceiro e último grupo de ferramentas oferece métodos de construções de espaço de estados diferentes para os sistemas temporais, pois geralmente a representação dos seus respectivos espaços de estados não são infinitas.

Além de este ambiente ter sido desenvolvido para ser facilmente integrado com outras ferramentas de verificação, TINA possui uma ferramenta para efetuar a verificação de sistemas chamada SELT (*model-checker*) que suporta estados e transições na expressão das propriedades. Para as propriedades encontradas como falsas, um contra exemplo pode ser fornecido, sendo computado e eventualmente executado pelo simulador.

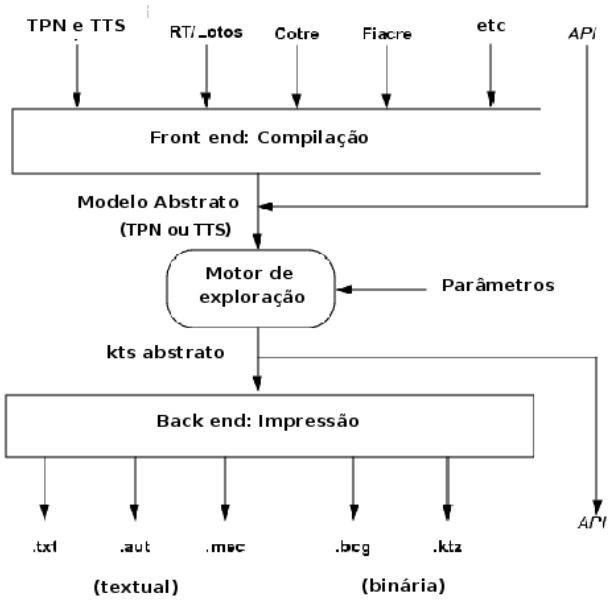


Figura 3.1: *Arquitetura da ferramenta* TINA *[4].*

A arquitetura funcional de Tina é dividida em três partes (Figura 3.1). A primeira parte (front-ends) compreende os compiladores utilizados para traduzir os modelos de entrada em um dos formalismos de base do ambiente, que são: Redes de Petri Temporal ou Sistemas de Transição Temporizados. A segunda parte pode ser interpretada como o *kernel* de Tina e consiste em um motor

de exploração de estados parametrizado pela classe de propriedades a serem preservadas. A terceira e última camada (*back-end*) promove a integração do ambiente TINA com ferramentas externas através da conversão dos resultados obtidos em representações legíveis por outras ferramentas. Maiores informações sobre a ferramenta TINA podem ser obtidas em [4].

Sistema de Transições Temporizadas - TTS

O compilador proposto nesta dissertação definiu como formalismo alvo o Sistema de Transições Temporizadas. Este formalismo foi selecionado porque ele é muito utilizado para verificar formalmente sistemas que são caracterizados como: concorrentes, assíncronos, distribuídos, paralelos ou não determinísticos. "Eles são uma generalização do sistema de transições de base através da associação de tempos mínimos e máximos para as transições" [17].

Um Sistema de Transição Temporizado $S = \langle \vartheta, \Sigma, \Theta, T, l, \mu \rangle$ é composto por seis parâmetros:

- um conjunto finito de variáveis ϑ;
- um conjunto de estados Σ. Cada estado $\sigma \in \Sigma$ é uma interpretação de ϑ ; significa que para cada variável $x \in \vartheta$ é relacionado a um valor $\sigma(x)$ que pertence ao seu domínio;
- um sub-conjunto de estados iniciais $\Theta \subseteq \Sigma$;
- um atraso inicial mínimo $l_\tau \in N$ para cada transição $\tau \in T$. A ausência de um atraso mínimo para uma transição é modelada como $l_{\tau_i} = 0$;
- um atraso máximo $\mu_\tau \in N \bigcup \{\infty\}$ para cada transição $\tau \in T$. É preciso que $\mu_\tau \geqslant l_\tau$ para toda $\tau \in T$, e que $\mu_\tau = \infty$ se τ é habilitada em qualquer um dos estados iniciais em Θ. A ausência de um atraso máximo para uma transição é modelada como $l_{\tau_i} = \infty$;
- um conjunto finito T de transições, incluindo a transição não ativa τ_l. Cada transição $\tau \in T$ é uma relação binária em Σ; em outras palavras, ela define para cada estado $\sigma \in \Sigma$ um conjunto, o qual pode ser vazio, de τ-sucessores $\tau(\sigma) \subseteq \Sigma$. A transição τ é sensibilizada por um estado σ se e somente se $\tau(\sigma) \neq 0$ e igualmente se as restrições temporais são respeitadas ($l_\tau \leq t \leq \mu_\tau$). As restrições temporais asseguram que as transições serão disparadas somente no intervalo de tempo permitido. Em particular, a transição repouso (*idle*) $\tau_e = \{(\sigma, \sigma) | \sigma \in \Sigma\}$ está habilitada em todos os estados.

A partir de [21], podemos interpretar este tipo de sistema como uma extensão da Rede de Petri Temporal (TPN) estruturada em duas partes: controle e dados.

A parte de *controle* é descrita pela Rede de Petri Temporal comum e descreve os encadeamentos de eventos e atividades. Assim, as variáveis que pertencem ao conjunto ϑ e que são importantes para o controle do sistema são representadas como lugares da TPN empregada.

A parte de *dados* descreve as variáveis que fazem parte do conjunto ϑ e não pertencem à estrutura de controle do sistema. Os cálculos, que são realizados sobre a estrutura de dados, são representados associando expressões condicionais e ações às transições. A Figura 3.2 mostra o exemplo de uma representação gráfica de uma transição TTS representada como uma TPN estendida com dados. A definição do formalismo TPN pode ser encontrada em [21].

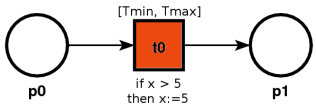


Figura 3.2: *Visualização de uma Transição Temporizada Estendida.*

A figura 3.2 pode ser interpretada como uma transição chamada t_0 com as características seguintes:

- restrição temporal: esta transição não pode ser disparada antes do tempo mínimo T_{min} ou após o tempo máximo T_{max};
- condição associada à transição: O lugar p_0 precisa possuir ao menos uma ficha e a variável x dever ser maior que cinco para que a transição t_0 possa ser disparada;
- ação associada à transição: Após o disparo da transição, o lugar p_1 receberá uma ficha e a variável x receberá o valor 5.

3.2 Expansão

A fase de expansão da descrição FIACRE é realizada internamente pelo compilador (*front-end*) e consiste em simplificar a descrição original de modo que o código resultante seja mais simples de ser codificado na linguagem alvo. Este código resultante é escrito em uma linguagem FIACRE simplificada chamada aqui de SUBFIACRE (Figura 3.3). Esta sub-linguagem é utilizada internamente pelo compilador para fazer a ligação entre as fases de expansão e geração de código. Nesta seção, esta sub-linguagem será detalhada, bem como as operações de expansão utilizadas para traduzir FIACRE para TTS. É importante ressaltar que devido ao caráter modular destas operações, elas podem ser posteriormente empregadas para auxiliar a tradução de FIACRE para outros formalismos matemáticos.

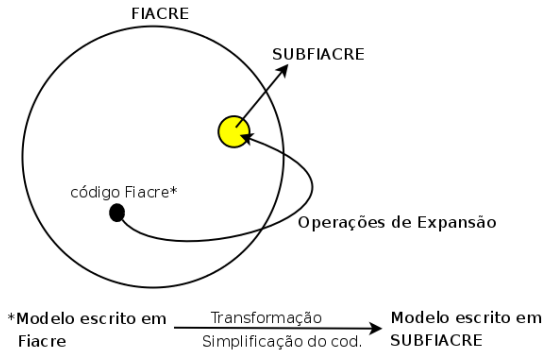


Figura 3.3: *Expansão de* FIACRE *para SUBFIACRE.*

3.2.1 SUBFIACRE

O formato intermediário SUBFIACRE pode ser considerado como um subconjunto simplificado de FIACRE. A Figura 3.4 mostra que um programa FIACRE, que pode ser interpretado como um componente construído a partir da composição paralela de outros componentes ou processos comunicantes, pode ser simplificado em apenas um processo. Este código resultante representa o estágio final da tradução porque sua estrutura é facilmente transformada em TTS-TINA.

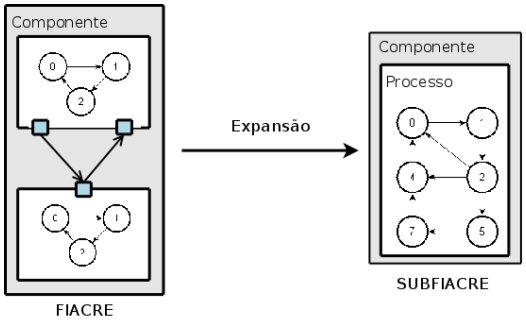


Figura 3.4: *Comparação de SUBFIACRE com* FIACRE.

Um programa SUBFIACRE compreende:

- uma lista de tipos;
- uma lista de prioridades;
- um componente;
- um processo formado por uma lista de estados e um conjunto de transições.

Um resumo da notação EBNF da sub-linguage SUBFIACRE é apresentado abaixo, a parte de dados e expressões foi removida da sintaxe abaixo e encontra-se no anexo A:

```
port_dec::=([in][out]port)*,':'channel
arg_dec::=([read][write]var)*,':'type
var_dec::=var*,':'type[':='initializar]
process_dec::=process name is
        states pstates*, init state
        [var var_dec*,]
        transition*transition::=IDENT in time_interval:from state+statement
statement::= |access+,':='exp+,
        |if exp then statement end
        |to state
        |statement ';' statement
component_decl::= component name is
        [priority (IDENT+|>IDENT+|)+,]
        composition
instance::=name
composition::= instance
program::=      [type_decl*]
                process_decl
                component_decl
```

Comparando a sintaxe da linguagem SUBFIACRE com FIACRE, é possível observar que diversas construções foram eliminadas, como por exemplo as comunicações ('?' e '!'), as construções não determinísticas (**select**), as bifurcações determinísticas (**elsif-else**), as atribuições não determinísticas (**any**), as múltiplas atribuições, entre outras. Por conseguinte, o modelo descrito nesse formato simplificado pode ser facilmente compilado para o formato alvo sem a necessidade de grandes operações. Esta afirmação decorre de três aspectos:

- menor complexidade porque a composição paralela entre os processos já foi realizada;
- apenas um processo, logo, apenas uma máquina de estado para codificar em TTS-TINA;
- transições simplificadas, ou seja, possuem somente as construções **if**, atribuições de variáveis (**:=**) e **to** (próximo estado).

3.2.2 Operações de Expansão

A etapa de Expansão pode ser considerada como uma atividade de re-escrita da descrição original, simplificando as construções não aceitas diretamente pelo formalismo alvo. Ela é realizada pela execução de diversas operações, respeitando uma ordem pré-estabelecida. Este procedimento de expansão é constituído por um conjunto de operações divididas em quatro grupos, são eles:

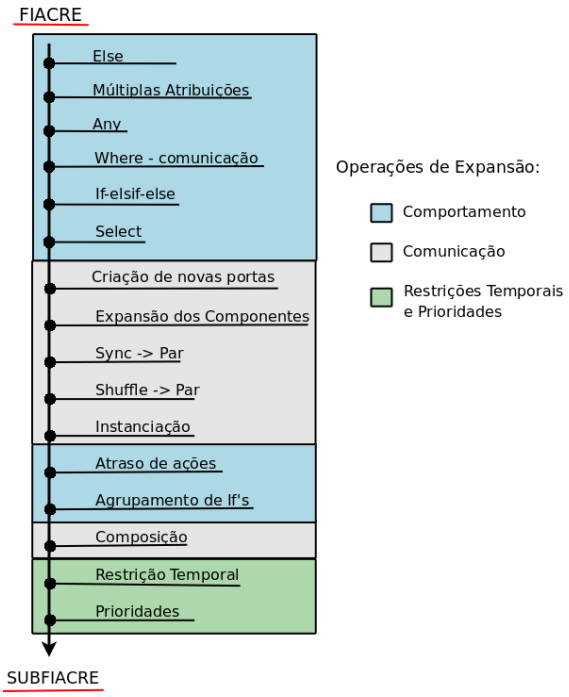


Figura 3.5: *Operações de Expansão.*

1. estrutura: as operações deste grupo são importantes para manter a estrutura do código coesa durante e após o processo de tradução;

2. comportamento: trata-se das operações de expansão aplicadas sobre a linguagem de comportamento para simplificar as atribuições (**any**) e ramificações não determinísticas nativas em FIACRE;

3. comunicação: são as operações que efetuam a composição paralela entre os processos e/ou componentes;

4. restrições Temporais e Prioridades: resolvem as restrições temporais e as prioridades entre as transições do sistema resultante.

3.2.3 Estrutura

As operações que fazem parte deste grupo são importantes para manter a estrutura do código coesa. Durante a tradução, a estrutura do código é alterada. Algumas operações inserem construções chamadas aqui de marcadas. Estas construções são de uso interno e precisam ser removidas no momento certo. Outras partes do código, devido à manipulação, começam a perder suas propriedades. Por estes motivos, é importante analisar a estrutura do código continuamente a fim de mantê-la coesa. Como estas operações estão diretamente relacionadas à árvore abstrata do compilador (ver apêndice D), elas não serão detalhadas aqui.

3.2.4 Comportamento

A expansão do comportamento FIACRE é sem dúvida a etapa mais importante da tradução e possui dois objetivos principais, são eles: a expansão das macro transições (ramificações não determinísticas ou determinísticas) em transições independentes e a reorganização das transições na forma condições-ações (***if*** *expressão* ***then*** *ação*). Tanto a expansão como a reorganização são imprescindíveis porque TTS (ver seção 3.1) não aceita macro transições e nem transições cuja condição e ação estejam entrelaçadas. As figuras 3.6 e 3.7 ilustram de forma sucinta o resultado esperado após a execução deste grupo de operações.

```
from wait
        select
                app1; x:=x+1;
                if x=1 then to emitdown
                else to wait end
        []      exit1; x:=x-1;
                if x=0 then to emitup
                else to wait end
        end
```

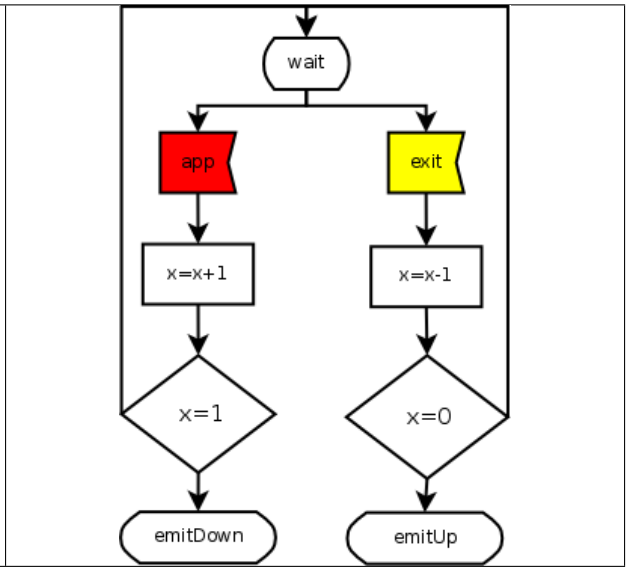


Figura 3.6: *a)Transição* FIACRE*. b)Representação Gráfica da Transição como um Fluxograma.*

A Figura 3.6a) mostra uma macro transição formada por dois ramos não determinísticos (**select**). Internamente, ambos os ramos apresentam imbricações condicionais (**if-else**). A Figura 3.6b) mostra

graficamente, de maneira informal, esta transição. Pode-se observar que esta macro transição permite quatro caminhos distintos de disparo, dependendo da porta na qual o evento de sincronização é recebido e do valor da variável *x* (condição **if**).

```
from wait app1;
        if x+1=1
        then x:=x+1; to emitdown end
from wait app1;
        if not(x+1=1)
        then x:=x+1; to wait end
from wait exit;
        if x=0
        then x:=x-1; to emitup end
from wait exit;
        if not(x-1=0)
        then x:=x-1; to wait end
```

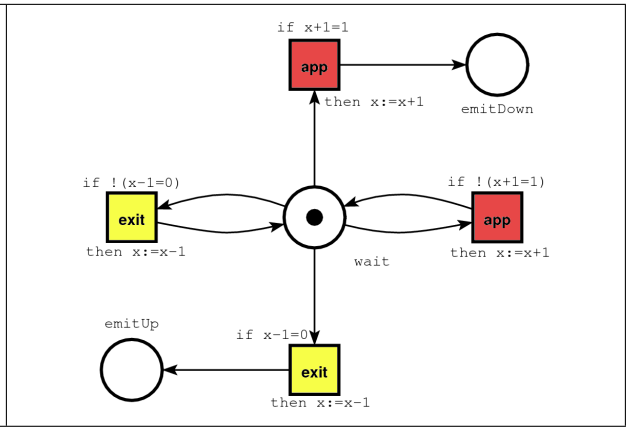


Figura 3.7: *a)Transição SUBFIACRE Resultante. b)Representação Gráfica da Transição Resultante em TTS.*

A operação de expansão da macro transição apresentada acima resulta em quatro transições com a forma condição-ação (ver Figura 3.7a). É importante observar que as relações de determinismo entre elas foram preservadas, pois os disparos destas transições ainda dependem das portas e do valor da variável *x*. A Figura 3.7b) mostra graficamente estas transições como um Sistema de Transições Temporizadas.

A Figura 3.8 mostra todas as operações necessárias para expandir o comportamento FIACRE, ressaltando a sua ordem de execução. Esta ordem deve ser respeitada porque as operações são dependentes um das outras.

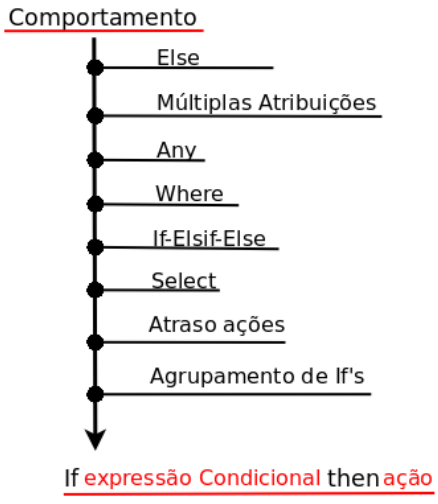
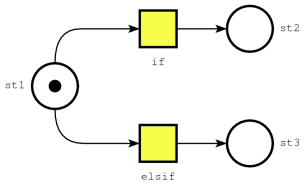


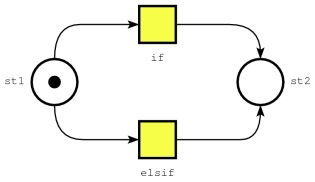
Figura 3.8: *Expansão do Comportamento.*

Estas operações são apresentadas em seguida respeitando a sua ordem de execução.

Expansão Else

A construção **if-elsif** pode apresentar um comportamento dual em FIACRE quanto à evolução da transição. Esta construção pode definir uma ramificação determinística que evolui para estados FIACRE diferentes (ver Figura 3.9a) ou para um estado FIACRE em comum (ver Figura 3.9b). Esta dualidade de comportamento decorre da forma como a construção **to** é empregada. No caso da Figura 3.9a), a construção **to** faz parte das ações da condição **if-elsif**. Já a Figura 3.9b) não possui a construção **to** como parte das suas ações, ou seja, as ações estão definidas externamente da construção **if-elsif**.





a) **if** `condição` **then** `...` ; **to** `st1`
elseif `...` ; **to** `st2` **end**

b) **if** `condição` **then** `...`
elseif `...` **end** ; **to** `st2`

Figura 3.9: *If-Elsif: a)evolui para estados* FIACRE *diferentes. b)evolui para o mesmo estado* FIACRE.

A operação de expansão **else** consiste em adicionar a opção else para todas as ramificações determinísticas **if-elsif** que não possuam a construção **to** como parte de suas ações. Esta operação é necessária porque, independente do que ocorrer nesta ramificação, a transição precisa evoluir para o próximo estado. Para simplificar este tipo de transição, esta operação faz a adição de um caminho alternativo (**else**) de disparo quando as condições **if-elsif** não são verdadeiras. A regra para efetuar esta operação de expansão é apresentada abaixo:

Regra: Expansão **Else**

Enunciado

if $\texttt{exp}_0$ **then** $\texttt{statement}_0$
elsif $\texttt{exp}_1$ **then** $\texttt{statement}_1$
$\vdots$
elsif $\texttt{exp}_i$ **then** $\texttt{statement}_i$
end$\,';'\,\texttt{statement}_{i+1}$
$\forall i \in N^+$

$\Downarrow$

Código Gerado

if $\texttt{exp}_0$ **then** $\texttt{statement}_0$
elsif $\texttt{exp}_1$ **then** $\texttt{statement}_1\,';'\,\texttt{statement}_{i+1}$
$\vdots$
elsif $\texttt{exp}_i$ **then** $\texttt{statement}_i\,';'\,\texttt{statement}_{i+1}$
else $\texttt{statement}_{i+1}$ **end**

A Figura 3.10 ilustra um exemplo de uma transição que inicia no estado *Não Ativo* e evolui para o estado *ativo*. Entre estes intervalos, ocorre uma ramificação determinística (**if-elsif**) condicionada ao valor de x. Contudo, esta transição não possui nenhum caminho explícito caso as condições **if-elsif** não sejam verdadeiras. Para simplificar a tradução, esta operação adicionou a opção **else** à transição.

```
from Nao Ativo
    if x> 2 then x:=2
    elsif x =2 then x:=0 end ; to Ativo
```

a) Antes da Adição de **else**.

$\Downarrow$ Expansão

```
from Nao Ativo
    if  x> 2 then x:=2; to Ativo
    elsif x =2 then x:=0; to Ativo
    else to Ativo end
```

b) Depois da Adição de **else**.

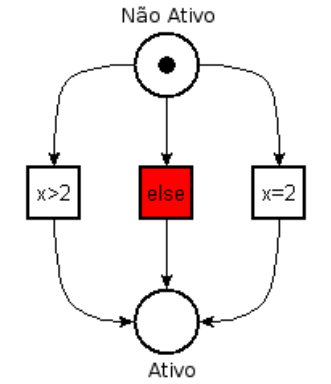


Figura 3.10: *Exemplo da Expansão da construção Else.*

Observando atentamente a transição resultante da Figura 3.10, nota-se que a construção **to** foi deslocada para dentro da construção condicional (**if-elsif-else**). Esta é outra simplificação para auxiliar a tradução, pois assim será mantida a estrutura (***if*** *expressão* ***then*** *ação*) para as transições

definidas no início.

Múltiplas atribuições

FIACRE permite a definição de múltiplas atribuições em uma mesma declaração. Esta operação faz a expansão destas múltiplas atribuições em uma declaração para cada atribuição. As regras abaixo formalizam esta operação:

Regra: Expansão de Múltiplas Atribuições

Enunciado	
1)	2)
$\texttt{access}_0\texttt{,...,access}_i\texttt{':='exp}_0\texttt{,...,exp}_i$	$\texttt{access}_0\texttt{,...,access}_i\texttt{':='}$ **any**
$\forall i \in N$	$\forall i \in N$
$\Downarrow$	$\Downarrow$
Código Gerado	Código Gerado
$\texttt{access}_0\texttt{':='exp}_0\texttt{';'}$	$\texttt{access}_0\texttt{':='}$ **any** $\texttt{';'}$
$\texttt{...';'}$	$\texttt{...';'}$
$\texttt{access}_i\texttt{':='exp}_i$	$\texttt{access}_i\texttt{':='}$ **any**

O exemplo abaixo ilustra uma transição que possui múltiplas atribuições em apenas uma declaração:

from `idle` **x,y,z := 1,2,3; to** `busy` $\longrightarrow$ **from** `idle` **x := 1; y := 2; z := 3; to** `busy`

Expansão Any

A construção **any** corresponde a uma atribuição (**...:=any**) não determinística de uma variável de acordo com o seu tipo. Esta operação consiste em substituir esta construção por ramos não determinísticos (**select**), sendo um ramo para cada valor que pertence ao domínio da variável. A regra de expansão para esta operação é apresentada abaixo:

Regra: Expansão **Any**

Enunciado

`access':='` **any**

Supondo que o domínio de *access* possui *n* elementos. $D = \{exp_0, \ldots, exp_i\}$

$\Downarrow$

Código Gerado

select

`access':='` exp_0

`'[]'` $\vdots$

`'[]'` `access':='` exp_i

end

$\forall i \in N$

Por exemplo, consideramos uma variável do tipo: *x:interval 0..3* cujo domínio é $D = \{x \in N \| x \leqslant 3\}$. A expansão da construção **x:=any** consiste em criar uma macro transição com quatro ramos não determinísticos, sendo um ramo para cada valor pertencente ao domínio *D* da variável *x*. Este exemplo é ilustrado abaixo:

```
var x:interval 0..3
from idle x:= ANY; to endState
```

O resultado deste exemplo é apresentado em seguida:

```
from idle select
        x:=0
[]      x:=1
[]      x:=2
[]      x:=3
end; to endState
```

A opção **where** adiciona uma restrição ao conjunto de valores possíveis de atribuição a partir da associação de um conjunto de inequações não lineares. Assim, para traduzir esta opção é necessário solucionar estas inequações. Devido à complexidade da tradução deste sistema de inequações, ela não será abordada neste trabalho.

Expansão Where-Comunicação

A linguagem FIACRE permite restringir a recepção de valores, em outras palavras, é possível prover uma expressão condicional opcional para definir quando uma recepção de valor pode ou não

ocorrer. Um aspecto importante relativo a esta construção é que a linguagem FIACRE garante a atomicidade da transição, ou seja, a transição ocorre somente se a expressão condicional **where** permitir, caso contrário, a transição não é disparada.

Esta restrição opcional pode ser simplificada fazendo a substituição da construção **where** pela construção condicional **if**. Para garantir a atomicidade da transição, todas as construções subseqüentes são inseridas no corpo da construção condicional (**then**), evitando assim o disparo prematuro da transição. A regra abaixo resume esta operação de expansão:

Regra: Expansão **Where**

Enunciado

`port ['':' profile]'?'var`$^{+}_{,}$ **where** `exp; statement`

$\Downarrow$

Código Gerado

`port ['':' profile]'?'var`$^{+}_{,}$`;` **if** `exp` **then** `statement` **end**

O exemplo abaixo mostra uma transição que emprega esta restrição:

from `idle pIn? xTemp` **where xTemp** $>$ **4**; **to** `endState`

A transição resultante aplicando-se a regra demonstrada acima é apresentada abaixo:

from `idle pIn? xTemp;` **if xTemp** $>$ **4 then** `to endState` **end**

Como se pode observar, a restrição imposta pela construção opcional **where** foi substituída por uma construção **if-then**. Assim, esta evoluirá do estado *idle* para o estado *endState* se e somente se $xTemp > 4$.

Expansão If-ElsIf-Else

FIACRE é uma linguagem elegante porque permite macro transições. Estas macro transições são formadas basicamente por duas construções: **if-elsif-else** (determinística) e **select** (não determinística). Para simplificar este processo de expansão de macro transições, vamos transformar uma construção determinística em uma não determinística, preservando o determinismo. Embora esta afirmação soe estranha à primeira vista, ela consiste simplesmente em substituir uma construção **if-elsif-else** por uma **select**, expandindo as expressões condicionais para preservar o determinismo entre

os ramos. Mais adiante, a construção **select** será expandida em transições independentes, finalizando o processo de expansão de macro transições. A regra para a expansão da construção **if-elsif-else** é apresentada abaixo:

Regra: Expansão **If-Elsif-Else**

Enunciado

if $\texttt{exp}_0$ **then** $\texttt{statement}_0$
elsif $\texttt{exp}_1$ **then** $\texttt{statement}_1$
$\vdots$
elsif $\texttt{exp}_i$ **then** $\texttt{statement}_i$
else $\texttt{statement}_{\texttt{else}}$ **end**

$\forall j, i \in N^+ | j \leqslant i$

$\Downarrow$

Código Gerado

select
if $\texttt{exp}_0$
then $\texttt{statement}_0$ **end**
[] **if** not $\texttt{exp}_0$ and $\texttt{exp}_1$
then $\texttt{statement}_1$ **end**
$\vdots$ if
[] **if** $\bigcap_{j=0}^{i-1}$ (not $\texttt{exp}_j$) and $\texttt{exp}_i$
then $\texttt{statement}_i$ **end**
[] **if** $\bigcap_{j=0}^{i}$ (not $\texttt{exp}_j$)
then $\texttt{statement}_{\texttt{else}}$ **end**
end

A Figura 3.11a) mostra todas as ramificações da macro transição da Figura 3.11b).

A macro transição não determinística resultante desta operação de expansão é mostrada na Figura 3.12.

Expansão da construção select's

Esta operação termina a expansão das macro transições FIACRE e consiste em criar novas transições, uma para cada ramificação possível.

```
from Nao Ativa
        if x>4 then to Ativa
        elsif  x<4 then
                if x=y then to Aberta
                else to Em Espera end
        else to Nao Ativa end
```

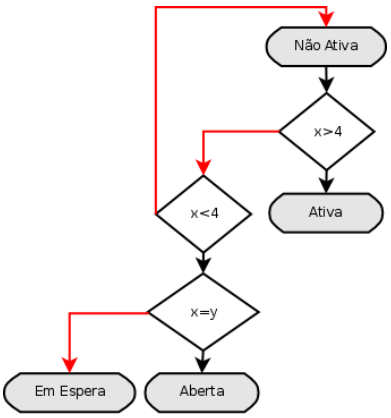


Figura 3.11: *a)Descrição* FIACRE *de uma macro transição. b) Fluxograma de uma Macro Transição* FIACRE.

```
from Nao Ativa
        select
                if x>4 then to Ativa end
        []      if not x>4 and x<4 and x=y
                        then to Aberta end
        []      if not x>4 and x<4 and not x=y
                        then to Em Espera end
        []      if not x>4 and not x<4
                        then to pasActif end
        end
```

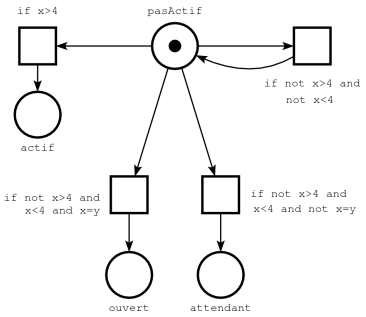


Figura 3.12: *a)Descrição SUBFIACRE após a expansão da macro transição. b) TTS da transição após a expansão da macro transição.*

Regra: Expansão Select

from state **select**

statement$_0$

$\vdots$

[] statement$_i$

end

$\forall i \in N^+$

$\Downarrow$

Código Gerado

from state statement$_0$

$\vdots$

from state statement$_i$

O exemplo abaixo ilustra uma macro transição não determinística (**select**):

```
from Espera
  select
        pOut! 4; to Fim
  []    pIn? Temp;
                select
                        to Fim
                []      to Espera
                end
  end
```

As transições resultantes após a expansão da construção **select** são apresentadas abaixo:

```
from Espera  pOut! 4; to Fim
from Espera  pIn? Temp; to Fim
from Espera  pIn? Temp; to Espera
```

Operação Atraso de Ações

A operação *Atraso de Ações* é de extrema importância para associar as condições-ações às transições. Logo, esta operação é responsável por reorganizar a parte de *dados* das transições na forma ***if*** *condições* ***then*** *ações*. Para reorganizar a transição, todas as atribuições (**:=**) devem ser deslocadas para depois das condições. Contudo, este deslocamento deve preservar as informações das atribuições para não alterar a execução da transição. A regra para esta operação é apresentada abaixo:

Regra: Expansão **Atraso de Ações**

Enunciado

$\texttt{access}_0\texttt{':='exp}_0\texttt{';' ...';' access}_i\texttt{':='exp}_i\texttt{';'}$ **if** `exp` **then** `statement` **end**

$\forall i \in N^+$

$\Downarrow$

Código Gerado

```
if exp then
        access0':='exp0';'
        ...';'
        accessi':='expi';'
        statement
end
```

A partir da regra definida acima, vamos efetuar o "atraso das ações" para o exemplo abaixo:

from `idle` **tp** $:=$ **x** $+$ **2**; **if** **tp**`>10` **then** **to** `endState`
else `to start` **end**

A atribuição $tp := x+2$ deve ser deslocada para após a expressão condicional $tp > 10$. Entretanto, para preservar a equivalência entre as transições, todas as referências da variável tp devem ser substituídas pela expressão $x+2$, conforme segue:

from `idle` **if** **x** $+$ **2** $>$ **10** **then** **tp** $:=$ **x** $+$ **2** ; `kwto endState`
else **tp** $:=$ **x** $+$ **2**; **to** `start` **end**

Operação para Agrupamento de If's

Algumas das operações de expansão para o comportamento resultam na adição de construções **if**. Como resultado, pode ocorrer seqüências de construções if's entrelaçadas. Estas construções entrelaçadas serão agrupadas para definir uma expressão condicional única e, por conseguinte, uma única seqüência de ação.

Regra: Expansão Agrupamento de If's

Enunciado

1)

if $\texttt{exp}_0$ **then**
if $\texttt{exp}_1$ **then** $\texttt{statement}_1$ **end**`';'`
$\texttt{statement}_0$
end

⇓

Código Gerado

if $\texttt{exp}_0$ `and` $\texttt{exp}_1$
then $\texttt{statement}_1$`';'` $\texttt{statement}_0$ **end**

2)

if $\texttt{exp}_0$ **then**
$\texttt{statement}_0$`';'`
if $\texttt{exp}_1$ **then** $\texttt{statement}_1$ **end**
end

⇓

Código Gerado

if $\texttt{exp}_0$ `and` $\texttt{exp}_1$
then $\texttt{statement}_0$`';'` $\texttt{statement}_1$ **end**

O exemplo abaixo mostra uma transição com duas construções if entrelaçadas:

```
from idle if x>4 then if y< 5 then to endState end end
```

Como se pode observar, a transição resultante é equivalente a original e possui uma única construção **if**.

```
from idle if x>4 and y< 5 then to endState end
```

3.2.5 Comunicação

O terceiro grupo de operações é o responsável por simplificar a comunicação síncrona realizada a partir da composição paralela dos processos e componentes. Esta simplificação ocorre sincronizando-se as máquinas de estados comunicantes dos processos e componentes do sistema. Esta sincronização inicia-se a partir da separação das portas sobrecarregadas. Continuando, os componentes são aplainados (*flatten*). Em seguida, os operadores de composição **sync** e **shuffle** são substituídos pelo operador **par**. Após a expansão dos operadores, os processos são instanciados, e por último, ocorre a sincronização das máquinas de estados comunicantes dos processos.

Operação para Criação de novas Portas

A primeira operação a ser executada deste grupo é a expansão das portas sobrecarregadas. Esta operação é efetuada sobre as portas que possuem mais de um perfil, e consiste em criar uma nova porta para cada perfil adicional. A regra referente a esta operação é apresentada abaixo:

Regra: Expansão **Portas Sobrecarregadas**

Enunciado

$$\texttt{[in] [out] port':' profile\_0} \mid \cdots \mid \texttt{profile}_i$$
$$\forall i \in N$$
$$\Downarrow$$

Código Gerado

$$\texttt{[in] [out] port\_0':' profile}_0\ \texttt{','}$$
$$\vdots\texttt{','}$$
$$\texttt{[in] [out] port\_i':' profile}_i$$

Por exemplo, o processo e1 apresentado abaixo possui uma porta sobrecarregada em sua interface (pInOut). Esta porta é formada por um perfil para receber os dados do tipo inteiro e outro para enviar dados do tipo booleano.

```
process e1 [in out pInOut : int | bool] is
.....................................................
from state1 pInOut? x; to state2
from state2 pInOut! true; to state2
.....................................................
component c1 is
port portLocal : int | bool
sync     e1[portLocal]
         e2[portLocal]
end
```

Como é possível observar, após a operação de expansão de portas sobrecarregas, será criada uma nova porta para cada perfil. A porta sobrecarregada *pInOut* deu origem a duas portas, uma chamada *pInOut_1* para entrada de dados do tipo inteiro e outra chamada *pInOut_2* para saída de dados booleanos. Esta operação não é executada somente sobre a interface do processo, ela também expande as portas locais dos componentes (*portLocal* $\Rightarrow$ *portLocal*_1 e *portLocal*_2).

```
process e1 [in out pInOut_1 : int, in out pInOut_2 : bool]
.....................................................
from state1 pInOut_1? x; to state2
from state2 pInOut_2! true; to state2
.....................................................
component c1 is
port portLocal_1 : int, portLocal_2 : bool
sync     e1[portLocal_1,portLocal_2]
         e1[portLocal_1,portLocal_2]
end
```

Operação para Substituição dos Operadores Sync

O operador **sync** é uma simplificação do operador **par** quando todas as portas são observáveis. Esta operação reescreve a composição substituindo o operador **sync** pelo operador **par**, colocando em evidências todas as portas observáveis locais.

Sendo *Pti* o conjunto de canais de uma instância e *C* o conjunto de canais do componente, o conjunto de portas observáveis da j-ésima instância é definido como: $Pl_j = \{\langle \bigcup_{n=0}^{k} \rangle Pti_n \bigcap Pti_j \bigcap C\}$ $\forall k,n \in N$. Esta operação é descrita pela regra apresentada abaixo:

Regra: Expansão Operador **sync**

$$\texttt{instance}_j\texttt{::=name ['['}Pti_j\texttt{']'][('}Pmi_j\texttt{')']}$$

$$\texttt{component\_decl::= }\textbf{component}\texttt{ name ['['}Pt\texttt{']'][('}Pm\texttt{')'] is}$$
$$[\textbf{var } V]$$
$$[\textbf{port } C]$$
$$[\textbf{priority } Pr]$$
$$C$$

Regra número 1:

$$\texttt{composition::=}\textbf{sync}\ \texttt{instance}_0$$
$$\vdots$$
$$\texttt{instance}_j$$
$$\textbf{end}$$

$$\forall j,n \in N$$
$$\Downarrow$$

Código Gerado

$$\texttt{composition::=}\textbf{par}\ \ \{\ \bigcup_{n=0}^{j} Pti_j\ \}\ \cap\ Pti\_0\ \}\ \cap\ C\ \ \texttt{'->'}\ \texttt{instance}_0$$
$$\vdots$$
$$\{\ \bigcup_{n=0}^{j} Pti_j\ \}\ \cap\ Pti\_j\ \}\ \cap\ C\ \ \texttt{'->'}\ \texttt{instance}_j$$
$$\textbf{end}$$

O exemplo abaixo mostra dois processos, e_1 e e_2, que se comunicam através das portas $port_1$ e $port_2$:

component c1 [port3:none] **is**
port port1: int, port2: bool
sync e1[**port3**,**port1**,**port2**]
e2[**port2**,**port1**]
end

O resultado da substituição do operador **sync** pelo **par** é apresentado abaixo:

component c1 [port3:none] **is**
port port1: int, port2: bool
par **port1**,**port2** $->$ e1[port1,port2]
port1,**port2** $->$ e2[port2,port1]
end

A porta nomeada *port3* não faz parte da lista de portas observáveis do operador **par** (C_o) -

instância do processo e_1 - porque esta porta não é uma canal de comunicação, ou seja, *port*3 não pertence ao conjunto C. Assim, esta porta não é observável dentro deste componente.

Operação para Substituição do Operador Shuffle

O operador **shuffle** também é uma simplificação do operador **par** quando nenhuma porta é observável, ou seja, não ocorre nenhuma comunicação entre os processos/componentes envolvidos. Esta operação realizará a substituição de todos os operadores **shuflle** por operadores **par**, deixando vazio o conjunto de portas observáveis. A regra para esta expansão é apresentada abaixo:

Regra: Expansão Operador **shuffle**

`composition::=`**shuffle** `composition`^{n} **end**

$\Downarrow$

Código Gerado

`composition::=`**par** `( '->' composition)`^{n} **end**

Um exemplo de substituição do operador **par** segue abaixo:

Shuffle
`train [app,exit]`
`train [app,exit]`
end

$\longrightarrow$

Par
`'->' train [app,exit]`
`'->' train [app,exit]`
end

Operação de Instanciação

Um processo ou um componente FIACRE pode ser visto como a abstração de uma classe de elementos que possuem características em comum. A instanciação desses elementos é feita através da passagem dos parâmetros formais e dos canais de comunicações. No contexto desse trabalho, optou-se por fazer esta instanciação em duas etapas, que são realizadas em momentos diferentes.

A primeira etapa realiza a instanciação levando em conta somente os parâmetros formais fornecidos. Esta operação resulta em um novo processo ou componente, dependendo da natureza do elemento em questão, no qual as variáveis locais são unificadas com os parâmetros formais passados pela instância. Estes novos elementos são diferentes dos originais, porque suas interfaces preservam somente os canais de comunicação. A segunda etapa da instanciação, a qual é a unificação dos canais com as portas do elemento, é feita diretamente pela operação de composição.

O exemplo abaixo ilustra um sistema com dois processos (e_1 e e_2) e um componente (c_1). Este componente é formado pela composição síncrona de duas instâncias, uma do processo e_1 ($e_1[port_1](0,true)$) e outra do e_2 ($e_2[port_1](4)$).

```
process e1 [out pOut_1: int] (msg:int, start:bool) is
.....................................................................
from state1 if start=true then to state2 end
.....................................................................
process e2 [in pIn_1: int] (cap:int) is
.....................................................................
component c1 is
port port1: int,
sync        e1[port1] (0,true)
                e2[port1] (4)
end
```

As instâncias $e_1\_0\_true[port1]$ e $e_2\_4[port1]$ resultam nos processos *e1_0_true [out pOut_1: int]* e *e2_4 [in pIn_1: int]*. A transição ***from*** *state1* ***if*** *true=true* ***then*** *to state2* ***end*** do processo e_1 ilustra o processo de unificação dos parâmetros formais efetuado no sistema de transições:

```
process e1_0_true [out pOut_1: int] is
.....................................................................
from state1 if true=true then to state2 end

process e2_4 [in pIn_1: int] is
.....................................................................
component c1
port port1: int, port2: bool
sync
        e1_0_true[port1]
        e2_4[port1]
end
```

Operação para Expansão dos Componentes

Um componente pode ser descrito a partir de subcomponentes interconectados por canais de comunicações (portas locais). No entanto, este conceito de hierarquia não é aceito por TTS-TINA. Logo, precisamos aplainar o componente principal.

A Figura 3.13 mostra um exemplo de "aplainamento" do componente principal chamado **main**.

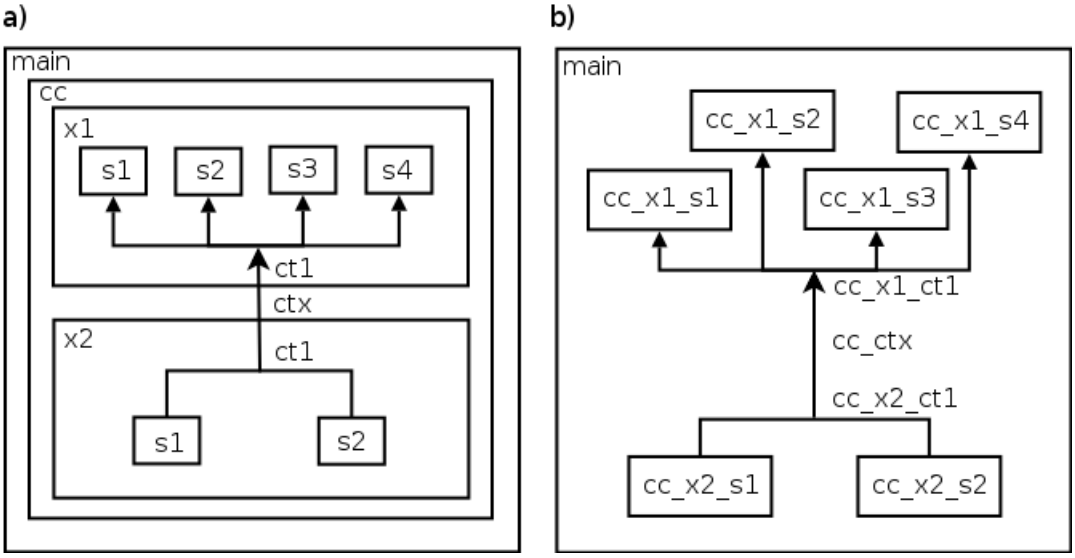


Figura 3.13: *a)Antes de "aplainar" o componente main b)Componente main "aplainado".*

O resultado desta operação é um componente descrito pela composição paralela dos processos elementares, que estão localizados nas extremidades de uma árvore estruturada associada ao modelo hierárquico do sistema. Durante esta modificação, conflitos de nomes são resolvidos a fim de preservar a equivalência entre as estruturas FIACRE e SUBFIACRE.

Operação de Composição dos Processos

A operação de composição pode ser resumida como a sincronização das máquinas de estados comunicantes dos processos. Esta sincronização ocorre através da união das transições etiquetadas, que compõem estas máquinas de estados, de acordo com a topologia da comunicação. Esta topologia é definida pelos operadores **par** em conjunto com a lista de portas observáveis para cada instância. Esta operação é dividida em quatro etapas, são elas:

1. Primeiro, um operador **par** fornece uma lista de instâncias, cada uma com o seu respectivo conjunto de portas observáveis.

2. Em seguida, é realizada a segunda etapa da instanciação dos processos, ou seja, as portas da interface do processo são unificadas com os canais de comunicação fornecidos pela instância.

Esta unificação alterará as etiquetas das transições que compõem a máquina de estado das instâncias.

3. A terceira etapa realiza a união das transições a partir do conjunto de portas observáveis de cada instância. Esta união ocorre entre as transições etiquetadas pelos canais de comunicação que fazem parte do conjunto de portas observáveis. Classificamos este conjunto como portas observáveis porque ele determina se as transições de uma instância podem ou não se unir com as de outra instância. Durante este processo de sincronização, os estados e as transições são renomeados com nomes únicos preservando a equivalência entre os modelos. As regras de união de transição são apresentadas abaixo:

Regra: União de Transições sem passagem de valor (**none**)

from $\texttt{stateI}_1$ port';'$\texttt{statement}_1$';' **to** $\texttt{stateF}_1$

$\vdots$

from $\texttt{stateI}_j$ port';'$\texttt{statement}_j$';' **to** $\texttt{stateF}_j$

$\forall j \in N_+$

$\Downarrow$

Código Gerado

from $\texttt{stateI}_1 \cdots \texttt{stateI}_j$ port ';'
$\texttt{statement}_1$';'$\cdots$';'$\texttt{statement}_j$';' **to** $\texttt{stateF}_1 \cdots \texttt{stateF}_j$

Esta regra promove a união entre transições sem a passagem de valor. A transição resultante consiste na união dos comportamentos de ambas as transições.

Regra: União de Transições **?** com **!**

from $\texttt{state1}_i$ port '?' $\texttt{var}_1$','...','$\texttt{var}_j$';'**to** $\texttt{state1}_f$

from $\texttt{state2}_i$ port '!' $\texttt{exp}_1$','...','$\texttt{exp}_j$';'**to** $\texttt{state2}_f$

$\forall i, j \in N_+$

$\Downarrow$

Código Gerado

from $\texttt{state1}_i$ $\texttt{state2}_i$ port '!' $\texttt{exp}_1$','...','$\texttt{exp}_j$';'
$\texttt{var}_1$=$\texttt{exp}_1$';'...';'$\texttt{var}_j$=$\texttt{exp}_j$';'
to $\texttt{state1}_f$ $\texttt{state2}_f$

A regra apresentada acima realiza a união de transições com os operadores '?' e '!'. Esta união promove a unificação dos parâmetros enviados ('!') com as variáveis selecionadas para recepção ('?'). A transição resultante preserva somente o operador '!', pois esta transição pode ainda fazer parte de outras composições.

Regra: União de Transições **!** com **!**

from $\texttt{stateI}_1$ port '!' $\texttt{exp}_{11}$','...','$\texttt{exp}_{1j}$';'
$\texttt{statement}_1$';' **to** $\texttt{stateF}_1$
⋮
from $\texttt{stateI}_i$ port '!' $\texttt{exp}_{i1}$','...','$\texttt{exp}_{ij}$';'
$\texttt{statement}_i$';' **to** $\texttt{stateF}_i$

$\forall\ i, j\ \in\ N_+$

⇓

Código Gerado

$\texttt{transition}_r$::=**from** $\texttt{stateI}_i \cdots \texttt{stateI}_i$ port '!' $\texttt{exp}_{11}$','...','$\texttt{exp}_{1j}$';'
if $\texttt{exp}_{11}$=$\texttt{exp}_{i1}$ **and** $\cdots$ **and** $\texttt{exp}_{1j}$=$\texttt{exp}_{ij}$
then $\texttt{statement}_1$';'$\cdots$';'$\texttt{statement}_j$';'
to $\texttt{stateF}_1 \cdots \texttt{stateF}_i$ **end**

A regra apresentada acima realiza a união de transições com os operadores '!' e '!'. Esta união cria um laço condicional entre as transições, ou seja, todos os valores enviados precisam ser iguais para ocorrer esta composição. A transição resultante preserva o operador '!' com uma lista de expressões, pois esta transição pode ainda fazer parte de outras composições. Apenas a primeira lista de expressões (*exp1*) é preservada porque os laços condicionais criado impõem que todas as expressões sejam iguais, não havendo necessidade de manter as outras listas.

Regra: União Transições **?** com **?**

from $\texttt{stateI}_1$ port '?' $\texttt{var}_{11}$','⋯','$\texttt{var}_{11j}$';'
$\texttt{statement}_1$';'**to** $\texttt{stateF}_1$
⋮
from $\texttt{stateI}_i$ port '?' $\texttt{var}_{i1}$','⋯','$\texttt{var}_{ij}$';'
$\texttt{statement}_i$';'**to** $\texttt{stateF}_i$
$\forall i, j \in N_+$

⇓

Código Gerado

$\texttt{transition}_r$::=**from** $\texttt{stateI}_1$ ⋯ $\texttt{stateF}_i$
port '?' $\texttt{var}_{11}$','⋯','$\texttt{var}_{1j}$';'
$\texttt{var}_{21}$':='$\texttt{var}_{11}$';'⋯';'$\texttt{var}_{2j}$':='$\texttt{var}_{1j}$';'
⋮
$\texttt{var}_{i1}$':='$\texttt{var}_{11}$';'⋯';'$\texttt{var}_{ij}$':='$\texttt{var}_{1j}$';'
$\texttt{statement}_1$';' ⋯ ';'$\texttt{statement}_i$';'
to $\texttt{stateF}_1$ ⋯ $\texttt{stateF}_i$

Esta última regra retrata a união de transições que possuem o operador do tipo "recepção" ('?'). Neste caso o operador '?' é preservado em conjunto com a primeira lista de variáveis(var_{1x}). As outras variáveis (var_{ix}) vão receber os valores fornecidos na primeira lista (var_{ix}:=var_{1x}).

4. A quarta e última etapa consiste em retornar um novo processo formado pelas transições resultantes. As etiquetas (portas) não serão removidas porque elas serão necessárias para resolver as Prioridades e Restrições Temporais.

O exemplo abaixo ilustra a composição entre duas instâncias (*requerente* e *requerido*) através do canal de comunicação l_2, os quais fazem parte do componente principal chamado *main*.

```
type mensagem:enum pedido, fechar, ok,nok,bip end
process requerente[in pIn, out pOut:mensagem] is
states requerer, esperar init requerer
var tmp:mensagem:=bip
from requerer pOut! pedido; to esperar
.....................................................
process requerido[in pIn,out pOut:message] is
```

```
states esperar, responde, aberto init esperar
var tmp:mensagem:=bip
from esperar pIn? tmp; to responde
.....................................................
from aberto pIn? tmp; to responde
.....................................................
component main
port l1, l2:message
par     l1,l2 -> requerente[l1,l2]
        l1,l2 ->  requerido[l2,l1]
end
```

O resultado da operação de composição é apresentado abaixo:

```
type mensagem:enum pedido, fechar, ok,nok,bip end
process process_main is
states requerer_requerente, esperar_requerente
esperar_requerido, responde_requerido, aberto_requerido
        init requerer_requerente esperar_requerido
var tmp_requerente:mensagem:=bip
         tmp_requerido:mensagem:=bip
.............................................................
from   requerer_requerente  esperar_requerido
         l2!; tmp_requerido := pedido;
         to esperar_requerente  responde_requerido  end
.............................................................
from requerer_requerente  aberto_requerido
         l2!; tmp_requerido := requete;
         to esperar_requerente  responde_requerido  end
.............................................................

component main is
port l1, l2:mensagem
process_main
```

3.2.6 Prioridades e Restrições Temporais

A resolução das Prioridades e Restrições Temporais consiste apenas em alterar a forma como esta informação é representada no sistema. Então, as prioridades e as restrições temporais, que estão inicialmente relacionadas aos canais de comunicação (portas locais), serão associadas em SUBFIACRE às transições para simplificar o processo de tradução.

Prioridades

Antes de explicar como esta operação é realizada, é importante mencionar que o formalismo TTS não possui uma representação explícita de prioridades. Contudo, TTS-TINA permite definir prioridades entre as transições. Com isto, esta operação consiste em substituir a lista de prioridades FIACRE, a qual é formada por nomes de portas locais, por uma outra lista formada por nomes de transições. Esta nova lista define a relação de prioridade entre as transições. Como FIACRE trabalha com transições anônimas, todas as transições serão nomeadas com nomes únicos. Para ilustrar esta operação, apresenta-se o exemplo a seguir:

```
process process_main is
states ..........................................................
var ..............................................................
        T1 : from demander attendant l2;
                if requete=requete
                        then tmp repondeur:=requete;
                        to attendant repondre end
        T2 : from demander ouvert l2;
                if requete= fermer
                        then tmp repondeur:=requete;
                        to attendant repondre end
        T3 : from repondre attendant l1;
                if tmp_repondeur=requete and ok=ok
                        then tmp_demandeur=ok ;
                        to ouvert ouverte end

component main is
port l1, l2:message
priority l1 > l2
```

```
        process_main
```

A lista de prioridades FIACRE(*priority l1 > l2*) deste exemplo define que a porta local l_1 é mais prioritária que l_2. Após a operação de expansão, esta lista será redefinida com os nomes das transições as quais as portas locais estavam associadas, neste caso, a lista resultante é *priority T3 > T1 T2*.

```
. . . . . .
component main is
port l1, l2:message
priority T3 > T1 T2
process_main
. . . . . .
```

Esta lista de prioridades SUBFIACRE interpreta o operador > da mesma forma que FIACRE, ou seja, os nomes alocados do lado esquerdo são mais prioritários que os do lado direito.

Restrição Temporal

A operação de resolução das restrições temporais é similar à operação de prioridades. Entretanto, as restrições neste caso são associadas diretamente às transições.

Apresentamos abaixo um exemplo SUBFIACRE que possui associado aos seus canais de comunicação (l_1 e l_2) a restrição temporal *[2,5]*.

```
process process_main is
states . . . . . .
. . . . . .
        T1:from demander attendant l2;
                if requete=requete
                then tmp repondeur:=requete;
                        to attendant repondre end
. . . . . .
component main is
port l1, l2 : message in [2,5]
        process_main
```

A restrição temporal *[2,5]* será associada às transições etiquetadas pelos canais l_1 e l_2. Neste caso, a transição T_1, que é etiquetada pelo canal de comunicação l_2, será associada à restrição temporal *[2,5]*. O sistema resultante SUBFIACRE com as restrições já associadas às transições é apresentado abaixo:

```
process process_main is
states ..........................................
.....................................................
        T1 in [2,5] : from demander attendant l2;
                if requete=requete
                then  tmp repondeur:=requete;
                        to attendant repondre
.....................................................
component main is
port l1, l2:message in [2..5]
        process_main
```

3.3 Geração - Tradução de SUBFIACRE para TTS

A segunda etapa desta tradução conceitual consiste na geração do formalismo alvo, em outras palavras, na tradução de SUBFIACRE em TTS.

3.3.1 Princípios de Tradução

Apesar de tratarmos aqui da tradução de SUBFIACRE para TTS, optamos por orientar esta tradução utilizando a representação gráfica adotada neste trabalho (ver seção 3.1). Esta escolha foi feita no sentido de auxiliar a compreensão do sistema TTS resultante, pois assim este pode ser representado a partir de uma Rede de Petri Temporal estendida.

O princípio desta tradução consiste em representar os dados do programa SUBFIACRE relativos ao controle do sistema como lugares da rede TPN. Os "cálculos" efetuados sobre estas variáveis serão interpretados como arcos da rede. Estas regras serão esclarecidas no decorrer desta seção.

3.3.2 Tipos de Dados

O formalismo matemático TTS não impõe nenhuma restrição relacionada ao domínio da estrutura de dados do sistema. Logo, a tradução da parte do código resultante SUBFIACRE referente à estrutura de dados é importante para questões de alocação de memória da ferramenta de verificação. Assim, sua tradução depende da ferramenta escolhida. O apêndice D mostra maiores detalhes da tradução dos tipos SUBFIACRE para TTS-TINA, formato de entrada do ambiente TINA.

3.3.3 Processos

A linguagem simplificada SUBFIACRE comporta somente um processo. A tradução deste processo pode ser considerada como a peça fundamental na transformação do código resultante SUBFIACRE em TTS.

A sintaxe SUBFIACRE abaixo mostra que um processo é formado por: uma lista de estados, uma de variáveis e uma de transições, conforme apresentado abaixo:

```
initalize::=literal
        | ('+'|'-')NATURAL
        | '['initializer*,']'
        | ''(field'='initializer)*,''
var_dec::=var*,':'type[':='initializar]
process_dec::=process name is
        states pstates*, init state
        [var var_dec*,]
        transition*
```

O formalismo SUBFIACRE, como FIACRE, interpreta a palavra **state** como o estado de um processo. Para TTS, estado é uma interpretação do conjunto ϑ (variáveis). Logo, considerando somente a definição dada pelo formalismo TTS, um estado SUBFIACRE é traduzido como uma variável que pertence ao conjunto ϑ. Porém, a partir dos princípios de tradução inicialmente definidos, vamos considerar que os estados SUBFIACRE fazem parte do controle do sistema e por isto serão interpretados como os lugares de uma Rede de Petri Temporal, distinguindo-os das outras variáveis do sistema.

A tradução das variáveis é feita de forma direta, ou seja, uma variável SUBFIACRE é automaticamente interpretada como uma variável TTS. Quanto à construção **initialize**, sua tradução é fundamental para definir o estado inicial do sistema.

A tradução do sistema de transições simplificado presente em SUBFIACRE depende da forma da transição. Uma transição SUBFIACRE pode assumir duas formas:

1. `IDENT` **in** `time_interval:` **from** $\texttt{state}^{+}$ $[\texttt{access}^{+}_{,}\texttt{':='exp}^{+}_{,}]^{*}_{;}$ **to** $\texttt{state}^{+}$
2. `IDENT` **in** `time_interval:` **from** $\texttt{state}^{+}$ **if** `exp` **then** $[\texttt{access}^{+}_{,}\texttt{':='exp}^{+}_{,}]^{*}_{;}$ **to** $\texttt{state}^{+}$ **end**

A tradução destas duas formas de transições segue 5 regras simples de tradução, são elas:

1. `IDENT` **in** `time_interval`

   Esta construção determina a restrição temporal associada à transição ($t_i \leq t \leq t_m$).

2. **from** $\texttt{state}^{+}$

   Esta construção determina os lugares "precedentes" de uma transição, ou seja, ela define os arcos de origem de uma transição.

3. **if** `exp` **then** `.....` **end**

   A construção **if** associa uma expressão condicional à transição.

4. $\texttt{access}^{+}_{,}\texttt{':='exp}^{+}_{,}\texttt{]}$

   Esta construção pode ser interpretada como as "ações" efetuadas sobre a estrutura de dados do sistema. Ela será associada à transição, assim, quando a transição for disparada, esta ação será executada.

5. **to** $\texttt{state}^{+}$

   Esta construção determina os lugares "seguintes" de uma transição, ou seja, ela define os arcos de destino de uma transição.

As Figuras 3.14 a) e b) foram obtidas a partir da aplicação das regras definidas acima sobre as duas formas de transições SUBFIACRE apresentadas anteriormente. As letras a) e b) são referentes às transições SUBFIACRE apresentadas anteriormente, respectivamente sem e com expressão condicional.

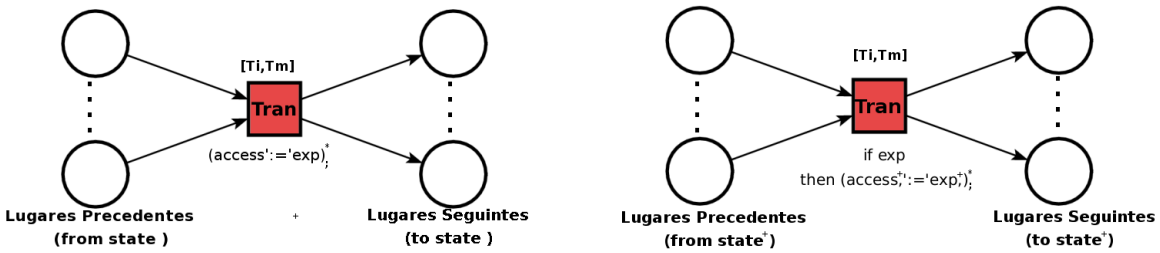


Figura 3.14: *Representação TTS das transições: a) sem expressão condicional. b)com expressão condicional*

3.3.4 Componente

O programa SUBFIACRE resultante possui apenas um componente. Conforme a sintaxe SUBFIACRE apresentada abaixo, este componente é formado por um conjunto de canais de comunicação, um conjunto de prioridades e uma instância de um processo.

```
component_decl::= component name is
        [port (port_dec [in time_interval])+,]
        [priority (port+|>port+|)+,]
        composition
instance::=name
composition::= instance
program::=      [type_decl*]
                [channel_decl*]
                process_decl
                component_decl
```

A tradução do componente SUBFIACRE consiste apenas em traduzir o processo que ele possui. As portas e a lista de prioridades não são mais importantes para a tradução porque todas as informações relevantes já foram associadas ao comportamento do sistema.

3.4 Compilador

A partir do esquema conceitual de tradução (Expansão e Geração) de FIACRE para TTS, foi desenvolvido um compilador para operacionalizar a verificação de sistemas FIACRE no ambiente TOPCASED. Um compilador pode ser definido de forma simples como: "um programa que lê um programa escrito em uma linguagem - a linguagem fonte - e traduz em um programa equivalente numa outra linguagem - a linguagem alvo"[22].

3.4.1 Definição e Objetivo

A tradução de FIACRE para TTS-TINA pode ser considerada como uma "compilação de código fonte para código alvo". O compilador recebe como entrada modelos escritos em FIACRE, que pode ser vista como uma linguagem de nível intermediário, e produz como resultado modelos escritos no formalismo TTS-TINA. O principal objetivo deste programa é realizar esta tradução preservando as informações e propriedades presentes no modelo original.

3.4.2 Arquitetura do Compilador

"Um compilador funciona por análise-síntese, ou seja, ao contrário de substituir cada construção da linguagem fonte por uma equivalente da linguagem alvo, ele começa por analisar o texto fonte a fim de construir uma representação intermediária que ele traduz por sua vez em linguagem alvo" [22].

O compilador implementado segue a arquitetura clássica de um compilador. Esta arquitetura é dividida em duas partes: análise (parte frontal) e síntese (parte final) (ver Figura 3.15). A parte frontal é responsável pela tradução do código fonte em uma representação intermediária. A parte final traduz esta representação intermediária no formato alvo. Esta abordagem permite dividir a complexidade do programa separando a parte frontal, que fica responsável pelas operações que envolvem a semântica da linguagem, da parte final, que se concentra na produção do código final de forma eficiente e correta. A representação intermediária consiste em uma estrutura de dados chamada de "árvore abstrata".

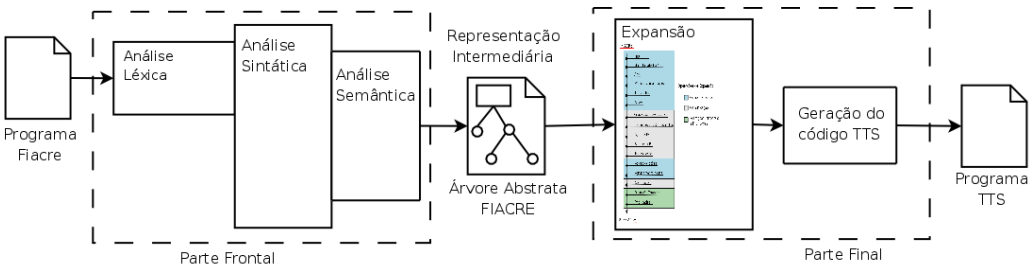


Figura 3.15: *Arquitetura do Compilador.*

Parte Frontal

A parte frontal é responsável pela criação do formato intermediário e por checar a sintaxe por possíveis erros lógicos. Para resolver esta tarefa, a parte frontal é dividida em 4 fases, são elas:

1. análise Léxica: divide o código fonte em pequenos pedaços (*"tokens"* ou *"terminais"*), cada um representando uma unidade atômica da linguagem, por exemplo, uma palavra-chave, um símbolo identificador ou um nome;

2. análise Sintática: identifica a estrutura sintática do código fonte. Está somente focado na estrutura, ou seja, identifica a ordem dos *tokens* e compreende a estrutura hierárquica no código;

3. análise Semântica: esta fase verifica os erros semânticos no programa fonte e captura as informações de tipo para a fase subseqüente de geração de código. Utiliza a estrutura hierárquica determinada pela fase de análise sintática, a fim de identificar os operadores e operandos das expressões e enunciados. Grande parte dos erros são descobertos nesta etapa. O documento [1] apresenta na íntegra a semântica da linguagem FIACRE;

4. representação Intermediária: O programa fonte é transformado em uma representação intermediária (árvore abstrata).

O anexo C demonstra detalhes da implementação dos analisadores léxicos e sintáticos para a linguagem FIACRE.

Parte Final

A parte final é responsável pela compilação do código no formato alvo. Nesta parte, a representação intermediária é simplificada e posteriormente traduzida na linguagem alvo. Estas tarefas são divididas em duas fases:

1. expansão: as operações de expansão descritas anteriormente são executadas com o objetivo de melhorar o código intermediário, de modo que o código resultante seja de fácil codificação na linguagem alvo.

2. geração de Código: a representação intermediária simplificada é transformada em código alvo. Como está descrito no anexo D, este código resultante compreende dois arquivos: um responsável pela estrutura da rede (**.net**) e outro responsável pelas manipulações das variáveis (**.c**).

Maiores informações sobre as fases de Expansão e Geração de Código são fornecidas no anexo D.

3.4.3 Ferramentas Utilizadas

O tradutor foi implementado utilizando a linguagem Standard ML (SML) [23] e o compilador MLton [24] para gerar os arquivos executáveis. SML é uma linguagem de programação que combina a elegância da programação funcional com a eficácia da programação imperativa. O analisador léxico e sintático foi implementado utilizando as ferramentas MLLex e MLYacc [25] [26] [27].

3.5 Exemplo e Testes

A Figura 3.16 apresenta a estrutura de controle do Sistema de Transição Temporizado resultante do exemplo clássico de dois trens cruzando uma passagem de nível, cuja descrição FIACRE foi apresentada no Capítulo 2. Este sistema possui 15 lugares e 17 transições. Os lugares da rede estão agrupados de acordo com os processos aos quais eles pertenciam.

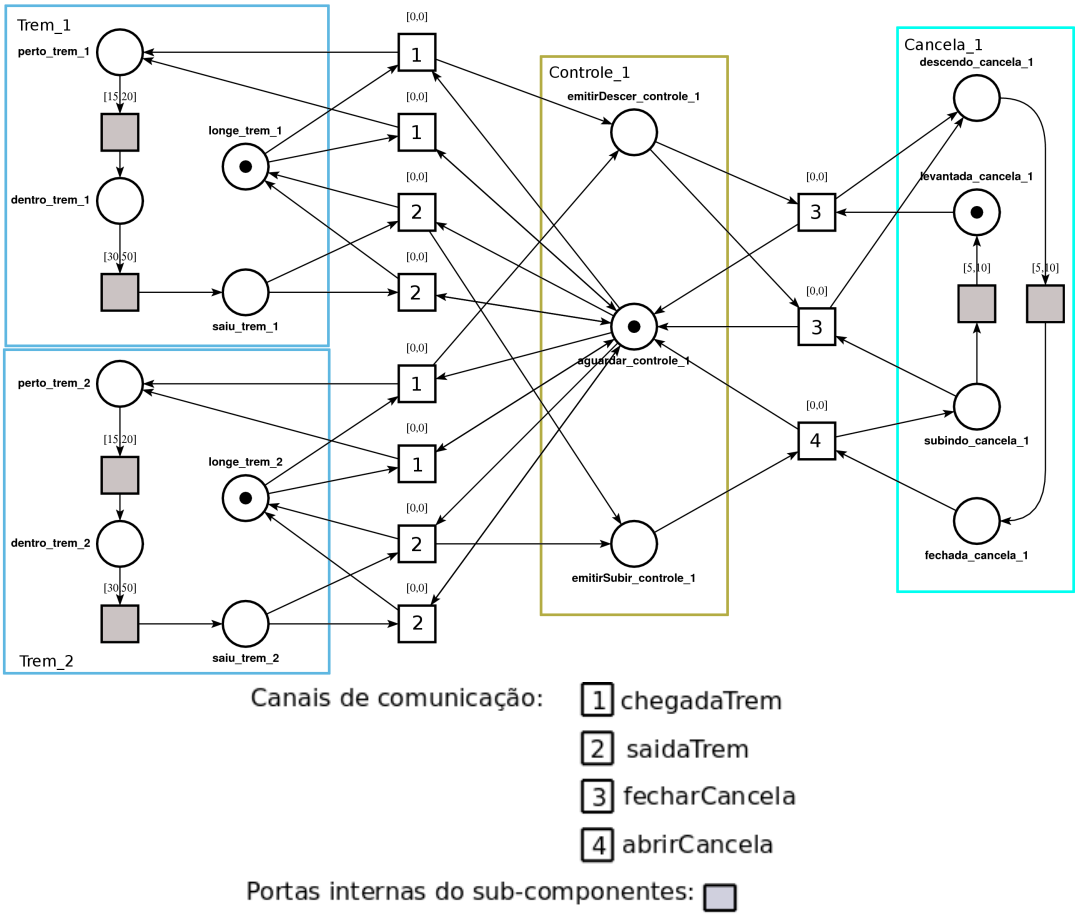


Figura 3.16: *TTS do exemplo clássico de trens cruzando uma passagem de nível.*

Como podemos observar, os lugares que modelam os estados dos processos do sistema (*trem_1*, *trem_2*, *controle_1* e *cancela_1*) estão agrupados nos seus respectivos retângulos. As transições estão organizadas de acordo com a porta responsável pela composição. Por exemplo, as transições coloridas com a cor vermelha são resultantes das operações de composição realizadas pelo canal de comunicação *chegadaTrem*. O código SUBFIACRE deste modelo se encontra em anexo (Anexo B).

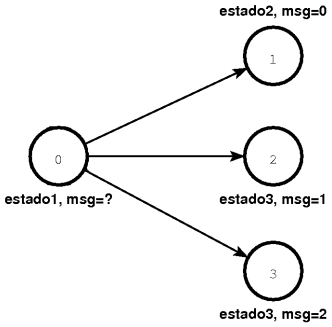
3.6 Conclusão

Este capítulo apresentou um esquema conceitual para traduzir modelos FIACRE para o formalismo TTS. Este esquema é realizado em duas etapas, chamadas expansão e geração, interligadas por um formato intermediário. A fase de expansão da descrição FIACRE compreende 16 operações, que simplificam a descrição original de modo que o código resultante seja mais simples de ser codificado na linguagem alvo. Este modelo simplificado é descrito em um formato intermediário, chamado SUBFIACRE, que pode ser considerado como uma descrição simplificada de FIACRE, muito próxima do formalismo TTS. A partir desta descrição simplificada, o código final em TTS é gerado.

A partir deste modelo conceitual, foi implementado um compilador (*front-end*) para tornar operacional a verificação de FIACRE com o ambiente TINA. Este compilador, que foi desenvolvido utilizando técnicas clássicas, é formado por duas partes: parte frontal e parte final. A parte frontal compreende os módulos responsáveis pela análise léxica, sintática, semântica e a produção de uma representação intermediária. A parte final é formada pelas operações de expansão e geração, que foram descritas no modelo conceitual, para traduzir a representação intermediária (árvore abstrata) no formato alvo, que é neste caso o formalismo TTS.

Este compilador vem sendo utilizado no contexto do projeto TOPCASED desde outubro de 2007, quando foi apresentado em uma reunião com os parceiros e órgãos financiadores do projeto. Alguns trabalhos preliminares foram conduzidos para estudar como FIACRE pode auxiliar no processo de verificação de propriedades esperadas de um sistema. Por exemplo, estudos estão sendo conduzidos com este compilador para identificar técnicas de abstração capazes de reduzir o espaço de estados gerado pelo modelo. Uma técnica simples consiste em identificar variáveis, que não são exaustivamente utilizadas pelo sistema, e armazenar valores fixos a fim de diminuir o espaço de estados gerado pelas ferramentas de verificação. A transição abaixo ilustra este caso:

```
from estado1 pIn? msg;
        if msg = 0 then to estado2
        else to estado3 end
```

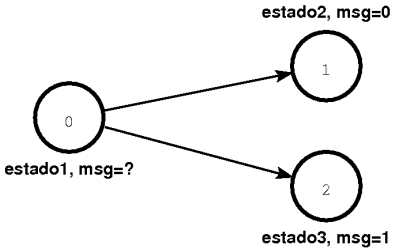


A transição acima aguarda um valor através da porta **pIn**. O valor recebido será armazenado na variável **msg**. Vamos supor que esta porta aceite somente valores no intervalo de 0 a 2 e que a

variável **msg** não é utilizada por nenhuma outra transição deste processo. Por conseguinte, os valores $\{1,2\}$ implicarão que a transição evoluirá para o **estado3**. Entretanto, o espaço gerado possuirá um estado separado para cada situação, ou seja, para cada valor possível da variável **msg**.

Para evitar estados excedentes para cada valor da variável *msg*, é possível armazenar um valor fixo, por exemplo *msg:=0* sempre que **msg!=0**:

```
from estado1 pIn? msg;
        if msg = 0 then to estado2
        else msg := 1; to estado3 end
```



Capítulo 4

Em Direção à Tradução de SDL para FIACRE: Estudo da Expressividade

Este capítulo relata um trabalho realizado durante a definição da Linguagem FIACRE e pode ser consultado na íntegra em [28].

Devido ao posicionamento de FIACRE como uma linguagem intermediária, esta linguagem precisa ser capaz de receber os modelos provenientes de diferentes linguagens de modelagem. Por isto, uma das empresas participantes do projeto TOPCASED[1], a Communication Systems (CS), levantou questões quanto à capacidade de FIACRE receber modelos provenientes da Linguagem para Especificação e Descrição (SDL[2]). Estas questões provêm do fato que SDL possui algumas características não suportadas em FIACRE de forma nativa. Este capítulo serve de exemplo de como abordar o problema da expressividade de uma linguagem de modelagem em relação à linguagem intermediária FIACRE.

Este capítulo inicia-se com uma breve descrição da linguagem SDL e em seguida aborda a problemática da tradução entre SDL e FIACRE. A seção 2 expõe o método de tradução proposto para demonstrar que FIACRE é capaz de expressar modelos SDL. A seção 3 ilustra uma ferramenta de tradução de modelos UML SDL para FIACRE, que está sendo implementada no projeto TOPCASED em conjunto com a empresa CS. As conclusões são apresentadas na seção 4.

[1] Communications & Systems - http://www.c-s.fr/
[2] Specification and Description Language

4.1 SDL - Specification and Description Language

SDL é uma linguagem de modelagem que foi proposta pela organização ITU[3] na recomendação Z100 [29]. Esta é uma linguagem para descrição de sistemas comunicantes amplamente utilizada no campo das telecomunicações. Sua arquitetura é definida como um sistema de blocos e cada bloco pode possuir internamente outro bloco ou um processo. Cada processo possui uma máquina de estado estendida que se comunica com outros processos ou o ambiente através de sinais. Maiores informações sobre a linguagem SDL podem ser encontradas em [30] e [31].

4.1.1 Problemática - Estudo da expressividade de FIACRE e SDL

O estudo da expressividade de FIACRE é muito importante devido ao seu caráter intermediário para o ambiente TOPCASED. Durante a definição desta linguagem, um dos trabalhos o qual estivemos envolvidos foi o de assegurar que era possível escrever modelos SDL em FIACRE preservando a atomicidade das transições SDL para garantir a equivalência entre os modelos. Neste texto não iremos abordar todas as construções SDL, somente as pertinentes ao domínio de sistemas embarcados e que não podem ser traduzidas diretamente. Dentro deste contexto, é de extrema importância assegurar a tradução correta de duas construções presentes em SDL, são elas: a comunicação assíncrona e a criação dinâmica de processos. Estas duas construções não são nativas em FIACRE e por isso não possuem tradução direta.

Comunicação Assíncrona

A comunicação assíncrona em SDL é realizada a partir da troca de sinais entre processos. Assim, cada processo possui uma fila de espera do tipo FIFO (Primeiro a entrar, Primeiro a sair) onde os sinais de entrada são armazenados. Uma transição que espera um ou diversos sinais de tipos diferentes, irá tratá-los de acordo com a ordem de entrada para julgar se eles são ou não esperados. Se o sinal analisado é esperado, ele é transmitido ao ramo da transição que o espera, senão ele é descartado e o próximo da fila é analisado. Este procedimento é repetido pela transição até encontrar um sinal esperado ou esvaziar a fila de espera [31]. A Figura 4.1 ilustra uma transição que pertence ao processo e1. Sua fila possui três sinais: disc, close e open. Sendo dado que o único sinal esperado pela transição é open, os outros serão descartados.

Criação Dinâmica

[3] International Telecommunication Union

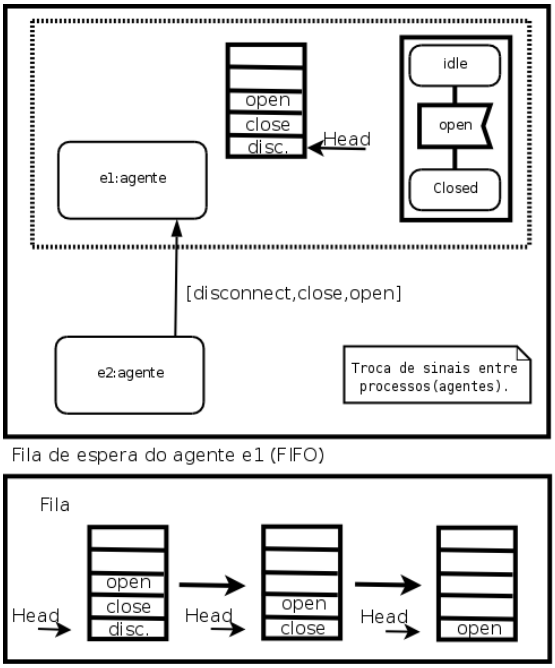


Figura 4.1: *Exemplo de uma Comunicação via uma Fila de Espera.*

SDL permite a criação dinâmica de instâncias. Este procedimento, o qual é graficamente assinalado por uma linha pontilhada entre os conjuntos de processos correspondentes (ver Figura 4.2), é realizado durante a execução de uma transição. Como as instâncias criadas fazem também parte de um conjunto de processos, o número máximo é limitado.

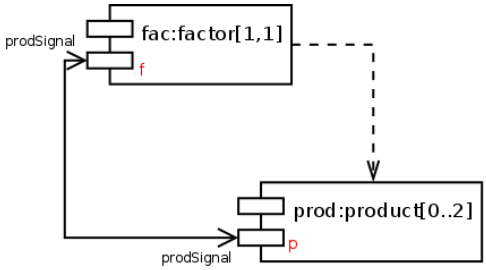


Figura 4.2: *Criação Dinâmica.*

Em SDL, as instâncias de processos pertencem a um conjunto de processos. A definição de um conjunto de processos começa pelo seu nome, em seguida pelo número mínimo e máximo de instâncias e eventualmente o nome do tipo de processos pode ser fornecido.

Uma instância pode ser criada pelo seu próprio conjunto de processos ou pela solicitação de uma outra instância (criação dinâmica). Ela pode deixar de existir quando executar a ação *stop*.

4.2 O Processo de Tradução

Para traduzir estas construções, as quais podem ser consideradas como as peças fundamentais desta tradução, foi escolhida uma abordagem baseada no uso de variáveis compartilhadas, pois esta permite manter a atomicidade das transições SDL em FIACRE. Por exemplo, uma abordagem baseada na comunicação por portas não seria capaz de reproduzir a comunicação assíncrona sem adicionar estados intermediários. Esta afirmação decorre do fato de que SDL permite a realização de duas comunicações (uma emissão e uma recepção) por transição e FIACRE aceita somente uma. Esta abordagem de variável compartilhada consiste em construir uma tabela de filas de espera para cada conjunto de processos para ser compartilhada por todos os conjuntos.

A alocação dinâmica também é traduzida em FIACRE através de variáveis compartilhadas. É importante ressaltar aqui que FIACRE é uma linguagem estática porque sua motivação principal é a verificação. Logo, a abordagem seguida aqui é que todas as instâncias precisam ser declaradas em FIACRE, porém precisamos criar um mecanismo para definir se ela está ativa ou não (*busy* ou *idle*). Devido também à atomicidade, optamos por adicionar uma variável compartilhada aos conjuntos de processos que aceitam a criação dinâmica. Esta variável fornece a informação em relação ao estado de alocação (ativo ou não) para cada instância do conjunto. Foi necessário utilizar estas variáveis porque o processo de alocação ocorre durante uma transição e ele precisa ser realizado antes do fim desta. Caso fosse usada uma outra abordagem, o processo de alocação seria concluído somente após a finalização da transição, criando sérios problemas de equivalência entre os modelos SDL e FIACRE.

O uso de variáveis compartilhadas é a peça chave para traduzir modelos SDL em FIACRE. Porém, são necessárias outras construções para completar esta tradução. Por exemplo, para operacionalizar esta abordagem baseada em variáveis compartilhadas, é preciso definir um "contexto" para o sistema. Este contexto consiste na criação de novos tipos para simplificar a tradução. Este estudo completo é apresentado em [28].

Exemplo

Como foi descrito anteriormente, tanto a comunicação assíncrona como a criação dinâmica de processos são traduzidas em FIACRE utilizando as variáveis compartilhadas. Por conseguinte, para simplificar este processo de tradução, estas variáveis são reunidas em grupos de variáveis compartilhadas, ordenadas de acordo com o conjunto ao qual o processo pertence. Logo, cada conjunto de processos SDL possuirá uma matriz de variáveis compartilhadas. A Figura 4.3 mostra graficamente algumas das variáveis compartilhadas para o exemplo apresentado na figura 4.2. Assim, cada conjunto de processos possui uma matriz de processos (fac, prod), que é na verdade um registro (*record*)

formado por uma fila de sinais(FIFO), quatro variáveis SDL - exigidas pela recomendação Z100: *self*, *offspring*, *parent* e *sender* - e um campo para sinalizar o estado de alocação da instância (**pstate**). O último campo (**pstate**) é opcional e só é necessário quando o processo pode ser criado dinamicamente (seta tracejada).

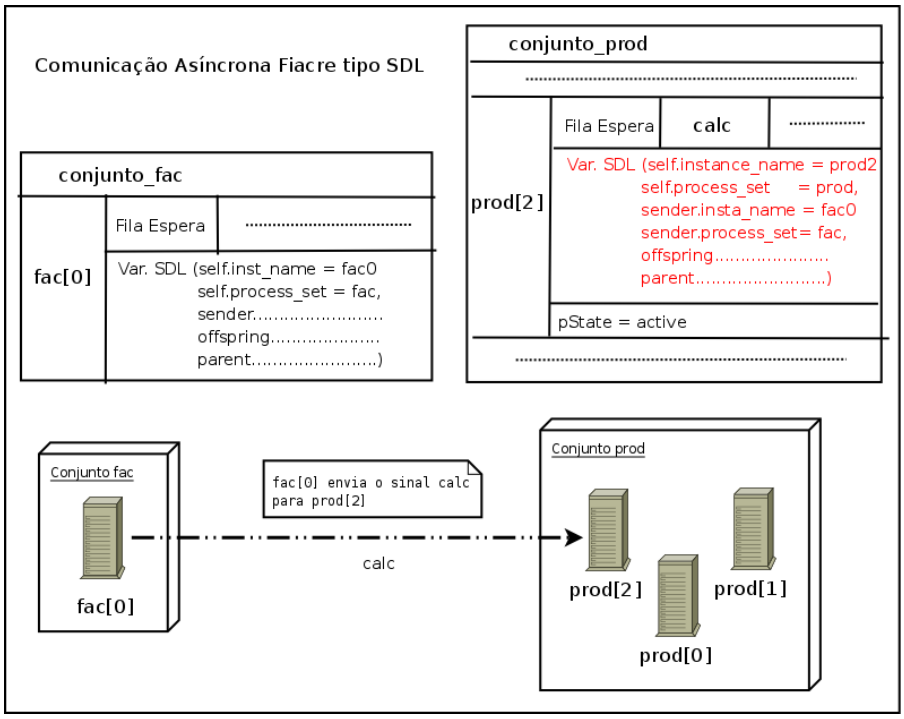


Figura 4.3: *Variáveis Compartilhadas.*

No relatório técnico [28], pode-se encontrar a tradução de SDL para FIACRE do exemplo clássico: "Cálculo da Função Fatorial Distribuído"

Considerações da Expressividade FIACRE

Este estudo foi muito importante durante a definição da linguagem FIACRE. A partir dele, três mudanças foram feitas na linguagem:

- Tipo união (*union*): a peça chave desta tradução é o uso de variáveis compartilhadas. Assim, cada conjunto de processos possui uma matriz de filas de espera do tipo FIFO. Como um processo pode aceitar tipos diferentes de dados, optou-se por criar estas filas como uniões (*union*) de todos os tipos de sinais aceitos pelo processo. Poderíamos ter usado o tipo *record*, entretanto, ocorreria um desperdício enorme de memória.

- Variáveis compartilhadas: antes deste estudo, FIACRE manipulava as variáveis pelo seu nome global, ou seja, não era possível definir nomes locais para referenciar uma variável global.

Devido às dificuldades encontradas para codificar SDL em FIACRE utilizando somente nomes globais, optamos por alterar a linguagem FIACRE e permitir a definição de nomes locais para referenciar as variáveis globais.

- Macros: apesar de macros não representarem uma alteração da linguagem, este estudo tornou evidente a necessidade de se definir macros parametrizados para a linguagem.

4.3 Mapeamento do profile UML-SDL para FIACRE

O estudo realizado foi importante para integrar a linguagem SDL na arquitetura de verificação do ambiente TOPCASED. Esta integração de SDL na arquitetura está sendo feita pela empresa CS utilizando a linguagem ATL em conjunto com o *plugin* TOPCASED para a ferramenta Eclipse. Para isto, a empresa participante isolou as construções abstratas e as suas inter-relações, seguindo as recomendações documentadas no documento SDL Z100, para implementar um meta-modelo da linguagem SDL. A partir deste meta-modelo, a CS optou por modelar um perfil (*profile*) UML para SDL. Com esta abordagem, podemos reutilizar muitas construções e mecanismos UML.

O modelo de entrada da ferramenta implementada pela CS utiliza este perfil UML para SDL. A Figura 4.4 mostra as etapas internas desta ferramenta. A primeira etapa transforma o meta-modelo de origem (SDL) no meta-modelo alvo (FIACRE). A segunda transforma esse meta-modelo alvo em linguagem concreta FIACRE.

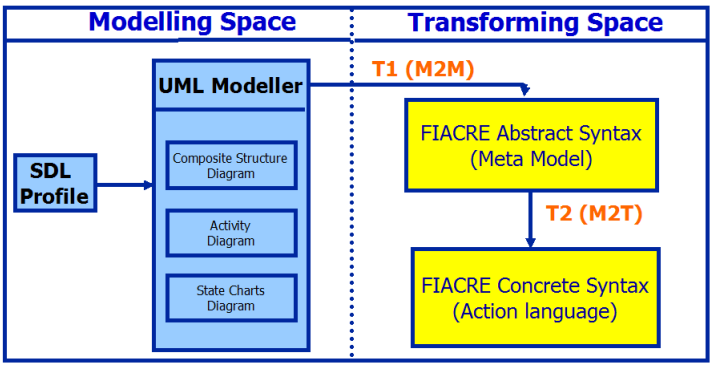


Figura 4.4: *Tradução de UML SDL para* FIACRE*[5].*

A parte da ferramenta denominada T1 (M2M) da Figura 4.4 emprega a abordagem de tradução de SDL para FIACRE apresentada neste capítulo. A empresa CS formalizou esta tradução através de um mapeamento da estrutura, do comportamento e da adição de 35 regras de transformação ATL. Estas regras foram formuladas para lidar com a criação dos tipos necessários para criar o contexto,

a criação dos macros, a tradução das construções de comunicação assíncrona e criação dinâmica de instâncias. Como esta formalização e implementação não foram desenvolvidas pelo grupo no qual o eu participei, e também devido a contratos de sigilo industrial, não serão apresentados maiores detalhes sobre esta implementação. É importante observar que esta cadeia ainda é um trabalho em andamento porque a empresa ainda não terminou a implementação desta ferramenta de tradução. Maiores informações podem ser encontradas em [5].

4.4 Conclusão

O estudo da tradução de SDL para FIACRE contribuiu para auxiliar na definição da linguagem FIACRE. Através dele, foi possível observar a necessidade de adicionar dois novos tipos de construções à linguagem e também de mudar a maneira de manipular as variáveis compartilhadas. A primeira construção foi à adição de macros para simplificar a codificação de procedimentos repetitivos. A segunda foi à adição do tipo "união" (**union**) para criar filas de espera polimorfas. A mudança em relação à manipulação das variáveis compartilhadas é que agora elas são manipuladas pelos componentes/processos através dos nomes locais e não mais pelos nomes globais. Com isto, a variável global passa a ser mais um parâmetro a ser passado ao processo no momento de sua instanciação.

Para finalizar, realizar a troca de mensagens através de variáveis compartilhadas nos permitiu manter a atomicidade das transições. Porém, como as portas para comunicação foram removidas, a arquitetura de comunicação se torna invisível. Este fato aumenta a dificuldade no retorno dos resultados obtidos com a verificação do sistema. Contudo, consideramos este estudo da expressividade apenas como um ponto de partida para a tradução de SDL para FIACRE porque ele abordou apenas a tradução da comunicação assíncrona e da criação dinâmica SDL para FIACRE.

Capítulo 5

Exemplo de Verificação Formal utilizando o Ambiente TOPCASED

Este capítulo apresenta um exemplo de uma cadeia de verificação para o ambiente TOPCASED utilizando o compilador desenvolvido neste trabalho. Esta cadeia selecionou um conjunto de diagramas UML[1] para modelar o sistema. Os modelos descritos em UML serão refinados e em seguida traduzidos para FIACRE. Esta descrição FIACRE refinada será verificada no ambiente TINA com o auxílio do compilador FIACRE para TTS-TINA (Capítulo 3). Apesar de este exemplo ser apenas um estudo preliminar, ele foi importante para estudar a viabilidade da utilização de linguagens não formais como parte da cadeia de verificação.

Este capítulo assume que o leitor possui conhecimentos básicos sobre a linguagem de modelagem UML [32].

5.1 Uma Cadeia de verificação a partir de Diagramas UML

O desenvolvimento de sistemas críticos precisa considerar diferentes domínios e aspectos. A linguagem de modelagem UML permite descrever um sistema a partir de um conjunto de diagramas integrados (diagramas de estrutura, comportamento, etc). Esta abordagem informal é muito conveniente nos primeiros passos da fase de modelagem do sistema, mas devido à sua fraqueza do ponto de vista semântico, estes diagramas não são suficientes para os passos seguintes e especialmente

[1] Unified Modeling Language

para a verificação formal por *Model-Checking*. É importante ressaltar que já existem trabalhos publicados que realizam a verificação automática de modelos descritos utilizando a linguagem UML, podemos citar [33] e [34]. Porém, o objetivo deste estudo é apenas apresentar um exemplo de uma cadeia de verificação completa utilizando o ambiente TOPCASED a partir de diagramas UML. Para de não aumentar a complexidade deste estudo, optamos por não utilizar *profiles* ou estereótipos UML. Em conjunto com os diagramas UML, os requisitos do sistema são descritos de forma textual. Em seguida, estes diagramas são refinados adicionando-se informações que permitam traduzi-los em FIACRE e os requisitos são formalizados em lógica temporal linear LTL. Dependendo da natureza do requisito, pode ser necessário adicionar processos classificados como observadores para verificar a propriedade formalizada em LTL (ver Figura 5.1).

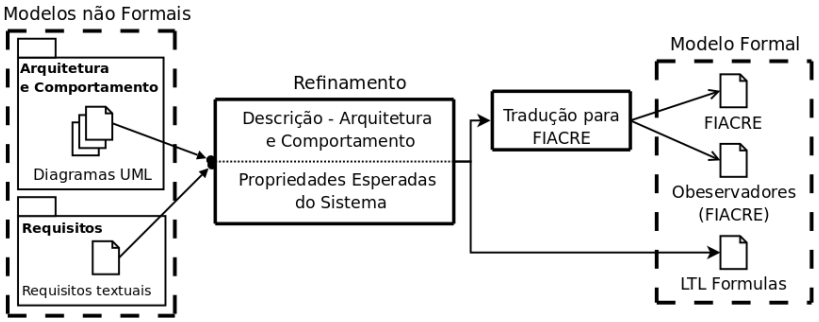


Figura 5.1: *Mapeamento UML e Requisitos para FIACRE e fórmulas LTL.*

A partir da descrição FIACRE, é possível traduzir esta descrição no formalismo Sistemas de Transição Temporizados (TTS) através do compilador desenvolvido (Capítulo 3). Após a tradução do modelo para TTS, um software chamado `tina`, que faz parte do ambiente TINA, é utilizado para construir uma abstração adequada do grafo de alcançabilidade do sistema como um Sistema de Transição Kripke (.ktz). Este grafo é confrontado com as propriedades formuladas em LTL (arquivo .ltl) por um *model-checker* chamado `selt` ([35]).

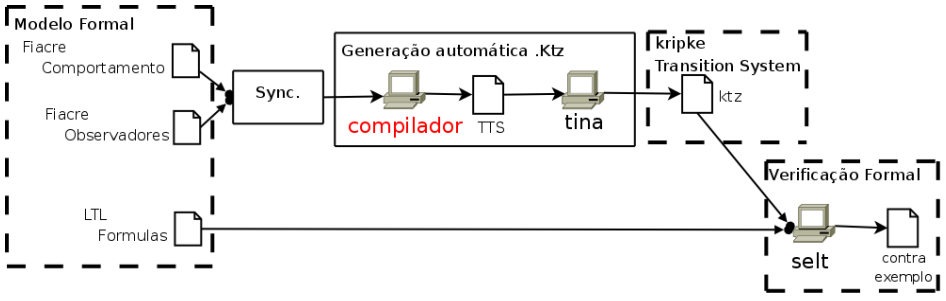


Figura 5.2: *Verificação de um modelo* FIACRE *utilizando o ambiente* TINA.

5.1.1 Diagramas UML

Considerou-se suficiente para este exemplo levar em conta os diagramas que seguem, sendo que esta abordagem pode ser estendida, se necessário for, para o uso de outros diagramas.

O comportamento e a arquitetura do sistema são inicialmente descritos por três diagramas UML, são eles:

- Diagramas de Classes, que permitem descrever a estrutura do sistema colocando em evidência as classes do sistema, seus atributos e as relações entre as classes.
- Diagramas de Estados, que ilustram o comportamento de uma classe. Este diagrama é formado pelos vários estados nos quais uma instância desta classe pode estar e pelas transições definidas entre estes estados. Neste exemplo, assumimos que os eventos disparadores das transições possuem os mesmos nomes que os conectores do diagrama *Composite* UML.
- Diagramas de Estrutura *Composite*, que mostram a estrutura interna de uma classe e as *colaborações* que esta estrutura torna possível. Este diagrama pode incluir *partes* internas da classe, *portas* que as instâncias internas utilizam para se comunicar uma com as outras e os *conectores* que fazem à ligação destas portas. Neste exemplo, consideramos que este diagrama permite somente comunicações síncronas entre as *partes*, ou seja, comunicações assíncronas não são permitidas.

Entretanto, apenas estes diagramas não são capazes de fornecer todas as informações necessárias para obter um modelo formal em FIACRE. Logo, algumas informações serão adicionadas na fase de refinamento do modelo.

5.1.2 Refinamento dos Diagramas

O refinamento dos diagramas UML consiste em adicionar informações ao modelo para preencher algumas lacunas existentes entre os diagramas selecionadas e a linguagem FIACRE. A adição de informações referentes à comunicação entre os componentes e as restrições temporais é uma etapa em direção a uma maior formalização do modelo. Para efeito de representação sintática, nós optamos por representar esta informações utilizando uma sintaxe do tipo álgebra de processos. Esta sintaxe torna mais claro este nível de refinamento porque define meios para expressar estas informações de forma correta e concisa. Nas próximas seções, as expressões sintáticas para adição destas informações serão apresentadas.

Comunicação

O diagrama *Composite* UML permite que um conector estabeleça a ligação entre múltiplas *partes* (ou instâncias). Entretanto, o tipo de comunicação não é claro no que diz respeito às quais instâncias estão enviando e quais estão recebendo, ou a existência de alguma sincronização entre os emissores ou receptores. Estas informações devem ser inseridas neste refinamento.

A representação sintática definida para inserir este tipo de informação é baseada em operadores associados aos conectores dos Diagramas UML *Composite*. Os operadores empregados são apresentados abaixo:

- $\|$: este operador simboliza a sincronização entre os processos;
- $|||$: este operador simboliza o paralelismo puro entre os processos, ou seja, não ocorre sincronização.

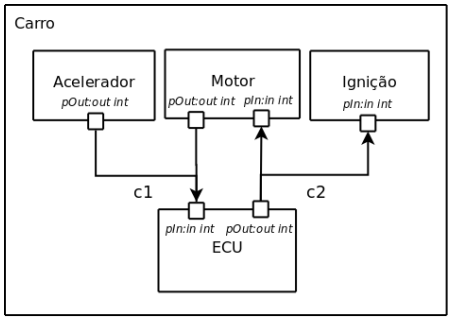


Figura 5.3: *Estrutura interna do componente Carro.*

A Figura 5.3 mostra o Diagrama *Composite* de um *Carro*. Sua estrutura interna é formada pelas *partes*: *Acelerador*, *Motor*, *Ignição* e *ECU* (Unidade de Controle - Eletronic Control Unit). Estas *partes* interagem através dos conectores c_1 e c_2. Associando os operadores LOTOS aos conectores, podemos descrever diferentes topologias de comunicação, por exemplo:

$c_1 : ECU \| \langle Acelerador ||| Motor \rangle$: especifica que a ECU sincroniza ora com o Acelerador, ora com o Motor;

$c_1 : ECU \| Motor \|$ *Ignição*: especifica que a ECU, o Acelerador e o Motor estão sincronizados.

Restrições Temporais

Após a formalização da topologia de comunicação, as restrições temporais são adicionadas nos diagramas de *Estados* UML. Esta adição ocorre associando-se operadores de restrição temporal aos eventos do Diagrama de *Estados* UML. Estes operadores são apresentados abaixo:

- *evento?t[0 < t <= 5]*: significa que a transição será disparada se e somente se o *evento* ocorrer em um intervalo de tempo menor que 5 unidades de tempo depois de sensibilizada;
- *evento!5*: significa que a transição será disparada se e somente se o *evento* ocorrer 5 unidades de tempo depois de sensibilizada.

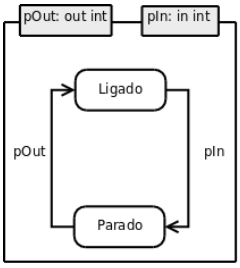


Figura 5.4: *Diagrama de Estados para o Motor.*

A Figura 5.4 ilustra o diagrama de *Estados* para a classe motor. Podemos associar uma restrição temporal para que a transição que aguarda o evento *pIn* dispare somente após 4 unidades de tempo depois de o evento ter ocorrido. Esta restrição pode ser formalizada utilizando os operadores apresentados acima: *pIn!4*.

Neste trabalho, optamos por inserir estas informações de refinamento nos rótulos dos conectores, para os operadores de composição paralela, e das transições, para as restrições temporais. Após esta adição de informação, referenciamos estes diagramas como: Diagrama *Composite* Refinado e Diagrama de *Estados* Refinado.

5.1.3 Refinamento dos Requisitos

Os requisitos expressos textualmente são mapeados em propriedades esperadas do sistema e posteriormente são formalizados em fórmulas de lógica temporal linear LTL [36]. Estas fórmulas são construídas a partir de um conjunto de variáveis proposicionais, operadores lógicos usuais ($\rightarrow, \neg, \bigwedge, V$) e de operadores temporais (*X, G, F, U* e *R*). Logo, este refinamento consiste em descrever as propriedades esperadas em fórmulas LTL substituindo as variáveis proposicionais pelos estados definidos nos Diagramas de Estados UML.

As propriedades a serem verificadas são divididas em dois grupos: qualitativas e de tempo real. As propriedades qualitativas recebem este nome porque admitem uma tradução direta, ou seja, apenas uma fórmula LTL é capaz de descrever a propriedade a ser verificada. As propriedades que envolvem tempo explícito, ou também chamadas de tempo real, não podem ser traduzidas diretamente porque

o Espaço de Estados gerado pela ferramenta de verificação - TINA - abstrai a informação temporal. A verificação destas propriedades implica na adição de observadores para "observar" se a restrição temporal está sendo respeitada. Seu uso consiste em sincronizá-los de alguma forma com os processos que eles devem "observar".

Uma forma mais avançada para representar os requisitos do sistema pode ser observada em [37], o qual utiliza um diagrama de Requisitos SYSML [38] para auxiliar o rastreamento entre os requisitos e os testes a serem realizados.

5.1.4 Tradução para FIACRE

A tradução é realizada aplicando-se um conjunto de regras sobre os diagramas UML *refinados* do sistema. As regras que são utilizadas para retirar informações dos diagramas UML são apresentadas abaixo, divididas de acordo com o diagrama ao qual se aplicam.

Esta tradução inicia-se a partir do diagrama UML de Classes, pois ele permite identificar se uma classe pode ser vista como um processo ou um componente em FIACRE. Este diagrama permite também identificar as variáveis locais e os parâmetros de um processo ou componente FIACRE. Estas informações são obtidas aplicando-se as regras formuladas abaixo:

Regra 1: Toda classe que possuir relações de composição (*Composition association*) ou de agregação (*agregation association*) com outras classes, será interpretada em FIACRE como um componente, pois estas associações definem que uma classe é formada por uma coleção de classes.

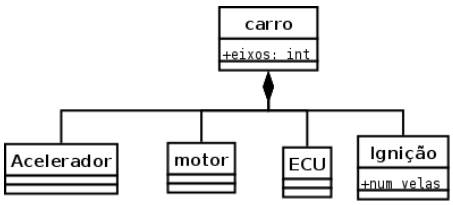


Figura 5.5: *Exemplo Diagrama de Classe UML.*

A Figura 5.5 mostra um diagrama de Classes composto por 5 classes: *carro*, *motor*, *ignição*, *acelerador* e *ECU*. Como a classe *carro* possui a associação de composição, ela será descrita em FIACRE como um componente formado pelas classes *motor*, *ECU*, *ignição* e *acelerador*. .

Regra 2: Como um complemento para a primeira regra, toda classe que não for formada por uma coleção de classes será descrita como um processo FIACRE. Este é o caso das classes *motor*, *ECU*, *ignição* e *acelerador* da Figura 5.5.

Regra 3: Os atributos **privados** (*private*) de uma classe definem as variáveis locais e os **públicos** (*public*) os parâmetros que compõem a interface do elemento FIACRE.

Neste estudo, apenas estas 3 regras são necessárias para extrair todas as informações necessárias deste diagrama. Construções como os métodos, as relações de herança, realização, não são necessárias para este estudo de verificação porque não agregam nenhuma informação relevante. Por exemplo, os métodos são informações pertinentes somente para a implementação do modelo e não para a verificação, pois estas construções esclarecem como uma determinada ação é efetuada. Por conseguinte, como o procedimento de verificação observa somente se uma dada ação é ou não possível de ser executada em um certo estado, saber como ela ocorre não auxilia este procedimento.

Após a identificação dos componentes e processos que compõem o sistema, podemos fazer duas afirmações quanto aos diagramas *Composite* e de Estados UML:

- todo diagrama *Composite* define um tipo de classe que é interpretado em FIACRE como um componente porque este diagrama descreve uma classe formada a partir da interação de outras classes.
- todo diagrama de *Estados* define um tipo de classe que é interpretado em FIACRE como um processo porque seu comportamento é descrito por uma máquina de estados.

A segunda parte desta tradução analisa os diagramas *Composite refinados* associados aos componentes, ou seja, as classes que são interpretadas como componentes FIACRE. Este diagrama é empregado para descrever a estrutura interna de um componente. As regras formuladas abaixo permitem abstrair esta informação do diagrama:

Regra 4: Os conectores do diagrama *Composite* UML refinado são descritos em FIACRE como canais de comunicação (portas locais) do componente que ele descreve. A Figura 5.3 mostra que o componente *carro* possui dois canais de comunicação: *c1* e *c2*.

Regra 5: As portas das *parts* do diagrama *Composite* são descritas como as portas que compõem a interface das classes que compõem o componente. Por exemplo, a Figura 5.3 mostra que o processo *motor* possui duas portas, ambas do tipo inteiro (*int*), sendo uma de entrada e outra de saída.

Regra 6: Os operadores composicionais utilizados para indicar o tipo da comunicação são diretamente traduzidos em operadores FIACRE seguindo o mapeamento abaixo:

- $\|\rightarrow$ **sync**;
- $|||\rightarrow$ **shuffle**;

O terceiro e último diagrama a ser considerado é o Diagrama de Estados *refinado*. Este diagrama descreve o comportamento de um processo. As regras que permitem abstrair o comportamento deste diagrama são apresentadas abaixo:

Regra 7: todos os estados do diagrama são descritos como estados de um processo FIACRE.

Regra 8: Os arcos que ligam os estados são interpretados como transições FIACRE. Estes arcos são divididos em três partes: evento, recepção ou emissão de valores, condição e ação (ver Figura 5.6). Como foram convencionados no início do trabalho, os eventos são nomeados com os mesmos nomes dos conectores que fazem parte do Diagrama *Composite*. As condições e ações são efetuadas sobre as variáveis(locais ou compartilhadas) do processo.

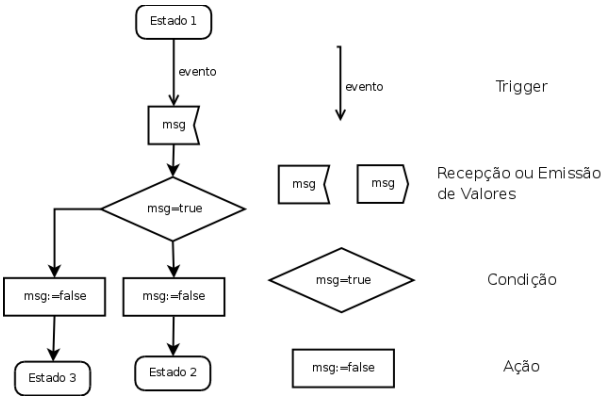


Figura 5.6: *Diagrama de Estados UML.*

A transição FIACRE referente à Figura 5.6 é apresentada abaixo:

```
from estado1 evento? msg;
        if msg = true then msg := false; to estado2
        else msg := false; to estado3 end
```

Regra 9: Os operadores para as restrições temporais são primeiramente associados aos canais de comunicação do componente e posteriormente traduzidos. A tradução consiste em calcular o intervalo de tempo referente a cada operador:

- *evento?t[x≤t≤y]*: a tradução FIACRE para esta restrição consiste em associar o intervalo *[x,y]* ao canal de comunicação ligado a este evento;

- *evento!t 5*: a tradução FIACRE para esta restrição consiste em associar o intervalo *[5,∞[* ao canal de comunicação ligado a este evento.

Caso mais de uma restrição temporal seja associada ao mesmo canal, a restrição resultante é a intersecção entre elas. Por exemplo, a Figura 5.7 apresenta um canal de comunicação (*canal*) que possui associado a ele duas restrições temporais, são elas: $t > 5$ e $t < 10$. A restrição temporal resultante para o canal de comunicação nomeado *canal* é o intervalo $5 < t < 10$ *[5,10]*.

A tradução em FIACRE da Figura 5.7 é mostrada abaixo:

```
process pr1 [in pEv:none]
states state1, state2 init state1
from state1 pEv; to state2

process pr2 [in pEv:none]
states state1, state2 init state1
from state1 pEv; to state2

component main is
port canal:none in [5,10]
        sync
                pr1[canal]
                pr2[canal]
        end
```

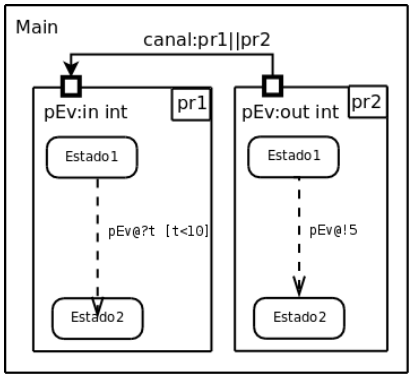


Figura 5.7: *Representação de Restrições Temporais.*

5.2 Exemplo: Modelagem e Verificação Formal de uma Fábrica

O exemplo escolhido para apresentar esta cadeia de verificação é uma Fábrica para a produção de peças (P) retirado de [37].

5.2.1 Descrição Informal do Problema

O objetivo desta linha de produção é produzir peças e, para isto, trabalhadores e máquinas estão disponíveis. Os trabalhadores são divididos em dois grupos: a) operários (O_x), os quais utilizam as máquinas e trabalham na linha de produção e, b) os técnicos (T_x), os quais fazem à manutenção das máquinas. Esta fábrica possui linhas de produção (L_x) que operam as máquinas (M_x) em uma dada ordem.

Os recursos da fábrica apresentados neste exemplo são:

- Humanos: os recursos humanos desta fábrica são formados por dois trabalhadores (O_1 e O_2) e um técnico de manutenção (T_1). O intervalo de tempo necessário para um técnico efetuar a manutenção de uma máquina é de 0 a 5 unidades de tempo e de um operário . A legislação trabalhista determina que um trabalhador não execute uma tarefa repetitiva por mais que 35 unidades de tempo.
- Mecânicos: a fábrica possui quatro máquinas (M_1, M_2, M_3 e M_4) para auxiliar na produção das peças. O fabricante destas máquinas garante o funcionamento correto somente se a manutenção for feita a cada 15 ciclos de trabalho consecutivos.

Para produzir o produto desejado (peças), é necessário o trabalho das duas linhas sobre o mesmo produto. O produto é trabalhado primeiro pela linha L_1 e em seguida pela linha L_2. Estas linhas não compartilham trabalhadores; O_1 trabalha somente em L_1 e O_2 somente em L_2. A linha L_1 utiliza as máquinas M_1, M_2 e M_3, nesta ordem. A linha L_2 utiliza três máquinas M_2, $M3$ e $M4$, também nesta ordem.

A partir destas informações sobre a Fábrica, podemos nos questionar de como afirmar que a Fábrica a ser construída cumprirá com alguns requisitos, por exemplo, a legislação trabalhista. Entre as questões a serem levantadas antes da implementação da linha de produção, destaca-se que : nenhum trabalhador pode trabalhar mais do que 35 unidades de tempo consecutivas; a manutenção das máquinas deve ocorrer a cada 15 ciclos de operação destas.

5.2.2 Modelagem UML

Para responder as questões colocadas na seção anterior, é desejável possuir um modelo formal desta fábrica. Porém, inicialmente optamos por modelar a linha utilizando os diagramas UML para descrever a estrutura e o comportamento do sistema. Em seguida, a linguagem FIACRE é empregada para refinar o modelo e TINA para realizar a verificação formal das propriedades formalizadas na lógicas LTL.

Modelagem UML

O diagrama de classes UML apresentado na Figura 5.8 mostra a relação de interdependência e hierarquia entre as entidades envolvidas na fábrica.

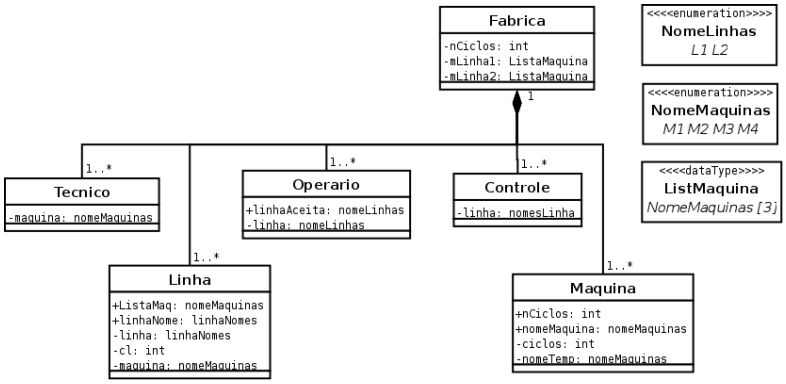


Figura 5.8: *Diagrama de Classes da Fábrica.*

A Figura 5.9 mostra a estrutura interna da Fábrica. Este diagrama mostra que a fábrica é controlada por uma unidade de controle (C_1) que gerencia duas linhas de produção (L_1 e L_2). Existem 4 máquinas(M_1, M_2, M_3 e M_4) e cada linha utiliza 3 máquinas, respectivamente $L_1(M_1, M_2, M_3)$ e $L_2(M_2, M_3, M_4)$. Estas linhas não compartilham trabalhadores; L_1 trabalha somente com o trabalhador O_1 e na linha L_2 somente com o trabalhador O_2. O técnico T_1 realiza a manutenção de todas as máquinas. Além do mais, este diagrama também permite interpretar a arquitetura de comunicação da fábrica.

Após a modelagem estrutural, é necessário modelar o comportamento do sistema. A Fábrica é formada por diversas classes e cada uma possui um comportamento distinto. Os operários são responsáveis por operar as máquinas respeitando o descanso garantido por lei a cada ciclo de trabalho. O técnico realiza a manutenção das máquinas quando lhe é solicitado. As máquinas executam uma tarefa quando solicitadas pelo controle das linhas de produção e não podem executar mais de 15 ciclos de trabalho consecutivos sem efetuar a manutenção preventiva. Os controles de linhas de produção gerenciam as máquinas. Cada linha possui sua própria seqüência de 3 máquinas para coordenar,

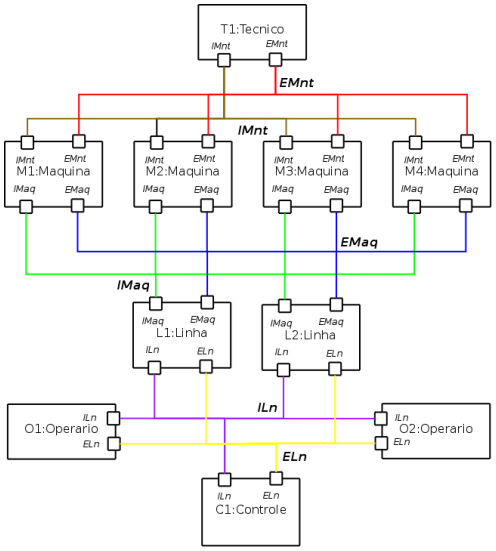


Figura 5.9: *Diagrama de Estrutura Composite do Fábrica.*

$L_1(M_1, M_2, M_3)$ e $L_2(M_2, M_3, M_4)$. O controle da produção gerencia a ordem de execução das linhas, primeiro a L_1 e depois L_2, e a alocação dos trabalhadores, O_1 para L_1 e O_2 para L_2.

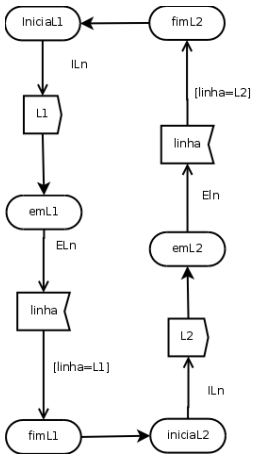


Figura 5.10: *Diagrama de Estados para o processo Controle da Produção.*

Por exemplo, A Figura 5.10 apresenta o Diagrama de Estado do processo *controle*. Este processo gerencia a ordem de execução das linhas, primeiro a L_1 e depois L_2, e a alocação dos trabalhadores, O_1 para L_1 e O_2 para L_2. Neste caso, a alocação dos trabalhadores O_1 e O_2 ocorrem a partir dos eventos *iniciaL1* e *iniciaL2*, respectivamente.

As máquinas são responsáveis por agregar valor ao produto produzido, ou seja, elas executam operações de trabalho nas peças. Porém, não podemos esquecer os requisitos de manutenibilidade antes de modelar as máquinas. Este processo não pode operar mais que *nCiclos* vezes consecutivas sem manutenção. A figura 5.11 mostra o Diagrama de Estados para este processo. Podemos observar

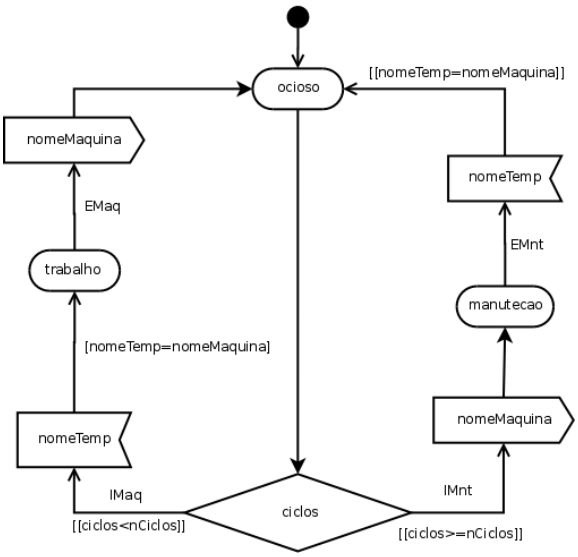


Figura 5.11: *Diagrama de Estados do processo Máquina.*

que o requisito de manutenibilidade é levado em conta pela condição $ciclo < nCiclos$ entre os estados *ocioso* e *manutenção*.

A Figura 5.12 mostra o diagrama de Estados do processo *técnico*. Como foi definido, o técnico é responsável pela manutenção das máquinas. Esta atividade é executada somente quando requisitada por uma máquina. Logo, este processo alterna seu comportamento entre dois estados: *ocioso* e *trabalho*. Como a Figura 5.12 mostra, o técnico espera uma requisição de manutenção (evento *IMnt*) para evoluir ao estado trabalho.

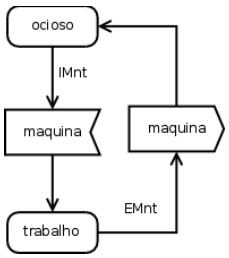


Figura 5.12: *Diagrama de Estados do processo Técnico.*

Além destas descrições na forma de diagramas UML da estrutura e do comportamento, existem outras informações que devem ser adicionadas, são elas: esquema de sincronização da fábrica, tempo de operação das máquinas e tempo de manutenção dos técnicos. Estas informações serão adicionadas na fase de refinamento.

Refinamento

O primeiro passo para refinar os diagramas UML é descrever o esquema de sincronização dese-

jado nos diagramas *Composite*. Analisando a Figura 5.9, podemos organizar o *layout* de comunicação da fábrica considerando três tipos de sincronização:

- sincronização par-a-par entre máquinas e o técnico de manutenção através das portas locais IMnt e EMnt: M_1-T_1,M_2-T_1, M_3-T_1, M_4-T_1.
- sincronização par a par entre as máquinas e linhas através das portas locais IMaq e EMaq: $M_1 - L_1$, $M_2 - L_1$, ..., $M_1 - L_2$, $M_2 - L_2$, etc.
- multi-sincronização entre as linhas, os trabalhadores e o controle da produção geral através das portas ILn e ELn: $L_1 - O_1 - C_1$ e $L_2 - O_2 - C_1$.

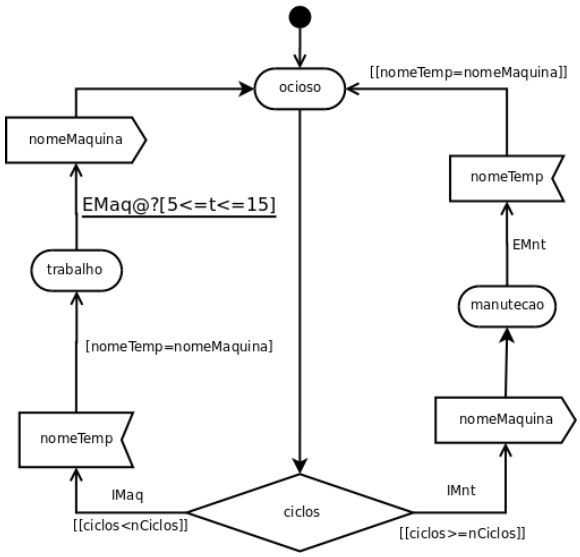
O refinamento deste diagrama (Figura 5.9) se dá associando-se os operadores de composição aos conectores. As associações são apresentadas abaixo:

- IMnt: $T_1 \| \langle M_1 ||| M_2 ||| M_3 ||| M_4 \rangle$
- EMnt: $T_1 \| \langle M_1 ||| M_2 ||| M_3 ||| M_4 \rangle$
- IMaq: $\langle L_1 ||| L_2 \rangle \| \langle M_1 ||| M_2 ||| M_3 ||| M_4 \rangle$
- EMaq: $\langle L_1 ||| L_2 \rangle \| \langle M_1 ||| M_2 ||| M_3 ||| M_4 \rangle$
- ILn: $C_1 \| \langle L_1 \| O_1 ||| L_2 \| O_2 \rangle$
- ELn: $C_1 \| \langle L_1 \| O_1 ||| L_2 \| O_2 \rangle$

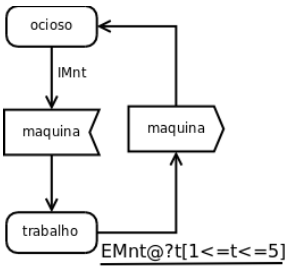
O segundo passo do refinamento é a adição de restrições temporais aos diagramas de Estados. Neste exemplo existem duas restrições temporais a serem inseridas. A primeira é o intervalo de tempo que uma máquina leva para executar um trabalho. A segunda é o intervalo de tempo necessário para um técnico realizar a manutenção de uma máquina. Estas restrições serão inseridas nos diagramas de estados, associando os operadores aos eventos que etiquetam as transições.

A Figura 5.13a) mostra o diagrama de Estados refinado do processo *máquina*. Neste diagrama, a expressão temporal *?[5$\leqslant t \leqslant$15]* está associada ao evento *EMaq*. A Figura 5.13b) mostra o diagrama de Estados refinado do processo *técnico*. Neste diagrama, a expressão temporal *?[0$< t \leqslant$5]* está associada ao evento *EMaq*.

Tradução



a) Diagrama de Estados refinado do processo Máquina.



b)Diagrama de Estados do processo Técnico.

Figura 5.13: *Diagramas de Estados Refinados.*

O primeiro diagrama UML a ser considerado para a tradução é o diagrama de classes UML apresentado na Figura 5.8. Efetuando-se as regras 1 e 2 podemos afirmar que a *Fábrica* é um componente FIACRE composto pelos processos *Operário*, *Técnico*, *Linha* e *Controle*. A regra 3 permite identificar os atributos destas classes como parâmetros da interface (*público*) ou variáveis locais (*privado*).

Aplicando-se a regra 4 é possível retirar do diagrama *Composite* (Figura 5.9) os canais de comunicação utilizados internamente pelo componente *fábrica*. A regra 5 permite identificar as portas dos processos. A Regra 6 permite traduzir em FIACRE os operadores LOTOS associados aos canais de comunicação do componente *fábrica*:

- IMnt: $T_1 \| \langle M_1 \| | M_2 \| | M_3 \| | M_4 \rangle$

```
sync   Ti[IMnt]
       shuflle
              M1[IMnt]
              M2[IMnt]
              M3[IMnt]
              M4[IMnt]
       end
end
```

- EMnt: $T_1 \| \langle M_1 \| | M_2 \| | M_3 \| | M_4 \rangle$

```
sync    Ti[EMnt]
        shuflle
                M1[EMnt]
                M2[EMnt]
                M3[EMnt]
                M4[EMnt]
        end
end
```

- IMaq: $\langle L_1 ||| L_2 \rangle \| \langle M_1 ||| M_2 ||| M_3 ||| M_4 \rangle$

```
sync    shuffle
                L1[IMaq]
                L2[IMaq]
        end
        shuflle
                M1[IMaq]
                M2[IMag]
                M3[IMaq]
                M4[IMaq]
        end
end
```

- EMaq: $\langle L_1 ||| L_2 \rangle \| \langle M_1 ||| M_2 ||| M_3 ||| M_4 \rangle$

```
sync
        shuffle
                L1[EMaq]
                L2[EMaq]
        end
        shuflle
                M1[EMaq]
                M2[EMag]
                M3[EMaq]
                M4[EMaq]
        end
end
```

- ILn: $C_1 \| \langle L_1 \| O_1 \| | L_2 \| O_2 \rangle$

```
sync    C1[ILn]
        shuffle
                sync    L1[ILn]
                        O1[ILn]
                end
                sync    L2[ILn]
                        O2[ILn]
                end
        end
end
```

- ELn: $C_1 \| \langle L_1 \| O_1 \| | L_2 \| O_2 \rangle$

```
sync    C1[ELn]
        shuffle
                sync    L1[ELn]
                        O1[ELn]
                end
                sync    L2[ELn]
                        O2[ELn]
                end
        end
end
```

A composição final do componente *Fábrica* é obtido efetuando-se a união de todos as composições apresentas acima. O exemplo abaixo ilustra a união das composições realizadas através dos canais de comunicação *IMnt* e *EMnt*, que pertencem ao componente *Fábrica*.

```
sync   Ti[IMnt]            sync   Ti[EMnt]            sync   Ti[EMnt,IMnt]
   shuflle                    shuflle                    shuflle
          M1[IMnt]                   M1[EMnt]                   M1[EMnt,IMnt]
          M2[IMnt]                   M2[EMnt]                   M2[EMnt,IMnt]
          M3[IMnt]     +             M3[EMnt]        →          M3[EMnt,IMnt]
          M4[IMnt]                   M4[EMnt]                   M4[EMnt,IMnt]
   end                        end                        end
end                        end                        end
```

O último passo antes de compor o componente *fabrica* é traduzir as restrições temporais associadas aos eventos nos diagramas de *Estados* refinado. Recapitulando, este exemplo possui duas restrições temporais, são elas: tempo de operação das máquinas e o tempo de manutenção das máquinas (ver Figura 5.13b). O tempo de operação das máquinas é descrito pela expressão *EMaq@?t[*$5 \leq t \leq 15$*]* retirada da Figura 5.13a). Logo, a restrição temporal $[5 \leq t \leq 15]$ será associada ao canal de comunicação *EMaq* do componente *fábrica*, resultando em: *EMaq:NomeMaquinas in [5,10]*. O tempo de manutenção das máquinas é descrito pela expressão *EMnt@?t[*$1 \leq t \leq 5$*]* retirada da Figura 5.13b). Logo, a restrição temporal $[1 \leq t \leq 5]$ será associada ao canal de comunicação *EMnt* do componente *fábrica*, resultando em: *EMnt:NomeMaquinas in [1,5]*.

Antes de apresentar o componente *Fábrica* descrito em FIACRE, é importante ressaltar que FIACRE trabalha com instâncias anônimas. Assim, os nomes M_1, M_2, M_3 e M_4 serão substituídos por *máquina*, os nomes L_1 e L_4 por *linha*, O_1 e O_2 por *operário*, T_1 por *técnico* e finalmente C_1 por *controle*. A descrição do componente *Fábrica* em FIACRE, que foi obtida aplicando as regras descritas acima, é apresentado abaixo:

```
component fabrica is
var nCiclos:int:=15
    mLinha1:ListMaquina:=[M1,M2,M3]
    mLinha2:ListMaquina:=[M2,M3,M4]
port    IMnt, IMaq:NomeMaquinas,
        EMnt:NomeMaquinas in [1,5],
        EMaq:NomeMaquinas in [5,10],
        ILn, ELn:NomeLinhas
sync
        Controle [ILn, ELn]
        shuffle
                sync    Operario[ILn, ELn]
                        Line[ILn, ELn, IMaq, EMaq] (mLinha1,L1)
                end
                sync    Operario[ILn, ELn]
                        Line[ILn, ELn, IMaq, EMaq] (mLinha2,L2)
                end
        end
        sync    Tecnico [IMnt, EMnt]
                shuffle Maquina[IMaq, IMnt, EMaq, EMnt] (nCicles, M1)
```

```
                        Maquina[IMaq, IMnt, EMaq, EMnt] (nCicles, M2)
                        Maquina[IMaq, IMnt, EMaq, EMnt] (nCicles, M3)
                        Maquina[IMaq, IMnt, EMaq, EMnt] (nCicles, M4)
                end
        end
end
```

Dando continuidade ao processo de tradução, vamos agora traduzir os diagramas de *Estado* refinados em processos FIACRE. Considerando o diagrama de *Estados* refinado do processo *controle* (Figura 5.10), aplicando-se as regras 10 e 11 podemos traduzir este diagrama no processo FIACRE apresentado abaixo:

```
process Controle [out ILn, in ELn:NomeLinhas] is
states iniciaL1, emL1, fimL1, iniciaL2, emL2, fimL2 init iniciaL1
var linha:NomeLinhas:=NN
        from iniciaL1 ILn! L1; to emL1
        from emL1 ELn? linha where linha=L1; linha:=NN; to fimL1
        from fimL1 to iniciaL2
        from iniciaL2 ILn! L2; to emL2
        from emL2 ELn? linha where linha=L2; linha:=NN; to fimL2
        from fimL2 to iniciaL1
```

A descrição FIACRE do diagrama de *Estados* refinado ilustrado na Figura 5.13a) é apresentada abaixo:

```
process Maquina[in IMaq, out IMnt, out EMaq, in EMnt: NomeMaquinas]
        (nCiclos:int, meuNome:NomeMaquinas) is
states ocioso, trabalho, mnt init ocioso
var ciclos:int:=0, nomeTemp:NomeMaquinas:=NN
        from ocioso if ciclos < nCiclos
                then IMaq? nomeTemp where nomeTemp=meuNome; to trabalho
                else IMnt! meuNome; to mnt end
        from trabalho EMaq! meuNome; ciclos:=ciclos + 1; to ocioso
        from mnt EMnt? nomeTemp where nomeTemp=meuNome; ciclos:=0; to ocioso
```

Os outros diagramas de estados e suas respectivas descrições FIACRE são demonstrados no anexo 5.

5.2.3 Exemplo: Representação Informal dos Requisitos e Refinamento em Lógica LTL

Neste exemplo, vimos que podemos especificar informalmente os requisitos abaixo:

- Legislação trabalhista:

  Descanso: uma pausa (5 unidades de tempo neste caso) deve ser garantida depois de cada ciclo de trabalho.

  Trabalho Contínuo: o trabalhador não pode trabalhar mais que 35 unidades contínuas sem descanso.

- Normas da Fábrica:

  Normas de Manutenção: as normas de manutenção são responsáveis por garantir o funcionamento da fábrica sem falhas.

  Manutenção das máquinas: a manutenção das máquinas deve ser garantida durante o processo de manufatura.

Para verificar se o modelo formal FIACRE da Fábrica atende às especificações ilustradas acima, é preciso mapear estes requisitos em propriedades esperadas do sistema. Estas propriedades são então formalizadas em fórmulas LTL. É importante ressaltar que todos os estados dos processos FIACRE podem ser utilizados como variáveis proposicionais destas fórmulas. Logo, esta formalização consiste em compreender quais variáveis proposicionais (*estados* FIACRE) permitem verificar a propriedade esperada do sistema. Por exemplo, o estado *manutenção* (Figura 5.11) dos processos máquinas podem ser utilizadas para verificar se uma máquina cumpre ou não o requisito de manutenção.

Antes de começar a formalizar as propriedades esperadas em LTL, precisamos detalhar como o compilador faz a resolução dos nomes dos estados. Por exemplo, o estado *manutenção* do processo *máquina* será resolvido no código resultante TTS como *manutencao_maquina_1*. Esta resolução segue a ordem: nome original do estado, nome do processo e número da instância. Este número da instância é colocado porque FIACRE trabalha com instâncias anônimas. Dando continuidade ao refinamento dos requisitos, as propriedades a serem verificadas são divididas em dois grupos: qualitativas e de tempo real.

As propriedades qualitativas recebem este nome porque admitem uma tradução direta, por exemplo:

- Descanso do trabalhador : " Sempre o operário *Opi* possuirá uma pausa após um ciclo contínuo de trabalho em uma linha", $[](fimL1\_Controle\_1 \Rightarrow pausa\_Operario\_i)$.
- Manutenção das máquinas : "Sempre existirá um momento no qual a máquina *Mi* estará em manutenção", $[] <> manutencaot\_Maquina\_i$;

Neste exemplo, o requisito "excesso de trabalho" é classificado como uma propriedade que envolve tempo, pois é necessário "observar" se o "o trabalhador *Opi* nunca trabalha em excesso". Classificamos como excesso aqui quando o trabalhador trabalha mais de 35 unidades de tempo consecutivas. Logo, vamos sincronizar um processo chamado de *observador* com os trabalhadores *operários* para "observar" se a propriedade é cumprida. A Figura (Figura 5.14) mostra essa sincronização.

```
process Observador [obPort:none] is
states ocupado, trabalhoExcesso init trabalho
from ocupado obPort; to trabalhoExcesso

component Op1 [in out IT, ET: nomeLinhas] is
port in tp:none in [5,5], ob:none in [35,35]
        sync
                Observador[ob]
                Operario[IT,ET,tp,ob] (L1)
        end
```

Figura 5.14: *Sincronização do Observador com o Trabalhador.*

Assim, a partir da Figura (Figura 5.14), podemos formalizar a propriedade 'O operário *Opi* nunca trabalha em excesso" através da fórmula LTL: $[]\neg trabalhoExcesso\_Observador\_i$.

5.2.4 Exemplo: Resultados da Verificação

Utilizando o compilador desenvolvido, podemos traduzir o modelo formal FIACRE para TTS-TINA automaticamente. A parte do controle do Sistema de Transição Temporizado obtido pode ser visualizado na Figura (5.15). Os lugares desta rede estão organizados de acordo com os processos aos quais eles pertenciam. Este sistema é formado por 65 lugares e 38 transições. Agora, utilizando o ambiente TINA (ver Figura 5.2) podemos verificar se as restrições formuladas na lógica LTL serão atendidas pela Fábrica. O espaço de estados obtido utilizando a ferramenta TINA possui 2818 estados e 3761 transições.

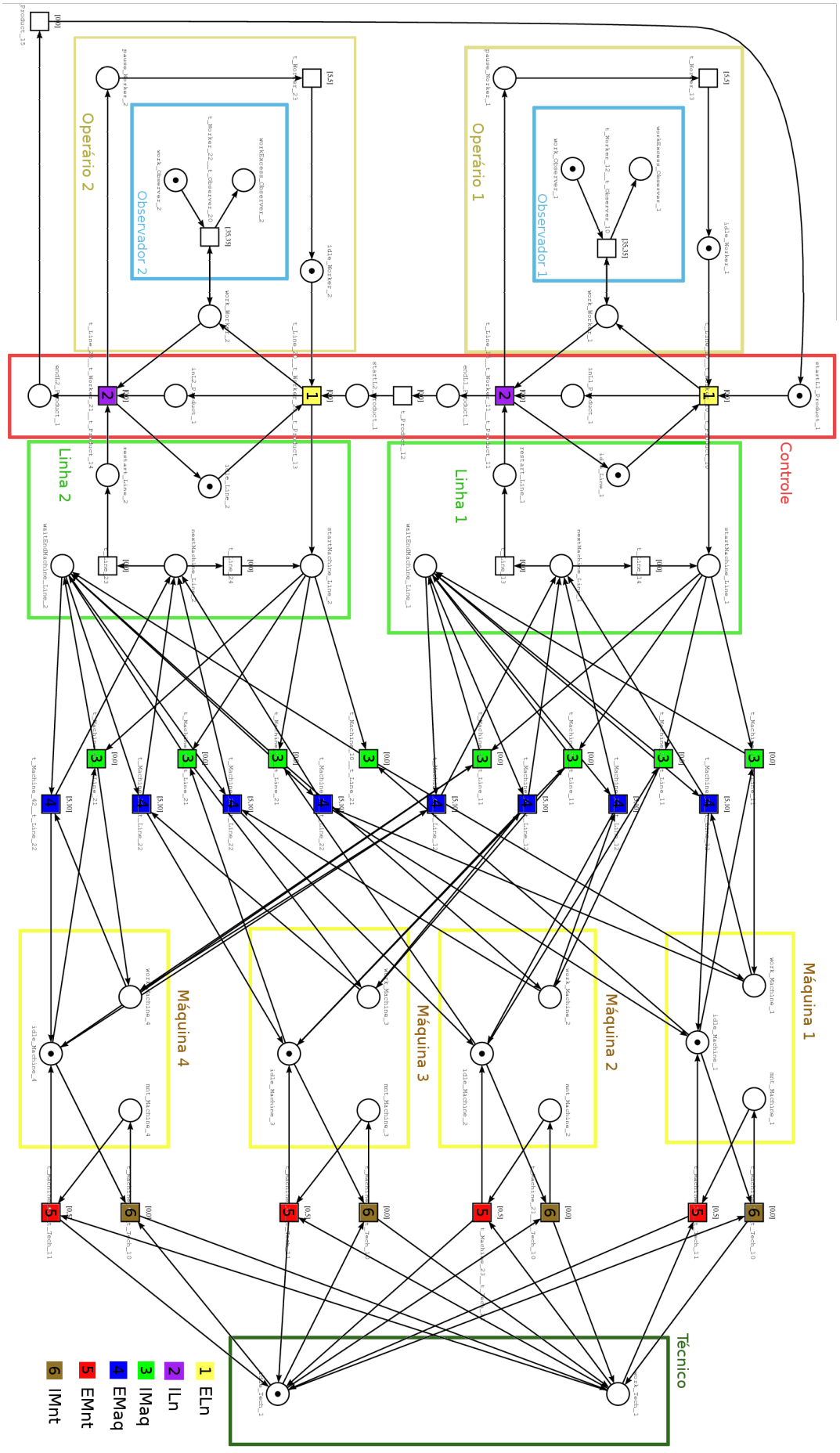


Figura 5.15: *Sistema de Transição Temporizado da Fábrica.*

O resultado da verificação das fórmulas LTL elaboradas utilizando-se o ambiente TINA é apresentado abaixo:

```
##Verifying Properties
Selt version 2.8.5 -- 10/24/07 -- LAAS/CNRS
ktz loaded, 2818 states, 3761 transitions
0.010s
#Pausa para o Operário.
#Linha 1 and Operário 1
LTL:[](fimL1_Controle_1 => <>pausa_Operario_1) -> TRUE.
#Linha 2 and Operário 2
LTL:[](fimL2_Controle_1 => pausa_Operario_2) -> TRUE.
#Manutenção Máquinas
#M1 LTL:[](<> manutencao_Maquina_1) -> TRUE
#M2 LTL:[](<> manutencao_Maquina_2) -> TRUE
#M3 LTL:[](<> manutencao_Maquina_3) -> TRUE
#M4 LTL:[](<> manutencao_Maquina_4) -> TRUE
#Excesso de Trabalho
LTL:[](- trabalhoExcesso_Observador_1) -> FALSE
 state 1240: idle_Line_2 .... workExcess_Observer_1 ........ [accepting all]
LTL:[](- trabalhoExcesso_Observador_2) -> FALSE
 state 835: idle_Line_1 ..... workExcess_Observer_2 ........ [accepting all]
```

Como podemos observar, o resultado acima mostra que o modelo formal FIACRE proposto cumpre grande parte dos requisitos. Os requisitos de manutenção das máquinas e pausa para descanso dos operários são cumpridos porque as fórmulas LTL relacionadas são verificadas. Por exemplo, os requisitos de manutenção são validados a partir da verificação das fórmulas $[] <> M_1\_mnt$, $[] <> M_2\_mnt$, $[] <> M_3\_mnt$ e $[] <> M_4\_mnt$. Porém, com os dados obtidos, o processo de verificação encontrou ao menos duas seqüências de eventos que nos levam a afirmar que os trabalhadores vão trabalhar em excesso. Esta afirmação é resultado da verificação das fórmulas $[]\neg\ Opi\_trabalhoExcesso_1$ e $[]\neg\ Opi\_trabalhoExcesso_2$.

Quando uma propriedade não é verificada, faz parte do processo de verificação formal analisar um contra exemplo para auxiliar na resolução do problema. O contra exemplo fornecido pelo software `selt` auxiliou a definir novas restrições temporais para o modelo. Alterando o tempo de operação das

máquinas de 5 a 15 para 5 a 10, pudemos obter um novo modelo capaz de cumprir todos os requisitos esperados (ver resultado abaixo). Os contra-exemplos obtidos não foram apresentados devido ao seus tamanhos, 1240 e 835 estados, mas permitiu achar os valores através do uso de simulações.

```
##Verifying Properties
Selt version 2.8.5 -- 10/24/07 -- LAAS/CNRS
ktz loaded, 870 states, 1060 transitions
0.000s
#Pausa para o Operário.
#Linha 1 and Operário 1
LTL:[](fimL1_Controle_1 => <>pausa_Operario_1) -> TRUE
#Linha 2 and Operário 2
LTL:[](fimL2_Controle_1 => pausa_Operario_2) -> TRUE
#Manutenção Máquinas
#M1 LTL:[](<> manutencao_Maquina_1) -> TRUE
#M2 LTL:[](<> manutencao_Maquina_2) -> TRUE
#M3 LTL:[](<> manutencao_Maquina_3) -> TRUE
#M4 LTL:[](<> manutencao_Maquina_4) -> TRUE
#Excesso de Trabalho
LTL:[](- trabalhoExcesso_Observador_1) -> unbound variable workExcess_Observer_1
LTL:[](- trabalhoExcesso_Observador_2) -> unbound variable workExcess_Observer_2
```

O resultado acima mostra que este novo modelo verifica todas as propriedades do sistema, ou seja, o sistema cumpre todas as propriedades esperadas. Os resultados "unbound variable trabalho Excesso_Observador_i" sinaliza que as fórmulas LTL $[]\neg$ *trabalhoExcesso_Observador*$_1$ e $[]\neg$ *trabalhoExcesso_Observador*$_2$ são verdadeiras, pois as variáveis proposicionais, que são os estados *trabalhoExcesso* dos processos observadores, não fazem parte do espaço de estados do sistema.

5.3 Conclusões

Nesta cadeia de verificação foram apresentados experimentos utilizando o ambiente TOPCASED para modelagem e verificação de sistemas complexos a partir de uma abordagem informal (Diagramas UML). Estes diagramas são então refinados através da adição das informações de

comunicação e restrições temporais (representadas com uma sintaxe do tipo LOTOS). Estes diagramas refinados são então traduzidos em FIACRE e os requisitos funcionais formalizados em lógica temporal linear (LTL). O modelo formal descrito em FIACRE é traduzido para o formalismo de entrada da ferramenta de verificação tina(tts). O *model-checker* chamado `selt` é utilizado para confrontar o espaço de estados gerado por tina (ktz) com as propriedades formuladas em LTL (.ltl).

Finalizando, este estudo foi importante como uma primeira abordagem do problema de verificação de linguagens não formais para o projeto TOPCASED. Para futuros trabalhos, propomos estudar a definição de uma descrição em diagramas UML, seguindo "Um Profile" FIACRE.

Capítulo 6

Conclusão

Este trabalho apresentou a arquitetura de verificação do ambiente TOPCASED, em conjunto com a linguagem intermediária formal FIACRE. Esta arquitetura foi desenvolvida com o objetivo de realizar a verificação formal de forma transparente para o usuário, dando liberdade quanto à escolha da linguagem de modelagem e da ferramenta para verificação formal. Para atingir este objetivo, optou-se por utilizar uma linguagem intermediária formal, chamada FIACRE, para promover o intercâmbio de informação entre a modelagem e a verificação do sistema.

A primeira atividade desenvolvida neste trabalho foi auxiliar na definição da linguagem intermediária FIACRE. O estudo da tradução de SDL para FIACRE, apresentado no capítulo 4, contribuiu para auxiliar nesta definição da linguagem FIACRE. Através dele, foi possível observar a necessidade de adicionar novos tipos de construções à linguagem e também de mudar a maneira de manipular as variáveis compartilhadas. Contudo, consideramos este estudo da expressividade de SDL e FIACRE apenas como um ponto de partida para a tradução de SDL para FIACRE, levando em conta que esta tradução abordou apenas dois aspectos, são eles: comunicação assíncrona e criação dinâmica.

A partir da definição da linguagem, deu-se início ao estudo de um esquema conceitual para traduzir modelos FIACRE para o formalismo TTS (Capítulo 3). Este esquema é realizado em duas etapas, chamadas expansão e geração, interligadas por um formato intermediário. A fase de expansão da descrição FIACRE compreende 16 operações, que simplificam a descrição original de modo que o código resultante seja simples de ser traduzido na linguagem alvo. Este modelo simplificado é descrito em um formato intermediário, chamado SUBFIACRE, que pode ser considerado como uma descrição simplificada de FIACRE, muito próxima do formalismo TTS. A partir desta descrição simplificada, o código final é gerado.

A partir deste modelo conceitual, foi implementado um compilador (*front-end*) para tornar operacional a verificação de FIACRE com o ambiente TINA. Este compilador, que foi desenvolvido utilizando técnicas clássicas, é formado por duas partes: parte frontal e parte final. A parte frontal compreende os módulos responsáveis pela análise léxica, sintática, semântica e a produção de um formato intermediário. A parte final é formada pelas operações de expansão e geração, que foram descritas no modelo conceitual, para traduzir o formato intermediário no formato alvo, que é neste caso o formalismo TTS.Este compilador vem sendo utilizado desde outubro de 2007 no projeto TOPCASED, quando foi apresentado para parceiros e órgãos financiadores do projeto.

Neste trabalho, é apresentado um experimento utilizando o ambiente TOPCASED para modelagem e verificação de um sistema a partir de uma abordagem informal (diagramas UML). Estes diagramas são então refinados a partir da adição de informações ao modelo. Estas informações são descritas com operadores e permitem expressar restrições temporais e comunicação síncrona. Em seguida estes diagramas refinados são traduzidos em FIACRE e os requisitos funcionais do sistemas são formalizados em lógica linear temporal LTL. O modelo formal descrito em FIACRE é compilado para o formalismo de entrada da ferramenta de verificação `tina` (*.tts*). Um *model-checker* chamado `selt` é utilizado para confrontar o espaço de estados gerado por `tina` (*.ktz*) com as propriedades formuladas em LTL (*.ltl*). Este estudo foi importante como uma primeira abordagem do problema de verificação de linguagens não formais.

As perspectivas futuras deste trabalho são integrá-lo com uma ferramenta capaz de reproduzir o contra-exemplo fornecido quando o resultado de uma verificação é negativo. Como FIACRE é uma linguagem intermediaria, o projeto prevê outras aplicações, que serão desenvolvidas posteriormente, para retornar o contra-exemplo para a linguagem de modelagem. Por conseguinte, o projeto TOPCASED poderá reproduzir um contra-exemplo fornecido na linguagem fonte, ou seja, na linguagem de modelagem. A Figura 6.1 mostra o percurso da verificação dos modelos FIACRE com o ambiente TINA. A primeira etapa começa pela tradução do modelo FIACRE para TTS através do compilador desenvolvido neste trabalho (*frac*). Durante esta tradução, o compilador criará um arquivo auxiliar (*.tn*) com as informações necessárias para retornar o contra-exemplo quando uma propriedade não é verificada. A segunda etapa consiste em obter um arquivo binário (*.ktz*) com a representação abstrata dos estados do modelo TTS. Este arquivo é produzido pelo motor de exploração TINA. A terceira etapa é a verificação das propriedades formuladas em lógica temporal linear (*.ltl*) com a ferramenta *Selt*, que faz parte do ambiente TINA. Quando o resultado de uma verificação é negativo, um contra-exemplo será fornecido por esta última ferramenta. Na quarta etapa, este contra-exemplo (*.scn*) será reproduzido (*debugger*) em FIACRE por uma ferramenta a ser desenvolvida posteriormente, chamada simbolicamente na figura por *ce-tool*. Assim, com o advento de novas ferramentas,

os contra-exemplos reproduzidos em FIACRE poderão ser igualmente reproduzidos nas diferentes linguagens de modelagem (UML, SDL, SysML, etc) suportadas pelo ambiente TOPCASED.

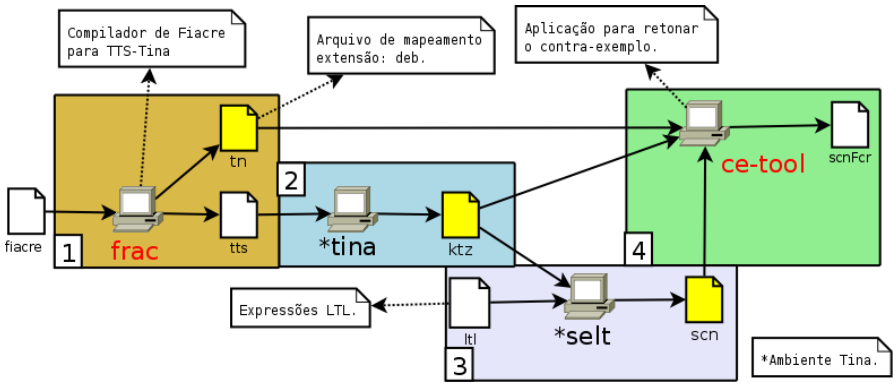


Figura 6.1: *Retorno de análise* TOPCASED.

Concluindo, graças ao compilador de FIACRE para TINA, a ferramenta a ser desenvolvida (*ce-tool*) e as ferramentas de transformação de modelos (UML, SysML, AADL, SDL, etc), o ambiente TOPCASED permitirá a verificação formal de forma transparente ao usuário, pois este poderá não somente verificar, como também analisar os resultados da verificação trabalhando somente com as linguagens de modelagem.

Referências Bibliográficas

[1] B. Berthomieu et al. The syntax and semantics of fiacre. Technical report, LAAS-CNRS, Toulouse, France, Janeiro 2008.

[2] F. Vernadat et F. Michel. Vérification de systeḿe de transitions[verification of transition systems]. *École d'été Temps Réel'97*, 1997.

[3] P. Farail et P. Gaufillet. Topcased: un environnement de développement open source pour les systèmes embarqués [topcased: an open source development environment for the embedded systems]. *Génie logiciel*, 1(73):pp. 16–20, 2005.

[4] B. Berthomieu, F. Vernadat, and P. Ribet. The tool tina construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14):2741 – 2756, 2004.

[5] A. Canals et al. An openembedd experimentation: "transformation from an sdl profiled uml model to a fiacre model". In *Proceedings of 4th European Congress ERTS EMBEDDED REAL TIME SOFTWARE 2008*, Toulouse, France, Janeiro 2008.

[6] N.G. Leveson. *Software: System Safety and Computers*, chapter Medical Devices: The Therac-25. ACM, New York, NY, USA, 1995. ISBN 0-201-11972-2.

[7] J.L. Lions. Ariane 5 - flight 501 failure. Report by the inquiry board, Paris, France, Maio 1996.

[8] Analysis guidebook for the verification of software and computer systems-volume ii: A practitioner companion. Technical Report http://hdl.handle.net/2014/22415, NASA, Jul 1997.

[9] P.G. Neumann. Illustrative risks to the public in the use of computer systems and related technology. *ACM SIGSOFT Software Engineering Notes*, 19(1):16–29, 1994.

[10] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, 1993. ISSN 0268-6961.

[11] J.P. Bowen and M.G. Hinchey. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. *10th international workshop on Formal methods for industrial critical systems*, pages 8–16, 2005.

[12] H. Garavel et al. CADP2006:A tool-box for the construction and analysis of distributed process. In *Proceedings of the 19rd International Conference on Computer Aided Verification CAV'07*, Berlim, Germany, 2007.

[13] D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[14] F. Allilaire et al. ATL-Eclipse Support for Model Transformation. In *Proceedings of the Eclipse Technology eXchange Workshop (eTX) at the ECOOP 2006 Conference*, Nantes, France, 2006.

[15] F. Allilaire and T. Idrissi. ADT: Eclipse development tools for ATL. In *Proceedings of EWMDA-2*, pages 171–178, 2004.

[16] J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc, Second Edition*. O'Reilly Media, Inc., Outubro 1992. ISBN 1-56592-000-7.

[17] T.A. Henzinger et al. Timed Transition Systems. In *Proceedings of (REX) Workshop*, pages 226–251. Springer, 1991.

[18] B. Berthomieu et al. Towards the verification of real-time systems in avionics the Cotre approach. *Electronic Notes in Theoretical Computer Science*, 80:203–218, Junho 2003.

[19] H. Garavel and F. Lang. *Formal Techniques for Networked and Distributed Systems-Forte 2002: 22nd IFIP WG 6.1*, volume 2529/2002. Springer Berlin/Heidelberg, Houston, Texas, USA, Novembro 2002. ISBN 978-540-00141-6.

[20] A. Basu et al. Modeling Heterogeneous Real-time Components in BIP. *Software Engineering and Formal Methods (SEFM 2006)*, pages 3–12, 2006.

[21] J. Cardoso e R. Valette. *Redes de Petri*. Editora da UFSC, 1997.

[22] J.D. Ullman, A.V. Aho, and R. Sethi. *Compiladores–Princípios Técnicas e Ferramentas*. LTC, 1995. ISBN 8521610572.

[23] Å. Wikström. *Functional programming using standard ML*. Prentice Hall International (UK) Ltd. Hertfordshire, UK, UK, 1987.

[24] S.T. Weeks. *MLton User Guide*, 2000.

[25] A.W. Appel et al. Userȁs Guide to ML-Lex and ML-Yacc. Technical report, Roger Price, 2002-2004.

[26] A.W. Appel, J.S. Mattson, and D. Tarditi. *A lexical analyzer generator for Standard ML*, 1989.

[27] D.R. Tarditi and A.W. Appel. *ML-Yacc Userȁs Manual*, Abril 2000.

[28] R. Saad et al. Schema de traduction sdl vers fiacre. Technical Report 07580, LAAS, 2007.

[29] ITU. *Specification and Description Language (SDL)*. Recommendation Z100. International Telecommunication Union (ITU), Agosto 2002.

[30] ITU. Specification and description language (sdl). Agosto 2002.

[31] L. Doldi. *Validation of Communications Systems with SDL: The Art of SDL Simulation and Reachability Analysis*. John Wiley and Sons, Abril 2003. ISBN 0470852860.

[32] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 2005. ISBN 0321267974.

[33] I.P. Paltor and J. Lilius. vuml: A tool for verifying uml models. In *Proceedings of 14th IEEE International Conference on Automated Software Engineering (ASE99)*, Outubro 1999.

[34] I. Schinz et al. The Rhapsody UML Verification Environment. In *Proceedings of Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 174–183, 2004.

[35] B. Berthomieu et F. Vernadat. Réseaux de petri temporels : méthodes d'analyse et vérification avec tina[timed petri nets: an approach for verification and analysis with tina]. *École d'été Temps Réel*, pages 18–22, 2006.

[36] S. Chaki et al. State/event-based software model checking. In *Proceedings of 4th International Conference on Integrated Formal Methods (IFM'04), Springer LNCS 2999*, pages 128–147, apr 2004.

[37] M.V. Linhares et al. Introducing the modeling and verification process in SysML. In *Proceedings of Emerging Technologies & Factory Automation, 2007. ETFA*, pages 344–351. IEE, Setembro 2007. ISBN 978-4244-0826-9.

[38] M. Hause, F. Thom, and A. Moore. *The Systems Modelling Language (SysML)*. White Paper. Artisan Software, 2004.

[39] S.C. Johnson. Yacc: Yet another compiler-compiler. Technical Report 32, Bell Laboratories, Murray hill, New Jersey, USA, 1975.

[40] M.E. Lesk. Lex: A lexical analyzer generator. Computing Science Technical Report 39, Bell Laboratories, Murray hill, New Jersey, 1975.

Apêndice A

BNF SUBFIACRE

A sintaxe BNF completa para o formato intermediário SUBFIACRE é apresentada abaixo:

```
tipe_id::=IDENT
enum::=IDENT
field::=IDENT
size::=NATURAL
integer::=['+'|'-']NATURAL
basic_type::=int|nat|bool
build_type::= interval integer '..' integer
        |enum enum^*_, end
        |array of size type
        |record(field^*_;':'type)^*_, end
        |union(field^*_;':'type)^*_, end
type::= basic_type | build_type | type_id
type_decl::= type type_id is type
var::=IDENT
literal::=NATURAL|true|false|enum|new NATURAL type
initalize::=literal
        | ('+'|'-')NATURAL
        | '['initializer^*_,']'
        | ''(field'='initializer)^*_,''
access::=var
        |access'['exp']'
```

```
        |access'.'field
unop::='-'|'S'|not|empty|dequeue|first
infixop::=or
        |and
        |'='|'<>'
        |'<'|'>'|'<='|'>='
        |'+'|'-'
        |'*'|'/'|'%'
binop::=enqueue
exp::=literal
        |access
        |unop exp
        |exp infixop exp
        |binop '(' exp ',' exp ')'
        |binop '('exp ',' exp')'
        |'('exp')'
        |exp'.?'field
port::=IDENT
channel_id::=IDENT
profile::=none|type*channel::=channel_id|profile
channel_id::=channel channel_id is channel
state::=IDENT
tag::=IDENT
name::=IDENT
time_interval::=('['|']')DECIMAL','(DECIMAL('['|']')|inf'[')
port_dec::=([in][out]port)*,':'channel
arg_dec::=([read][write]var)*,':'type
var_dec::=var*,':'type[':='initializar]
process_dec::=process name is
        states pstates*, init state
        [var var_dec*,]
        transition*transition::=IDENT kwin time_interval:from state+statement
statement::= |access+,':='exp+,
        |if exp then statement end
        |to state
```

```
        |statement ';' statement
component_decl::= component name is
        [priority (IDENT+|>IDENT+|)+,]
        composition
instance::=name
composition::= instance
program::=      [type_decl*]
                process_decl
                component_decl
```

Apêndice B

Descrição SUBFIACRE para o exemplo clássico de dois Trens

A descrição SUBFIACRE resultante para o exemplo apresentado no capítulo 2 é apresentada abaixo:

```
process train_8_train_7_train_6_train_5_controler_1_barrier_2
    is
    states up_barrier_2, lowering_barrier_2, down_barrier_2, raising_barrier_2,
        wait_controler_1, emitdown_controler_1, emitup_controler_1, far_train_5,
        close_train_5, on_train_5, left_train_5, far_train_6, close_train_6, on_train_6,
        left_train_6, far_train_7, close_train_7, on_train_7, left_train_7, far_train_8,
        close_train_8, on_train_8, left_train_8
        init  up_barrier_2 wait_controler_1 far_train_5 far_train_6 far_train_7 far_train_8
    var x_controler_1:int := 0
        from wait_controler_1 far_train_8
                if x_controler_1 + 1 = 1 then x_controler_1 := x_controler_1 + 1;
                                                to close_train_8 emitdown_controler_1
                end
        from wait_controler_1 far_train_8
                if not x_controler_1 + 1 = 1 then x_controler_1 := x_controler_1 + 1;
                                                to close_train_8 wait_controler_1
```

```
        end
 from close_train_8 to on_train_8
 from on_train_8 to left_train_8
 from wait_controler_1 left_train_8
        if x_controler_1 - 1 = 0 then x_controler_1 := x_controler_1 - 1;
                                      to far_train_8 emitup_controler_1
        end
 from wait_controler_1 left_train_8
        if not x_controler_1 - 1 = 0 then x_controler_1 := x_controler_1 - 1;
                                      to far_train_8 wait_controler_1
        end
 from wait_controler_1 far_train_7
        if x_controler_1 + 1 = 1 then x_controler_1 := x_controler_1 + 1;
                                      to close_train_7 emitdown_controler_1
        end
from wait_controler_1 far_train_7
        if not x_controler_1 + 1 = 1 then x_controler_1 := x_controler_1 + 1;
                                      to close_train_7 wait_controler_1
        end
 from close_train_7 to on_train_7
 from on_train_7 to left_train_7
 from wait_controler_1 left_train_7
        if x_controler_1 - 1 = 0 then x_controler_1 := x_controler_1 - 1;
                                      to far_train_7 emitup_controler_1
        end
 from wait_controler_1 left_train_7
        if not x_controler_1 - 1 = 0 then x_controler_1 := x_controler_1 - 1;
                                      to far_train_7 wait_controler_1
        end
 from wait_controler_1 far_train_6
        if x_controler_1 + 1 = 1 then x_controler_1 := x_controler_1 + 1;
                                      to close_train_6 emitdown_controler_1
        end
 from wait_controler_1 far_train_6
        if not x_controler_1 + 1 = 1 then x_controler_1 := x_controler_1 + 1;
```

```
                                        to close_train_6 wait_controler_1
        end
from close_train_6 to on_train_6
from on_train_6 to left_train_6
from wait_controler_1 left_train_6
        if x_controler_1 - 1 = 0 then x_controler_1 := x_controler_1 - 1;
                                        to far_train_6 emitup_controler_1
        end
from wait_controler_1 left_train_6
        if not x_controler_1 - 1 = 0 then x_controler_1 := x_controler_1 - 1;
                                        to far_train_6 wait_controler_1
        end
from wait_controler_1 far_train_5
        if x_controler_1 + 1 = 1 then x_controler_1 := x_controler_1 + 1;
                                        to close_train_5 emitdown_controler_1
        end
from wait_controler_1 far_train_5
        if not x_controler_1 + 1 = 1 then x_controler_1 := x_controler_1 + 1;
                                        to close_train_5 wait_controler_1
        end
from close_train_5 to on_train_5
from on_train_5 to left_train_5
from wait_controler_1 left_train_5
        if x_controler_1 - 1 = 0 then x_controler_1 := x_controler_1 - 1;
                                        to far_train_5 emitup_controler_1
        end
from wait_controler_1 left_train_5
        if not x_controler_1 - 1 = 0 then x_controler_1 := x_controler_1 - 1;
                                        to far_train_5 wait_controler_1
        end
from up_barrier_2 emitdown_controler_1 to wait_controler_1 lowering_barrier_2
from raising_barrier_2 emitdown_controler_1 to wait_controler_1 lowering_barrier_2
from down_barrier_2 emitup_controler_1 to wait_controler_1 raising_barrier_2
from lowering_barrier_2 to down_barrier_2
from raising_barrier_2 to up_barrier_2
```

```
componente main is
 port app:none in [0,0], exit:none in [0,0], down:none in [0,0], up:none in [0,0],
        lower_barrier_1:none in [5,10], raise_barrier_1:none in [5,10],
        enter_train_1:none in [20,30], leave_train_1:none in [30,50],
        enter_train_2:none in [20,30], leave_train_2:none in [30,50],
        enter_train_3:none in [20,30], leave_train_3:none in [30,50],
        enter_train_4:none in [20,30], leave_train_4:none in [30,50]

        process train_8_train_7_train_6_train_5_controler_1_barrier_2
```

Apêndice C

Compilador: Analisador Léxico e Sintático

O analisador léxico e sintático empregado neste trabalho foi gerado utilizando as ferramentas ML-Lex e MLYacc, que são variantes das ferramentas Lex e Yacc para a linguagem SML. Estas ferramentas, que foram primeiramente introduzidas por Lesk [39] e Johnson [40], permitem ao programador declarar tanto a semântica como a sintaxe da linguagem de forma explícita, concentrando-se em "o que fazer com o programa reconhecido" e não em como "reconhecê-lo". Esta abordagem economiza tempo e facilita a manutenção do programa.

A ferramenta ML-Lex gera um analisador léxico escrito na linguagem SML. Esta ferramenta utiliza padrões para comparar cadeias de caracteres (*string*) inseridos (via teclado, entrada de arquivos, etc), convertendo-os em fichas (*tokens*). Estas fichas são na verdade representações numéricas das cadeias de caracteres.

O analisador léxico gerado procura por identificadores na cadeia de caracteres fornecida (entrada), os inserindo em uma tabela de símbolos. Esta tabela também pode possuir outras informações, tais como o tipo do dado (inteiro, real, etc). Todas as referências subseqüentes a estes identificadores serão referenciadas pelo índice do símbolo na tabela.

A ferramenta ML-Yacc é utilizada para gerar um analisador sintático também escrito em SML. O analisador gerado produz uma árvore abstrata a partir de uma base de regras. Estas regras têm por objetivo impor uma estrutura hierárquica nas fichas (*tokens*) criadas pelo analisador léxico, resultando na árvore abstrata. A Figura C.1 ilustra como estes analisadores trabalham em conjunto.

Maiores informações sobre ML-Lex e ML-Yacc podem ser encontradas em:[16],[25] e [27].

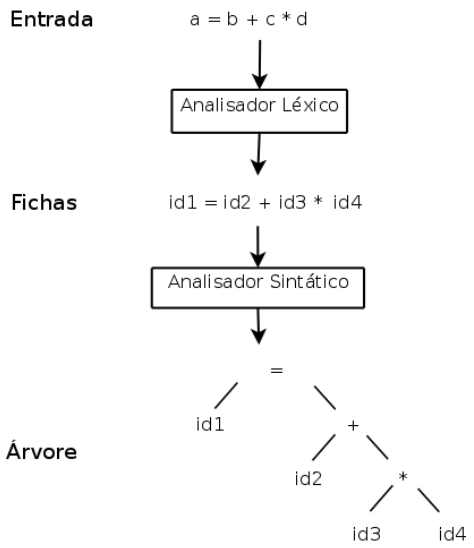


Figura C.1: *Analisador Léxico e Sintático.*

C.1 Especificação ML-Lex

Uma especificação ML-Lex possui a forma geral:

Declarações do Usuário
%%
Definições ML-Lex
%%
Regras

Na primeira seção, "Declarações do Usuário", o programador pode inserir valores para serem utilizados pelas regras. As expressões regulares podem ser nomeadas para auxiliar a legibilidade do código na seção "Definições ML-Lex". Estados iniciais e alguns comandos também podem ser inseridos nesta seção. Na seção "Regras", o programador deve inserir as regras necessárias para efetuar a análise léxica. Estas regras são divididas em duas partes: uma expressão regular e uma ação. A expressão regular define a classe de palavras que a regra busca. A ação é um fragmento de programa a ser executado quando uma regra é encontrada.

Um fragmento da especificação ML-Lex para a linguagem FIACRE é apresentado abaixo:

```
#Declarações do Usuário
structure Tokens = Tokens
```

```
val lineno = ref 0
val charno = ref 0

fun incl n = (lineno := n + !lineno; charno := 0)
fun incc n = (charno := n + !charno; !charno)
fun mkpos z =   let val lc = (!lineno, !charno)
                in charno := z + !charno; lc end
val comment = ref false;
.....................................................................
%%
#Definições ML-Lex
alpha=[A-Za-z];
digit=[0-9];
ws = [\ \t];
underline = "_";
dot = ".";
comma = ",";
%%
#Regras
\n => (incl 1; lex());
{ws} => (incc 1; lex());
"interval" => (Tokens.INTERVAL(mkpos 8));
"enum" => (Tokens.ENUM(mkpos 4));
"record" => (Tokens.RECORD(mkpos 6));
"union"         => (Tokens.UNION(mkpos 5));
"queue" => (Tokens.QUEUE(mkpos 5));
"process" => (Tokens.PROCESS(mkpos 7));
"component" => (Tokens.COMPONENT(mkpos 9));
"init" => (Tokens.INIT(mkpos 4));
"var" => (Tokens.VAR(mkpos 3));
.....................................................................
```

C.2 Especificação ML-Yacc

A especificação ML-Yacc também é formada por três partes, são elas:

Declarações do Usuário
%%
Declarações ML-Yacc
%%
Regras

Na seção “Declarações do Usuário”, o programador pode declarar variáveis e funções que são acessíveis pelas outras seções.

A seção “Declaração ML-Yacc” é utilizada para inserir as declarações requeridas e opcionais. Dentre as declarações requeridas estão às listas de estados terminais e não terminais. Nesta seção o programador pode, caso necessário, definir a relação de precedências entre os terminais.

Na última seção, chamada aqui de “Regras”, o programador descreve a gramática da linguagem utilizando uma variante da Backus Naur Form(BNF).

Apresentamos abaixo um fragmento da especificação ML-Yacc para a linguagem FIACRE:

```
%%
%pos int
#Estados Terminais
%term INTERVAL  | ENUM  | RECORD  | QUEUE  | PROCESS  | INIT
| VAR  | FROM of int  | ANY  | WHERE  | IF of int  | OF
| THEN  | ELSE  | ELSIF of int  | SELECT  | END  | TO of int
| ARRAY  | T_BOOL  | T_NAT  | T_INT  | TRUE  | FALSE
| FULL  | FIRST | EMPTY  | DEQUEUE  | ENQUEUE | NOT
| IN | OUT  | NONE | STATES  | PRIORITY  | PORT
.....
#Resolução de Associações
%right NOT COERCE
%left OR
```

```
%left AND
%left EQ NE
%left LT LE GT GE
%left ADD SUB
%left MUL DIV MOD

#Estados não Terminais
%nonterm
  start of Fiacre
| declarationList of Declaration list
| processDecl of Declaration
| portDeclarations of PortDecl list
| portDecls of PortDecl list
| portDecl of PortDecl
| portEntries of (line * PortAttr list * string) list
| portEntry of line * PortAttr list * string
| portAttrs of PortAttr list
| varDeclarations of ParamDecl list
| varDecls of ParamDecl list

%%
#Regras
start: declarationList   (Program declarationList)

(* note: allows declaration list without main *)
declarationList:  ([])
     | IDENT          ([Main (#2 IDENT,#1 IDENT)])
     | typeDecl declarationList (typeDecl::declarationList)
     | channelDecl declarationList (channelDecl::declarationList)
     | processDecl declarationList (processDecl::declarationList)


(* process declarations *)

processDecl: PROCESS IDENT portDeclarations varDeclarations IS
```

```
     STATES identsComma INIT IDENT lvarDeclarations transitions
(ProcessDecl (#2 IDENT1,#1 IDENT1,...........

(* component declarations *)

processDecl: COMPONENT IDENT portDeclarations varDeclarations IS
     lvarDeclarations locportDeclarations priorityDeclarations composition
(ComponentDecl (#2 IDENT1,#1 IDENT1,..........
................................................................................
```

Apêndice D

Compilador: Árvore Abstrata FIACRE e Geração do Código Alvo

Neste apêndice mostramos detalhes da implementação das etapas de expansão e geração do código. Na primeira seção apresentamos a árvore abstrata, na segunda algumas partes da implementação das operações de expansão e na terceira a geração do código TTS.

D.1 Árvore Abstrata

A representação intermediária, que é utilizada para interligar a parte frontal com a final, consiste em uma árvore abstrata do código FIACRE. Esta árvore é descrita em SML como um novo tipo de dado parametrizado chamado *dataype* **FIACRE**. A árvore abstrata implementada compilador é apresentada abaixo:

```
type line = int;
datatype Fiacre = Program of Declaration list
        | ProgramRef of DecRef list
and Declaration =  TypeDecl of line * string * Type
        | ChannelDecl of line * string * Channel
        | ProcessDecl of line * string * PortDecl list * ParamDecl list
                * StatesDecl * VarDecl list * Transition list
        | ComponentDecl of line * string * PortDecl list * ParamDecl list
                * VarDecl list * locPortDecl list * PriorDecl list * Composition
```

```
        | Main of int * string

and Type =  BasicType of BasicType
        | NamedType of string * Type option ref (*To locate the base type*)
        | BuiltType of BuiltType
        | NoType                                (* for typechecker  *)
        | AnyType                               (* for typechecker  *)
and BasicType = Bool | Nat | Int
and BuiltType = Interval of int * int
        | Enum of string list
        | Record of (string * Type) list
        | Union of (string * Type) list
        | Array of int * Type
        | Queue of int * Type
and Exp = ParenExp of Exp
        | LitExp of Literal
        | AccessExp of Access
        | InfixExp of Infix * Type option ref * Exp * Exp
        | BinopExp of Binop * Type option ref * Exp * Exp
        | UnopExp of Unop * Type option ref * Exp
        | UtestExp of Exp * string * Type option ref
and Literal = IntLit of int
        | BoolLit of bool
        | QueueLit of int * Type
        | EnumLit of string * Type option ref
and Initializer = LitInit of Literal
        | EnumInit of string
        | RecordInit of (string * Initializer) list
        | UnionInit of string * Initializer
        | ArrayInit of Initializer list
and Access = VarAccess of string * Type option ref
        | ArrayAccess of Access * Exp * Type option ref
        | RecordAccess of Access * string * Type option ref
        | UnionAccess of Access * string * Type option ref
and Infix = AND | OR | LE | LT | GT | GE | EQ | NE | ADD
```

```
        | SUB | MOD | MUL | DIV
and Binop = Enqueue
and Unop = MINUS | NOT | COERCE | FULL | EMPTY | DEQUEUE | FIRST
and Channel = NamedChannel of string * Channel option ref
        | ProfileChannel of Profile (* [] means none *)
        | UnionChannel of Channel * Channel
        | FlatChannel of Profile list (* for typechecker *)
and Statement = StNull
        | StBlock (*for Comunication transformation constrained by the where option*)
        | StCond of line * Exp * Statement * (line * Exp * Statement) list * Statement optic
        | StSelect of Statement list
        | StSequence of Statement * line * Statement
        | StTo of line * string
        | StPlusTo of string list (*When there is more then one target state.
                                    Used by the composition operation.*)
        | StWhile of line * Exp * Statement
        | StAssign of line * Access list * Exp list
        | StAny of line * Access list * Exp option
        | StSignal of line * string * Profile option * Profile option ref
        | StInput of line * string * Profile option * Profile option ref * string list * Exp
        | StInputList of line * string * string list list (*Input composition among transiti
        | StOutput of line * string * Profile option * Profile option ref * Exp list
and Bound =  Open of real | Closed of real | Infinite
and VarAttr = READ | WRITE
and PortAttr = IN | OUT
and Composition = ShuffleComp of line * Composition list
        | SyncComp of line * Composition list
        | ParComp of line * (string list * Composition) list
        | InstanceComp of Instance * DecRef option (*TTS process reference*)
        | EmptyComp     (*Internal Use - Used only by Implode components normalisation *)

withtype
        TypeDecl = line * string * Type
and     ChannelDecl = line * string * Channel
and     PortDecl = (line * PortAttr list * string) list * Channel
```

```
and     ParamDecl = (line * VarAttr list * string) list * Type
and     StatesDecl = line * string list * string
and     VarDecl = (line * string) list * Type * Initializer option
and     PriorDecl = (line * string) list * (line * string) list
and     locPortDecl = ((line * PortAttr list * string) list * Channel) list * (Bound * Bound)
and     Instance = (line * string) * string list list * Exp list
and     Transition = line * string * Statement
and     TimedTransition = Tag * line * string list * Statement * (Bound * Bound)
        (*Transitions with time constraints - Added by Timed Transformation*)
and     Profile = Type list;
```

D.2 Memória - Tabelas Globais

Este compilador não utiliza tabelas globais para representar a árvore abstrata. Optou-se por utilizar a própria árvore abstrata para as etapas de Expansão e Geração de código. Contudo, esta árvore foi estendida com *ponteiros* para permitir a atualização dos dados e também para tornar mais rápido o acesso aos dados desejados. A árvore abstrata estendida é apresentada abaixo:

```
and DecRef =
        TTSProcess of (line * string * PortDecl list *
                ParamDecl list * (line * string list * string list) *
                VarDecl list * locPortDecl list * PriorDecl list *
                TimedTransition list) ref
        | TypeRef of (line * string * Type) ref
        | ChannelRef of (line * string * Channel) ref
        | ComponentRef of (
                line *
                string *
                PortDecl list *
                ParamDecl list *
                VarDecl list *
                locPortDecl list *
                PriorDecl list *
                Composition) ref
```

```
    | MainRef of (int * string) ref
```

D.3 Geração - Tradução de SUBFIACRE para TTS-TINA

Após a tradução do modelo FIACRE em SUBFIACRE, o compilador efetua a geração do código alvo escrito no formato TTS-TINA. Esta operação não compreende somente os aspectos relativos aos formalismos matemáticos TTS, mais também como TINA manipula o dados.

D.3.1 Formato de entrada TINA (TTS-Tina)

O formato alvo da tradução é um sistema TTS descrito em um formato aceito pela ferramenta de verificação TINA. Este formado consiste em dois arquivos: o primeiro com a extensão .net, responsável pela definição estrutural da Rede de Petri Temporal responsável pelo controle do sistema, e outro com a extensão .c, responsável pela parte de dados do sistema (estrutura de dados, condições e ações). Maiores informações sobre este formato pode ser encontradas na documentação do ambiente TINA[4].

Arquivo .net Este arquivo define uma estrutura da Rede de Petri Temporal, ou seja, os lugares, os arcos e as transições com suas respectivas restrições temporais. O arquivo .net relativo ao exemplo dado na Figura 3.2 é mostrado abaixo:

```
Net exemplo
pl p0 (1)
tr t_nom [Tmin,Tmax]  p0 -> p1
```

As palavras chaves deste arquivo são apresentadas a seguir:

- **net** : define o nome da rede. Ex : net exemplo.
- **pl** : determina a marcação inicial. Ex : pl p0 (1).
- **tr** : esta palavra define uma nova transição a partir de quatro parâmetros. O primeiro declara o nome da transição. O segundo parâmetro é relativo à restrição temporal. Os dois últimos são os lugares de início e de chegada da transição. Ex: tr t_nom [Tmin,Tmax] p0 -¿ p1.

Arquivo .c

Este arquivo é responsável pela extensão da Rede de Petri em um Sistemas de Transição Temporizados. Através deste arquivo é possível manipular as variáveis do sistema e associar expressões condicionais e ações as transições declaradas no arquivo .net. As palavras chaves do arquivo .c são demonstradas no exemplo abaixo relativo à transição apresentada na Figura 3.2:

- Cabeçalho: Os cabeçalhos a seguir são impostos pela ferramenta TINA: *math.h*, *string.h*, *stdio.h* e *avl.h*.

- Estrutura valor(**value**) : esta estrutura define a estrutura de dados do sistema, ou seja, todos os dados que pertencem ao sistema são declaradas aqui.

```
struct value { int x; }
struct value initval;
```

- Função compara (**compare_value**): esta função executa a comparação lexicográfica entre os tipos de dados criados na estrutura value.

```
bool compare_value(struct value *v1, struct value *v2){
  if(v1->x > v2->x) {return 1;}
  if(v1->x < v2->x) {return -1;}
  return 0;}
```

- Função início(**initial**): esta função inicia as variáveis do sistema.

```
Key initial () {
initval.x = 0;
 init_storage(compare_value,free_value);
 return store (&initval);}
```

A função armazenar *store* que se executa no fim da função *intial* é responsável para armazenar os dados no estado do sistema.

- Matriz Transições (**transtable**): Esta matriz possui todos os nomes das transições declaradas no arquivo .net.

```
char *transtable[1] = {"t_nom"};.
char **transitions (int *sz) {
 *sz = 1; /*Número de transições.*/
 return transtable;}
```

- Função condição (**pre_(nome da transição)**): esta função será criada para todas as transições que possuírem uma expressão condicional associada no modelo (construção if). Ela é identifica pelo nome "pre" seguido pelo índice(posição numérica) do nome da transição na matriz **transtable**. Toda transição declarada no arquivo **.net** que possuir uma função "pre" associada, será disparada somente se esta função retornar o valor *true*.

```
bool pre_1 (key s) {
struct value *v = lookup(s);
return ( (v->x > 5) ?1:0);}
```

- Função ação - **act_(nome da transição)** : Toda a ação de manipulação sobre as variáveis do sistema é desempenhada por funções nomeadas com o prefixo "act" seguido pelo índice do nome da transição na matriz transtable. Estas funções são executadas quando a transição é disparada.

```
key act_1(key s){
  struct value *v = lookup(s);
  struct value *new = (struct value *) malloc(sizeof(struct value));
 memcpy(new,v,sizeof(struct value));
  new->x = 5;
  return store(new); }
```

- Função imprimir estado (**sprint_state**): esta função imprime o valor das variáveis no estado corrente.

```
int sprint_state (int sz, char *buff, key s) {.
struct value *v = lookup(s);
char temp[255];  strcpy(safe,"\0");
  sprintf(temp,"x=%i,",v->x);
 if(sz < strlen(temp)) {
  return -1; }
  else {
  sprintf(buff,temp);
  return strlen(buff); } }
```

- Função valores observáveis (**obs_values**): todas as variáveis observáveis que fazem parte do sistema são retornadas para o ambiente TINA através desta função.

```
char *obsnames[1] = {"x"};
char **obs_names (int *count) {*count = 1; return obsnames;}
int ovalues[1];
int *obs_values (key s) {
struct value *v = lookup(s);
ovalues[0] = v->x;
return ovalues;}
```

D.3.2 Geração do código TTS-TINA (Tipos)

A codificação do modelo SUBFIACRE para TTS-TINA inicia-se a partir da declaração dos tipos de dados no arquivo .c. Esta etapa é descrita na linguagem C e aceita diferentes tipos de dados. A tradução dos tipos é apresentada a seguir: **Nat**, **Int**, **Interval**, **Bool**, **Enum**, **Union** e **Array** Os tipos **nat** (natural), **int** (inteiro), **interval** (intervalo), **bool** (booleano) e **enum** (enumeração), **union** (união), **array** (matriz) são compatíveis com a linguagem C e sua tradução é feita diretamente.

FIACRE	TTS-TINA
type newNat is nat; type newInt is int; type newInterval interval 0..5 type newBool is bool; type newEnum is enum e1,e2 end; type newRecord is union field1:bool, field2:int end type newArray is array of 2 int;	unsigned int newNat; int newInt; int newInterval; bool newBool; enum newEnum {e1, e2}; union newRecord { boot field1; int field2; }; Int newArray[2];

Record

O tipo record é traduzido utilizando o tipo structure da linguagem C como mostra o exemplo abaixo:

FIACRE	TTS-TINA
type newRecord is record field1:bool, field2:int end	structure newRecord { bool field1; intfield2;};

Queue

O tipo queue (fila) é um tipo especial FIACRE que consiste em uma fila do tipo FIFO (o primeiro a entrar é o primeiro a sair). Este tipo é traduzido como em duas variáveis: uma matriz com a mesma dimensão que a fila e um inteiro para marcar a posição do último elemento inserido na fila. Além disso, cinco funções são criadas para os operadores full, empty, dequeue, enqueue e first.

FIACRE	TTS-TINA
Type newQueue is queue of 4 int;	int newQueue_values[4]; int newQueue_head;

Variáveis

A etapa seguinte da codificação consiste na declaração das variáveis no arquivo .c. Como foram apresentadas anteriormente, todas as variáveis que fazem parte dos sistemas são declaradas como parte da estrutura value. Nesta estrutura não existe nenhuma distinção entre as variáveis definidas como compartilhadas ou locais no modelo Original. Um exemplo é dado abaixo:

FIACRE	TTS-TINA

`var x,y :int, start :bool`	`struct value {`
	`int x;`
	`int y;`
	`bool start; };`

A distinção entra as variáveis compartilhadas e locais é definida pela função obs_value. As variáveis locais, ao contrário das globais, não serão retornadas por esta função.

D.3.3 Codificação de uma Transição SUBFIACRE em TTS-Tina

Como o formato TTS-TINA é dividido em dois arquivos. Uma transição SBFIACRE é simultaneamente traduzida em ambos os arquivos. A parte referente à estrutura da rede (estado inicial e final) é descrito no arquivo .net. Já a expressão condicional e as manipulações descritas nas transições são codificadas no arquivo .c. A figura abaixo apresenta uma transição escrita no formato SUBFIACRE e sua respectiva tradução nos arquivos .net e .c.

from idle if $x3$ then $y := x+2$; to endState	
.net	**.c**

`tr nome`
`[0,w[ idle -> endState`
	`bool pre_1 (key s) {`
	`struct value *v = lookup(s);`
	`return (( v->x>3)?1:0); }`
	`key act_1(key s) {`
	`struct value *v = lookup(s);`
	`struct value *new = (struct value *)`
	`malloc(sizeof(struct value));`
	`memcpy(new,v,sizeof(struct value));`
	`new->y=new->x+2;`
	`return store(new); }`

Como é possível observar, o arquivo **.net** possui a declaração estrutural da transição, ou seja, o nome da transição (*nome*), a restrição temporal (*[0,w[*) e os estados de início e fim ($idle \rightarrow endState$).

A expressão condicional ($x3$) e as manipulações sobre as variáveis ($y := x+2$) que fazem parte da transição SUBFIACRE são descritas no arquivo **.c** através das funções pre_1 e act_1. Recapitulando:

- pre : A expressão condicional da transição SUBFIACRE é traduzida na função com o prefixo *pre*.
- act : As ações sobre a estrutura de dados do sistema de uma transição SUBFIACRE são traduzidas na função com prefixo *act*.

Apêndice E

Fábrica: Modelo FIACRE e Diagramas de Estado

Este apêndice apresenta os Diagramas de Estado e o modelo formal FIACRE para o exemplo apresentado no Capítulo 5.

E.1 Tipos de dados

Para auxiliar a codificação em FIACRE desde modelo, três novos tipos de dados foram criados:

- Nome das linhas: Tipo enumeração que possui os nomes das linhas de produção (L_1 e L_2). Declaração FIACRE: type NomeLinhas is enum NN, L1, L2 end.
- Nome das máquinas:Tipo enumeração que possui os nomes das máquinas (M_1, M_2, M_3 e M_4). Declaração FIACRE: type NomeMaquinas is enum NN, M1, M2, M3, M4 end.
- Lista de máquinas: É uma matriz que armazena os nomes de 3 máquinas. Esta matriz é utilizada para configurar as máquinas operadas pelas linhas de produção. Declaração FIACRE:type ListaMaquina is array of 3 NomeMaquinas.

E.2 Process: Diagramas de Estado e modelo FIACRE

Os Diagramas de estados de cada processo, juntamente com as suas respectivas descrições FIACRE, são apresentados abaixo:

E.2.1 Linha de Produção

O produto produzido por esta fábrica é trabalhado em duas linhas de produção. Cada linha possui um trabalhador dedicado e utiliza três máquinas das quatro máquinas que a Fábrica possui. As máquinas utilizadas e a ordem de execução variam para cada linha. A Figura E.1 mostra que cada processo (*linha*) possui uma matriz (*listaMaq*) de *cl* posições. Esta matriz é do tipo enumeração (*nomeMaquinas*) e possui o nome das máquinas que a linha está configurada para operar.

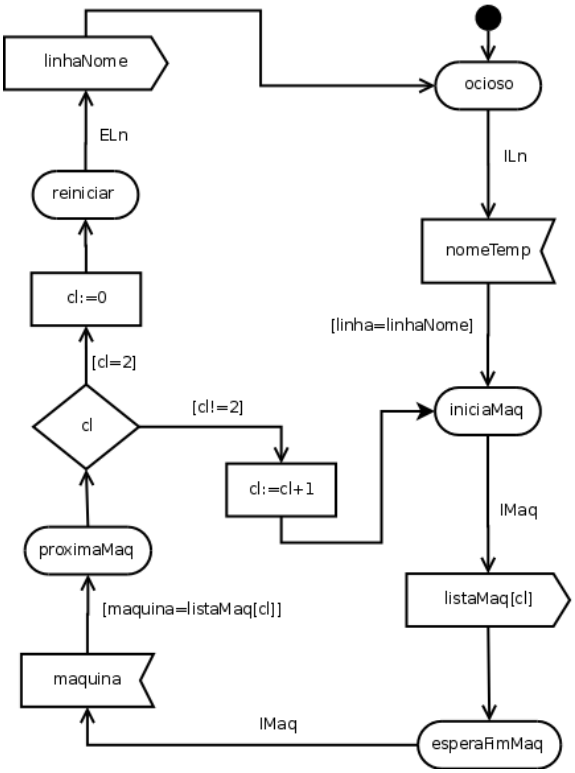


Figura E.1: *Diagrama de Estados do processo Linha.*

A descrição FIACRE para o processo *linha* é apresentada abaixo:

```
process linha [in ILn, out ELn:NomeLinhas, out IMaq, in EMaq:NomeMaquinas]
        (listaMaq:ListMaquina, linhaNome:NomeLinhas) is
states ocioso, reiniciar, iniciaMaq, esperaFimMaq, proximaMaq init ocioso
var linha:NomeLinhas:=NN, cl:interval 0..2:=0, maquina:NomeMaquinas:=NN
        from ocioso
                ILn? linha where linha=linhaNome; to iniciaMaq
        from iniciaMaq
                IMaq! listaMaq[cl]; to esperaFimMaq
```

```
        from esperaFimMaq
                EMaq? maquina where maquina=listaMaq[cl]; to proximaMaq
        from proximaMaq
                if cl=2 then cl:=0; to reiniciar
                else cl:=cl+1; to iniciaMaq end
        from reiniciar
                ELn! linhaNome; to ocioso
```

E.2.2 Operário

Pela definição do exemplo (Capítulo 5), o operário trabalha em uma linha de produção e faz uma pausa para descanso a cada ciclo contínuo de trabalho. A figura E.2 mostra o Diagrama de Estados para o processo *operário*.

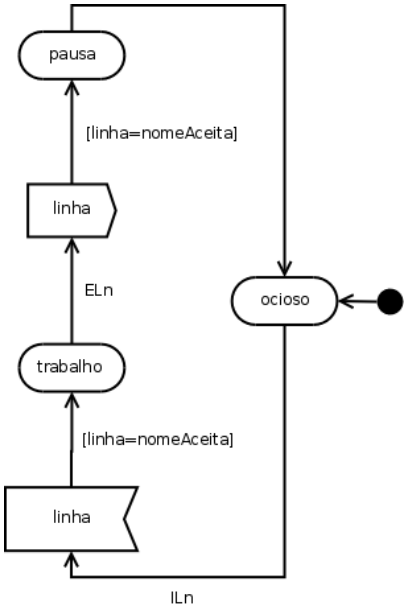


Figura E.2: *Diagrama de Estados do processo operário.*

A descrição FIACRE para o processo *operário* é apresentada abaixo:

```
process Operário [in ILn, in ELn: NomeLinhas, Observer:none]
                (linhaAceita:NomeLinhas) is
states ocioso, trabalho, pausa init ocioso
var linha:NomeLinhas:=NN
        from ocioso
```

```
        ILn? linha where linha=linhaAceita; to trabalho
from trabalho
select
        ELn? linha where linha=linhaAceita; to pausa
[]      Observer; to trabalho
end
from pausa to ocioso
```