Análise de Traço com Geração de Diagnósticos para Testes de Comportamento de uma Implementação de Protocolo de Comunicação em Presença de Falhas

Márcio Roberto Stefani

Dissertação de Mestrado

Análise de Traço com Geração de Diagnósticos para Testes de Comportamento de uma Implementação de Protocolo de Comunicação em Presença de Falhas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Márcio Roberto Stefani e aprovada pela Banca Examinadora.

Campinas, 7 de maio de 1997.

Eliane Martins

Eliane Martins
Instituto de Computação - UNICAMP
(Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Instituto de Computação
Universidade Estadual de Campinas

Análise de Traço com Geração de Diagnósticos para Testes de Comportamento de uma Implementação de Protocolo de Comunicação em Presença de Falhas

Márcio Roberto Stefani

maio 1997

Banca Examinadora:

- Eliane Martins
  Instituto de Computação - UNICAMP (Orientadora)
- Ana Maria de Alencar Price
  Instituto de Informática - UFRGS
- Ricardo de Oliveira Anido
  Instituto de Computação - UNICAMP
- Tomasz Kowaltowski
  Instituto de Computação - UNICAMP

Tese de Mestrado defendida e aprovada em 07 de maio de 1997 pela Banca Examinadora composta pelos Professores Doutores

Profª. Drª. Ana Maria de Alencar Price

Prof. Dr. Tomasz Kowaltowski

Prof. Dr. Ricardo Anido

Prof. Dr. Eliane Martins

Aos meus pais João e Virgínia e à minha irmã Denise, por tudo que têm feito por mim, durante todos os anos de minha vida.

Agradecimentos

A Deus, pela vontade e coragem.

À profª Eliane Martins, pela orientação e estímulo no decorrer do trabalho.

Ao CNPq e FAPESP, pelo apoio financeiro.

À Selma Bássiga Sabião, pela valorosa e inestimável ajuda nos momentos difíceis.

Ao Acires Dias, Fátima, Tiago e Lucas, que são uma verdadeira família para mim.

À Vicentina e Felice Coiro, por me acolherem como um filho.

Ao Luiz Carlos Pantojo, Hemerson Grando e Bito, pela convivência fraterna e amiga.

À Cristiane Grando, pelo apoio e pelo incentivo.

Ao Marcos Salenko, pelo suporte no projeto da ferramenta.

Aos amigos Raul Cesar Requião, Maurício Atanásio, Ana Waleska, Arédis Sebastião de Oliveira, Edmar Lourenço Borges, Emerson Madureira, Lúcio Dutra, Dinalva (*in memorian*), Eduardo Camponogara, Vicente De Luca, Sávio S. Ipiranga, Solange Sobral, Luiz Eduardo da Silva, Cristina Toyota, Amanda Rosa Apolinário, Patrícia Ropelatto, Rogério De Carlo, Marília Coelho, Marcelo Marcos Barbosa, Ernesto Freud, Delano, e outros igualmente importantes pela alegria que trouxeram no decorrer do trabalho.

Aos professores Ana Maria de Alencar Price, Tomasz Kowaltowski e Ricardo Anido, por serem membros da banca examinadora.

Aos trabalhadores brasileiros que, através de seus impostos, sustentam a estrutura universitária neste País.

Resumo

Uma dificuldade comum aos testes de *software* é o problema do oráculo: como determinar se os resultados dos testes estão corretos? O oráculo é um mecanismo que analisa os resultados dos testes com base numa referência para o comportamento do *software*. Essa referência precisa ser a mais confiável possível.

Neste trabalho foi desenvolvido um método de análise de resultados para testes de comportamento e testes por injeção de falhas por *software* aplicados à uma implementação de protocolo de comunicação. Para representar o comportamento do protocolo, utilizou-se um modelo baseado em máquina finita de estados estendida. Foi desenvolvido também o projeto de uma ferramenta que coloca esse método em prática. Essa ferramenta de análise de resultados deve ser incorporada ao ATIFS, um Ambiente integrado de Testes baseado em Injeção de Falhas por Software. O mecanismo de análise de resultados é um analisador de traço de execução. Um traço é um histórico da execução dos testes. Com base no modelo do comportamento e no traço de execução, o analisador de traço produz as seguintes saídas: veredictos sobre as saídas produzidas por uma implementação sob teste, diagnósticos de erros e informações sobre a ativação de mecanismos de detecção de erros e tolerância a falhas.

Abstract

A common difficulty to software testing is the oracle problem: how to determine whether or not test results are correct? The oracle is a mechanism that analyses test results by using a reference for the software behavior. This reference should be as reliable as possible.

In this work, it had been developed a test result analysis method for behavior tests and software fault injection tests applied to a communication protocol implementation. To represent the protocol behavior, it had been used a model based on extended finite state machine. It had been also developed a design of a tool that puts this method into practice. This test result analysis tool will be embedded in the ATIFS, an integrated test environment based on software fault injection. The mechanism of test result analysis is a trace analyzer. A trace is an execution history of the tests. The following outputs are produced by the trace analyzer based on the behavior model and the trace: verdicts about the outputs produced by an implementation under test, error diagnoses, and informations about activation of the error detection and fault tolerance mechanisms.

Conteúdo

PARTE II: DESENVOLVIMENTO DE UMA FERRAMENTA DE ANÁLISE DE TRAÇO COM GERAÇÃO DE DIAGNÓSTICOS

Lista de Figuras

Lista de Tabelas

Nomenclatura

AF	Active Ferry clip
ASP	Abstract Service Primitive
CCITT	Comité Consultatif Internationale de Télégraphie et Téléphonie.
DFD	Diagrama de Fluxo de Dados
EFSM	Extended Finite State Machine
FDT	Formal Description Techniques
FIA	Fault Injection Agent
FIC	Fault Injection Controller
FSM	Finite State Machine
ISO	International Standards Organization
IUT	Implementation Under Test
LOTOS	Language Of Temporal Ordering Specification
LSAP	Lower Service Access Point
LT	Lower Tester
MTF	Mecanismo de Tolerância a Falhas
OMT	Object Modeling Technique
OSI	Open Systems Interconnection
PCO	Point of Control and Observation
PDU	Protocol Data Unit
PF	Passive Ferry clip
SAP	Service Access Point
SUT	System Under Test
TS	Test System
TSC	Test Sequence Controller
USAP	Upper Service Access Point
UT	Upper Tester
V&V	Verificação e Validação

1. Introdução

Nos últimos anos, a necessidade de garantir a qualidade de um *software* vem aumentando cada vez mais. Um dos meios para se atingir o objetivo da qualidade é a atividade de teste. Essa atividade tem alto custo e ainda requer intensa atividade humana.

Raramente, os testes ocupam um lugar notável no processo de desenvolvimento de *software* e, geralmente, são realizados de forma inadequada, além de serem deixados para a última fase do desenvolvimento e, até mesmo, economizados quando o tempo e os recursos são escassos. A maioria das melhoras de qualidade e produtividade em testes tem sido alcançada com a automatização do processo de teste através de ferramentas e da incorporação efetiva dos testes no processo de desenvolvimento. As principais etapas do processo de teste são: geração, execução e análise dos resultados dos testes. Esta última etapa tem como objetivo verificar se o *software* se comportou corretamente durante a execução dos testes.

A automatização da análise de resultados de testes é uma necessidade que cresce proporcionalmente ao aumento do número de testes necessários para exercitar todas as funcionalidades do *software* testado. O número de testes necessários, por sua vez, cresce em função do aumento da complexidade do *software*. Quando o volume de testes executados é grande, a análise dos resultados é difícil de ser feita manualmente, além de ser propensa a erros e exigir muito tempo.

Para que a análise de resultados possa ser automatizada, é necessária a existência de um modelo para o *software* testado. Esse modelo pode ser usado tanto para a geração de casos de testes quanto para a análise dos resultados dos testes executados. Na análise dos resultados, o modelo é a referência para o comportamento do *software*. Essa referência precisa ser a mais confiável possível. A análise de resultados pode, ainda, ser complementada pela geração de diagnósticos de erros, sempre que possível.

Protocolos de comunicação (capítulo 4), como qualquer outro *software*, também precisam ser testados. Entretanto, a análise dos resultados dos testes de protocolos precisa levar em conta a natureza distribuída dos protocolos. A análise de traço (capítulo 8) de

execução é uma técnica que pode ser usada para analisar resultados de testes de protocolos. Um traço de execução é um histórico da execução dos testes. A análise de traço é realizada após a execução dos testes. As interações que foram observadas durante a execução dos testes são investigadas tomando-se como referência o modelo do comportamento do protocolo. Ao final da análise, veredictos informam se as interações observadas estão de acordo com o modelo, ou não.

Esse trabalho desenvolve um método que combina a análise de traço com geração de diagnósticos de erros para testes de comportamento de protocolos, além de fornecer informações sobre a presença de falhas durante a execução dos testes. Para isso, o modelo do comportamento do protocolo deve levar em consideração alguns tipos de falhas e o traço deve conter indicações da presença de falhas.

O objetivo inicial do trabalho era desenvolver uma ferramenta de análise de resultados para o ATIFS (capítulo 6), um ambiente de testes que provê suporte a todas as tarefas do processo de teste. Com base nisso, procurou-se responder as seguintes perguntas:

Como determinar se os testes produziram resultados corretos ?

Como gerar diagnósticos para os erros detectados durante a análise dos resultados?

Como levar em conta a presença de falhas durante a execução dos testes?

Como saber se os mecanismos de detecção de erros e tolerância a falhas[1] foram ativados na presença de falhas?

Para atingir os objetivos do trabalho foram investigadas as técnicas de análise de resultados de testes tanto para *softwares* de comunicação quanto para os demais tipos de *software*. Foi necessário compreender os testes por injeção de falhas (capítulo 5) e conhecer quais tipos de falhas seriam considerados nesse estudo. Escolhida uma técnica de análise de resultados, pensou-se na adaptação da técnica aos testes de protocolos em presença se falhas e partiu-se para a definição de uma ferramenta que executasse a tarefa de análise. Essa ferramenta tem como principais funções: produzir um veredicto de conformidade para cada teste[2] executado, gerar diagnósticos de erros e verificar a ativação dos mecanismos de

[1] Daqui em diante, o termo "mecanismo de detecção de erros e tolerância a falhas" será mencionado como "Mecanismo de Tolerância a Falhas" (MTF).

[2] Entenda-se por teste, um caso de teste.

tolerância a falhas implementados no *software* testado. Foi desenvolvida, então, a especificação da ferramenta, segundo a modelagem OMT [RBP+91]. A partir dessa especificação será implementada a ferramenta.

O conteúdo dessa dissertação está dividido em duas partes: a primeira parte procura dar ao leitor os conceitos básicos sobre a terminologia usada na área de tolerância a falhas (capítulo 2), sobre testes de *software* em geral (capítulo 3), sobre testes de protocolos de comunicação (capítulo 4) e testes por injeção de falhas (capítulo 5). Também é apresentado o ATIFS, um ambiente integrado de testes baseado em injeção de falhas por software (capítulo 6), onde o resultado desse trabalho será usado.

A segunda parte do trabalho desenvolve uma ferramenta para analisar resultados para testes de comportamento e testes por injeção de falhas por *software* aplicados à uma implementação de protocolo de comunicação. No capítulo 7 é apresentado o contexto da análise de resultados, descrevendo-se os componentes necessários à análise. No capítulo 8 são apresentados o algoritmo da análise de traço e os produtos dessa análise, juntamente com um exemplo de aplicação. Em seguida é apresentada uma modelagem da ferramenta de análise de resultados de testes para futura implementação (capítulo 9). Segue-se com o capítulo de conclusão sobre o trabalho (capítulo 10), as referências bibliográficas e o apêndice que descreve o manual de uso da ferramenta desenvolvida.

Parte I

Contexto do Trabalho

Essa parte inicial do trabalho procura introduzir conceitos necessários à compreensão do método de análise de resultados de testes desenvolvido na Parte II. Inicialmente são descritos conceitos básicos da área de tolerância a falhas. No capítulo 3 são descritos conceitos relativos aos testes de software de forma geral, onde o principal aspecto a ser observado é a análise de resultados dos testes.

No capítulo 4, introduz-se conceitos sobre a organização dos protocolos de comunicação em camadas, os testes voltados para protocolos, as arquiteturas mais comuns para tais testes e, por fim, o problema da análise de resultados em testes de protocolos.

O capítulo 5 descreve os principais aspectos dos testes por injeção de falhas, as formas de aplicação da injeção de falhas e o problema da análise de resultados para esse tipo de teste.

No capítulo 6 é descrito um ambiente de testes baseado em injeção de falhas por software (ATIFS). Esse ambiente está em desenvolvimento e visa prover suporte a todas as atividades do processo de teste. O método desenvolvido nesse trabalho deve ser colocado em prática como sendo um módulo do ATIFS.

2. Conceitos Básicos

O uso de sistemas computacionais em aplicações que envolvem riscos (econômicos e de vidas humanas) tem sido cada vez mais freqüente. Esses sistemas precisam ter a capacidade de manter a execução correta de seus programas e operações de E/S por um determinado período de tempo. Esses sistemas devem ser, então, tolerantes a falhas. Mais particularmente, entende-se por um sistema tolerante a falhas [LL87] aquele que, sem intervenção manual, tem a capacidade de tratar falhas físicas e falhas induzidas pelo homem e, mesmo assim, manter seu comportamento como definido em sua especificação.

Esse capítulo descreve brevemente os conceitos sobre: segurança no funcionamento; tolerância a falhas; a terminologia adotada sobre o que é falha, erro e defeito; verificação e validação no contexto de engenharia de *software* e de tolerância a falhas.

2.1 Segurança no Funcionamento e Tolerância a Falhas

A segurança no funcionamento de um sistema é a propriedade que permite a seus usuários ter confiança no serviço fornecido por esse sistema. O serviço fornecido por um sistema é o seu comportamento, tal como ele é visto por seu(s) usuário(s); um usuário é um outro sistema (humano ou físico) que interage com o sistema considerado [Lap95].

Um meio para se conseguir a segurança no funcionamento é o uso de mecanismos de tolerância a falhas. Esses mecanismos implementam os conceitos de tolerância a falhas (explicitamente baseados na redundância), tanto por *hardware* como por *software*. Dessa forma, sistemas tolerantes a falhas têm a capacidade de continuar a fornecer o serviço especificado mesmo na presença de falhas em alguns de seus componentes.

2.2 Falha, Erro e Defeito

Um **defeito** ocorre num sistema quando o serviço fornecido não está conforme à especificação, sendo esta especificação uma descrição da função ou do serviço esperado do sistema. Um **erro** (*error*) é a parte do estado do sistema que pode levar a um defeito (*failure*): um erro afetando

o serviço indica que um defeito ocorreu ou está para ocorrer. A causa direta de um erro é uma **falha** (*fault*).

Não existe ainda um consenso quanto à terminologia a ser usada em português; os termos usados aqui estão baseados em [LL87].

2.3 Verificação e Validação

No contexto de engenharia de *software*, Verificação e Validação (V&V) é um conjunto amplo de atividades que visa garantir, segundo [Pre92], que:

a) o *software* implemente corretamente uma função específica (verificação);

b) *o software* que foi construído esteja de acordo com as exigências do cliente (validação).

Os testes (capítulo 3) são atividades de V&V que desempenham um papel extremamente importante, porém muitas outras atividades também são necessárias.

No que diz respeito à validação de sistemas tolerantes a falhas, além das funcionalidades normais, deve-se levar em conta também os MTFs, acionados quando ocorrem falhas/erros no sistema. Nesse contexto, a atividade de validação é chamada de **validação da tolerância a falhas**, e visa garantir a confiança na capacidade do sistema em fornecer o serviço especificado mesmo em presença de falhas/erros. Essa validação engloba os seguintes aspectos [Lap95]:

a) **eliminação de falhas**: envolve a verificação e diagnóstico de erros, e a correção de falhas;

b) **previsão de falhas**: visa a execução de uma avaliação do comportamento do sistema no que se refere à ativação e ocorrência de falhas.

Um ponto crucial na validação de sistemas tolerantes a falhas diz respeito à validação dos seus MTFs. A importância da validação desses mecanismos se deve a duas razões principais:

1) A presença de falhas de projeto/implementação nesses mecanismos pode levar a deficiências no comportamento dos mesmos quando em presença de falhas para as

quais eles foram projetados para tratar; essas deficiências podem levar o sistema a não mais fornecer o serviço correto (conforme à especificação).

2) O efeito da eficiência dos mecanismos de tolerância a falhas sobre medidas, tal como a confiabilidade do sistema.

O uso de técnicas formais de verificação (como prova de programas), é altamente desejável para garantir a corretude do sistema, em particular, de seus MTFs. O uso de métodos analíticos (modelos de Markov, ...) também é necessário para a obtenção de medidas da eficiência desses mecanismos. No entanto, devido à complexidade dos sistemas tolerantes a falhas, sua aplicação é limitada a partes do sistema e à consideração de um número reduzido de falhas.

Estas técnicas devem então ser complementadas com a validação experimental. A **injeção de falhas** (capítulo 5) é uma técnica que vem sendo muito utilizada na validação de MTFs, pois permite validá-los em presença de entradas especiais, para as quais eles foram construídos para tratar: as falhas.

Os testes por injeção de falhas permitem que sejam cobertos os dois aspectos da validação mencionados anteriormente, onde se pode observar que:

a) na **eliminação de falhas**, o objetivo é reduzir ao máximo as falhas de projeto/implementação existentes nos MTFs;

b) na **previsão de falhas**, o objetivo é avaliar a eficiência dos MTFs, obtendo-se estimativas sobre eles.

3. Teste de Software

Dado que qualquer programa não está livre de erros [Pre92, Bei90], testa-se um programa[3] com a finalidade de encontrar a máxima quantidade de erros possível. Algumas regras estabelecidas em [Mye79] demonstram os objetivos dos testes:

1) A atividade de teste é o processo de executar um programa com a intenção de descobrir erros.

2) Um bom caso de teste é aquele que tem uma elevada probabilidade de detectar um erro ainda não descoberto.

3) Um teste bem sucedido é aquele que detecta um erro ainda não descoberto.

Além de detectar erros, os testes mostram se as funções de *software* estão trabalhando de acordo com a especificação, e se os requisitos de desempenho foram cumpridos. Mais ainda, os resultados dos testes podem ser usados para se obter a medida da confiabilidade e alguma indicação da qualidade do *software*. Os testes, no entanto, não podem mostrar que um programa está correto e não contém erros; só podem mostrar que erros estão presentes. Essa tarefa de demonstrar a ausência de erros é, normalmente, executada através de técnicas como provas de correção de programa [DMM+87], embora já existam técnicas de testes que provem a ausência de certos tipos de erros no *software*.

3.1 O Processo de Teste

O processo de teste é composto de atividades complexas que, geralmente, tomam grande parte do tempo total do desenvolvimento de um *software*. A Figura 1, adaptada de [Pre92], ilustra esse processo. Duas classes de entrada principais são fornecidas ao processo de teste: (1) a **Configuração do Software**, que inclui a especificação de requisitos do *software*, a

[3] Os termos "programa" e "software" usados durante o texto referem-se à uma implementação.

especificação de projeto e o código-fonte; (2) a **Configuração dos Testes**, que inclui um plano e um procedimento de teste e quaisquer ferramentas de testes que venham a ser usadas. O **Modelo do Sistema** pode ser usado para gerar casos de testes. O **critério** para geração dos testes determina qual a quantidade suficiente de testes a ser aplicada.

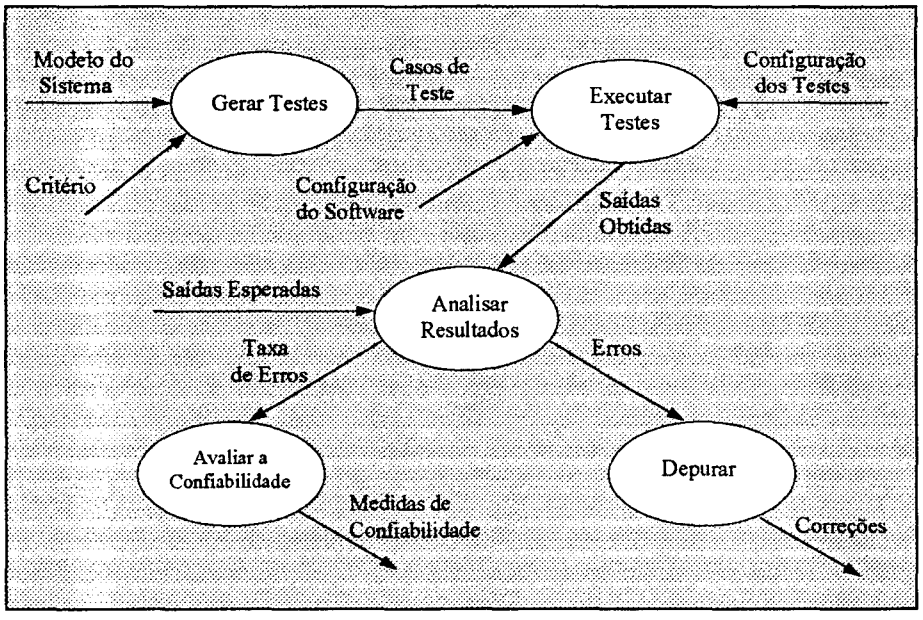

Figura 1: Processo de teste

De acordo com [Pre92], os testes são executados e as saídas obtidas são comparadas às saídas esperadas. Quando uma discrepância é encontrada, um erro é detectado e a depuração é iniciada. Se erros que requerem modificação de projeto são encontrados com uma certa regularidade, a qualidade e a confiabilidade do *software* são suspeitas, e é aconselhável aplicarem-se novos testes. Por outro lado, se as funções de *software* parecem estar trabalhando corretamente e os erros encontrados são facilmente corrigidos, duas conclusões podem ser consideradas: (1) a qualidade e a confiabilidade do *software* são aceitáveis; (2) os testes são inadequados para descobrir erros mais graves.

Se os testes não detectarem erros, os casos de testes podem não ter sido pensados o suficiente e, provavelmente, erros ficaram escondidos no *software*. Estes erros, eventualmente, serão descobertos pelos usuários e corrigidos durante a fase de manutenção, quando o custo por erro corrigido é maior do que se fosse feito durante a fase de desenvolvimento.

Os resultados acumulados durante os testes podem também ser analisados através de modelos de confiabilidade [Pre92, DMM+87], que usam dados das taxas de erros detectados para predizer a tendência futura da ocorrência de erros.

Quanto à geração de casos de testes, duas estratégias podem ser adotadas [Bei90, Mye79]: (1) **caixa preta**, onde o testador[4] vê o programa como uma caixa preta e não está preocupado com sua estrutura interna, interessado-se em encontrar circunstâncias nas quais o programa não se comporta conforme sua especificação; (2) **caixa branca**, onde o testador examina a estrutura lógica do programa e deriva os casos de testes a partir dessa estrutura.

Conforme a complexidade do *software* testado aumenta, maior é o número de testes necessários. Por isso, é bastante desejável a automatização do processo de teste, já que ele é responsável por uma porção significativa dos esforços e dos custos de um projeto de *software*. Essa automatização é uma tarefa bastante trabalhosa, visto que cada uma das tarefas mostradas na Figura 1 são bastante complexas.

3.2 Análise de Resultados

Uma dificuldade comum aos testes de *software* é o **problema do oráculo** [Wey82]: como determinar se os resultados dos testes estão corretos? O oráculo é um mecanismo que analisa os resultados dos testes com base numa referência para o comportamento do *software*. Essa referência precisa ser a mais confiável possível. A saída produzida por um oráculo é uma sentença, denominada veredicto, para cada caso de teste. Um veredicto informa se um teste produziu saídas corretas, conformes à referência para o comportamento do *software*.

Existem diversas propostas para resolver esse problema, entretanto não há ainda uma solução genérica. Quando existe uma especificação que descreva o comportamento do *software*, os resultados dos testes podem ser verificados com relação a ela. Entretanto, quando

[4] Um testador, nesse caso, é a pessoa que conduz os testes.

não há uma especificação para o *software*, algumas técnicas informais são propostas para contornar o problema [Wey82], tais como:

a) **Simplificação da entrada:** simplifica e reduz o número de entradas para que o volume de saídas seja pequeno e possa ser verificado manualmente; essa técnica não é aconselhada porque muitas combinações de entradas deixam de ser testadas e, com isso, muitos erros podem deixar de ser detectados.

b) **Pseudo-oráculo**: duas ou mais versões independentes de um mesmo programa, geradas a partir dos mesmos requisitos, por equipes distintas, recebem as mesmas entradas e suas saídas são comparadas; se todas as versões produzirem as mesmas saídas, são consideradas corretas; caso contrário, todas devem ser investigadas. Se, entretanto, os requisitos da especificação do sistema estiverem incorretos, mesmo que todas as versões produzam as mesmas saídas, essas saídas podem estar incorretas, e o veredicto poderá ser falso . Essa técnica é também conhecida como *back-to-back testing* ou teste por comparação [Pre92].

c) **Oráculo parcial**: significa estabelecer faixas plausíveis para os resultados dos testes, isto é, com alguma experiência sobre o funcionamento do sistema, alguns usuários *experts* podem assegurar que algumas faixas de valores de saída estão incorretas, mesmo sem conhecer a resposta exata. Por exemplo, o valor do faturamento de uma empresa multinacional jamais poderia ser R$ 50,00 no ano. Já, não se pode dizer a mesma coisa do valor R$ 5.900.000,00.

d) **Comparação com similares**: consiste em testar um programa em um conjunto de "problemas padrões" que já foram usados para testar e comparar programas destinados a realizar a mesma tarefa. Isso evita o *overhead* de produção de versões independentes somente para comparar suas saídas.

e) **Uso de diferentes máquinas**: esse método é usado, principalmente, para aplicações científicas e matemáticas. Propõe-se executar um programa em máquinas com diferentes precisões numéricas e comparar os resultados dessas execuções.

Pode-se observar que nos testes onde as técnicas listadas acima são usadas, não há preocupação em se exercitar todas as partes do *software*. A preocupação maior se refere às

partes mais usadas do *software* durante a fase operacional. Apesar disso, algumas dessas técnicas podem ser combinadas, o que aumenta a probabilidade de se encontrarem erros. Um fato a observar é que, na ausência de uma especificação para o comportamento do *software*, a análise de resultados é pouco precisa e, geralmente, manual, podendo levar o testador a falsos veredictos.

Quando existe uma especificação que inclui um modelo do comportamento do *software*, a automatização das tarefas de teste se torna mais factível. Os testes podem ser gerados com base nesse modelo e a análise de resultados pode tomá-lo como referência para verificar se as saídas obtidas durante a execução dos testes estão corretas. A automatização da análise de resultados é bastante desejável, visto que o volume de saídas dos testes é, geralmente, muito grande. A análise manual é demorada e propensa a erros. Por isso, algumas formas de análise automatizada são propostas:

a) **Análise embutida** nos casos de testes [Bei90, Ric94]: cada caso de teste carrega tanto as entradas a serem aplicadas quanto as saídas esperadas para cada uma das entradas. Durante a execução dos testes, a saídas observadas são comparadas às saídas esperadas, e assim são gerados veredictos para cada interação de teste. Nesse caso, a existência do modelo do comportamento é útil na geração dos testes para a obtenção das saídas esperadas, mas desnecessário durante a execução dos mesmos.

b) **Análise separada** dos casos de testes: significa que a obtenção das saídas esperadas é feita separadamente da geração dos testes. As entradas dos casos de testes não vêm acompanhadas das saídas esperadas. As saídas obtidas podem ser verificadas durante ou depois da execução dos testes tomando-se como referência um modelo do comportamento do *software* testado.

A análise embutida é mais simples de ser implementada pois consiste na comparação das saídas obtidas com as saídas esperadas. Entretanto, a previsão das saídas esperadas com antecedência pode ser uma tarefa bastante complexa e demorada, e nem sempre é possível fazê-la com exatidão, especialmente nos testes por injeção de falhas (capítulo 5), pois falhas podem ocorrer durante a execução dos testes, alterando o comportamento normal do *software* testado.

A análise separada é mais complexa de ser implementada mas usa um modelo como referência para o comportamento do *software*. É necessário um analisador que usa as entradas dos casos de testes e o modelo de referência para determinar se as saídas observadas durante a execução dos testes estão corretas. Duas formas de análise podem ser feitas nesse caso:

1) **Análise *on-line***: pressupõe-se que existe uma representação executável do modelo do comportamento do *software*, a qual é executada em paralelo com a implementação do *software* em teste. Essa forma é baseada nos conceitos de **observador** e **executante**, introduzidos em [AAD79]. O **observador** é o modelo em forma executável e o **executante** é a implementação sob teste (IUT).

2) **Análise *off-line***: analisa-se o traço de execução, também chamado de histórico de execução [JB83], como é denominada a seqüência de eventos observados durante os testes. Esta forma de análise é particularmente eficaz no caso de testes estatísticos [Mar95], onde o número de testes aplicados é, geralmente, muito grande, e não se sabe com antecedência, quais casos de testes serão aplicados.

No próximos capítulos serão apresentados os principais aspectos de testes de protocolos de comunicação e de testes por injeção de falhas. O problema do oráculo voltará a ser abordado para cada um deles.

4. Validação de Protocolos de Comunicação

Um dos mais importantes princípios da área de redes de computadores é estruturar a rede como um conjunto de camadas hierárquicas [Tan89, Hol91], cada uma sendo construída utilizando as funções e serviços oferecidos pelas camadas inferiores.

Cada camada (ou nível) deve ser pensada como um programa ou processo, implementado por *hardware* ou por *software*, que se comunica com o processo correspondente em outra máquina. As regras que governam a conversação de um nível N qualquer são chamadas de **protocolo de nível N**.

Dados transferidos em uma comunicação de nível específico "descem" verticalmente através de cada nível adjacente da máquina transmissora até o nível 1 (que é o nível físico onde, na realidade, há a única comunicação horizontal entre máquinas), para depois "subir" verticalmente através de cada nível adjacente da máquina receptora até o nível de destino. Os limites entre cada nível adjacente são chamados *interfaces*.

Cada nível (N) da rede oferece um conjunto de serviços ao nível superior (N+1), usando funções realizadas no próprio nível e serviços disponíveis no nível inferior (N-1). Um serviço representa um conjunto de funções oferecidas a um usuário por um fornecedor. O serviço oferecido por um fornecedor é acessado por um usuário através de um **ponto de acesso ao serviço** (SAP). A Figura 2 mostra um serviço de comunicação do ponto de vista de dois usuários. Os usuários interagem com o serviço de comunicação através de interações chamadas **primitivas de serviço**, trocadas através dos SAPs. As primitivas de serviço são os serviços fornecidos por uma camada à outra.

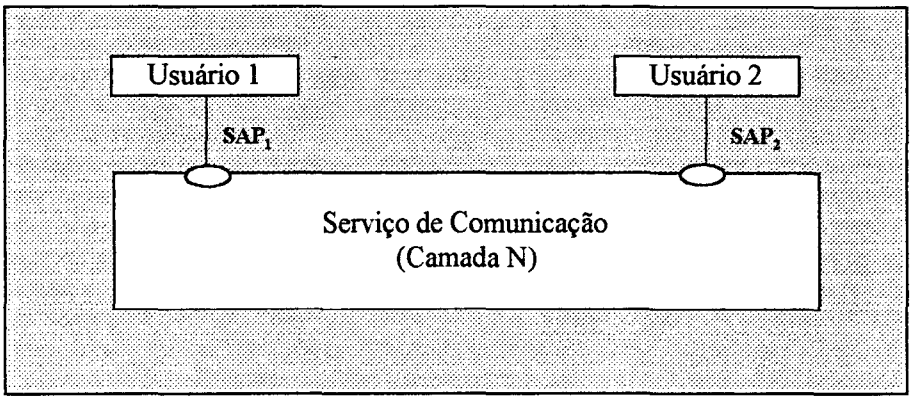

Figura 2:serviço de comunicação do ponto de vista de dois usuários

Os elementos ativos (implementações de protocolos) das camadas são denominados **entidades**. Uma entidade pode ser uma entidade de *software* (um processo) ou *hardware* (uma placa de interface de rede). Entidades da mesma camada em máquinas diferentes são denominadas **entidades pares** ou **parceiras.** A Figura 3 traz uma visão mais detalhada do serviço de comunicação, mostrando duas **entidades pares de protocolo**, que se comunicam através de um serviço da camada inferior (N-1). A definição dos requisitos de comportamento exigidos para uma entidade de protocolo é chamada **especificação do protocolo** [BP94], e engloba um modelo com as interações nos **pontos de acesso superior**[5] (USAP) e **inferior**[6] (LSAP), além de identificar, geralmente, vários tipos de **unidades de dados de protocolos** (PDU, ou mensagens) que são trocadas entre as entidades de protocolo através do serviço da camada inferior (N-1).

Quanto à especificação do protocolo, é importante o uso de uma especificação formal. Descrições narrativas da especificação em linguagem natural, apesar de serem fáceis de se compreender, geralmente contêm ambigüidades e dificultam a verificação de quão completo e correto é o protocolo testado. Nesse contexto, muitos modelos têm sido propostos para especificar protocolos, incluindo máquinas finitas de estado (FSM), redes de Petri, gramáticas formais, linguagens de programação de alto nível, álgebra de processos, tipos abstratos de

[5] Ponto de acesso onde ocorre troca de interações entre a camada N e a camada superior a ela (N+1)

[6] Ponto de acesso onde ocorre troca de interações entre a camada N e a camada inferior a ela (N-1)

dados e lógica temporal. Os mais simples, como FSM, redes de Petri e gramáticas formais, foram freqüentemente estendidos para manipular propriedades específicas de protocolos.

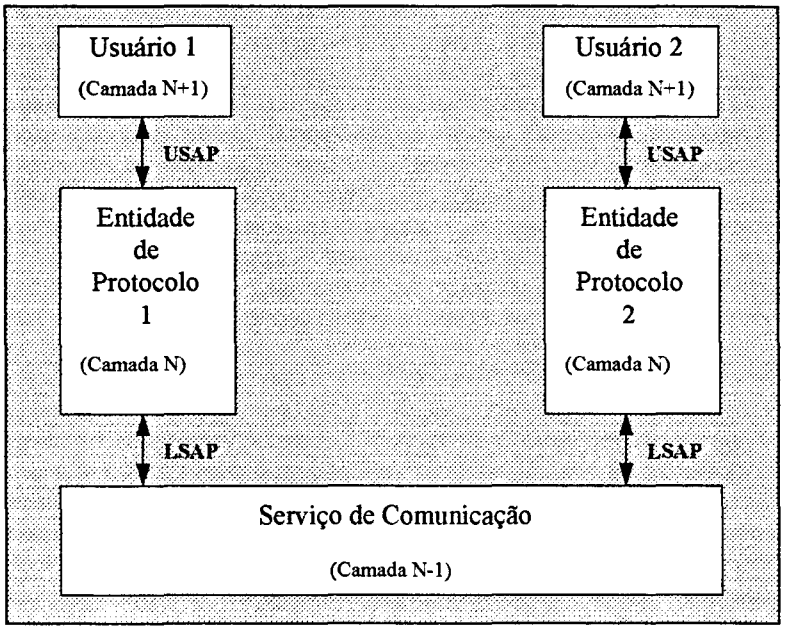

Figura 3: Visão mais detalhada do serviço de comunicação da Figura 2

Com o trabalho de padronização na interconexão de sistemas abertos (OSI/ISO), grupos de trabalho em técnicas de descrição formal (FDT) propuseram três linguagens para especificar protocolos, descritas brevemente em [BP94]: (1) Estelle: baseada em máquinas finitas de estado estendidas (EFSM), onde as extensões se referem a parâmetros de interação, variáveis adicionais de estado, envolvendo definição de tipos, expressões e declarações na linguagem Pascal; (2) LOTOS: baseada em cálculo algébrico para sistemas de comunicação, usa o conceito de FSM e de processos paralelos que se comunicam através de mecanismo de *rendezvous* entre dois ou mais processos; (3) SDL: também baseada no modelo de EFSM, usa o conceito de tipos de dados abstratos com adição de uma notação de variáveis de programa e estruturas de dados, similar à notação usada em Estelle, porém essa notação é baseada em CHILL, uma linguagem de programação recomendada pelo CCITT.

4.1 Testes de Protocolos

A implementação de um protocolo é um *software*. Assim, a validação de um protocolo cobre as seguintes atividades [Saj84]: verificação (seção 2.3) em cada estágio do desenvolvimento do protocolo, predição de desempenho, teste da implementação e avaliação de desempenho.

Uma implementação de protocolo é testada de forma a assegurar compatibilidade com outras implementações do mesmo protocolo. Essa tarefa é chamada de **teste de conformidade do protocolo** [BP94]. A ISO recomenda, para efeito de certificação de padrões de protocolos, que os testes de protocolos sejam do tipo caixa preta, pois podem ser feitos por terceiros, sem necessidade de acesso ao código-fonte[7]. Isso implica que uma especificação do protocolo deve ser fornecida, a qual é a base para a derivação dos casos de testes e análise de resultados dos testes [BP94].

O padrão da ISO IS9646-1 recomenda quatro tipos de testes a serem aplicados na implementações de protocolo [IS9646, Ray87]:

1) **testes de conformidade**: quatro tipos de testes distintos são propostos para a indicação de conformidade:

    a) **testes de interconexão básica**: detectam causas graves de não conformidade, como, por exemplo, quando uma IUT[8] não é capaz de se conectar com outra IUT ou não têm as principais características de seu padrão implementadas;

    b) **testes de capacitação**: verifica se as aptidões observáveis de uma IUT estão de acordo com os requisitos de conformidade estática do protocolo;

    c) **testes de comportamento**: provê testes mais práticos e completos possíveis de forma a cobrir a maior faixa possível de requisitos de conformidade

[7] Um protocolo, como qualquer outro software, também pode ser testado de forma estrutural, isto é, através de testes caixa branca.

[8] *Implementation Under Test*

dinâmica especificados pelo padrão do protocolo, dentro das aptidões da IUT;

d) **testes de resolução de conformidade**: provê diagnósticos mais definitivos possíveis para resolver se uma implementação satisfaz requisitos específicos.

2) **testes de interoperabilidade**: verificam se duas IUTs conseguem se interagir;

3) **testes de desempenho**: medem as características de desempenho de uma IUT, tal como *throughput* e tempo de resposta sob várias condições;

4) **testes de robustez**: verificam a reação da IUT sob várias condições de erro, inclusive as situações específicas da IUT mas não previstas na norma.

Para certos padrões de protocolo, os comitês de padronização têm definido seqüências padronizadas de testes de conformidade, o que corresponde a um grande número de casos de testes. A execução desses casos de testes provê uma segurança razoável de que as implementações testadas seguem todas as regras definidas pelo padrão do protocolo. Funções específicas fora do padrão do protocolo não são cobertas pelos casos de testes padronizados.

Sobre as seqüências de testes citadas acima, elas são definidas [IS9646, Sar88, Ray87] como sendo um conjunto de grupos de teste. Um grupo de teste é formado por um conjunto de casos de testes relacionados. Cada grupo tem um objetivo específico, por exemplo, grupo que testa estabelecimento de conexão, transferência de dados, etc. Um caso de teste é composto por um conjunto de eventos de teste (interações de entrada a serem fornecidas à IUT).

Em algumas situações, entretanto, os casos de testes padronizados não são suficientes. Algumas das razões são discutidas a seguir [BB89]:

a) Durante testes de conformidade de uma implementação, é desejável executar testes que não foram previstos pelo implementador. Para isso, novos casos de testes precisam ser selecionados.

b) Enquanto casos de testes padronizados OSI são definidos para verificar a conformidade de uma implementação com relação à especificação do protocolo, cada implementação geralmente tem de satisfazer requisitos adicionais que são

específicos do sistema em que devem funcionar. Por esta razão, casos de testes específicos para verificar esses requisitos precisam ser projetados e executados.

c) Em testes com árbitro [Bel89, BDZ89], por exemplo, os casos de testes padronizados OSI não podem ser usados. Os testes de arbitragem são executados quando duas implementações, já bem testadas, não se interoperam corretamente. Para determinar a razão para o problema, os testes de arbitragem são executados numa configuração que inclui um módulo que observa as interações entre duas IUTs que se comunicam. Esse módulo inclui uma função que decide quais implementações se comportam conforme a especificação do protocolo.

d) Casos de testes padronizados não são, geralmente, aplicados na depuração de uma nova implementação. Nas primeiras fases de implementação é freqüentemente desejável que sejam executados casos de testes *ad hoc* e verificar o comportamento da implementação nessas situações.

Além da padronização de seqüências de testes, os comitês de padronização da ISO também estabeleceram arquiteturas de testes, onde os testes padronizados (e também os testes específicos de uma implementação de protocolo) podem ser executados. Essas arquiteturas são abordadas na próxima seção.

4.2 Arquiteturas de Testes

Como foi mostrado anteriormente na Figura 3, uma entidade típica de protocolo tem dois SAPs. A existência de tais pontos é uma característica que distingue uma implementação de protocolo dos demais *softwares*. Surgiram, então, algumas arquiteturas de testes específicas para tal categoria de *software*.

O padrão internacional descrito em [IS9646] define quatro tipos básicos de arquiteturas[9] para realizar testes de conformidade em protocolos: local, distribuída, coordenada e remota.

[9] As referências [Bel89, BDZ89, BP94] também apresentam as arquiteturas de testes padronizadas. Em [Ray87] é descrito um resumo do padrão internacional IS9646, da mesma forma que em [Lin90].

A arquitetura de testes local é apresentada na Figura 4 e foi adaptada de [Lin90]. Nela, pode-se notar que os pontos de acesso a serviço inferior e superior (LSAP e USAP) da IUT são acessados, respectivamente, pelo **testador inferior**[10] (LT) e **testador superior**[11] (UT). Os testadores[12] são sincronizados por um módulo de coordenação de testes. Os testadores, o módulo de coordenação de testes e a IUT residem no mesmo computador. A IUT troca primitivas de serviço (ASP) com ambos os testadores através de **Pontos de Controle e Observação** (PCO). Um PCO é um SAP usado por um testador com o propósito de estímulo e observação da IUT durante a execução dos testes.

A arquitetura de testes local equivale ao teste de *software* tradicional, não sendo específica de teste de protocolos.

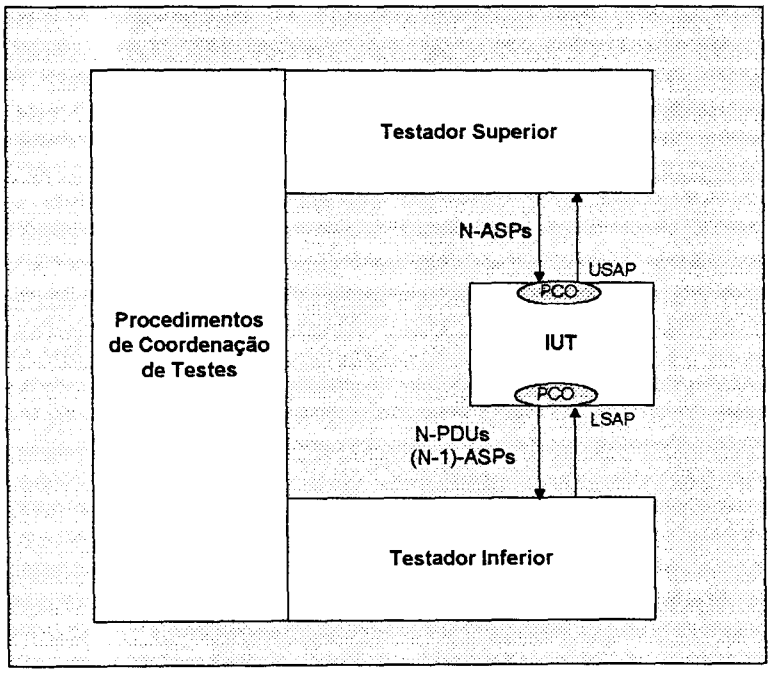

Figura 4: Arquitetura de testes local

[10] Componente que controla a liberação de entradas apenas para o ponto de acesso inferior da IUT

[11] Componente que controla a liberação de entradas apenas para o ponto de acesso superior da IUT

[12] Entenda-se por testador, nesse caso, um programa que está localizado no sistema de teste e que controla a liberação de entradas para estimular a IUT. O mesmo termo é usado, às vezes, para designar a pessoa que está realizando os testes.

As outras arquiteturas (distribuída, coordenada e remota) são chamadas de **externas** e assumem que o SAP inferior (LSAP) é acessado pelo testador inferior através do serviço de comunicação da camada N-1. O testador inferior reside no sistema de teste (TS), num computador distinto de onde reside a IUT, chamado de **sistema em teste** (SUT). A Figura 5 mostra a estrutura de uma arquitetura distribuída e ilustra as características descritas acima.

As arquiteturas distribuída e coordenada requerem um testador superior e um testador inferior. O testador superior troca interações com a IUT através do SAP superior (USAP) da IUT. O testador inferior troca interações com a IUT através do SAP inferior (LSAP) da IUT. A IUT e o testador superior residem no mesmo computador (SUT). O testador inferior reside num computador remoto (TS). O SUT é conectado ao TS através do serviço de comunicação da camada inferior à IUT (camada N-1).

A diferença entre a arquitetura distribuída e a coordenada é que a arquitetura coordenada conta com um protocolo que suporta coordenação e sincronização de ações de testes de ambos os testadores (UT e LT) à distância.

A arquitetura de teste remota, diferente das duas outras arquiteturas externas (distribuída e coordenada), exclui o SAP superior do processo de teste porque, em algumas situações práticas, a IUT está embutida num sistema complexo, de forma que o testador superior não pode ser incluído dentro do sistema em teste.

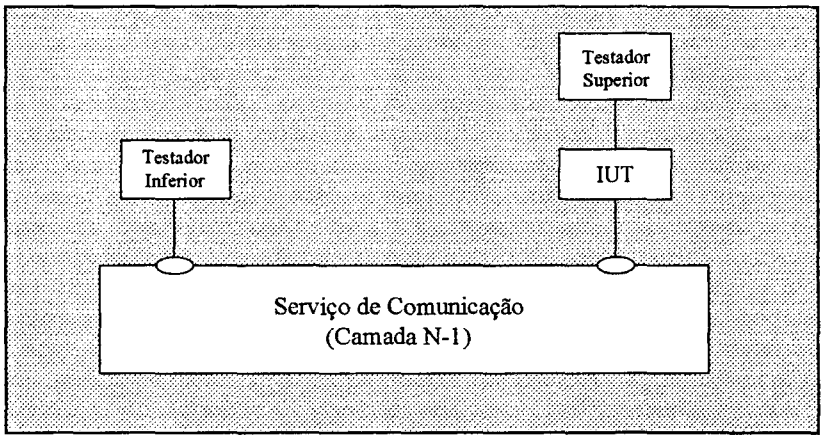

Figura 5: Arquitetura de testes distribuída

Essa variedade de arquiteturas é adequada para diferentes situações de teste e cada arquitetura tem um conjunto de falhas detectáveis que pode ser explorado. Contudo, cada

arquitetura tem um impacto diferente sobre a testabilidade: controlabilidade e observabilidade da IUT.

Um dos aspectos que deve ser observado em cada uma das arquiteturas apresentadas é o poder de detecção de erros que cada uma oferece. A arquitetura de testes local têm maior poder de detecção de erros que as arquiteturas externas. Entretanto, sua aplicação em testes de conformidade de protocolos é, às vezes, difícil [DB85], visto que ela requer acesso direto à *interface* através da qual a IUT se comunica com o serviço de comunicação da camada inferior (N-1). Dessa forma, a maioria das ferramentas de testes precisam residir no mesmo computador que a IUT. As arquiteturas de testes externas evitam essas dificuldades, tendo a maior parte do sistema de teste num computador remoto. Isso permite que o sistema de teste possa ser acessado à distância por vários SUTs[13].

Apesar das arquiteturas externas permitirem que as ferramentas de testes residam em computador remoto, o UT[14] deve residir no SUT. Isso dificulta a sincronização entre o LT, que está no sistema de teste, e o UT, que está no SUT [ZDH88].

O conceito de ***ferry***, proposto por Zeng [ZR85], sugere que o UT no sistema em teste seja substituído por um simples *ferry*[15] e que o UT original seja deslocado para o sistema de teste, no mesmo computador onde reside o LT. Os dados de testes são transferidos por um canal geralmente independente da IUT, denominado **canal *ferry***. Como resultados, a sincronização entre o UT e o LT é assegurada sem dificuldades porque eles estão no mesmo computador [ZDH88]. Além disso, a facilidade de levar os testes de um SUT para outro foi bastante melhorada [ZDH88].

A Figura 6, retirada de [CLPZ90], mostra a arquitetura ***ferry clip***, ilustrando o conceito *ferry*, que deu origem a essa arquitetura. A principal idéia desse conceito é transportar dados de testes transparentemente do SUT para o TS[16], permitindo que, tanto o UT, quanto o LT residam no TS. Um *passive ferry* firma-se à IUT como um *clip* (grampo) e recebe os dados de testes enviados pelo *active ferry*, localizado no TS. O módulo *Test Manager*, que contém o UT

[13] *System Under Test*

[14] *Upper Tester*

[15] a idéia do nome vem de "*ferry boat*" = "barco de travessia"

[16] *Test System*

e o LT, comunica-se com o *Active Ferry* através do módulo *Encoder/Decoder*. A função deste último é codificar os dados de testes enviados pelo *Test Manager* num formato conhecido pela *interface* da IUT. Da mesma forma, os dados recebidos pelo *Active Ferry* (vindos da IUT) são enviados ao *Test Manager* através do *Encoder/Decoder*, que os converte num formato que o *Test Manager* compreende.

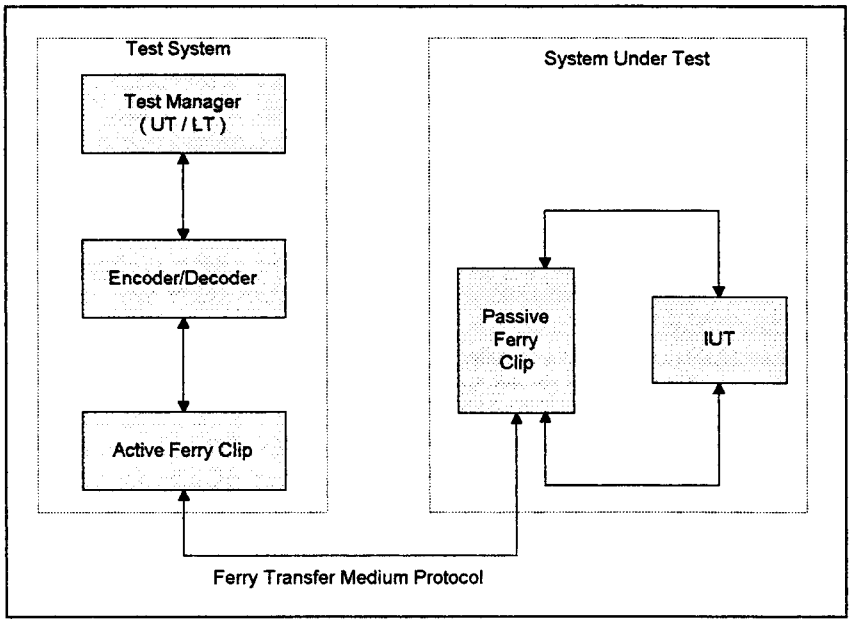

Figura 6: Arquitetura Ferry Clip

4.3 Análise de Resultados

Na seção 3.2 foram apresentadas as principais técnicas de análise de resultados de testes. Duas daquelas técnicas, descritas aqui em maiores detalhes, surgiram em estudos realizados com testes de protocolos:

a) **Análise *on-line*** [AAD79, MDA85, AALC91, DJC94]: esse tipo de análise é baseado no princípio da redundância e introduz o conceito de **observador-executante**: o **executante** é uma implementação clássica do sistema e o **observador** é uma representação executável do modelo do comportamento do sistema, cujas saídas são comparáveis às saídas do executante. Ambos são

implementados a partir da mesma especificação. O observador é executado em paralelo com o executante, ambos recebendo as mesmas entradas de testes, sendo que suas saídas são comparadas continuamente em tempo de execução, possibilitando detecção *on-line* de erros. Para cada erro detectado dessa forma, o executante é considerado incorreto porque o comportamento do observador, dirigido por um modelo formal, é suposto estar sempre em conformidade com a especificação do sistema. Conseqüentemente, a escolha do modelo implementado pelo observador é crucial. Para garantir que o comportamento do observador esteja correto, este modelo precisa ser validado o máximo possível. Em [MDA85, DJC94] o modelo implementado pelo observador é representado por Redes de Petri pelas seguintes razões: (1) são adequadas para modelar processos seqüenciais que se comunicam; (2) podem ser analisadas por métodos automáticos e (3) são executáveis por simuladores simples e fáceis de implementar. Na arquitetura em camadas de um sistema distribuído, o observador precisa de um modelo para cada camada. Cada entidade distinta de protocolo é representada por uma Rede de Petri deduzida da especificação do protocolo, geralmente descrita por FSMs.

b) **análise *off-line*** [BDZ89, Bel89, DB85, JB83]: durante a execução dos testes, as entradas dos casos de testes e as saídas produzidas pelo sistema são observadas e armazenadas num traço de execução. Após a execução dos testes, o traço é lido por um analisador que deverá informar se a seqüência de interações observadas está correta ou não, usando como referência um modelo do comportamento do sistema. Em [BDZ89, Bel89], o modelo do comportamento é descrito em LOTOS, uma linguagem definida pelo padrão 8807 da ISO. Os demais trabalhos estudados usam diagramas de transição de estados.

Um traço é um conjunto ordenado de eventos observáveis. Há dois tipos de traço no contexto de teste de protocolos de comunicação: (1) traço global: é o traço observado em todos os pontos de interação da **arquitetura de teste** com a IUT; (2) traço local: é o traço obtido de um subconjunto de pontos de interação observáveis. A projeção [BDZ89] é uma operação que pode ser aplicada ao traço global para reduzi-lo a um traço local corresponde a um subconjunto de pontos de interação.

As técnicas apresentadas acima são as principais técnicas para análise de resultados de protocolos. Ambas são realizadas separadamente da geração dos testes e têm como requisito a disponibilidade do modelo do comportamento do protocolo.

No próximo capítulo serão apresentados os testes por injeção de falhas, um meio para validar sistemas tolerantes a falhas. A idéia de se estudar técnicas de testes de protocolos e também de injeção de falhas tem como objetivo a realização de testes de protocolos tolerantes a falhas.

5. Testes por Injeção de Falhas

Uma das dificuldades no desenvolvimento de sistemas tolerantes a falhas diz respeito à sua validação. Essa dificuldade vem do fato de que os testes destes sistemas requerem a consideração de situações anormais, que ativem seus mecanismos de tolerância a falhas.

Uma técnica que vem sendo muito utilizada para este fim é a injeção de falhas [Mar95], que consiste em introduzir falhas/erros em um sistema, de maneira controlada, e observar o seu comportamento. A injeção de falhas tem sido reconhecida como um método para validar a tolerância a falhas e abrange ambos os objetivos desse processo de **validação**: **eliminação** e **previsão** de falhas (seção 2.3). Genericamente falando, a injeção de falhas é uma técnica que tem por objetivo descobrir deficiências da tolerância a falhas.

5.1 Formas de Aplicação

As formas de injeção mais empregadas são a simulação de falhas, a injeção física de falhas e a injeção de falhas por *software*, apresentadas a seguir:

a) **Simulação de falhas**: falhas lógicas são introduzidas num modelo do sistema. Esse modelo pode representar o comportamento ou a estrutura (em termos de portas e/ou transistores) do sistema. As falhas são injetadas no modelo com o intuito de analisar o processo de ativação de falhas e propagação de erros, a fim de avaliar o comportamento dos MTFs.

b) **Injeção física de falhas**: falhas físicas são aplicadas a um protótipo do *hardware* (e/ou do *software*) do sistema. Essa forma de injeção de falhas visa o estudo do comportamento de MTFs implementados por *hardware*, bem como dos *softwares* tolerantes a falhas.

c) **Injeção de falhas por software**: falhas lógicas são introduzidas em um protótipo do *software* (e/ou do *hardware*) do sistema, com o objetivo de emular a conseqüência de falhas físicas. Essa técnica não necessita de equipamento especial de *hardware*,

como é o caso da injeção física de falhas. Além disso, a controlabilidade e observabilidade do sistema durante os testes é a mesma da simulação de falhas, mas sem apresentar o tempo elevado de processamento desta última. É uma técnica adequada para validar MTFs implementados por *software*, por ter a capacidade de injetar condições de erro específicas que permitam ativar esses mecanismos, o que não pode ser garantido na injeção física de falhas.

Independente da forma de aplicação, a injeção de falhas também requer que seus resultados sejam analisados a fim de se obter um veredicto sobre o comportamento dos MTFs. Na próxima seção, é abordado o problema do oráculo para os testes por injeção de falhas.

5.2 Análise de Resultados

De acordo com [Mar95], a solução adotada para análise de resultados de testes na maioria dos estudos relacionados à injeção de falhas é a comparação dos resultados da execução do sistema sem falhas e a execução do sistema com falhas injetadas, usando a idéia da técnica conhecida como *back-to-back testing* [Pre92]. Esse é um tipo de oráculo, descrito no item 3.2 como **pseudo-oráculo**. O problema envolvido nessa técnica é que a execução do sistema sem falhas é considerada correta, mas se ela não está funcionando de acordo com sua especificação, deixa de ser uma referência confiável para o comportamento da IUT.

É importante a existência de um oráculo e mais importante ainda é que ele seja confiável. A partir do capítulo 7 desenvolve-se uma técnica de análise de resultados de testes por em injeção de falhas, a qual busca um oráculo mais confiável que os testes por comparação mencionados acima.

6. ATIFS

ATIFS [Mar95] é um Ambiente integrado de Testes baseado em Injeção de Falhas por Software, que provê suporte para o desenvolvimento e execução de testes, bem como para a análise dos resultados obtidos nos testes, objeto de estudo desse trabalho. Uma importante característica do ATIFS é o uso da especificação do sistema a ser testado [17], a qual pode servir de base tanto para a geração automatizada dos testes, quanto para referência a ser usada na análise dos resultados.

Um outro aspecto importante do ATIFS, e que o diferencia da maioria dos ambientes de testes por injeção de falhas, é que ele fornece suporte tanto para a verificação do comportamento do sistema em presença de falhas, quanto para a avaliação de medidas da qualidade do sistema, como a confiabilidade, por exemplo. Permite, também, ao usuário a definição de testes estatísticos, possibilitando a obtenção de estimativas das medidas procuradas e do erro cometido nessas estimativas.

6.1 Características do Ambiente de Testes

O estudo descrito em [Mar95] apresenta um ambiente para testes que visa cobrir os dois aspectos da validação de sistemas tolerantes a falhas: a verificação, que corresponde à eliminação de falhas, e a avaliação, que corresponde à previsão de falhas. O ATIFS é composto de várias ferramentas ou subsistemas, que dão suporte às diferentes atividades de testes; estas ferramentas devem funcionar de forma integrada; portanto, a arquitetura do ATIFS contém também os componentes que devem dar suporte a esta integração.

As funcionalidades a serem oferecidas por esse ambiente visam cobrir todas as atividades do processo de teste, que são as seguintes:

a) desenvolvimento dos testes;

[17] Os testes no ATIFS referem-se a implementações

b) especificação dos testes;

c) execução dos testes;

d) análise dos resultados.

Essas atividades constituem os principais subsistemas do ATIFS, como mostra a Figura 7. Além das funcionalidades mencionadas, o desenvolvimento do ambiente deverá levar em conta algumas metas como:

- **Independência da implementação sob teste**: é necessário um investimento elevado tanto em tempo quanto em recursos humanos e materiais para se desenvolver tal tipo de ambiente. Por isso é interessante que o mesmo seja o mais independente possível da implementação sob teste, de forma a ser aplicável ao teste de diferentes sistemas.
- **Facilidade de adaptação a diferentes configurações de teste**: o ambiente deve permitir a realização de diferentes tipos de testes. Assim sendo, ele deve ser facilmente adaptável a diferentes configurações de teste, requerendo o mínimo de mudanças na sua estrutura.
- **Facilidade de adaptação a diferentes sistemas**: os sistemas abertos permitem a interconexão de sistemas heterogêneos (equipamentos de diversos portes e características, sistemas operacionais variados). Portanto, o ambiente deve ser utilizável em diferentes sistemas de comunicação, rodando sob diferentes sistemas operacionais, requerendo para tanto um mínimo de alterações.
- **Convivialidade**: o ambiente deve oferecer uma interface amigável ao usuário, permitindo a sua intervenção nas diferentes fases de teste.
- **Facilidade de extensão**: o ambiente deve ser expansível, isto é, o acréscimo de novas funcionalidades não deve requerer mudanças estruturais profundas.

O ATIFS, inicialmente, deve testar sistemas especificados através de máquinas finitas de estados estendidas (EFSM), por estas serem comuns na especificação de sistemas de comunicação, principalmente protocolos. Entretanto, devido à facilidade de extensão, novos

tipos de especificação poderão ser usados, requerendo um mínimo de mudanças na estrutura do ambiente.

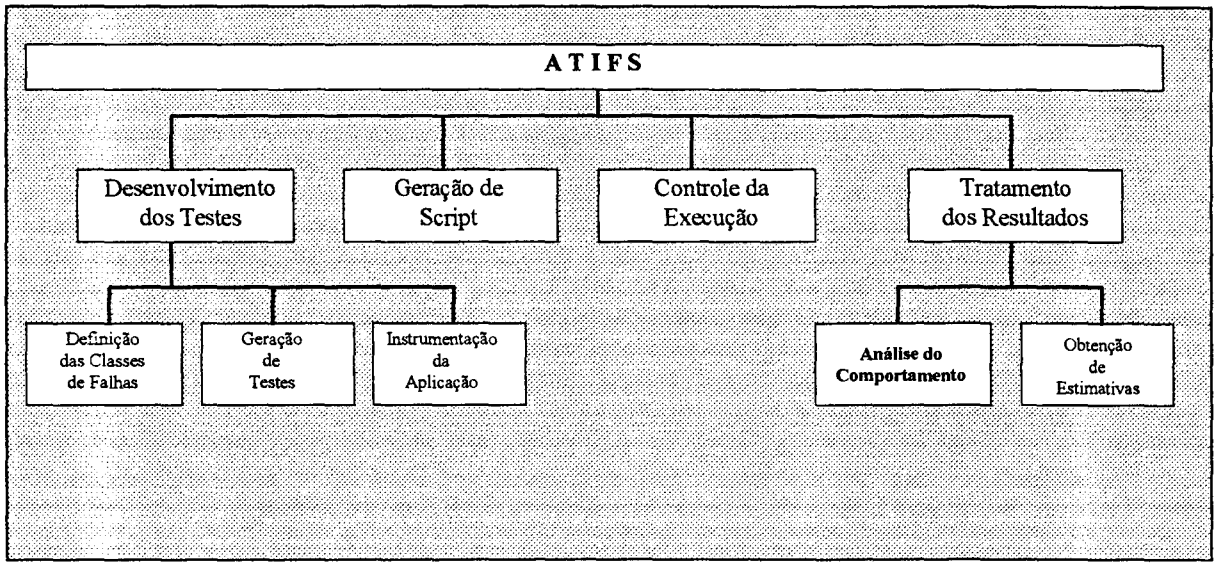

Figura 7: Estrutura do ATIFS

Como o ambiente é composto por uma série de subsistemas que necessitam trabalhar de forma integrada, uma arquitetura em camadas foi proposta para garantir essa integração (Figura 8). Para atingir essa integração completa o ambiente contém, além dos subsistemas já mencionados, os seguintes componentes [Pre92]: um banco de dados, que vai servir como um depósito global, onde são armazenadas todas as informações de testes; um subsistema para o controle de objetos, gerenciando a troca de informação; um mecanismo para controle dos subsistemas e uma interface com o usuário que permite o acesso consistente entre as ações realizadas pelo usuário e os subsistemas que compõem o ambiente. Os principais subsistemas são descritos brevemente nas seções seguintes.

6.2 Desenvolvimento dos Testes

Este subsistema compreende as funções de geração de testes, definição das falhas e instrumentação da IUT. A função de **geração de testes** trata da obtenção dos casos de testes, esta obtenção podendo se dar de maneira manual ou automática. Na geração manual, os casos de testes são fornecidos pelo usuário. Na geração automática, os casos de testes são obtidos a partir da especificação do sistema. A função de **definição das falhas** tem por objetivo auxiliar o usuário na definição das classes de falhas a serem consideradas para injeção, seus atributos e o método para injetá-las. A função de **instrumentação da implementação** tem como objetivo auxiliar o usuário na preparação da implementação para os testes.

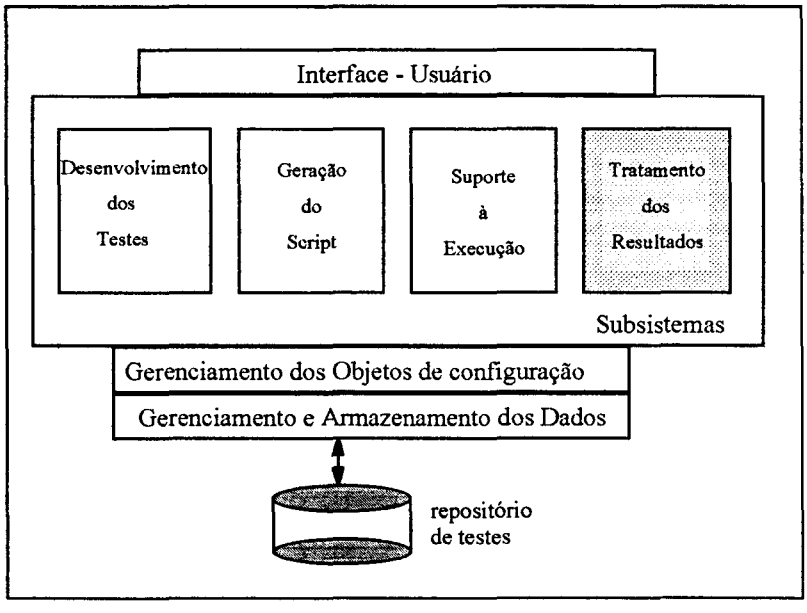

Figura 8: Arquitetura do ATIFS em camadas

6.3 Geração de Script

Este subsistema permite que o usuário descreva os testes a serem realizados. Cada seqüência de testes constitui um **experimento** e sua descrição inclui:

- os casos de testes a serem aplicados;
- os tipos de falhas a serem injetadas;

- o número de falhas a serem injetadas;
- o momento de injeção;
- o número de nós do sistema, e quais nós serão submetidos à injeção;
- os processos a serem executados em cada nó, o que inclui a implementação sob teste e os testadores;
- a duração do experimento;
- onde serão armazenados os resultados coletados.

O ATIFS prevê a realização de testes estatísticos, com o objetivo de obter medidas da eficiência dos mecanismos de tolerância a falhas do sistema. Para garantir a qualidade destas medidas, é preciso realizar um grande número de experimentos. Portanto, é necessário automatizar o encadeamento de múltiplos experimentos. Uma das dificuldades neste encadeamento diz respeito à sincronização e reiniciação de vários processos, residentes nos diferentes nós. A reiniciação do sistema pode consistir simplesmente em recarregar os processos, ou pode ser mais complexa, exigindo o *reset*[18] de todos os nós. A definição de como encadear os experimentos também é parte da geração do script.

6.4 Controle de Execução

Este subsistema tem como funções:

- Controlar a execução dos testes, responsabilizando-se pela ativação e interação com o SUT (sistema alvo), de acordo com o script gerado.
- Monitorar o sistema alvo, coletando os dados observados durante os testes no sistema alvo, armazenando-os para, mais tarde, analisá-los.

Os dados coletados durante a execução dos experimentos incluem:

- As interações (entradas e saídas) observadas.
- O histórico das falhas injetadas.
- O histórico dos testes realizados, contendo, entre outras, a identificação dos testes realizados, os métodos utilizados para obtê-los, os resultados obtidos e a versão da implementação submetida aos testes.

[18] Por *reset* entenda-se a reiniciação total do nó, incluindo a recarga do sistema operacional.

- As informações de auxílio ao diagnóstico dos erros existentes na implementação e que venham a ser detectados durante os testes: *dumps* de memória, valores de variáveis de contexto, etc.

Além disso, esse subsistema deve permitir que o usuário possa interagir com o sistema durante a execução dos experimentos, fornecendo-lhe os comandos que permitam obter informações parciais sobre os dados coletados, bem como interromper, recomeçar ou abortar os experimentos.

6.5 Tratamento dos Resultados

Este componente é responsável pelo tratamento dos dados coletados durante os testes, e compreende as seguintes funções: análise do comportamento observado e obtenção de estatísticas.

A **análise do comportamento** tem por objetivo verificar se o comportamento observado está de acordo com o que foi especificado. Esta função pode ser feita manualmente, ou seja, o usuário verificando se os resultados obtidos dos testes são os resultados esperados. No entanto, quando o número de experimentos se torna elevado, é necessário que a análise possa ser feita automaticamente.

A **obtenção de estatísticas** vai permitir ao usuário definir os parâmetros de qualidade que se deseja obter, tais como a cobertura da tolerância a falhas e a latência, bem como o método de estimação a ser empregado para obtê-los. Aplicando-se o método escolhido o usuário terá não somente o valor desejado, mas também o erro associado à estimativa. Além disso, é possível ao usuário visualizar, através de gráficos, os principais resultados das análises realizadas.

Para maiores informações sobre os demais subsistemas de apoio ao processo de teste no ATIFS e para maiores detalhes sobre o ambiente, consulte [Mar95].

Parte II

Desenvolvimento de uma Ferramenta de Análise de Traço com Geração de Diagnósticos

Essa segunda parte do trabalho tem como objetivo desenvolver uma ferramenta de análise de resultados de testes. O método usado no desenvolvimento da ferramenta baseia-se na análise de traço. O capítulo 7 descreve, inicialmente, os tipos de testes considerados na análise, o porquê da escolha da análise de traço, seguindo-se pela escolha da arquitetura de testes e pela definição do modelo usado para descrever o comportamento da implementação de protocolo a ser testada. Define-se também o formato que deve ter um traço de execução e como o traço é obtido durante a execução dos testes.

No capítulo 8, são descritos os aspectos da análise de traço com e sem a aplicação da injeção de falhas; o tratamento necessário ao traço quando se aplica a injeção de falhas; os algoritmos usados na análise de traço; os produtos dessa análise e, por fim, uma aplicação exemplo do método de análise.

7. Contexto da Análise de Resultados

Nesse capítulo serão descritos os componentes necessários para a realização da análise de resultados de testes para uma implementação de protocolo na presença de falhas.

7.1 Tipos de Testes Considerados na Análise de Resultados

O ATIFS, apresentado no capítulo 6, provê o suporte a diversos tipos de testes. Entretanto, a análise de resultados, nesse trabalho, é realizada, para dois tipos de testes[19]:

a) Testes de comportamento (seção 4.1) das funções normais do protocolo.

b) Testes de comportamento dos mecanismos de tolerância a falhas.

O item a) se refere à verificação da conformidade dinâmica das funções normais da IUT em relação à especificação do protocolo. O item b) equivale a testar a **tolerância a falhas**, visto que se preocupa com as partes da IUT preparadas para tratar de falhas. Como será visto adiante na seção 7.4, o modelo do comportamento da IUT deverá incorporar o tratamento das falhas para que o item b) possa ser realizado. A injeção de falhas por *software* (seção 5.1) é usada para exercitar os mecanismos de tolerância a falhas.

7.2 Tipo de Análise de Resultados Adotado

Na escolha da técnica para realizar a análise de resultados dos testes, alguns fatores precisaram ser ponderados: os tipos de testes que seriam realizados; o contexto do ambiente onde os testes seriam executados; as limitações que seriam impostas pela técnica de análise adotada. Assim, foram identificadas as seguintes características a serem levadas em consideração na análise de resultados:

[19] Todos os testes mencionados aqui se referem a testes de uma implementação.

a) Conforme mencionado na seção 7.1, serão realizados testes de comportamento e de tolerância a falhas.

b) Os testes serão executados no ATIFS, um ambiente que provê injeção de falhas por *software*, conforme descrito no capítulo 6.

c) A análise de resultados não pode impor limitações à seleção de entradas de teste e não deve interferir na execução dos testes.

A proposta desse trabalho é automatizar a análise de resultados. Por isso, não foram cogitados métodos de análise de resultados para os casos onde não há uma especificação disponível para a IUT. Isso porque as técnicas existentes para tais situações (seção 3.2) são pouco confiáveis e parciais no que diz respeito à cobertura dos testes.

A análise embutida nos casos de testes (seção 3.2) não foi adotada porque exige que os resultados esperados para cada caso de teste sejam obtidos antes da execução dos testes. Em testes por injeção de falhas, ela iria impor limitações à seleção de entradas (casos de testes e de falhas), visto que a presença de falhas durante os testes pode mudar os resultados esperados. Dessa forma, a injeção de falhas precisaria ser determinística: sabendo-se o exato momento e local da injeção de uma falha, ainda é possível de se prever com antecedência as saídas esperadas para os testes. Por outro lado, uma das estratégias do ATIFS é realizar injeção de falhas de forma não-determinística e assincronamente às entradas normais de testes. Nesse contexto, torna-se difícil prever, com antecedência e com exatidão, as saídas esperadas para os testes, já que as falhas podem ocorrer a qualquer instante e podem desviar o comportamento da IUT.

A análise *on-line* (seção 4.3) é adequada para a eliminação de falhas (seção 2.3), mas requer um modelo executável do sistema, ou seja, é necessário construir um simulador para concretizar o observador. Além disso, quando há uma discrepância entre a execução da IUT (o executante) e a execução do simulador do modelo do comportamento da IUT (o observador), significa que a IUT não está de acordo com o modelo de seu comportamento. Assim, a análise pode parar para que a falha seja tratada, o que pode ser uma limitação para a previsão de falhas (seção 2.3).

Na análise *off-line* (seção 4.3), todos os testes são executados e, durante a execução, armazena-se o histórico da observação das entradas fornecidas pelos casos de testes, das saídas produzidas pela IUT e das falhas injetadas durante a execução dos testes (no caso dos testes por injeção de falhas). Esse tipo de análise visa a eliminação de falhas e não limita a previsão de falhas por ser independente da execução dos testes. Além disso, não impõe restrições na seleção de entradas de testes e torna prático o tratamento de falhas injetadas e a obtenção de diagnósticos de erros.

A técnica de análise adotada nesse trabalho foi a análise *off-line* devido aos motivos citados acima e, também, por ser a mais adequada ao ATIFS, já que a injeção de falhas é uma das características marcantes desse ambiente de testes e é realizada sobre uma implementação. Se fosse adotada a análise *on-line*, seria necessário injetar falhas no executante (IUT) e no observador (simulador) (seção 4.3). Além disso, a análise *off-line* só é realizada após o término dos testes e da injeção de falhas, não causando qualquer impacto sobre a execução dos testes, o que não se pode dizer sobre a análise *on-line*.

A análise *off-line* é caracterizada por um analisador de traços de execução, que será descrito em detalhes no capítulo 8.

7.3 Arquitetura de Testes Adotada

Na seção 4.2, foram identificadas as arquiteturas de testes padronizadas pela ISO: local, remota, distribuída e coordenada. Na primeira, o sistema de teste e a IUT residem no mesmo computador; nas demais, chamadas externas, o sistema de teste reside num computador remoto, embora o UT[20] resida no mesmo computador que a IUT. Além dessas quatro arquiteturas, existe também a arquitetura *ferry*, cuja principal idéia é transportar dados de testes do TS[21] ao SUT[22], transparentemente, permitindo que o UT e o LT[23] residam no TS.

[20] *Upper Tester*

[21] *Test System*

[22] *System Under Test*

[23] *Lower Tester*

A arquitetura de testes adotada influencia a observação das interações ocorridas durante os testes. Apesar disso, a escolha de uma arquitetura de testes não faz parte do escopo desse trabalho, já que essa escolha também está associada aos objetivos e às estratégias do ambiente de testes em que a arquitetura vai ser usada.

No ATIFS, foi adotada a arquitetura *ferry clip* [ZDH88, CLPZ90]. Algumas adaptações foram realizadas para tornar possível a injeção de falhas por *software*. A Figura 8 mostra o resultado dessas adaptações. Nela podemos identificar alguns componentes que vão determinar o conteúdo do traço de execução que será analisado. Esses componentes estão descritos na Tabela 1.

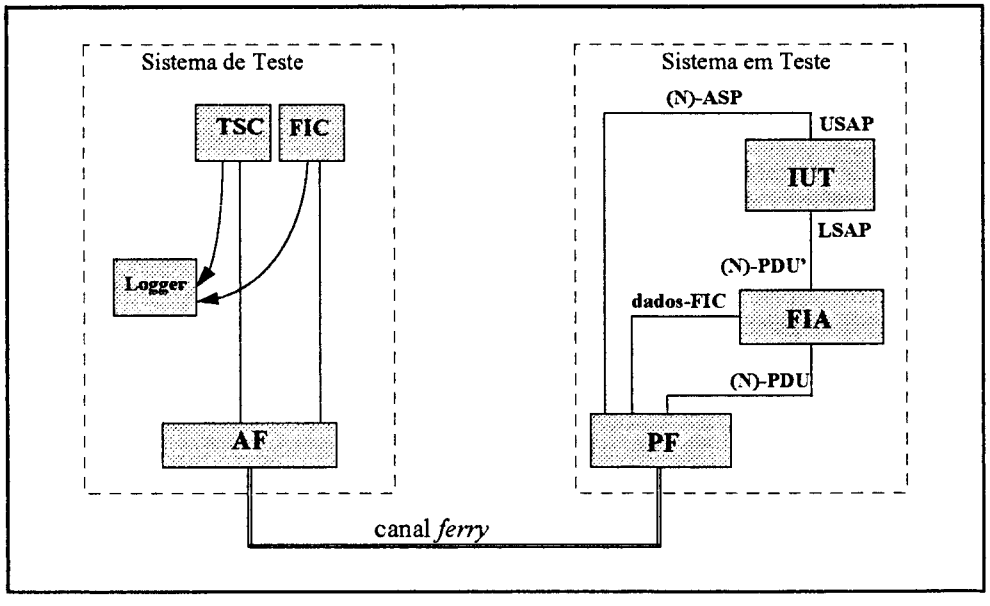

Figura 8: Adaptação da arquitetura ferry para testes por injeção de falhas em uma única entidade de protocolo

Além da adaptação da arquitetura *ferry clip* para testar uma IUT, também foi adaptada uma arquitetura para testar várias IUTs ao mesmo tempo (Figura 9). Nela podemos pensar em realizar a análise de traço do sistema como um todo, incluindo várias entidades de protocolo ao mesmo tempo. Foi definida de modo a abranger testes de interoperabilidade (seção 4.1) das diversas entidades e também a validação de propriedades globais do sistema.

Tabela 1: Componentes da Arquitetura Ferry Clip adaptada para Injeção de Falhas

Componente	Descrição
Sistema de Teste	nó que controla a execução dos testes e a injeção de falhas
Sistema em Teste	nó que contém a implementação de protocolo a ser testada
IUT	implementação de protocolo que está sendo testada
TSC	*Test Sequence Controller*: libera as entradas dos casos de testes que vão estimular a IUT
FIC	*Fault Injection Controller*: libera as falhas a serem injetadas nas entradas fornecidas à IUT
FIA	*Fault Injection Agent*: aplica as falhas liberadas pelo FIC
AF	*Active Ferry Clip*: agente do canal de comunicação *ferry* do lado do sistema de teste. Está ligado ao PF, descrito a seguir, através do canal *Ferry*.
PF	*Passive Ferry Clip*: componente que recebe as instruções do sistema de teste e as transmite para a IUT ou para o FIA, além de transmitir para o sistema de teste as saídas fornecidas pela IUT e a identificação de quais interações receberam falhas
Canal *Ferry*	canal de comunicação por onde passam as informações de controle dos testes; esse canal é independente do canal por onde passa o tráfego normal de informações na rede, além de ter um protocolo de comunicação próprio e suposto ser confiável
Logger	armazena as interações de entrada, de saída e falhas injetadas na IUT observadas durante a execução dos testes

Esse trabalho, entretanto, se limita a verificar o comportamento de cada IUT individualmente, tanto se ela estiver trabalhando em conjunto com outras IUTs (Figura 9), quanto se ela for a única IUT do sistema em teste, situação mostrada na Figura 8. No segundo caso, o poder de detecção de erros é maior porque o sistema de teste controla e observa a IUT em dois PCOs, enquanto que, no primeiro caso, geralmente, apenas um PCO está disponível. Apesar disso, o primeiro caso retrata uma situação mais próxima da realidade. Maiores detalhes sobre o poder de detecção de erros das arquiteturas de testes são dados na seção 7.5.2.

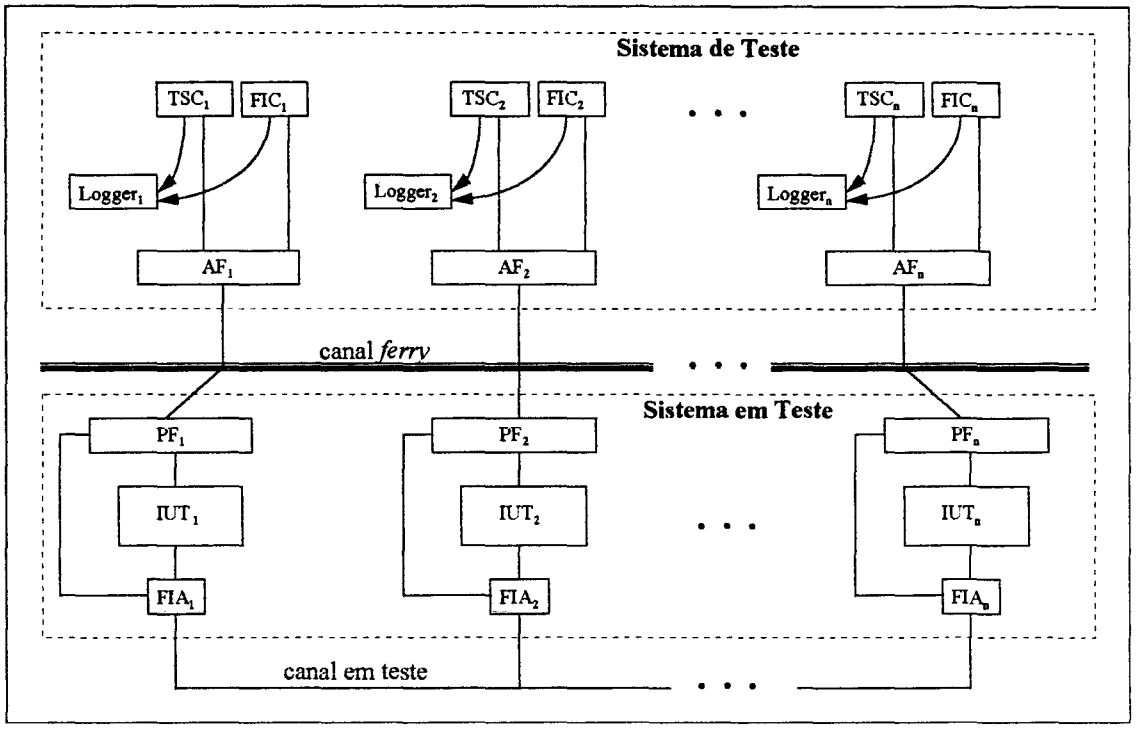

Figura 9: Arquitetura de teste para várias IUTs

7.4 O Modelo do Comportamento

Máquinas finitas de estados são, freqüentemente, usadas para expressar o comportamento de protocolos de comunicação [Hol91, GB93, BP94, RDT95a].

Uma máquina finita de estados (FSM) é uma tupla $<S, s_0, X, Z, g>$ [RDT95a], onde **S** é um conjunto finito não-vazio de estados , $s_0 \in S$ é o estado inicial da FSM, **X** é um conjunto finito não-vazio de entradas, **Z** é um conjunto finito não-vazio de saídas, **g** é uma função de transição definida como $\mathbf{g}: S \times X \rightarrow S \times Z$. A notação $g(s_i, x) = (s_j, z)$ significa que a FSM no estado s_i realiza uma transição para o estado s_j quando a entrada x é aplicada, produzindo a saída z.

Uma FSM pode ser representada por um grafo direcionado G(V,E)[24], onde S = V e cada transição $g(s_i, x) = (s_j, z)$ corresponde a uma aresta $e \in E$, direcionada de s_i para s_j com o rótulo x/z.

[24] V = conjunto de vértices; E = conjunto de arestas

Uma FSM é completamente especificada (também se diz FSM completa) se, para cada estado $s_i \in S$ e para cada entrada $x_i \in X$, há uma transição que sai de s_i com a entrada x_i [RDT95a].

Uma FSM é determinística se, para um dado estado $s_i \in S$ e uma entrada $x_i \in X$, há, no máximo, uma transição que sai de s_i com a entrada x_i [RDT95b].

Para melhorar o poder de descrição de uma FSM, variáveis adicionais são introduzidas no modelo matemático. Essas variáveis são usadas para especificar condições para a execução de transições e cálculos cumpridos durante as transições [HUK95]. Esse modelo é chamado de máquina finita de estados estendida (EFSM).

No ATIFS, uma IUT deve ter seu comportamento descrito na forma de EFSM, cuja extensão se refere a variáveis que indicam o tratamento de falhas nas transições.

A composição desse modelo EFSM é dada pela tupla $< S, s_0, X, Z, F, g >$, onde:

S conjunto finito de estados; $S = \{s_0, s_1, s_2, \ldots, s_n\}$

s_0 estado inicial do modelo

X conjunto finito de entradas; $X = \{ x_0, x_1, x_2, \ldots, x_n\}$

Z conjunto finito de saídas; $Z = \{ z_0, z_1, z_2, \ldots, z_n\}$

F conjunto finito de tipos de falhas previstas no modelo; $F = \{f_0, f_1 f_2, \ldots, f_n\}$, sendo que f_0 representa ausência de falhas;

g função de transição; é definida como $\boldsymbol{g}: S \times X \times F \rightarrow S \times Z$, onde $g(s_i,x,f) = (s_j,z)$ significa que o modelo EFSM no estado s_i realiza uma transição para o estado s_j, produzindo a saída **z**, quando a entrada **x** e a falha **f** são aplicadas.

Assim como uma FSM, uma EFSM também pode ser representada por um grafo direcionado G(V,E), onde **S** = V e cada transição $g(s_i, x, f) = (s_j, z)$ corresponde a uma aresta $e \in E$, direcionada de s_i para s_j com o rótulo x/f/z, onde x é uma entrada, f é uma falha, z é uma saída.

No contexto desse trabalho, uma entrada também é chamada de interação de entrada. O mesmo é válido para uma saída. A composição de uma interação de entrada **x** ou uma interação de saída **z** é dada por:

interação ::= <SAP> <ação> <evento>

<SAP> ::= U | L

<ação> ::= *send* | *receive*

<evento> ::= <id-evento> <lista de parâmetros>

onde:

<SAP> é o ponto de acesso a serviço onde a interação deve ocorrer. Um SAP pode ser **U** ou **L**. Se for **U** (**U**pper), significa que a interação ocorre numa troca de informações com a camada superior (N+1) (capítulo 4), correspondendo, portanto, ao ponto de acesso superior (USAP); se for **L** (**L**ower), significa que a interação ocorre numa troca de informações com a camada inferior (N-1), correspondendo, portanto, ao ponto de acesso inferior (LSAP).

<ação> é o tipo de ação sobre a interação. Uma ação pode ser ***send*** ou ***receive***. Pode corresponder a um envio (***send***) ou um recebimento (***receive***) da interação através do SAP especificado.

<evento> é composto de um identificador do evento (**id-evento**), que é o tipo da interação transmitida. Também compõe o evento uma **lista de parâmetros**, que corresponde aos dados adicionais a serem transmitidos junto ao id-evento.

A composição da identificação de uma falha (f_n) é dada por:

<tipo> <lista de parâmetros>

onde:

<tipo> é o tipo de falha tratada durante uma transição do modelo. Os tipos de falhas previstos no modelo são descritos na Tabela 2.

<lista de parâmetros> são informações que acompanham o identificador do tipo de falha e que são úteis ao FIC e ao FIA.

Tabela 2: Tipos de falhas considerados no modelo do comportamento

TIPO DE FALHA	CÓDIGO	AÇÃO EXECUTADA PELO INJETOR DE FALHAS
alteração	f_1	altera o conteúdo de uma entrada liberada pelo TSC[25] e a envia para a IUT
duplicação	f_2	duplica uma entrada liberada pelo TSC e envia para a IUT duas interações idênticas
retardamento	f_3	retém, durante um tempo ***t***, uma entrada liberada pelo TSC; em seguida, libera a entrada para a IUT
supressão	f_4	suprime uma entrada liberada pelo TSC e a IUT não a recebe

Nesse trabalho, para fins de análise de traço (capítulo 8), será considerada apenas a parte de controle do modelo, não sendo usadas nem a lista de parâmetros do evento da interação, nem a lista de parâmetros da identificação de falha, que representam a parte de dados.

A Figura 10 mostra a representação gráfica de um modelo de comportamento para um protocolo de comunicação tomado como exemplo. O modelo é um grafo orientado onde os nós são os estados e as arestas são as transições. Cada aresta está associada a um par de interações (entrada/saída) e a uma indicação de tratamento de falha. Os seguintes conjuntos podem ser identificados nesse modelo:

S = {INI, VT1, TIP, FIM } (INI = estado inicial)

X = { 21, 0102, 82, EOF, DIF }

Z = { A1, A2, A3, A4, A5, A6 }

F = { f_0, f_1 }

Se houver uma indicação f_0 numa transição, significa que ela equivale à uma função normal da IUT; caso contrário, equivale a uma função de tratamento de falha (transições com linhas pontilhadas). A entrada **DIF** faz parte do tratamento de erros pois ela representa todas

[25] *Test Sequence Controller*

as entradas que não estejam especificadas nas transições que saem do estado corrente do modelo. Por exemplo, no estado **INI**, ela representa todas entradas diferentes de **21** e de **EOF**.

Para simplificar a representação do modelo na Figura 10, não foram representadas as listas de parâmetros mencionadas anteriormente. As ações ***send*** e ***receive*** são representadas, respectivamente, por **!** e **?** [PM92, KSNM91], tanto na representação gráfica quanto na representação tabular do modelo.

Um exemplo da interpretação dessa notação pode ser dado para a transição **<L?EOF> <f_0>/<U! A2>**, entre os estados INI e FIM do modelo da Figura 10: o bloco <L?EOF> significa que uma interação do tipo EOF é recebida pelo SAP inferior da IUT; <f_0> significa que a transição corresponde a uma função normal do modelo; <U! A2> significa que uma mensagem de tipo A2 será enviada ao SAP superior, como resposta (saída) à mensagem do tipo EOF. A conseqüência dessa transição é a mudança do estado INI para o estado FIM.

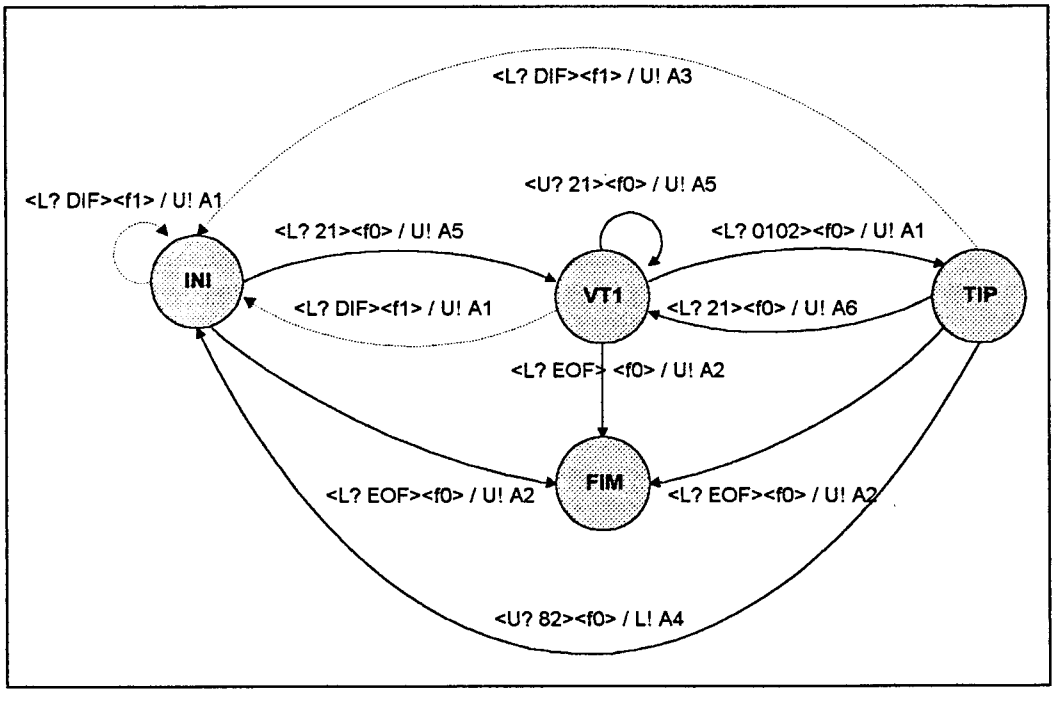

Figura 10:Modelo do Comportamento baseado em Máquina Finita de Estados Estendida

A Tabela 2 mostra a representação do modelo na forma de tabela de transições da Figura 10.

Tabela 3: Representação tabular do modelo do comportamento da Figura 10

Estado Corrente	Entrada	Saída	Falha Tratada	PróximoEstado
INI	L? 21	U! A5	f_0	VT1
INI	L? DIF	U! A1	f_1	INI
INI	L? EOF	U! A2	f_0	FIM
VT1	U? 21	U! A5	f_0	VT1
VT1	L? 0102	U! A1	f_0	TIP
VT1	L? EOF	U! A2	f_0	FIM
VT1	L? DIF	U! A1	f_1	INI
TIP	L? 21	U! A6	f_0	VT1
TIP	L? DIF	U! A3	f_1	INI
TIP	U? 82	L! A4	f_0	INI
TIP	L? EOF	U! A2	f_0	FIM

A princípio, algumas observações são feitas sobre as características do modelo:

a) Pode ser não-determinístico: mais de um estado pode ser atingido com a mesma interação de entrada, a partir do mesmo estado corrente.

b) Deve ser, preferencialmente, completo. Caso não seja e os casos de testes exercitem transições não especificadas, a análise de traço detectará o modelo incompleto e poderá ser paralisada.

c) Deve corresponder a uma máquina de Mealy [Gil62, Hol91], onde cada transição está associada a uma entrada e uma saída correspondente.

7.5 O Traço de Execução

Um traço de execução (também chamado histórico de execução [JB83]) de uma implementação é uma seqüência de interações das quais a implementação participa. O traço é obtido da seguinte forma: a implementação sob teste é estimulada por uma seqüência de testes. As entradas dessa seqüência, bem como as saídas produzidas pela implementação, são "observadas" pelo sistema de teste e armazenadas em meio persistente na forma de um histórico de execução, ou seja, um traço.

Baseado na organização de uma seqüência de teste (seção 4.1), o traço pode ser dividido em grupos de testes, casos de testes e interações de testes. Além disso, as interações devem estar organizadas de forma compatível com o modelo do comportamento da IUT. O formato do traço de execução nesse trabalho é definido na seção seguinte.

7.5.1 Formato do Traço

O formato geral de um traço de execução é dado na Figura 11:

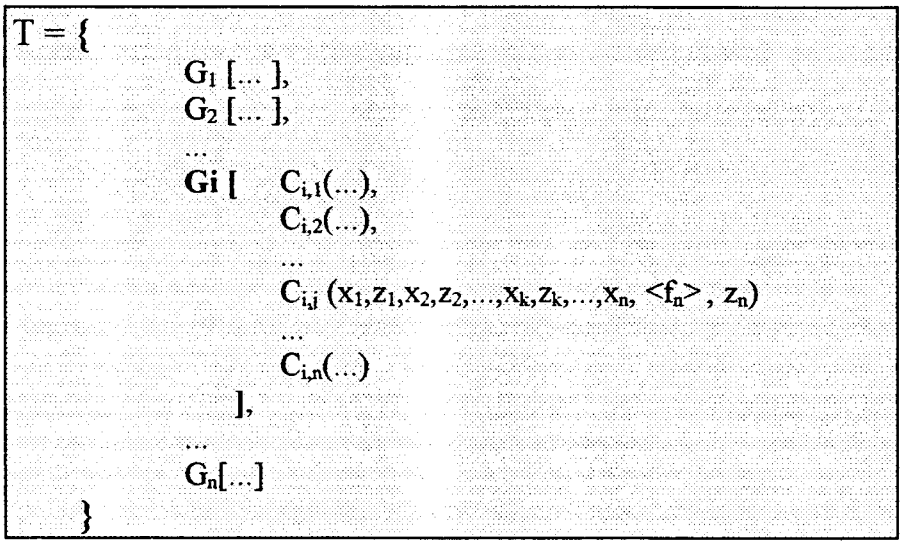

Figura 11: Formato de um traço de execução

onde:

T é o traço resultante da observação da execução de uma seqüência de testes

$\mathbf{G_i}$ é a identificação do grupo de teste i

$\mathbf{C_{i,j}}$ é a identificação do caso de teste *j* dentro do grupo de teste *i*

$\mathbf{x_k}$ é a interação de entrada k

$\mathbf{z_k}$ é a interação de saída k

$\mathbf{f_n}$ é a identificação de que a falha **n** foi injetada na interação x_n

Ao final de cada traço de execução há uma identificação do número de casos de testes inicial e o número de casos de testes que puderam ser aplicados na IUT. Essa informação é útil na obtenção dos produtos da análise de traço (ver seção 8.5). A Tabela 4 mostra um exemplo de traço de execução pronto para ser analisado. Cada linha situada abaixo da identificação do

caso de teste, está dividida em duas partes: a primeira é uma interação de entrada e a segunda é a interação de saída correspondente à entrada.

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_1
L? 21	U!A5
L?0102	U!A1
L?21	U!A6
L?EOF	U!A2
Caso de Teste	C_2
L? 21	U!A5
L?0102	U!A1
L?21	U!A6
L?EOF	U!A2

(a)

Traço	T_1
Grupo de Teste	G_2
Caso de Teste	C_3
L? 21	U!A5
	U!A2
L?0102	U!A1
L?EOF	U!A2
Caso de Teste	C_5
L? 21	U!A5
<f1> L?0102	U!A1
L?DIF	U!A3
L?EOF	U!A2

(b)

N° de casos de testes: 4	N° de casos de testes aplicados: 4

Tabela 4: Exemplo de traço de execução. A parte (a) refere-se ao grupo de teste G1 e a parte (b) refere-se ao grupo de teste G2, ambos do mesmo traço T1

O traço descrito na Tabela 4 corresponde à execução de testes sobre uma IUT cujo modelo está descrito na Figura 10, no qual é considerado apenas um tipo de falha (f_1). Uma posição em branco dentro do traço significa a ausência de uma interação naquela posição.

7.5.2 Obtenção do Traço de Execução

A arquitetura de testes adotada é a arquitetura *ferry clip* (seções 4.2 e 7.3) adaptada para o uso de injeção de falhas por *software* (seção 5.1). Nesse contexto, o traço de execução pode ser obtido em duas situações distintas:

a) Quando se considera que uma única IUT e dois PCOs[26] estão disponíveis: um PCO superior, para trocar informações com o UT, e um PCO inferior, para troca de informações com o LT. Essa situação equivale à Figura 8.

[26] *Point of Control and Observation*

b) Quando duas ou mais IUTs estão trabalhando em conjunto e apenas um PCO está disponível (PCO superior). O LT não existe pois as interações ocorridas no SAP inferior da IUT são referentes às entidades pares (implementações da mesma camada que se interoperam com a IUT considerada). Essa situação equivale à Figura 9.

No caso do item a), o poder de detecção de erros é maior devido ao número de PCOs disponíveis [DB85, BDZ89].

No caso do item b), apesar de termos as informações apenas do PCO superior para analisar, a IUT está trabalhando em conjunto com outras entidades pares, uma situação mais real do que caso a). Para se ter, mesmo nesse caso, as interações do PCO inferior, seria necessário um **árbitro** [BDZ89] ou espião que observasse tudo o que se passa no canal em teste entre duas IUTs. Esse componente precisaria ainda reconhecer a linguagem do protocolo que está sendo testado.

A obtenção do traço será considerada para o item a). Essa escolha se deve ao fato de que o traço de execução deverá passar por um tratamento antes de ser usado na análise dos resultados, e o traço obtido na situação a) é mais completo que o traço obtido na situação b). Logo, o tratamento do traço no caso a) é o mesmo para o caso b).

No caso considerado, o sistema de teste faz o papel da camada (N+1), por meio do UT, e da camada (N-1), por meio do LT. Têm-se, então, dois PCOs disponíveis para a realização dos testes. Isso significa que as interações ocorridas entre a IUT e os testadores UT e LT, além das falhas injetadas durante os testes, são observadas e armazenadas no traço de execução.

Além de considerar a IUT em conjunto com outras ou não, há duas situações distintas para a obtenção de um traço:

a) **Sem injeção de falhas**: o traço contém apenas interações de entrada e de saída.

b) **Com injeção de falhas**: o traço contém as mesmas informações do caso a) e também as indicações de falhas injetadas durante a execução dos testes.

Essas duas situações serão apresentadas nas seções seguintes.

7.5.2.1 Obtenção do Traço na Execução dos Testes sem Injeção de Falhas

Considerando a arquitetura de testes apresentada na Figura 8, o TSC é o componente que controla a liberação de entradas para a IUT. Quando uma entrada é enviada para a IUT através do canal *ferry*, a saída produzida pela IUT, em resposta a essa entrada, é devolvida ao sistema de teste pelo mesmo canal. No momento em que o TSC enviou a entrada para a IUT, também a enviou para o Logger, que armazena o traço de execução. A resposta da IUT chega ao TSC que, por sua vez, a envia para o Logger e libera uma nova entrada para a IUT. Esse procedimento dura até que a seqüência de testes termine ou seja interrompida.

Um exemplo de traço obtido nessas condições é o traço do caso de teste C1 da Tabela 4. Nesse caso, o traço está pronto para ser analisado, sem necessidade de qualquer tratamento.

7.5.2.2 Obtenção do Traço na Execução dos Testes com Injeção de Falhas

Quando se aplica a injeção de falhas, temos duas fontes assíncronas de entradas para a IUT: o TSC, responsável pelas entradas normais da IUT, e o FIC responsável pelas falhas que devem ser injetadas nas entradas. Assim, o conteúdo do traço, até então restrito a interações de entrada e de saída, passa a indicar também as falhas que foram injetadas e quais entradas foram afetadas pelas falhas.

O instante em que uma falha deve ser aplicada é determinado pelo FIC. Entretanto ele não determina qual a interação que vai receber um caso de falha que ele gera. A falha a ser aplicada é enviada ao FIA, no SUT. Quando este último a recebe, aplica a falha na primeira interação de entrada que vier do TSC. Toda entrada liberada pelo TSC é enviada para a IUT e para o *Logger* para compor o traço de execução. Quando o FIA interceptou uma interação para aplicar a falha ele envia uma mensagem para o FIC dizendo qual foi a interação que recebeu a falha e qual foi a falha injetada. O FIC, por sua vez, envia essa informação para o *Logger*. Assim, o traço passa a ter, além das interações de entrada e de saída, as informações sobre as falhas que foram injetadas.

Como foi descrito acima, a injeção de falhas implica em informações adicionais no traço. Essas informações nos dão um guia de quais interações foram atingidas pelas falhas injetadas pelo FIA e quais tipos de falhas foram aplicados.

Considera-se, nesse trabalho, quatro tipos de falhas (Tabela 2) que representam as causas de erros nos meios de comunicação, fazendo com que mensagens sejam perdidas, alteradas, duplicadas ou retardadas.

Nesse contexto, antes que o traço obtido seja analisado, é necessário que ele esteja organizado na forma adequada ao modelo do comportamento da IUT, o que equivale a dizer que o traço deve conter uma interação de saída para cada interação de entrada (devido ao modelo ser uma máquina de *Mealy*). Detalhes da preparação do traço para a análise são descritos no capítulo 8.

8. Análise de Traço com Geração de Diagnósticos

Neste capítulo, serão apresentados os aspectos da execução do analisador de traços e da geração de diagnósticos de erros. Também serão mostradas as informações complementares sobre o funcionamento dos mecanismos de tolerância a falhas, produzidas durante a análise de traço.

Nesse trabalho, considera-se o uso de um analisador de traços nos testes da implementação de um protocolo, visto que também pode ser usado para verificar a consistência de uma especificação mais detalhada de um protocolo em relação à uma versão mais abstrata dessa especificação, durante a fase de projeto do protocolo [JB83].

Dois fatores podem influenciar na análise de traço: (1) a injeção de falhas; (2) a disponibilidade de PCOs. Assim, a análise de traço pode ser feita das seguintes formas:

a) Análise de traço sem injeção de falhas, onde o traço contém as interações da qual a IUT participou, sendo que, durante a execução dos testes, nenhuma falha foi injetada pelo sistema de teste.

b) Análise de traço com injeção de falhas, onde o traço contém as interações da qual a IUT participou mais as informações sobre falhas injetadas.

c) Análise de traço onde este contém as interações ocorridas em apenas um PCO, independentemente de se executar ou não a injeção de falhas.

As seções 8.1 a 8.2 apresentam os tipos de análise de traço mencionados acima. Na seção 8.4 é descrito o algoritmo geral para análise de traço; na seção 8.5 são apresentados os produtos da análise de traço e na seção 8.6 é apresentada uma aplicação exemplo do analisador de traços.

8.1 Análise de Traço sem Injeção de Falhas

O principal componente da análise de traço é um analisador que tem como entradas o modelo do comportamento da IUT e o traço de execução observado durante a execução dos testes. Conforme ilustrado na Figura 12, o traço de execução é obtido durante a execução dos casos de testes na IUT. Terminada a execução, o analisador de traços verifica se a seqüência de

interações armazenadas no traço é uma seqüência possível de ser obtida no modelo do comportamento da IUT. Os produtos da análise são os veredictos e diagnósticos para cada caso de teste.

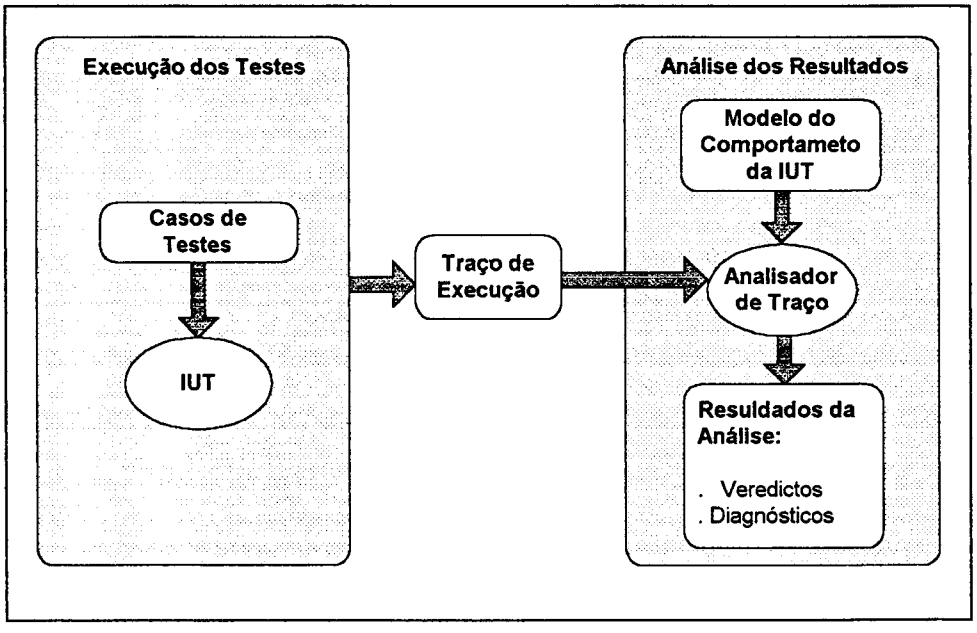

Figura 12: Obtenção e Análise do Traço de Execução

No contexto de protocolos de comunicação, a análise de traço verifica se a implementação de um protocolo está conforme sua especificação, que é dada por um modelo formal.

Como os protocolos governam a comunicação entre diferentes componentes dentro de um sistema computacional distribuído [BDZ89, Hol91], há uma implementação do protocolo presente em cada um desses componentes. O teste de um protocolo pode envolver o teste de mais de uma implementação ao mesmo tempo. Neste sentido, podemos ter dois tipos de traço de execução: local e global [Sar88, BDZ89,]. O traço local é o conjunto das interações ocorridas em um ou mais pontos de interação (SAP) de uma IUT. O traço global é o conjunto de todas as interações observadas em todas as IUTs.

O traço global proporciona maior poder de detecção de erros do que o traço local [BDZ89]. Entretanto, para se ter o traço global ordenado, é necessário ter *clocks* sincronizados num ambiente distribuído, uma tarefa que não é fácil de ser realizada [BDZ89, Lam78]. O traço local, por outro lado, é facilmente ordenado por se referir a um único *clock*.

A análise de traço, nesse trabalho, refere-se à análise do comportamento dinâmico da implementação de um protocolo e de seus mecanismos de tolerância a falhas. Uma única implementação é analisada de cada vez, mesmo que os testes tenham sido realizados em várias IUTs ao mesmo tempo. Portanto, considera-se o traço local.

A operação do analisador de traço consiste em ler um traço seqüencialmente e, após cada interação lida, verificar os possíveis estados nos quais o modelo do comportamento da IUT pode, possivelmente, estar.

Se o modelo do comportamento da IUT for não-determinístico, é possível que, para uma interação de entrada lida do traço, haja mais de uma transição possível. De fato, para cada interação de entrada o analisador de traço considera todas as transições possíveis.

8.2 Análise de Traço com Injeção de Falhas

O uso da técnica de injeção de falhas por *software* durante os testes implica na existência de informações sobre falhas no traço de execução. Os mesmos componentes descritos na Figura 12 são usados, embora haja algumas modificações e também alguns componentes adicionais. Considera-se ainda a disponibilidade de dois PCOs

As modificações são as seguintes: (1) a IUT recebe, além das entradas funcionais dos casos de testes, os casos de falhas a serem injetadas nas entradas que chegam à IUT; (2) um tratamento do traço é necessário antes que ele seja analisado, devido à ordem em que o ambiente de testes registra as informações no traço; (3) os modelos de falhas previstos no modelo do comportamento da IUT são usados no tratamento do traço; (4) além dos veredictos e diagnósticos de erros, também são fornecidas informações sobre a ativação dos mecanismos de tolerância a falhas. As mudanças com relação à Figura 12 estão em destaque na Figura 13 com linhas pontilhadas.

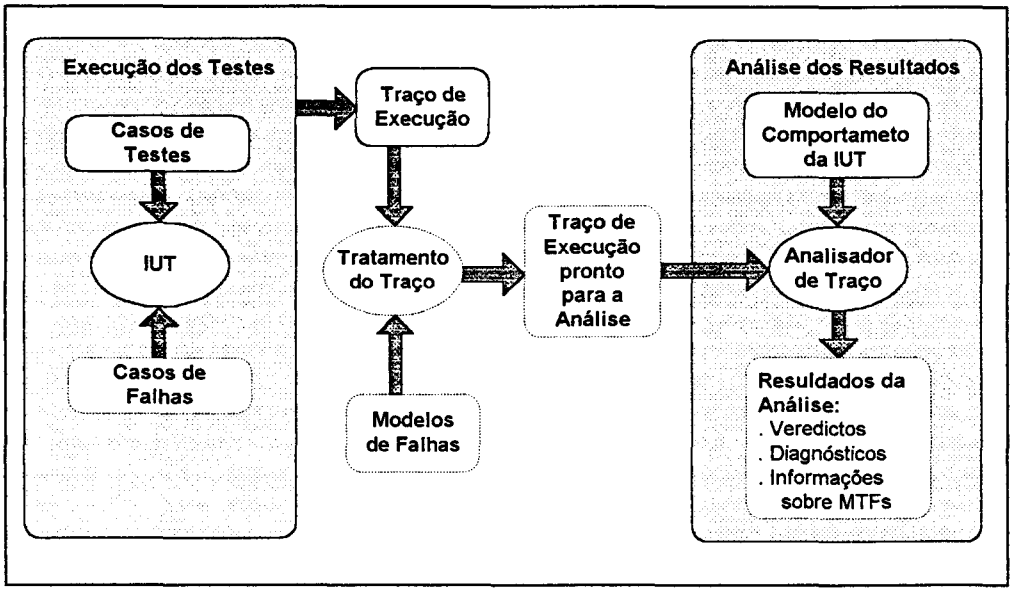

Figura 13: Obtenção e Análise do Traço de Execução com Injeção de Falhas

Um passo importante descrito na Figura 13 é o tratamento do traço de acordo com os modelos de falhas previstos no modelo do comportamento da IUT. O traço é tratado conforme o significado de cada falha injetada. Vários casos diferentes podem ser considerados nesse tratamento. Após o tratamento do traço, retorna-se ao mesmo caso da análise sem injeção de falhas, exceto pelas informações fornecidas sobre a ativação dos MTFs.

Esse tratamento do traço é baseado em hipóteses sobre o funcionamento da IUT. Essas hipóteses são simplificadoras pois é difícil prever o comportamento da IUT em presença de falhas. Além disso, é preciso abordar o problema da organização do traço com a presença de falhas, mesmo que seja de forma simplificada.

Para cada tipo de falha descrito na Tabela 2 o traço será tratado de uma forma diferente. Para fins de exemplo, considere a seguinte situação: seja $\mathbf{x_n}$ uma interação de entrada enviada para a IUT e $\mathbf{z_n}$ uma interação de saída produzida pela IUT em resposta à entrada $\mathbf{x_n}$. Suponha agora uma seqüência de entradas $X = \{x_1, x_2, x_3, x_4, x_5\}$ aplicada à IUT, referente a um único caso de teste. Em resposta a essa seqüência X, estando em presença de falhas, a IUT pode produzir um conjunto distinto de saídas (Z) para cada tipo de falha que venha a ser injetada.

As interações presentes no traço de execução estão na ordem em que foram observadas pelo ambiente de testes.

As seções seguintes apresentam o tratamento do traço para cada tipo de falha descrita na Tabela 2. Esse tratamento é necessário porque quando se realiza a injeção de falhas, o ambiente de testes registra as interações no traço numa ordem incompatível com o modelo do comportamento da IUT. Isso porque os componentes que realizam essa tarefa (TSC e FIA/FIC) são assíncronos.

Nos traços que serão apresentados nas seções 8.2.1 a 8.2.4, toda interação que estiver identificada como falha (f_n) deverá ser um elemento neutro na análise do traço, servindo apenas como indicador de uma falha que foi injetada durante os testes. Durante o tratamento do traço, o indicador de falha (f_n) serve como identificador da interação que recebeu falha: aquela imediatamente anterior ao indicador de falha no traço.

Nas seções seguintes, o destaque de algumas interações nos traços de execução significa que essas interações foram enviadas ao traço pelo FIA. As demais interações foram enviadas pelo TSC.

8.2.1 Tratamento do Traço com Falha f_1 (Alteração de Entradas)

A Tabela 5 mostra o traço de execução que reflete que a interação L?x_3 foi alterada de acordo com a falha f_1. A interação L?$\mathbf{x_3}$ foi atingida pela falha $\mathbf{f_1}$, e a interação L?$\mathbf{x_3'}$ representa, na verdade, a interação L?$\mathbf{x_3}$ afetada pela falha $\mathbf{f_1}$. A interação L?$\mathbf{x_3}$ está presente no traço por ter sido registrada no momento em que foi liberada pelo TSC. Entretanto, como existe um injetor de falhas (FIC), que funciona assincronamente ao TSC, descobriu-se mais tarde (no traço) que a IUT não recebeu a interação L?$\mathbf{x_3}$, mas sim a interação L?$\mathbf{x_3'}$.

Tabela 5: Traço com falha f1

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_1
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
$L?x_3$	
<f1> L?x3'	$U!z_3$'
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

O tratamento da falha f_1, então, é composto dos seguintes passos:

1) verificar se existe uma falha presente no traço; nesse caso é f_1, que sabemos ser uma alteração;

2) eliminar a interação anterior à posição da falha. O TSC envia para o traço todas as entradas que ele libera para a IUT. Como ele é assíncrono ao FIC, que gera os casos de falha, não sabe quais entradas receberão falhas. Quando o FIA injeta uma falha, ele envia para o FIC a identificação da falha que foi injetada e a interação que recebeu a falha. Essa informação é enviada para o traço através do FIC. Dessa forma, o que vale para o traço é a interação observada pelo FIA. A mesma interação, liberada pelo TSC está redundante e deve ser eliminada do traço.

O resultado desse tratamento é mostrado na Tabela 6 e já pode ser usado na análise do traço.

Tabela 6: Traço após tratamento da falha f1

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_1
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
<f1> L?x3'	$U!z_3'$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

8.2.2 Tratamento do Traço com Falha f_2 (Duplicação de Entradas)

A Tabela 7 mostra o traço de execução que reflete que a interação x_3 foi duplicada de acordo com a falha f_2. A IUT recebeu, de fato, as entradas $L?x_3'$ e $L?x_3''$ consecutivamente. Respondeu $U!z_3'$ em resposta à $L?x_3'$ e $U!z_3''$ em resposta à L?x3".

Tabela 7: Traço com falha f2

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_2
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
$L?x_3$	
<f2> L?x3'	
$L?x_3''$	
	$U!z_3'$
	$U!z_3''$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

O tratamento é constituído dos seguintes passos:

1. verificar se existe uma falha presente no traço; nesse caso é f_2, que sabemos ser uma duplicação, onde o FIA envia para a IUT a interação interceptada e em seguida uma cópia idêntica da mesma;
2. eliminar a interação anterior à posição da falha pois ela foi enviada em seguida pelo FIA;
3. transpor a posição das interações x_3'' e z_3' para corrigir a seqüência em que o traço foi armazenado.

O resultado desse tratamento é mostrado na Tabela 8.

Tabela 8: Traço após tratamento da falha f2

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_2
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
<f2> L?x3'	$U!z_3'$
$L?x_3''$	$U!z_3''$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

8.2.3 Tratamento do Traço com Falha f_3 (Retardamento de Entradas)

Quando uma interação é retardada por um certo tempo **t**, o que se quer é verificar o mecanismo de *timeout* da IUT. Há um tempo de espera para que uma interação chegue à IUT. Entretanto, se o atraso da interação for maior que esse tempo, uma decisão é tomada pela IUT. Nesse contexto, podemos identificar alguns casos interessantes dentro do caso de retardamento de entradas, supondo sempre a interação x_3 sendo retardada:

a) O tempo de retardamento é menor que o tempo de *timeout* da IUT

Nesse caso, ocorreu um atraso na chegada de uma interação à IUT e o mecanismo *timeout* não foi ativado. O traço da Tabela 9 representa esta situação e deve sofrer o mesmo tratamento dado ao caso da falha de alteração (f_1).

Tabela 9 : Traço com falha f3 e retardamento menor que timeout

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_3
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
$L?x_3$	
<f3> L?x3'	$U!z_3'$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

A interação $L?x_3$ foi atrasada por um tempo *t*, inferior ao tempo de espera determinado para o mecanismo de *timeout*. Dessa forma, a interação $L?x_3'$ representa a interação $L?x_3$ chegando à IUT após o tempo *t* de atraso.

O resultado do tratamento do traço é mostrado na Tabela 10.

Tabela 10: Traço após tratamento da falha f3 e retardamento menor que timeout

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_3
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
<f3> L?x3'	$U!z_3'$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

b) O tempo de retardamento é maior que o tempo de *timeout* da IUT

Nesse caso, o mecanismo de *timeout* deve ter sido ativado e a IUT responde ao atraso da interação. Supõe-se, inicialmente, que o tempo de retardamento não é superior ao dobro do tempo de *timeout* da IUT.

Na Tabela 11 está representado o traço onde a interação $L?x_3$ foi atrasada por um tempo *t*, superior ao tempo de espera determinado para o mecanismo de *timeout*. Dessa forma, a interação $L?x_3'$ representa a interação $L?x_3$ chegando à IUT após o tempo *t* de atraso. Entretanto $U!z_3'$ surgiu no traço antes mesmo que a indicação da falha injetada. Isso porque

antes que o FIA liberasse a entrada $L?x_3'$, ocorreu *timeout* na IUT e ela respondeu com $U!z_3'$, que chegou antes ao traço.

Tabela 11: Traço com falha f3 e retardamento maior que timeout

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_4
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
$L?x_3$	$U!z_3'$
<f3> L?x3'	
	$U!z_3''$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

O tratamento do traço é feito da seguinte forma:

1. verificar se existe uma falha presente no traço; nesse caso é f_3, que sabemos ser um retardamento;
2. eliminar a primeira interação de entrada imediatamente antes da indicação de falha;
3. para cada uma das saídas sem entrada correspondente, que se encontram imediatamente antes da indicação de falha, inserir uma interação de entrada nula, indicando a ausência de entrada.

O traço resultante após o tratamento acima é mostrado na Tabela 12.

Tabela 12: Traço após tratamento da falha f3 e retardamento maior que timeout

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_4
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
null	$U!z_3'$
<f3> L?x3	$U!z_3''$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

Suponha agora um caso onde o *timeout* ocorre da IUT seja de 10 segundos de espera, e a entrada $L?x_3$ sofreu um atraso de 21 segundos (supõe-se aqui que o número de vezes que a IUT tenta responder com *timeout* seja maior ou igual a 2). Nesse caso, a IUT responde três vezes a esse atraso antes que o FIA libere a entrada $L?x_3'$. O traço obtido nesse caso está representado na Tabela 13.

Tabela 13: Traço com falha f3 e retardamento 2 vezes maior que timeout

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_5
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
$L?x_3$	$U!z_3'$
	$U!z_3''$
<f3> L?x3'	$U!z_3'''$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

Como pode ser visto no traço acima, $U!z_3'$ e $U!z_3''$ surgiram antes mesmo que a indicação da falha injetada (f_3). Isso porque antes que o FIA liberasse a entrada $L?x_3'$ ocorreu *timeout* na IUT por duas vezes ($L?x_3'$ representa a entrada $L?x_3$ após o atraso provocado pela falha f_3). O algoritmo, nesse caso, é o mesmo usado para tratar o traço da Tabela 11. O resultado do tratamento está descrito na Tabela 14.

Tabela 14: Traço após tratamento da falha f3 e retardamento 2 vezes maior que timeout

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_5
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
null	$U!z_3'$
null	$U!z_3''$
<f3> L?x3'	$U!z_3'''$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

8.2.4 Tratamento do Traço com Falha f_4 (Supressão de Entradas)

Supondo que a interação $L?x_3$ tenha sido suprimida e tenha ocorrido *timeout* por duas vezes. Esse caso é parecido com o caso da falha f_3, quando o tempo de retardamento é maior que o tempo de *timeout*. A diferença é que, nesse caso, uma interação atingida pela falha f_4 nunca chega à IUT, por se tratar de uma falha de supressão e não de atraso.

Na Tabela 15, a interação $L?x_3$ está presente no traço porque o TSC a enviou para lá. Entretanto, antes que ela chegasse à IUT, foi suprimida pelo FIA. Portanto, não deve constar no traço.

Tabela 15: Traço com falha f4

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_6
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
$L?x_3$	$U!z_3'$
	$U!z_3''$
<f4>	
	$U!z_3'''$
$L?x_4$	$U!z_4$
$L?x_5$	$U!z_5$

O tratamento é, então, composto dos seguintes passos:

1. verificar se existe uma falha presente no traço; nesse caso é f_4, que sabemos ser uma supressão, onde o FIA retém uma interação destinada à IUT e esta não a recebe;
2. eliminar a primeira interação de entrada imediatamente antes da indicação de falha;
3. inserir após o indicador de falha de supressão, uma interação de entrada nula, a fim de completar a seqüência a ser reconhecida no modelo do comportamento da IUT.
4. se houver, antes da indicação de falha, saídas sem entrada correspondente, inserir para cada uma delas, uma interação de entrada nula; isso porque certamente ocorrerá *timeout* antes que a IUT perceba que a entrada foi, de fato, suprimida.

O resultado desse tratamento é mostrado na Tabela 16.

Tabela 16: Traço após tratamento da falha f4

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_6
$L?x_1$	$U!z_1$
$L?x_2$	$U!z_2$
null	$U!z_3'$
null	$U!z_3''$
$<f_4>$	
null	$U!z_3'''$
L?x4	$U!z_4$
$L?x_5$	$U!z_5$

8.3 Análise de Traço com um único PCO

Quando apenas uma das interfaces de uma IUT está disponível para observação de interações durante a execução de testes, significa que temos apenas um ponto de controle e observação (PCO). Isto implica que o traço de execução (seção 7.5) dos testes contém interações referentes a apenas um ponto de acesso (SAP) do protocolo. Entretanto, no modelo do comportamento da IUT, geralmente, ambos os SAPs são previstos: o inferior e o superior.

Quando o traço contém interações referentes a apenas um dos SAPs e o modelo prevê os dois SAPs, será necessário realizar uma projeção [BDZ89] no modelo. Essa operação equivale a desconsiderar no modelo todas as interações que estão associadas ao SAP que não está disponível para a observação dos testes.

Quando uma projeção é feita, menos informações serão consideradas durante a análise de resultados e o poder de detecção de erros, comparado à análise com informações de dois PCOs, é menor [BDZ89, DB85].

A situação caracterizada pela disponibilidade de apenas um PCO não é influenciada pela execução da injeção de falhas.

8.4 Algoritmo de Análise do Traço

O principal objetivo do algoritmo de análise é detectar quais casos de testes produziram resultados corretos e quais detectaram resultados incorretos, com base no modelo do comportamento da IUT. De forma complementar, serão fornecidas informações sobre ativação dos mecanismos de tolerância a falhas. Essas informações dão uma idéia de quais MTFs foram ativados na presença de falhas para as quais foram projetados para tratar.

O algoritmo de análise de traço é representado pelo procedimento Analisar, que está descrito no quadro da Figura 14. Ele invoca um procedimento de recuperação quando detecta um erro. Entende-se por recuperação a tentativa de ignorar um erro e tentar continuar a análise normalmente. Toda tentativa de recuperação de erro implica na geração de um diagnóstico. O objetivo dos diagnósticos é ajudar a procurar as causas dos erros detectados.

Os dois procedimentos invocados pelo procedimento **Analisar** são:

1) **Verificar_MTF** (Figura 15): verifica se um MTF foi ativado corretamente ou se deveria ser ativado.

2) **Recuperar** (Figura 16): tenta a recuperação de um erro detectado e gera um diagnóstico para o erro.

```
Procedimento Analisar (Traço, Modelo, Caminho_Atual)
Se não for fim do traço então
    Se não for fim do caso de teste então
        Se é possível avançar no modelo então
            Se houver mais de um caminho possível então
                armazenar uma lista com os caminhos possíveis a partir do estado atual
                guardar um ponto de retrocesso com o estado atual do modelo e a posição atual do traço
                escolher o primeiro caminho disponível
            Fim-Se
            invocar o procedimento Verificar_MTF
            avançar no traço e no modelo
            Analisar (Traço, Modelo, Caminho_Atual)
        Senão
            incrementar N    // número de erros a serem recuperados no caso de teste atual
            Se N <= que máximo permitido então
                invocar o procedimento Recuperar para cada um dos caminhos que partem do estado atual
                Se foi possível fazer recuperação então
                    continuar análise a partir da posição atual do modelo e do traço
                Senão
                    passar para o próximo caso de teste do traço e posicionar o estado inicial do modelo
                Fim-Se
                Analisar (Traço, Modelo, Caminho_Atual)
            Senão
                subtrair de N o número de erros do caminho atual e abandonar esse caminho
                voltar ao último ponto de retrocesso
                Se houver um próximo caminho a seguir na lista de possíveis caminhos então
                    escolher um novo caminho disponível no modelo
                    Analisar (Traço, Modelo, Caminho_Atual)
                Senão
                    Se houver mais pontos de retrocesso então
                        volta ao ponto de retrocesso mais próximo e escolher um novo caminho disponível
                        Analisar (Traço, Modelo, Caminho_Atual)
                    Senão
                        pular para o próximo caso de teste do traço
                        Analisar (Traço, Modelo, Caminho_Atual)
                    Fim-Se
                Fim-Se
            Fim-Se
        Fim-Se
    Senão        // significa que o caso de teste chegou ao fim
        Se N = 0 então
            exibir Veredicto "caso de teste é válido" e informação sobre ativação de MTFs
        Senão
            Se houver mais caminhos não investigados Então
                zerar N e voltar ao último ponto de retrocesso que tenha caminhos não investigados
                Analisar (Traço, Modelo, , Caminho_Atual)
            Senão
                exibir veredicto "caso de teste é inválido", diagnósticos de correção dos erros detectados e
                informação sobre ativação de MTFs
            Fim-Se
        Fim-se
        exibir Informações sobre ativação de MTFs e zerar N para o próximo caso de teste
    Fim-Se
```

Figura 14: Algoritmo de Análise de Traço

```
Procedimento Verificar_MTF
Se f_M ≠ f_0 então                    // f_M = tipo de falha presente no modelo
        Se existe f_T então             // f_T = tipo de falha presente no traço
            Se f_T = f_M então
                Exibir ("MTF foi ativado corretamente")
            Senão
                Exibir ("MTF foi ativado equivocadamente")
            Fim-Se
        Senão
            Exibir("MTF foi ativado sem a presença de falhas no traço")
        Fim-Se
Senão
        Se existe f_T então
            Exibir("MTF deveria ser ativado e não foi")
        Senão
           Exibir("Nenhum MTF foi ativado")
        Fim-Se
Fim-Se
```

Figura 15: Algoritmo de verificação de MTFs

```
Procedimento Recuperar (tipo_de_diagnóstico, caminho_atual)
Caso tipo_de_diagnóstico
        1: Análise de interações errôneas
                Avançar no modelo supondo que há interação faltando no traço
                Se interação do traço = interação do modelo então
                    GerarDiagnóstico("interação i deveria ser j no traço do caso de teste C_n")
            Fim-Se
        2: Análise de interações sobrando
                Avançar no traço supondo que há uma interação sobrando nele
                Se interação do traço = interação do modelo então
                    GerarDiagnóstico("interação i está sobrando na posição traço do caso de teste Cn ")
                Fim-Se
        3: Análise de interações faltando
                Avançar no modelo supondo que há interação faltando no traço
                Se interação do traço = interação do modelo então
                    GerarDiagnóstico("interação i está faltando no traço do caso de teste C_n ")
                Fim-Se
  Fim-Caso

Se foi possível recuperação e diagnóstico Então
        Retornar ponto atual do modelo e do traço para continuar a análise do caso de teste
Senão
        Retornar ("impossível recuperação")
Fim-Se
```

Figura 16: Algoritmo de Recuperação de Erros e Diagnósticos

8.5 Produtos da Análise de Traço

A análise de traço tem vários produtos. O principal deles é o veredicto para cada caso de teste, com o fim de detectar se um caso de teste detectou algum erro. Para ajudar a se levantar as causas dos erros detectados, são fornecidos diagnósticos. Para verificar se um MTF foi ou não ativado na presença de falhas, são fornecidas indicações a respeito.

São mencionadas em seguida, antes mesmo da apresentação dos produtos da análise propriamente ditos, as situações de erro que a análise de traço tenta detectar. Logo em seguida são apresentados os produtos da análise que permitem a cobertura dessas situações.

8.5.1 Tipos de Erros Detectados

Os seguintes tipos de erros podem ser detectados através dos produtos da análise de traço:

a) Saídas inválidas: quando saídas geradas pela IUT não pertencem ao conjunto de interações de saída do modelo de seu comportamento.

b) Modelo incorreto: quando, para uma dada interação de entrada lida do traço, não há no modelo do comportamento nenhuma transição especificada que conduza a um dos possíveis estados desse modelo.

A Tabela 17 ilustra as decisões (ações) tomadas na análise de traço de acordo com certas condições. As condições são as seguintes:

a) Modelo completo: significa que o modelo do comportamento, baseado em EFSM, está completo (seção 7.4).

b) Traço completo: significa que todos os casos de testes propostos inicialmente foram executados. O número de casos de testes inicial e o número de casos de testes executados são obtidos no final de cada traço, conforme mostrado na Tabela 4.

c) Erros Encontrados: indica se erros foram detectados, ou não, durante a análise.

d) Recuperação com Sucesso: indica se a recuperação de um erro detectado teve sucesso durante a análise.

Dadas as condições a serem analisadas e as decisões (ações) a serem tomadas, segue abaixo a interpretação da Tabela 17, separada em casos de colunas com resultados coincidentes:

a) Colunas 1, 6, 9 e 12: nenhum erro foi encontrado durante a análise. As demais condições não são levadas em consideração, portanto, a ação 1 é selecionada.

b) Colunas 2, 4, 7 e 10: erros foram detectados mas sempre foi possível a recuperação para continuar a análise. As demais condições não são levadas em consideração, portanto, a ações 2 e 6 são selecionadas.

c) Colunas 3 e 5: em ambos os casos, erros foram detectados e não foi possível a recuperação para continuar a análise. Observa-se que o modelo é completo, portanto, os erros detectados são referentes às saídas da IUT incompatíveis com o modelo, o que pode indicar uma possível falha de projeto que tenha sido corrigida somente na implementação. As ações 3, 5, 6 e 7 são selecionadas.

d) Colunas 8 e 11: em ambos os casos, erros foram detectados e não foi possível a recuperação para continuar a análise. Entretanto, o modelo não é completo e, por isso, pode-se supor tanto uma falha de projeto (transições ausentes ou especificadas incorretamente) quanto uma falha de projeto que tenha sido corrigida somente na implementação. Ações 3, 4, 5, 6 e 7 são selecionadas. A ação 6 foi selecionada nesse caso porque, mesmo que a recuperação do erro detectado não tenha sucesso, um diagnóstico para esse erro é emitido.

Tabela 17: Situações cobertas pela análise de traço

Condições		1	2	3	4	5	6	7	8	9	10	11	12
1	Modelo Completo	S	S	S	S	S	S	N	N	N	N	N	N
2	Traço Completo	S	S	S	N	N	N	S	S	S	N	N	N
3	Erros Encontrados	N	S	S	S	S	N	S	S	N	S	S	N
4	Recuperação com Sucesso	-	S	N	S	N	-	S	N	-	S	N	-
Ações													
1	Veredicto: "Passou"	X					X			X			X
2	Veredicto: "Falhou"		X		X			X			X		
3	Veredicto: "Inconclusivo"			X		X			X			X	
4	Advertência* "Possível Falha de Projeto"								X			X	
5	Advertência "Possível Falha de Projeto corrigida durante a implementação "			X		X			X			X	
6	Emissão de Diagnósticos de Erro		X	X	X	X		X	X		X	X	
7	Paralisação da Análise			X		X			X			X	

* A falha de projeto pode ter dois motivos: transições não especificadas e transições especificadas incorretamente no modelo.

8.5.2 Veredictos

Um veredicto é uma declaração que informa se um caso de teste detectou, ou não, erros. O resultado também pode ser inconclusivo, se uma condição não é satisfeita para fins de análise. Os veredictos possíveis considerados nesse trabalho estão baseados em [Ray87, JB83]:

1) "Passou" : significa que as saídas produzidas pela IUT, em resposta às entradas dos casos de testes, estão corretas conforme o modelo do comportamento da IUT.

2) "Falhou" : significa que as saídas produzidas pela IUT não estão conformes às saídas especificadas no modelo do comportamento.

3) "Inconclusivo": nada se pode dizer porque vários erros foram desencadeados em seguida de forma que não foi possível realizar a recuperação desses erros com sucesso. A causa para esse problema pode ser um modelo incompleto, um modelo especificado incorretamente, ou falhas que tenham sido corrigidas na implementação mas não no modelo.

8.5.3 Diagnósticos

A função básica da análise de traço é determinar a validade de um traço de execução, o que é reportado através dos veredictos mencionados na seção 8.5.2. Entretanto é extremamente útil o fornecimento de diagnósticos [Bel89, BP94] O objetivo dos diagnósticos é ajudar na eliminação de falhas (seção 2.3), mostrando possíveis causas de um erro detectado. Podem ajudar também na previsão de falhas (seção 2.3), quando tentam explicar os resultados ruins obtidos da avaliação do sistema, tais como baixa cobertura da tolerância a falhas ou baixa confiabilidade.

Os diagnósticos fornecidos durante a análise de resultados de testes, propostos nesse trabalho, consideram as seguintes causas possíveis de erros:

a) **Interações erradas**: não há correspondência entre uma interação presente no traço de execução e sua correspondente no modelo.

b) **Interações a mais**: existe uma interação sobrando no traço.

c) **Interações a menos**: uma das interações que deveria estar no traço está ausente.

Um diagnóstico é gerado quando se tenta a recuperação de um erro detectado para que a análise do traço possa prosseguir mesmo após erro. O número de tentativas (N) de recuperações pode ser escolhido pelo usuário do analisador de traços. Portanto, se $N > 1$, pode haver mais de um diagnóstico para cada caso de teste, dependendo do número de recuperações. Se $N = 0$, apenas os veredictos são fornecidos para os casos de testes. O algoritmo de recuperação de erros e geração de diagnósticos está descrito na seção 8.4.

8.5.4 Informações sobre MTFs

Independentemente do veredicto dado a um caso de teste, são fornecidas informações sobre quais MTFs foram ativados e se foram ativados corretamente na presença de falhas. Isso pode ser conseguido devido à indicação dos tipos de falhas presentes no traço de execução e também à identificação das transições que correspondem aos MTFs dentro do modelo do comportamento da IUT.

Esse tipo de informação pode ajudar na procura das causas para erros detectados, na explicação de uma baixa confiabilidade do *software*, por exemplo, e na obtenção de medidas tais como a probabilidade de um erro ser tolerado [AT95]. Os seguintes casos podem ser identificados no que diz respeito ao comportamento dos MTFs:

a) **MTF foi ativado**: quando, durante a análise do traço, uma transição do modelo que indica o tratamento de algum tipo de falha está sendo exercitada. Podemos, então, detectar as seguintes situações:

1) Se o caso de teste "passou", significa que ele produziu saídas corretas, provavelmente porque um MTF tratou uma falha que possa ter ocorrido;

2) Se o caso de teste "falhou", significa que produziu resultados incorretos embora um MTF tenha sido ativado. Isso pode ser uma indicação de que o MTF não esteja funcionando conforme sua especificação.

Em ambas as situações acima, é importante notar que se o traço indica a injeção de algum tipo de falha, essa é a causa da ativação do MTF. Caso contrário, o MTF foi ativado mesmo sem a presença de falhas injetadas, o que leva a crer que uma falha fora do experimento o tenha ativado. E, ainda, se uma falha indicada no traço não é a mesma que foi ativada no modelo, significa que um MTF foi ativado equivocadamente.

b) **MTF não foi ativado**: quando nenhuma transição que indica o tratamento de tipo de falha está sendo exercitada. Podemos, então, detectar as seguintes situações:

1) Se o caso de teste "passou", significa que ele produziu saídas corretas por execução normal (sem falhas) do sistema ou porque um erro foi mascarado (o sistema continua a fornecer o serviço correto, mesmo em presença de erros);

2) Se o caso de teste "falhou", significa que produziu resultados incorretos, provavelmente porque uma falha não foi detectada e tolerada por um MTF.

Em ambas as situações acima, se o traço indica a injeção de algum tipo de falha, significa que o MTF deveria ser ativado e não foi. Caso contrário, se o traço indica a ausência de falhas injetadas, e se o caso de teste é inválido, significa que nenhum MTF foi projetado para tratar da falha que ocorreu no sistema.

8.6 Uma Aplicação Usando Análise de Traço e Geração de Diagnósticos

Esta seção descreve uma aplicação do algoritmo proposto para análise de traço (seção 8.4). Os resultados dessa aplicação foram obtidos através de uma implementação inicial do algoritmo de análise. Essa implementação foi construída na linguagem de programação C.

O modelo da Figura 10 é usado nesse exemplo. A Figura 17 é uma repetição da Figura 10 para facilitar o acompanhamento do exemplo junto ao modelo usado como referência. O traço de cada caso de teste é analisado separadamente, obtendo-se como produtos: o veredicto, os diagnósticos e as informações sobre os MTFs (quando houver injeção de falhas). Logo em seguida a cada caso de teste, há um comentário de interpretação dos produtos da análise do traço. Os algoritmos descritos na seção 8.4 são usados para produzir o resultado da análise do traço dos seguintes casos de testes:

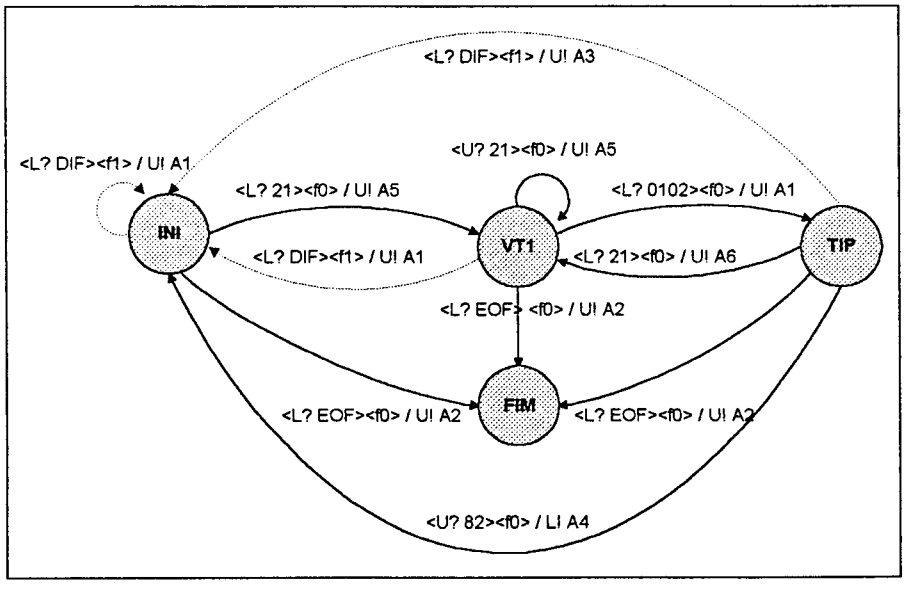

Figura 17: Modelo do comportamento usado como referência no exemplo de aplicação

a) Traço do Caso de teste C1

Tabela 18: Traço do caso de teste C1

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_1
L? 21	U!A5
L?0102	U!A1
L?21	U!A6
L?EOF	U!A2

Resultados da análise do traço da Tabela 19:

Veredicto: "passou"
Nenhum MTF ativado
Nenhum diagnóstico de erro

Comentário: caso de teste que não detectou erros

b) Traço do Caso de teste C2

Tabela 19: Traço do Caso de Teste C2

Traço	T_1
Grupo de Teste	G_1
Caso de Teste	C_2
L? 21	
L?0102	U!A1
L?21	U!A6
L?EOF	U!A2

Resultados da análise do traço da Tabela 19:

Veredicto: "falhou"
Nenhum MTF ativado
Diagnóstico 1: Interação U!A5 está faltando na posição 2

Comentário: caso de teste detectou um erro de supressão de interação, e o diagnóstico sugere que deveria haver uma interação *t* na segunda posição do traço de C_2

c) Traço do Caso de teste C3

Tabela 20: Traço do Caso de Teste C3

Traço	T_1
Grupo de Teste	G_2
Caso de Teste	C_3
L? 21	U!A5
	U!A5
L?0102	U!A1
L?EOF	U!A2

Resultados da análise do traço da Tabela 20:

Veredicto: "falhou"
Nenhum MTF ativado
Diagnóstico 1: Interação U!A2 está sobrando na posição 3

Comentário: caso de teste detectou um erro de sobra de interação, e o diagnóstico sugere que a interação *y*, localizada na terceira posição do traço de C_3, não deveria existir.

d) Traço do Caso de Teste C5

Tabela 21: Traço do Caso de Teste C5

Traço	T_1
Grupo de Teste	G_2
Caso de Teste	C_5
L? 21	U!A5
L?0102	U!A1
<f1> L?DIF	U!A3
L?EOF	U!A1

Veredicto: "passou"
MTF ativado para a falha f_1
Nenhum diagnóstico de erro

Comentário: caso de teste não detectou erros, embora haja uma indicação de que a falha f_1 foi injetada durante a execução do caso de teste C_5 (a interação **<f1> L?DIF** foi enviada ao traço pelo FIA). Além disso, um MTF foi ativado e, pelo veredicto, podemos supor que ele trabalhou corretamente.

9. Especificação da Ferramenta

Nesse capítulo é descrita a especificação da ferramenta de análise de resultados proposta neste trabalho. Para a análise e projeto da ferramenta foi utilizada a metodologia OMT [RBP+91]. A OMT combina três visões de modelagem: modelo de objetos, modelo dinâmico e modelo funcional.

A ferramenta de análise de resultados é representada por esses três modelos. O modelo de objetos descreve os aspectos estáticos da ferramenta. O modelo dinâmico representa os aspectos relativos à seqüência de operações ocorrida no decorrer do tempo. O modelo funcional representa os aspectos relativos às transformações de valores.

Em seguida é apresentada uma breve descrição do que a ferramenta deve realizar, e logo após são descritos os modelos de objeto, dinâmico e funcional, referentes à metodologia usada.

9.1 Descrição da Ferramenta

A meta a ser atingida pela ferramenta de análise de resultados de testes é fornecer um veredicto para cada interação de teste (evento de teste), diagnósticos de correção de erros detectados num caso de teste, e informações sobre a ativação, ou não, de mecanismos de tolerância a falhas durante a execução dos testes. Essas tarefas são executadas por um analisador de traço de execução dos testes.

O analisador de traço usa um modelo de referência (modelo do comportamento) da IUT para verificar o traço (histórico de execução) obtido durante a execução dos casos de testes (capítulo 8). O modelo de referência descreve o comportamento operacional que a IUT deve apresentar. O traço contém todas as interações de testes fornecidas (entradas) e todas as interações produzidas (saídas) pela IUT, mais as falhas injetadas durante os testes.

Após a análise do traço de execução, o analisador também produz um relatório de testes contendo aspectos estatísticos da execução dos testes.

Os aspectos de *interface* são independentes da ferramenta e estão definidos em [Gui96].

9.2 Modelo de Objetos

O modelo de objetos é composto por um diagrama de objetos, uma descrição dissertativa de suas classes, seus relacionamentos e um dicionário de dados que descreve cada classe separadamente juntamente com suas operações e atributos.

9.2.1 Descrição do Diagrama de Objetos

O diagrama de objetos para a ferramenta de análise de resultados é apresentado na Figura 18. A classe **Analisador de Traço** é o componente principal do modelo de objetos. Para gerenciar suas interações com a *interface*, essa classe possui um **Controlador**, que por sua vez, contém uma **Visão**. Esta última representa a *interface* com o usuário da ferramenta..

Um objeto **Controlador** gerência a interação entre o objeto **Analisador de Traço** e o objeto **Visão**. Para isso, o objeto **Controlador** deve conhecer os serviços oferecidos pelos objetos **Analisador de Traço** e **Visão**, mas não tem conhecimento de nenhum detalhe interno de como esses objetos prestam esses serviços. O módulo de interface com o usuário é independente do desenvolvimento da ferramenta. Maiores detalhes sobre o uso do *framework* no desenvolvimento de ferramentas interativas para um ambiente integrado de testes podem ser obtidas em [Gui96].

Quando o **Analisador de Traço** é invocado pelo usuário, é preciso informar qual é o **Traço de Execução** a ser verificado. Selecionado o traço, o analisador toma como referência o **Modelo do Comportamento** da IUT para analisá-lo.

O objeto **Modelo do Comportamento** é baseado em máquina finita de estados estendida e é composto de um ou mais objetos **Estado** e um ou mais objetos **Transição**. Um objeto **Estado** pode ser origem ou destino de um ou mais objeto **Transição**.

O objeto **Traço de Execução** é composto de objetos **Grupo de Teste** e de objetos **Grupo de Falha**. Um **Grupo de Teste** é composto de objetos **Caso de Teste**, e estes são compostos por objetos **Interação de Teste**. Uma **Interação de Teste** pode ser tanto uma entrada fornecida à IUT, quanto uma saída produzida pela IUT.

Um **Grupo de Falha** é composto de objetos **Caso de Falha**, e estes são compostos por objetos **Interação de Falha**. Uma **Interação de Falha** representa uma falha que foi injetada num dos objetos **Interação de Teste**.

Quando o **Analisador de Traço** verifica o **Traço de Execução**, associa a cada **Caso de Teste** um veredicto, caso todos os objetos **Interação de Teste** de entrada tenham um objeto **Interação de Teste** de saída correspondente correto. Também associa a cada **Caso de Teste**, os diagnósticos de erros quando erros são detectados na análise, e informações sobre ativação de mecanismos de tolerância a falhas acionados durante a execução do **Caso de Teste**.

Ao final da verificação do **Traço de Execução**, o **Analisador de Traço** produz um objeto **Relator dos Resultados**, que reúne informações do atributo ***status*** do objeto **Traço de Execução,** dos objetos **Grupo de Teste** e **Grupo de Falha**, e dos objetos **Caso de Teste** e **Caso de Falha.**

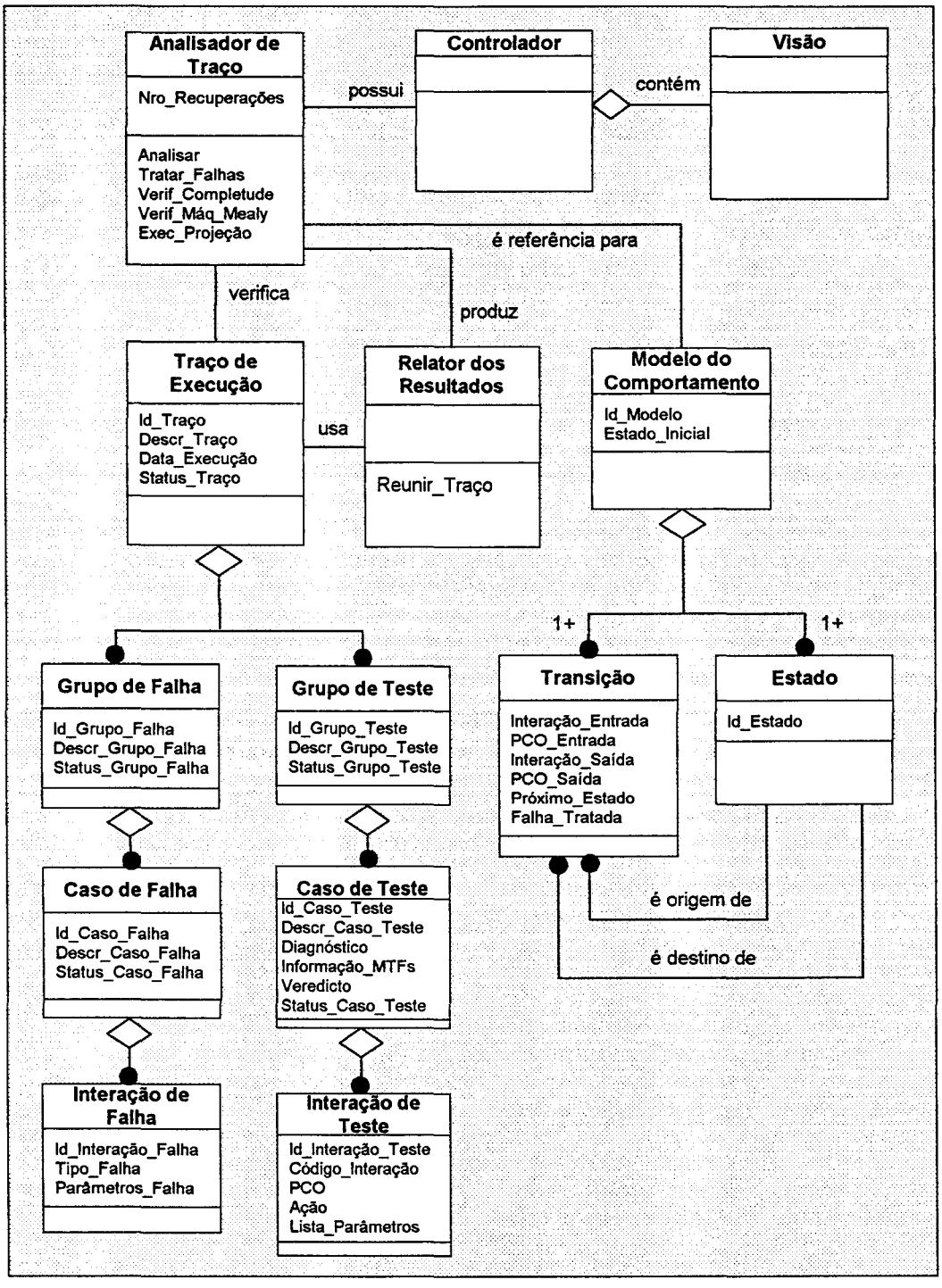

Figura 18: Diagrama de objetos da ferramenta

9.2.2 Dicionário de Dados

A seguir é apresentado o dicionário de dados para o modelo de objetos da Figura 18. Nele são descritas as classes, os atributos e as operações do diagrama de objetos.

Classe *Visão*: classe da interface usuário responsável pela comunicação visual com o usuário. É uma classe prestadora de serviços (operações de interface) à classe *Controlador* .

Classe *Controlador*: esta classe representa o controlador do Analisador de Traços. Coordena as interações entre a classe *Visão* (interface usuário) e a classe *Analisador de Traço* (ferramenta).

Classe *Analisador de traço*: inclui métodos que geram os produtos da análise de traço.

Atributos:

Nro_Recuperações: número máximo de tentativas de recuperação de erros por caso de teste

Operações:

Analisar: gerar veredictos, diagnósticos e informações sobre MTFs.

Tratar_Falhas: colocar o traço na forma adequada ao modelo do comportamento

Verif_Completude: verificar se o modelo do comportamento é completo

Verif_Máq_Mealy: verificar se o modelo do comportamento é uma máquina de Mealy

Exec_Projeção: executar a projeção num dado PCO, o que compreende isolar as interações do modelo relativas ao PCO considerado.

Classe *Modelo do Comportamento*: tabela de transições que representa uma máquina finita de estados estendida. Esta, por sua vez, representa o comportamento da IUT. É composta da agregação de um conjunto de *Estados* e um conjunto de *Transições*, cujas quantidades podem indicar o tamanho e complexidade do modelo.

Atributos:

Id_Modelo: identificação para se localizar o modelo comportamento

Estado_Inicial: estado inicial do modelo, a partir de onde será dado o início à análise de um traço de execução

Classe *Estado*: estados do *Modelo do Comportamento* que compõem, juntamente com as transições, a tabela de transições.

Atributos:

Id_Estado: identificação única do estado na modelo do comportamento.

Classe *Transição*: transições que descrevem os estímulos (entradas) e respostas (saídas) que exercitam os estados do *Modelo do comportamento*.

Atributos:

Interação_Entrada: interação de entrada que estimula um estado
PCO_Entrada: PCO onde a interação de entrada foi observada
Interação_Saída: resposta à interação de entrada
PCO_Saída: PCO onde a interação de saída foi observada
Próximo_Estado: estado que será atingido em função do estado atual do modelo e do estímulo recebido (entrada)
Falha_Tratada: descreve o tipo de falha que a transição trata.

Classe *Traço de execução*: conjunto de informações que foram observadas em certos pontos de controle e observação da IUT durante a execução dos testes.

Atributos:

Id_Traço: identificação para localização do traço.
Descr_Traço: Descrição do traço.
Data_Execução: Data em que o traço foi produzido
Status_Traço: contém número de grupos de testes que o traço contém

Classe *Grupo de Falha*: uma seqüência de falhas é dividida em grupos de falhas para fins específicos como alteração de variáveis de memória, falhas de comunicação ou do processador.

Atributos:

Id_Grupo_Falha: identificador do grupo de falha
Descr_Grupo_Falha: descrição do grupo de falha
Status_Grupo_Falha: contém número de casos de falhas do grupo de falha

Classe *Caso de Falha*: um grupo de falha é dividido em casos de falha, cada um com um objetivo de falha específico.

Atributos:

Id_Caso_Falha: identificador do caso de falha
Descr_Caso_Falha: descrição do caso de falha
Status_Caso_Falha: contém o número de interações de falha dentro do caso de falha

Classe *Interação de Falha*: um grupo de falha é dividido em casos de falha, cada um com um objetivo de falha específico.

Atributos:

Id_Interação_Falha: identificador da interação de falha
Tipo_Falha: código do tipo de falha
Parâmetros_Falha: parâmetros relativos à falha aplicada

Classe *Grupo de teste*: um grupo de teste identifica o objetivo comum dos casos de testes que o compõem.

Atributos:

Id_Grupo_Teste: identificador do grupo de teste
Descr_Grupo_Teste: descrição do grupo de teste
Status_Grupo_Teste: contém o número de casos de testes do grupo de teste e o número de casos, de fato, executados

Classe *Caso de teste*: um grupo de teste é dividido em casos de testes, cada um com um objetivo de teste específico.

Atributos:

Id_Caso_Teste: identificador do caso de teste
Descr_Caso_Teste: descrição do caso de teste
Diagnóstico: diagnósticos de erro para interações do caso de teste
Informação_MTFs: informação sobre a ativação de MTFs durante a execução do caso de teste
Veredicto: veredicto ("passou", "falhou", "inconclusivo") para o caso de teste.
Status_Caso_Teste: contém o número de interações de teste dentro do caso de teste e o número de interação de teste executadas no caso de teste

Classe *Interação de Teste*: no contexto do traço de execução, uma interação é parte do resultado de um caso de teste, podendo esta corresponder tanto a um valor de entrada do caso de teste quanto a um valor de saída da IUT.

Atributos:

Id_Interação_Teste: identificador da interação de teste
Código_Interação: código da interação de teste
PCO: Ponto de controle e observação onde a interação foi observada
Ação: envio ou recebimento
Lista_Parâmetros: parte de dados da interação

Classe Relator dos Resultados: reúne os resultados da análise de traço em forma de relatório de testes para o usuário. Esses resultados podem ser vistos tanto para um traço recém analisado quanto para outros traços analisados anteriormente.

Operações:

Reunir_Traço: reunir os *status* do Traço de Execução, Grupo de Falha, Grupo de Teste, Caso de Teste, Caso de Falha, mais as informações sobre veredictos, diagnósticos e informações sobre MTFs.

9.3 Modelo Dinâmico

A ferramenta de análise de resultados de testes tem como principal componente um analisador de traços de execução. O processamento desse analisador é estritamente seqüencial, sendo que o único estímulo externo que ele pode receber é um sinal de interrupção do processamento. Dessa forma, não é representado aqui o modelo dinâmico da ferramenta.

9.4 Modelo Funcional

Nesta seção são apresentados os aspectos funcionais da ferramenta. Inicialmente são identificados os dados de entrada e de saída dos principais objetos da ferramenta. (Figura 19). O usuário interage com a visão (*interface*). A visão e o Analisador de Traço trocam informações através do **controlador** a fim de manter independência entre as funcionalidades da ferramenta e a interface [Gui96].

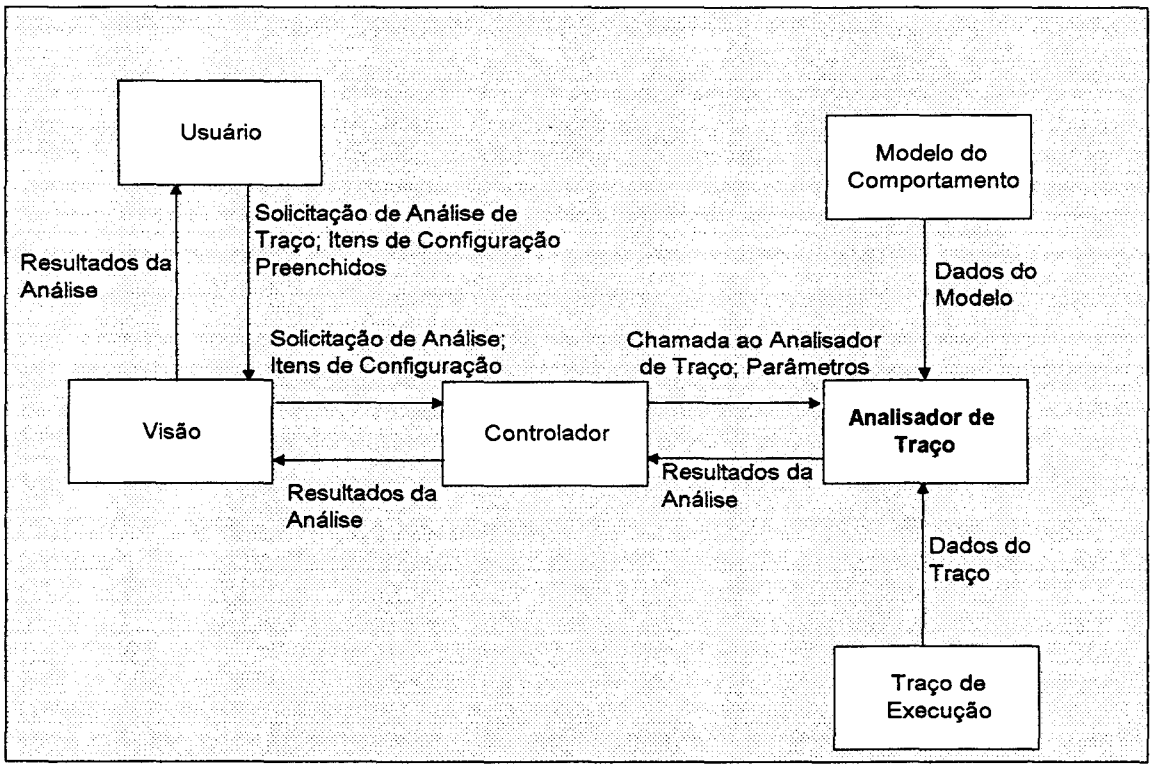

Figura 19: Dados de entrada e saída para a ferramenta de Análise de Traço

Em seguida é apresentado na Figura 20 o Diagrama de Fluxo de Dados (DFD) da classe **Analisador de Traço** num primeiro nível de detalhe. As elipses representam os processos (métodos) da classe. As setas indicam fluxos de dados. Os nomes envolvidos por duas retas paralelas são os depósitos de dados. Os detalhes apresentados ainda não chegaram num nível

de detalhe suficiente para o uso do DFD como base para uma implementação. Entretanto, mostram os principais processos do Analisador.

O Analisador de Traço executa os processos de verificação do modelo (Verif_Máq_Mealy e Verif_Completude), projeção do modelo (Exec_Projeção), tratamento do traço quanto às falhas injetadas (Tratar_Falhas) e, como sua principal função, oferece a análise do traço (Analisar).

As entradas para o analisador vêm do modelo do comportamento e do traço. As solicitações e configuração para a análise vêm do controlador.

A classe Analisador de Traço é responsável pela análise do traço, produzindo veredictos, diagnósticos e informações sobre a ativação dos MTFs. Para tanto, é necessário verificar se o modelo do comportamento da implementação sob teste corresponde à uma máquina de Mealy (processo Verif_Máq_Mealy) e se ele é completo (processo Verif_Completude). Se for necessário, será executada a projeção (processo Exec_Projeção) no modelo a fim de isolar as interações ocorridas em apenas um SAP da IUT. Quando falhas são injetadas durante os testes, é necessário realizar um tratamento do traço (processo Tratar_Falhas) de forma a deixá-lo num formato adequado para que a sua análise possa ser executada. A análise do traço, propriamente dita (processo Analisar), só pode ser executada quando o traço está organizado adequadamente e quando o modelo do comportamento corresponde a uma máquina de Mealy. As saídas do processo Analisar são os Resultados da Análise, que correspondem aos veredictos, diagnósticos de erros e informações sobre MTFs. Os depósitos de dados armazenam o traço de execução, o modelo do comportamento e o modelo eventualmente projetado[27].

[27] O modelo projetado tem o mesma estrutura do modelo do comportamento, entretanto, representa as interações de apenas um ponto de controle e observação da IUT.

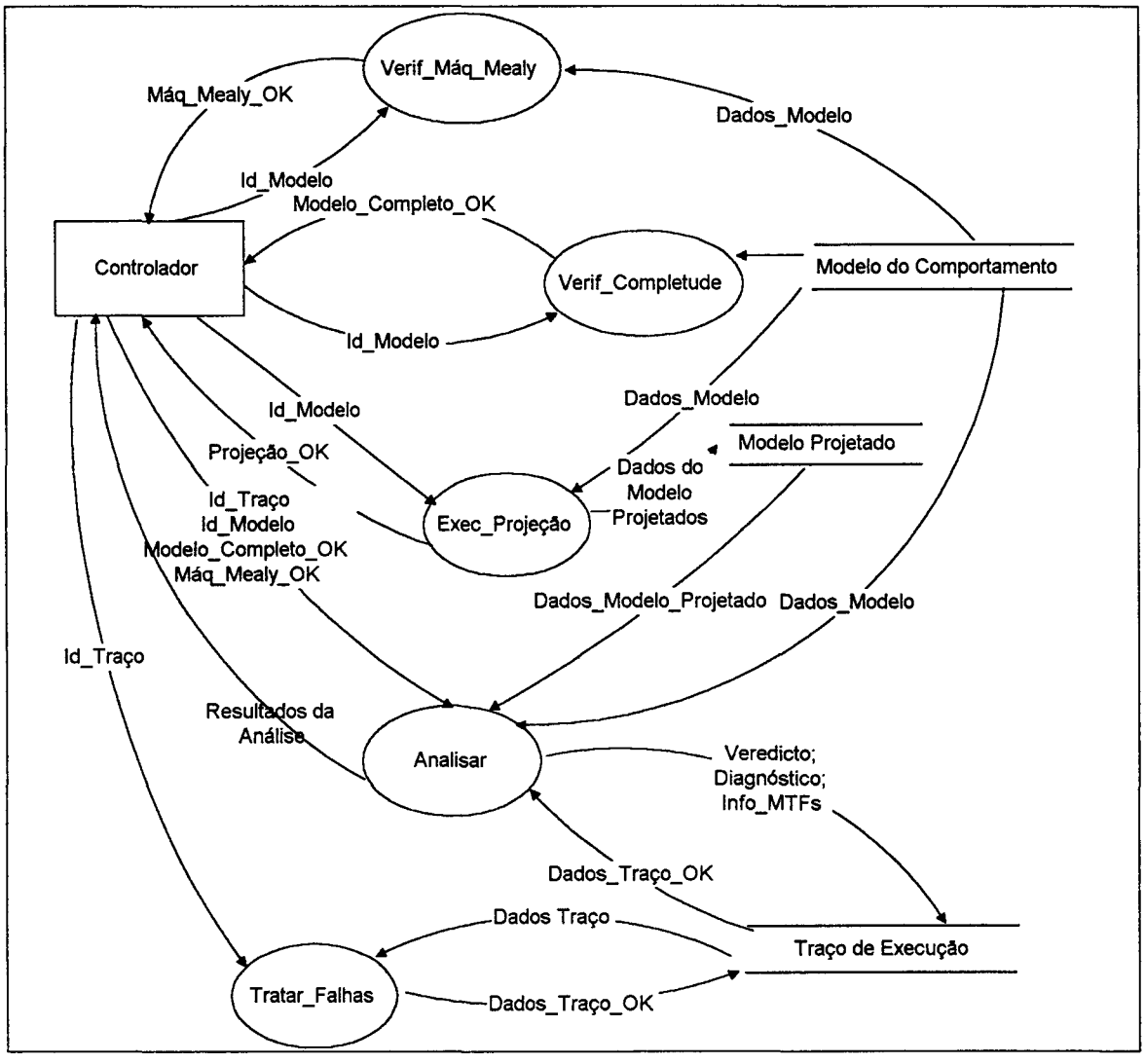

Figura 20: DFD da classe Analisador de Traço

A Figura 21 mostra a expansão do processo Analisar do DFD da Figura 20. O processo Analisar lê o traço de execução (processo Ler Traço) à medida que o compara ao modelo do comportamento da IUT (processo Ler Modelo) . Os veredictos são obtidos (processo Produzir Veredicto) com base na comparação do traço com o modelo e na informação sobre a completude do modelo. Os diagnósticos são gerados (processo Gerar Diagnóstico) com base no resultado da comparação do traço com o modelo e com os dados do traço e do modelo. As informações sobre a ativação dos MTFs são obtidas (processo Verif_MTFs) com base nos dados do modelo e do traço.

O algoritmo executado no processo Analisar e dos procedimentos invocados por ele estão descritos na seção 8.4.

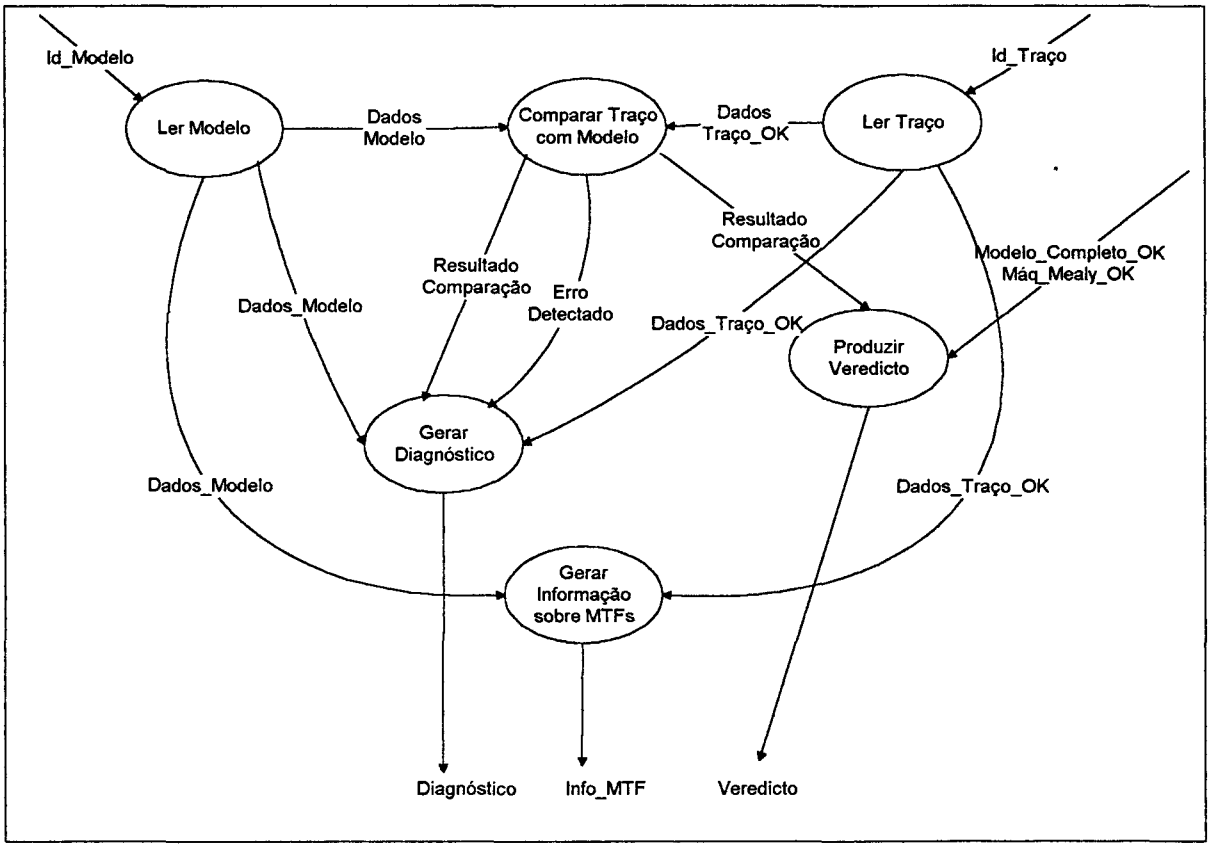

Figura 21: Expansão do processo Analisar

10. Conclusão

Este capítulo traz as conclusões obtidas durante o desenvolvimento desse trabalho. São abordados os objetivos iniciais do trabalho, as metas atingidas, as contribuições, as críticas ao trabalho e as extensões futuras.

10.1 O Que se Pretendia Realizar

O principal objetivo do trabalho, desde seu início, era desenvolver uma ferramenta de análise de resultados de testes que pudesse ser usada numa versão inicial do ATIFS (capítulo 6), que tem uma proposta concreta para ser um ambiente de teste de *software* completo.

10.2 O Que foi Realizado e Quais Foram as Dificuldades

O trabalho foi iniciado com vários estudos sobre análise de resultados de testes. Verificou-se um crescimento significante da importância desse tipo de estudo, principalmente nos últimos cinco anos, quando vários trabalhos começaram a surgir (a grande maioria no exterior).

Como o ATIFS tem como objetivo inicial o teste de protocolos de comunicação, foram investigadas com maior profundidade as técnicas de análise de resultados para testes de protocolos. Dessas técnicas investigadas, foi selecionada a análise de traço, por ser mais adequada ao ATIFS e por não interferir na execução dos testes.

Partiu-se então para a adaptação dessa técnica ao uso de injeção de falhas por *software*, o que fez surgir uma série de dificuldades. Devido ao tempo disponível para o trabalho, essas dificuldades tiveram que ser contornadas de forma parcial (com hipóteses sobre o funcionamento do sistema de testes), que é o caso do tratamento de falhas injetadas presentes no traço de execução.

O traço deve sofrer um tratamento antes de ser analisado devido à ordem em que o ambiente de testes grava as interações de testes no traço de execução. Esse tratamento é

dependente dos tipos de falha considerados no modelo do comportamento da IUT. Por isso cada tipo de falha implica uma situação diferente.

Após a definição de quais operações seriam necessárias ao traço, partiu-se para a desenvolvimento de um analisador de traço. Chegou-se a uma especificação contendo os três modelos da metodologia OMT.

Uma implementação inicial do algoritmo de análise de traço também foi realizada para gerar alguns exemplos de análise. Também foi construído um protótipo da interface, que pode ser visto no apêndice, na descrição do manual de uso da ferramenta especificada.

10.3 Contribuições

Contribui-se, principalmente, com a área de injeção de falhas porque auxilia na verificação do comportamento dos mecanismos de tolerância a falhas durante os testes. O tipo de análise realizado sobre os resultados dos testes por injeção de falhas foi proposto para substituir a análise por comparação de versões (com e sem falhas).

10.4 Extensões Futuras

As seguintes extensões são propostas para o atual trabalho:

a) Conclusão da implementação da ferramenta.

b) Experimentos para verificar a viabilidade do algoritmo de análise de traço para modelos de comportamento com número elevado de estados. Para isso, a ferramenta de análise desenvolvida deverá ser submetida a testes de protocolos reais.

c) Experimentos que comparem o desempenho do algoritmo de análise de traço quando o modelo do comportamento descreve o tratamento de falhas, com o desempenho quando da descrição apenas das funções normais da IUT.

d) Teste da ferramenta integrada ao ATIFS.

e) Investigação detalhada do tratamento de falhas no traço.

f) Extensão do método de análise de resultados para testes multicamadas (duas ou mais camadas adjacentes do sistema de comunicação consideradas como uma única camada para fins de realização de testes).

g) Investigação sobre a possibilidade de uso da ferramenta para testes estruturais (caixa branca)

10.5 Limitações e Críticas ao Trabalho

Este é um trabalho inicial na área de análise de resultados de testes. Claramente, ele não serve para analisar resultados de testes a partir de qualquer tipo de especificação. Não foi investigado também qual seria a mudança necessária à ferramenta se outro tipo de modelagem de comportamento (tais como ESTELLE, LOTOS, ou redes de Petri) fosse adotado.

Somente uma IUT é analisada de cada vez devido aos problemas de sincronização envolvidos na organização de um traço global de um sistema distribuído.

11. Referências Bibliográficas

[AAD79] J. M. Ayache, P. Azema, M. Diaz. Observer: a Concept for On-line Detection of Control Errors in Concurrent Systems. *Proc. FTCS-9*, Wisconsin, Madison, USA, 1979.

[AALC91] D. R. Avresky, J. Arlat, J. -C. Laprie, Y. Crouzet. Guiding the Process of Fault Injection for Testing Fault Tolerance. Relatório interno LAAS n. 91-387. dez. 1991.

[AT95] D. R. Avresky, P. K. Tapadiya. A Method for Developing a Software-based Fault Injection Tool. Relatório técnico n. 95-021 DCS Texas A&M University, 1995.

[BB89] G. v. Bochman, Omar B. Bellal. Test Result Analysis with Respect to Formal Specifications. Relatório interno do departamento de informática da Universidade de Montreal, Canadá, 1989.

[BDZ89] G. v. Bochmann, R. Dssouli, J. R. Zhao. Trace Analysis for Conformance and Arbitration Testing. *IEEE Transaction on Software Engineering*, vol. 15, n. 11, pp. 1347-56, nov. 1989.

[Bei90] B. Beizer. Software Testing Techniques, 2. ed., Thomson Comp. Press, 1990.

[Bel89] Omar B. Bellal. Analyse Automatique de Résultats de Tests Appliquée aux Protocoles de Communication. Tese de mestrado, Universidade de Montreal, Canadá, nov. 1989.

[BP94] G. v. Bochmann, A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. *ACM Software Engineering Notes*, pp. 109-24, ago. 1994.

[CLPZ90] S. T. Chanson, B. P. Lee, N. J. Parakh, H. X. Zeng. Design and Implementation of a Ferry Clip Test System. *9th IFIP Workshop Protocol Specification, Testing and Verification*, pp. 101-118, 1990.

[DB85] R. Dssouli, G. v. Bochmann. Error Detection with Multiple Observers, in *5th IFIP Workshop Protocol Specification, Testing and Verification*, pp. 483-94, Toulouse, França, 1985.

[DJC94] M. Diaz, G. Juanole, J.-P. Courtiat. Observer - a Concept for Formal On-line Validation of Distributed Systems. *IEEE Transactions on Software Engineering*, vol. 20, n.12, pp. 900-13, dez. 1994.

[DMM+87] R. A. DeMillo, W. . McCracken, R. J. Martin, J. F. Passafiume. Software Testing and Evaluation, The Benjamin Cummings, 1987.

[GB93] A. Ghedamsi, G. v. Bochmann. Diagnosing Multiple Faults in Finite State Machines. Publicação Interna n. 859, Département d'informatique et de recherche opérationnelle, Université de Montreal, jan. 1993.

[Gil62] Arthur Gill. Introduction to the Theory of Finite-state Machines, McGraw-Hill, 1962.

[Gui96] M. S. Guimarães. Um Framework de Ferramenta Interativa para um Ambiente Integrado de Testes. Dissertação de Mestrado, Instituto de Computação, Universidade Estadual de Campinas, dez. 1996.

[Hol91] G. J. Holzmann. Design and Validation of Computer Protocols, Prentice Hall, 1991.

[HUK95] O. Henniger, A. Ulrich, H. König. Transformation of Estelle Modules Aiming at Test Case Generation. *IWPTS - 8th International Workshop on Protocol Test Systems*, pp. 45-60, Evry, França, set. 1995.

[IS9646] OSI Conformance Testing Methodology and Framework.

[JB83] C. Jard, G. v. Bochmann. An Approach to Testing Specifications. *The Journal of Systems and Software*, pp. 315-23, mar. 1983.

[KSNM91] K. Katsuyama, F. Sato, T. Nakakawaji, e T. Mizuno. Strategic Testing Environment with Formal Description Techniques. *IEEE Transactions on Computers*, vol. 40, n. 4, abr. 1991.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21, n. 7, pp. 558-65, jul. 1978.

[Lap95] J. C. Laprie. Dependability - Its Attributes, Impairments and Means. *Predictably Dependable Computing Systems, ESPRIT Basic Research Series*, Springer-Verlag, 1995.

[Lin90] R. J. Linn Jr.. Conformance Testing for OSI Protocols. *Computer Networks and ISDN Systems*, 18, 1990, pp. 203- 219.

[LL87] J. C. B. Leite, O. G. Loques F°. *II SCTF*, cap. 4 do mini-curso intitulado: Introdução à Tolerância a Falhas, Campinas, SP, 1987.

[Mar95] E. Martins. ATIFS: um Ambiente de Testes baseado em Injeção de Falhas por Software. Relatório interno DCC-95-24, Departamento de Ciência da Computação, Universidade Estadual de Campinas, dez. 1995.

[MDA85] R. Molva, M. Diaz, J. M. Ayache. Observer: a Run-time Checking Tool for Local Area Networks, *5th IFIP Workshop Protocol Specification, Testing and Verification*, pp. 495-506, Toulouse, França, 1985

[Mye79] G. J. Myers. The Art of Software Testing, John Wiley & Sons, 1979.

[NS92] K. Naik, B. Sarikaya. Testing Communication Protocols. *IEEE Software*, pp. 27-37, jan. 1992.

[PM92] R. L. Probert, O. Monkewich. TTCN: The International Notation for Specifying Tests of Communications Systems. *Computer Networks and ISDN Systems*, 23, pp. 417- 437, 1992.

[Pre92] R. S. Pressman. Software Engineering: a Practitioner's Approach, 3. ed., McGraw-Hill, 1992.

[RDT95a] T. Ramalingam, Anindya Das, K. Thulasiraman. On Testing and Diagnosis of Communication Protocols Based on the FSM Model. *Computer Communications*, vol. 18, n. 5, pp. 329-338, mar. 1995.

[RDT95b] T. Ramalingam, Anindya Das, K. Thulasiraman. A Unified Test Case Generation Method for the FSM Model Using Context Independent Unique Sequences.. *IWPTS - 8th International Workshop on Protocol Test Systems*, pp. 289-305, Evry, França, set. 1995.

[Ray87] D. Rayner. OSI Conformance Testing. Computer Networks and ISDN Systems, 14, pp. 79-98, 1987.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. Object-oriented Modeling and Design, Prentice-Hall, 1991.

[Ric94] D. J. Richardson. TAOS: Testing with Analysis and Oracle Support. *ACM Software Engineering Notes*, ago. 1994.

[Saj84] M. Sajkowski. Protocol Verification Techniques: Status quo and Perspectives. 4th IFIP Workshop Protocol Specification, Testing and Verification, pp. 697-720, 1984.

[Sar88] B. Sarikaya. Protocol Test Generation, Trace Analysis and Verification Techniques. *2nd Workshop on Software Testing, Verification and Analysis*, pp. 123-30, Banff, Canadá, jul. 1988.

[Tan89] A. S. Tanenbaum. Computer Networks, 2. ed., Prentice-Hall, 1989.

[Wey82] E. J. Weyuker. On Testing Non-testable Programs. *The Computer Journal*, vol. 25, n. 4, pp. 465-70, 1982.

[ZDH88] H. X. Zeng, X. F. Du, C. S. He. Promoting the "Local" Test Method with the New Concept "Ferry Clip". *8th IFIP Workshop Protocol Specification, Testing and Verification*, pp. 231-241, 1988.

[ZR85] H. X. Zeng, D. Rayner. The Impact of the Ferry Concept on Protocol Testing. *5th IFIP Workshop Protocol Specification, Testing and Verification*, pp. 533-544, 1985.

Apêndice

Manual de Uso da Ferramenta Desenvolvida

Nesta seção são apresentados detalhes da interface que o usuário da ferramenta deve conhecer para poder analisar traços de execução e obter os resultados dessa análise.

A tela principal da ferramenta contém um conjunto de opções de *menu* geral (Arquivo, Editar, Ajuda) que dizem respeito à operações como abertura, fechamento e impressão de arquivos, recursos de *cut* e *paste*, e ajuda ao usuário. O opção de *menu* referente às funções específicas da ferramenta de análise de resultados de testes está representado na Figura 22 como "Análise de Resultados". Se essa opção for selecionada, dois itens estarão disponíveis: (1) Análise de Traço: é a solicitação para a execução da análise de traço; (2) Resultados de Análises: é a solicitação para se obter resultados de análises já executadas.

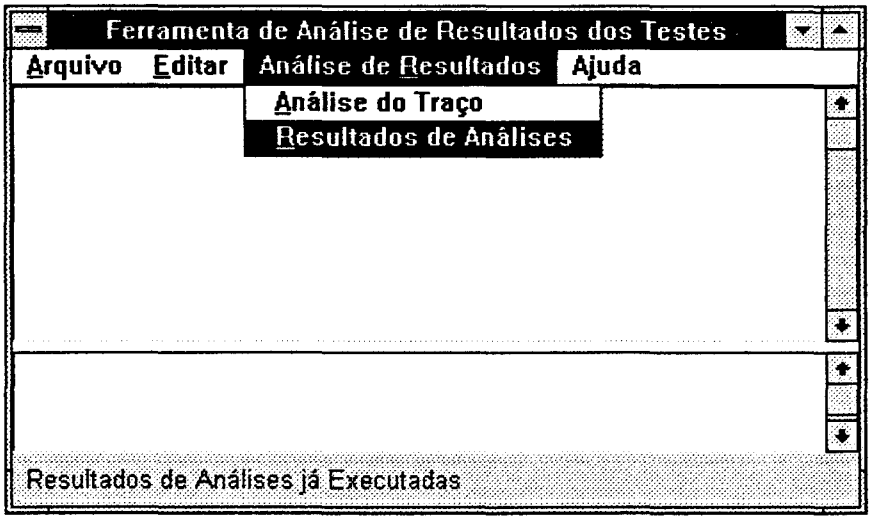

Figura 22: Tela principal da ferramenta de análise de resultados. de testes

Quando o item "Análise do Traço" do *menu* "Análise de Resultados" é selecionado, é solicitado ao usuário o preenchimento de um conjunto de itens de configuração necessários para o início da análise. A configuração que o usuário deve informar está descrita na Figura 23. O botão "Selecionar Traço de Execução..." solicita a localização do traço de execução a ser analisado através do quadro de diálogo mostrado na Figura 24. O botão "Selecionar Modelo EFSM..." solicita a localização do modelo do comportamento da mesma forma que o botão

"Selecionar Traço de Execução...". A caixa de texto "Estado Inicial do Modelo: " refere-se ao estado do modelo do comportamento em que a análise do traço deve iniciar. A caixa de texto "Máximo de Erros por Recuperação: " identifica o número máximo de tentativas de recuperação que o analisador deve tentar para cada caso de teste quando um erro é detectado. Os demais objetos da Figura 23 são descritos adiante.

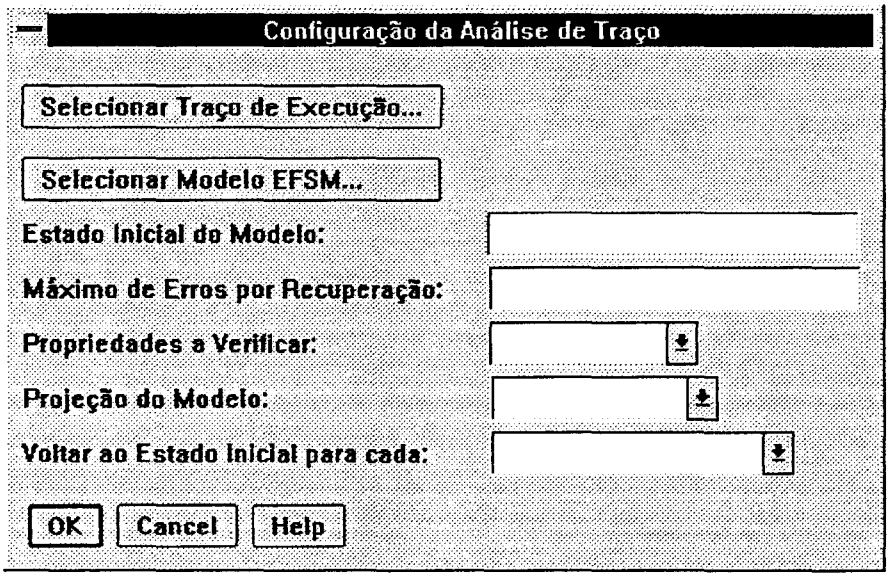

Figura 23: Quadro de diálogo de configuração da análise de traço

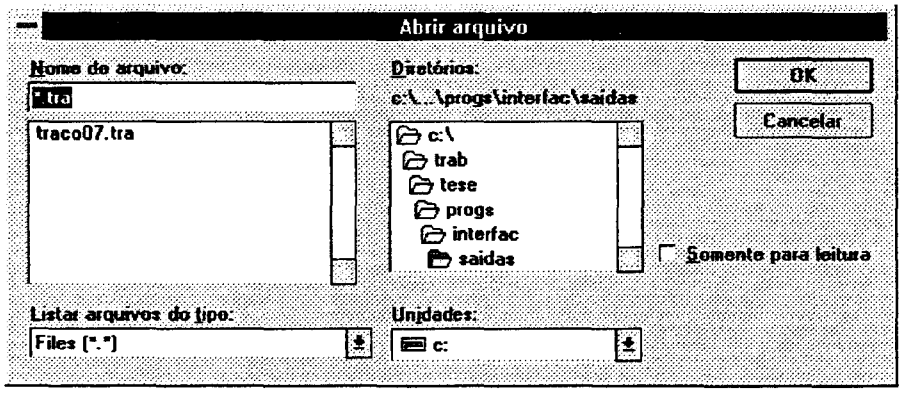

Figura 24: Caixa de seleção para os arquivos de traço de execução e de modelo do comportamento

Os três últimos itens de configuração do analisador de traço apresentados na Figura 23 estão em destaque nas três próximas figuras, respectivamente.

A Figura 25 mostra as opções para o item "Propriedades a Verificar: ". O usuário pode optar por:

a) **Nenhuma**: nenhuma propriedade do modelo é verificada antes do início da análise.

b) **Mealy**: é verificado se o modelo do comportamento é uma Máquina de Mealy.

c) **Completude**: é verificado se o modelo comportamento está completo.

d) **Ambas**: as duas propriedades anteriores são verificadas.

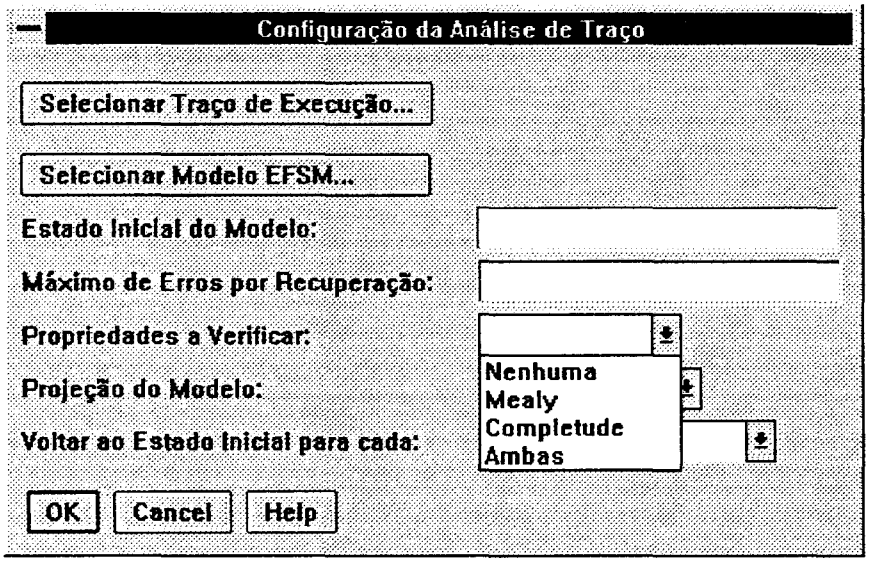

Figura 25:Seleção das propriedades a serem verificadas no modelo do comportamento

A Figura 26 mostra as opções para o item "Projeção do Modelo: ". O usuário pode optar por:

a) **Nenhuma**: nenhuma projeção é executada.

b) **PCO Inferior:** é realizada a projeção das interações do modelo do comportamento que dizem respeito ao Ponto de Controle e Observação inferior da IUT.

c) **PCO Inferior:** é realizada a projeção das interações do modelo do comportamento que dizem respeito ao Ponto de Controle e Observação superior da IUT.

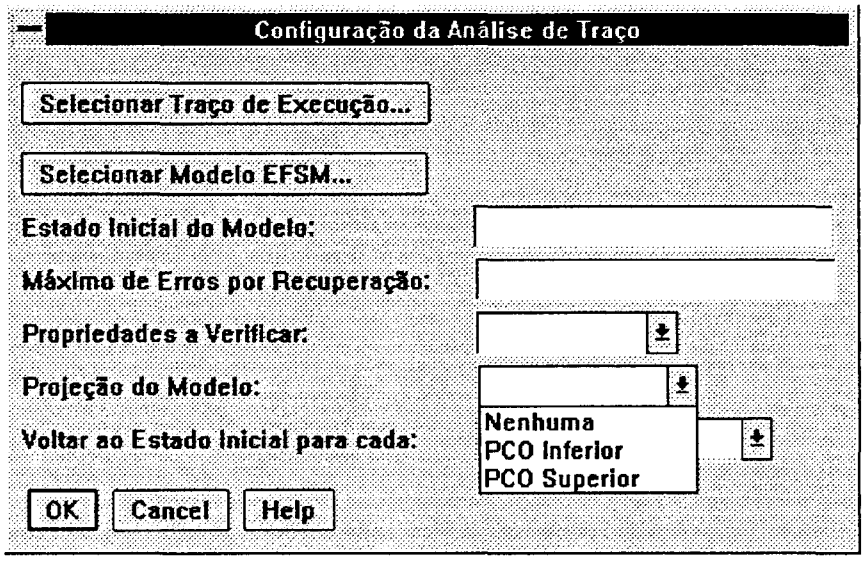

Figura 26: Seleção do Ponto de Controle e Observação dos teste para projeção

A Figura 27 mostra as opções para o item "Voltar ao Estado Inicial para cada: ". O usuário pode optar por:

a) **Caso de Teste**: significa que o analisador de traço volta ao estado inicial do modelo a cada novo caso de teste analisado.

b) **Grupo de Teste:** significa que o analisador de traço volta ao estado inicial do modelo a cada novo grupo de teste analisado.

c) **Seqüência de Teste:** significa que o analisador de traço volta ao estado inicial do modelo a cada nova seqüência de teste analisada.

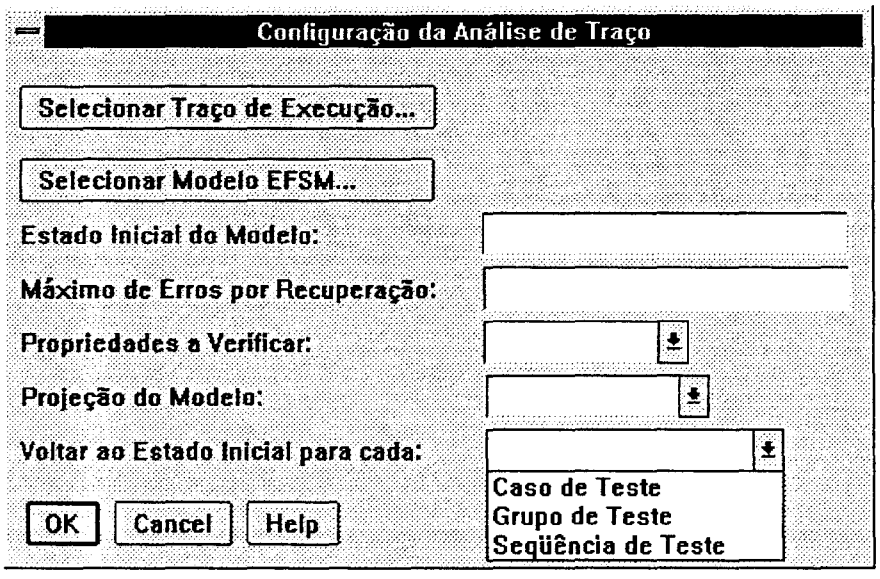

Figura 27: Seleção do tipo de seqüência durante a análise

Após preencher a configuração necessária, dá-se início ao processo de análise do traço de execução. Os passos da análise são mostrados na tela da ferramenta (Figura 28). Essa tela é divida em duas áreas de texto, na área maior, logo abaixo da linha de *menu*, é possível realizar edição de texto, abrir arquivos para anotações sobre a análise, etc. Também pode ser usada para mostrar informações sobre a análise. A área de texto menor, situada na porção inferior da tela mostra informações de acompanhamento da análise. Maiores detalhes sobre o projeto da interface com o usuário estão descritos em [Gui96].

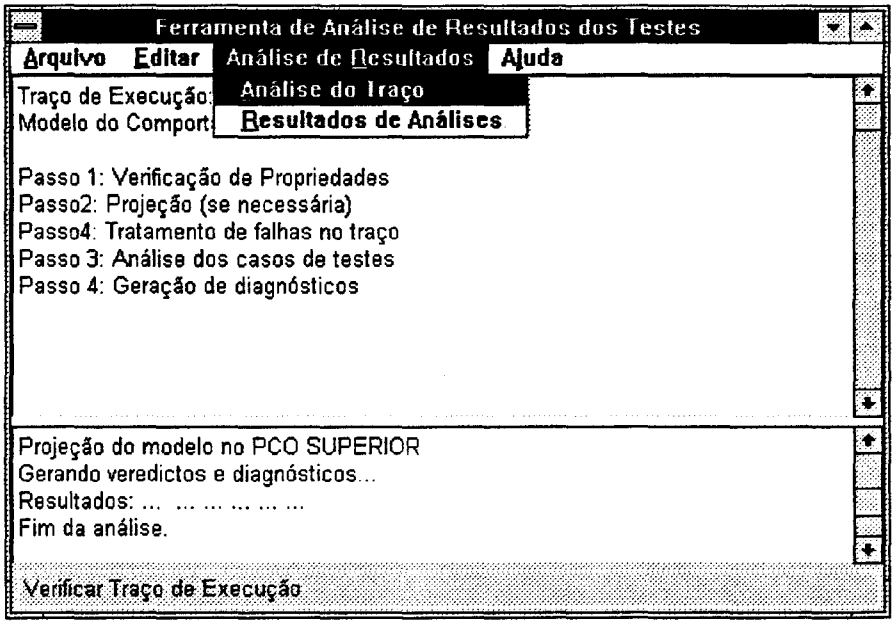

Figura 28: Tela durante a execução da análise do traço

Quando a análise de um traço é executada, um relatório sobre a análise é exibido na tela da ferramenta. Entretanto, se o usuário desejar ver os resultado da análise de um traço anterior, deve selecionar a opção "Resultados de Análises" do *menu* "Análise de Resultados", que pode ser visto na Figura 22.